How to Design Knowledge Programs

in Axioma

From first programs to data structures
and knowledge representation

Vlad Evenhar

Reviewed working edition
September 2026

© 2026 Vlad Evenhar.
Created and designed by Vlad Evenhar.
Implemented by AI agents.

Contents

How to Design Knowledge Programs — in Axioma 1

Chapter 1 · Arithmetic 7

Chapter 2 · Variables and Naming 21

Chapter 2a · Loops 30

Chapter 3 · Functions 35

Chapter 4 · The Design Recipe 44

Chapter 5 · Conditionals and Booleans 52

Chapter 6 · Arrays as Lists 71

Chapter 7 · Recursion 81

Chapter 8 · Trees via Concepts 98

Chapter 9 · Mutual Recursion 105

Chapter 10 · Higher-Order Functions 111

Chapter 11 · Lambda 149

Chapter 12 · Local Definitions 155

Chapter 13 · Mutually Recursive Data Designs 161

Chapter 14 · Generative Recursion 170

Chapter 15 · Accumulator-Style Recursion 180

Chapter 16 · Mutation 188

Chapter 17 · Streams 217

Chapter 18 · Types and Data Modeling 230

Chapter 19 · Enums and Subranges as Invariants 249

Chapter 20 · Multi-Valued Logic **257**

Chapter 21 · Logic Programming **269**

Chapter 22 · The Meta-Circular Evaluator **286**

Volume II — Data Structures and Beyond **297**

Chapter 23 · Linked Lists from Concepts **298**

Chapter 24 · Stacks, Queues, and Deques **309**

Chapter 25 · Hash Tables and Dictionaries **318**

Chapter 26 · Balanced Trees **327**

Chapter 27 · Graphs as Reified Relationship Concepts **333**

Chapter 28 · The Graphs Landscape — Four Kinds, One Language **340**

Chapter 29 · Conceptual Spaces (Gärdenfors) **355**

Chapter 30 · Gödelization **364**

Chapter 31 · Lojban-in-Axioma **374**

Chapter 32 · Natural Semantic Metalanguage (Wierzbicka) **382**

Chapter 33 · Schank's Conceptual Dependency **392**

Chapter 34 · The MCP Bridge **403**

Chapter 35 · Big-O and Empirical Complexity **413**

Chapter 36 · The Sorting Showcase **426**

Chapter 37 · Graph Algorithms **440**

Chapter 38 · State Machines Revisited **453**

Chapter 39 · Event Loops & Reactors **461**

Chapter 40 · The Cascade Reactor **476**

Chapter 41 · Stack Programming **488**

Chapter 42 · Neuro-Symbolic Patterns **501**

Volume III — AxiomaCascade **512**

Chapter 43 · Cascade Architecture, in One Diagram 513

Chapter 44 · The Knowledge Graph Schema 524

Chapter 45 · The Ingest Pipeline 536

Chapter 46 · The Truth-Value Lifecycle 548

Chapter 47 · Reasoning Algorithms — NetworkX in Cascade 560

Chapter 48 · Axioma Rule Firing in Production 570

Chapter 49 · Writing Custom `.ax` Rules for Cascade 582

Chapter 50 · The Agent Swarm 592

Chapter 51 · Reactor Internals & Scheduled Tasks 602

Chapter 52 · Operating Cascade at Scale 613

Chapter 53 · SQL as a Pedagogical Surface 623

Chapter 54 · The Closed Hindley–Milner Island 630

Appendix A · Quick Syntax Reference 654

Appendix B · The Design Recipe — One-Page Summary 676

Appendix C — Solutions 680

Chapter 1 · Solutions 681

Chapter 2 · Solutions 686

Chapter 2a · Solutions 690

Chapter 3 · Solutions 693

Chapter 4 · Solutions 697

Chapter 5 · Solutions 701

Chapter 6 · Solutions 709

Chapter 7 · Solutions 713

Chapter 8 · Solutions 718

Chapter 9 · Solutions 722

Chapter 10 · Solutions **727**

Chapter 11 · Solutions **735**

Chapter 12 · Solutions **738**

Chapter 13 · Solutions **742**

Chapter 14 · Solutions **749**

Chapter 15 · Solutions **755**

Chapter 16 · Solutions **759**

Chapter 17 · Solutions **770**

Chapter 18 · Solutions **775**

Chapter 19 · Solutions **785**

Chapter 20 · Solutions **790**

Chapter 21 · Solutions **797**

Chapter 22 · Solutions **807**

Chapter 23 · Solutions **814**

Chapter 24 · Solutions **820**

Chapter 25 · Solutions **828**

Chapter 26 · Solutions **834**

Chapter 27 · Solutions **839**

Chapter 28 · Solutions **848**

Chapter 29 · Solutions **851**

Chapter 30 · Solutions **857**

Chapter 31 · Solutions **860**

Chapter 32 · Solutions **865**

Chapter 33 · Solutions **870**

Chapter 35 · Solutions **874**

Chapter 36 · Solutions **882**

Chapter 37 · Solutions **889**

Chapter 38 · Solutions **894**

Chapter 39 · Solutions **899**

Chapter 40 · Solutions **907**

Chapter 41 · Solutions **916**

Chapter 42 · Solutions **924**

Chapter 43 · Solutions **931**

Chapter 44 · Solutions **938**

Chapter 45 · Solutions **943**

Chapter 46 · Solutions **948**

Chapter 47 · Solutions **954**

Chapter 48 · Solutions **960**

Chapter 49 · Solutions **966**

Chapter 50 · Solutions **972**

Chapter 51 · Solutions **978**

Chapter 52 · Solutions **984**

Chapter 54 · Solutions **990**

Appendix D · Where to Read Next **997**

Appendix E · Introspection and Debugging **1001**

Appendix F · Cross-language translation **1024**

Appendix G — How the Axioma Interpreter Works **1029**

How to Design Knowledge Programs — in Axioma

© 2026 Vlad Evenhar.

Created and designed by Vlad Evenhar. Implemented by AI agents.

A unified textbook for the **Axioma** programming language — covering a learning path from `1 + 2` through data structures and knowledge representation. The title plays on Felleisen, Findler, Flatt, and Krishnamurthi's *How to Design Programs* (HtDP, MIT Press, 2018, 2nd ed.), whose design-recipe pedagogy the early chapters follow faithfully (all prose, examples, and exercises here are original) — but the book's scope grows past traditional HtDP to cover data structures, knowledge representation, and a separately maintained AxiomaCascade application route.

What this book is

A textbook in the **HtDP tradition**: design recipes drive everything; data definitions come before code; examples become tests; the structure of the data shapes the structure of the program. The book is organized into **Volumes**:

- **Volume I — Foundations** (Chapters 1–22, plus Chapter 2a and addendum Chapter 54). The CS1-level material — arithmetic, variables, loops, functions, the design recipe, conditionals, arrays, recursion, trees, higher-order functions, lambdas, mutual recursion, generative recursion, accumulators, mutation, streams, Concepts, enums and subranges, multi-valued logic, logic programming, and the meta-circular evaluator. Chapter 54 is the closed Hindley–Milner island (read after Chapter 19).
- **Volume II — Data Structures and Beyond** (Chapters 23+). The CS2-level material — linked lists, stacks, queues, hash tables, balanced trees, graphs. Followed (as the volume grows) by algorithm analysis, stateful systems, knowledge representation, and the neuro-symbolic frontier.
- **Volume III — AxiomaCascade** (draft; review deferred). Chapters covering the Cascade reasoning-engine layer that uses Axioma as one of its substrates.

It is **not** an Axioma reference manual — that already exists at ../../manual/Axioma Manual.md. This book is a learning path. The manual is the look-up index.

Revision coverage

This is a **reviewed working edition**, checked against the interpreter at revision `19a9398d6` (September 6, 2026). The language is still evolving. The current pass repairs older examples and covers the recent numeric, collection, control-flow, type-contract, metaprogramming and discovery changes. It also corrects mathematical explanations, documented outputs and prerequisites.

The automated suite inventories chapter and solution fences and exercise starters. It executes the declared standalone and contextual examples, compares documented output, and checks intentional failures. Exercise holes, foreign code and external integrations have explicit classifications. A successful test run does not mean every unfinished exercise was solved or every function's possible inputs were tested.

Deferred route: Cascade product integration, the MCP bridge and associated external-service examples require their own repository and environment review. Those chapters remain marked as drafts. Chapter 53 documents a current SQL NULL limitation instead of presenting its failing example as successful. See CURRENCY_AUDIT.md for scope, corrections and remaining boundaries. This is not yet a final print edition.

Audience

Two audiences, depending on entry point:

- **First-time programmers** (CS1 audience) — start at Chapter 1. High-school students who have used a calculator can begin here. No prior programming experience assumed.
- **Programmers exploring Axioma's KR features** — already comfortable with one language, want to learn Axioma's signature material. Skim Volume I for orientation, dive into Volume II.

Reading formats

The book ships in three formats, all generated from the same markdown sources:

- Markdown — every `.md` file in this directory is readable on its own.
- HTML — `HtDKP-in-Axioma.html`, single-file with table of contents.
- PDF — `HtDKP-in-Axioma.pdf`, growing with each chapter.

After editing any markdown source, regenerate the HTML and PDF with:

```
./build.sh
```

Requires `pandoc`, LuaLaTeX, STIX Two Text/Math, and the fallback fonts named in `fonts-fallback.tex`. The build uses LuaLaTeX so Unicode symbols in code remain readable; see `build.sh` for the platform font requirements.

How to read the book

Use the main sequence when learning programming from the beginning. The optional extensions can wait until the canonical form feels familiar. Chapters sometimes reuse earlier helpers; load the named prelude rather than treating every code fence as an independent file. Independent alternatives and language-mode changes are identified in the text.

A first programming course. Start with Chapters 1–9, then abstraction in 10–15. Continue with state and data modeling in 16–21. Return to Chapter 22's interpreter and macro material after writing several small programs.

Axioma as an additional language. Start with the numeric distinctions in 1–2, calls and receivers in 3, collections and matrices in 6, and higher-order functions/broadcasting in 10. Then study binding and mutation in 16, Concepts and schemas in 18–19, and logic in 20–21. Appendix E is the working companion for inspecting values and diagnosing errors. Familiarity with another language does not settle Axioma's indexing, exact-number or scope rules.

Optional application routes. For data structures, read 23–27 alongside algorithm analysis in 35–37. For stateful systems, read 38–39 and the stack chapter 41. Knowledge representation in 28–33 and SQL in 53 build on the logic chapters. Read the Hindley–Milner addendum 54 as a separate language mode. The Cascade route can be skipped entirely without losing the core course.

Inside the interpreter. Appendix G follows one small program through the current Go implementation, with a source map and a guided experiment. Read it after Chapters 3 and 12, or alongside Chapter 22's teaching evaluator.

Each chapter follows the same shape:

1. **Concept** — the new idea, motivated by something you already know.
2. **Worked example** — a small problem solved step-by-step.
3. **Design recipe** stages where they apply (data → signature → examples → template → body → tests).
4. **Exercises** — graduated difficulty; the last one is always open.
5. **Reflection** — why we did it this way, what comes next.

You'll need a running Axioma installation. Throughout the book we assume you're using the **beginner subset**:

```
axioma --learn                          # interactive single-shot tutor
axioma --recipe                         # six-stage Design Recipe wizard
axioma --language axioma/beginner ex.ax # restricts to canonical forms
axioma --typecheck ex.ax                # static type check before run
```

If you write `#language axioma/beginner` at the top of a file, the beginner restrictions apply automatically for that file. Most exercises in this book are written that way. Other real dialects show up later, each as a *language*, not a flag on the host:

- Chapter 21 — `#language axioma/knowledge-core` (monotonic Horn proof-core)
- Chapter 41 — `#language axioma/rpn` (additive Forth return stack; infix still runs)
- Chapter 54 — `#language axioma/hm` (closed Hindley–Milner island; `println` is out, `expect` is in, untypable programs exit 1)

`axioma/core` and `axioma/functional` are known names with no AST gate — they run as host. This book does not teach them. AppA lists the full roster. Bare `#language axioma` is not a name.

Keeping examples current

From the repository root, `make test-textbook` builds the current interpreter and runs the source-linked checks. The manifest in `verification/examples.json` records each code block's role and the case that exercises runnable material. Editing code requires reviewing its expected behavior, not merely replacing a

hash. Contributor instructions require a textbook impact decision alongside the Manual, quick reference, website and editor for language changes.

`make textbook` creates HTML, PDF and a build receipt. The commit check compares those hashes with the staged sources; CI repeats the example checks. This keeps missing updates visible while leaving explanatory quality to editorial review.

Table of contents

The generated table of contents links directly to the chapters. The repository also keeps an editorial outline; its draft-status tracker is not repeated in the reader edition. A quick map:

Volume I — Foundations

Part	Chapters	Focus
I. Fixed-Size Data	1–5	Arithmetic, variables, functions, conditionals, booleans
II. Arbitrary-Size Data	6–9	Lists, recursion, trees, mutual recursion
III. Abstraction	10–12	Higher-order functions, lambdas, locals
IV. Intertwined Data	13–14	Mutually recursive data, generative recursion
V. Accumulators & State	15–17	Mutation, stateful objects, streams
VI. Axioma additions	18–22, **54**	Concepts, enums + subranges, multi-valued logic, logic programming, the meta-circular evaluator; Ch.54 is a later addendum — the closed Hindley–Milner island

Volume II — Data Structures and Beyond

Part	Chapters	Focus
VII. Data Structures	23–27	Linked lists, stacks/queues/deques, hash tables, balanced trees, graphs

Part	Chapters	Focus
X. Knowledge Representation (pilots drafted)	28–33	Graphs landscape (Sowa/Peirce/Context); Conceptual Spaces (Gärdenfors); Gödelization; Lojban-in-Axioma; NSM (Wierzbicka); Schank's CD
VIII. Algorithm Analysis (planned, ch. nums TBD)	—	Big-O, sorting showcase, graph algorithms
IX. Stateful Systems (planned, ch. nums TBD)	—	State machines, event loops, the Cascade reactor
XI. Symbolic + Neural (planned, ch. nums TBD)	—	Stack programming, MCP bridge, neuro-symbolic patterns

Volume III — AxiomaCascade

Part	Chapters	Focus
XII. Cascade Foundations	34	Architecture map; the 4 layers; where Axioma sits; CLI walkthrough
XIII. Cascade Knowledge Graph (planned)	43–45	SQLite schema, ingest pipeline, truth-value lifecycle
XIV. Cascade Reasoning (planned)	46–48	NetworkX + Axioma; writing custom `.ax` rules
XV. Operating Cascade (planned)	49–52	Agent swarm, scheduled tasks, scaling

Status

Volume I complete (Ch. 1–22) + Volume II drafted (Ch. 23–42) + Volume III complete (Ch. 43–52) + Ch. 53 (the SQL surface) + Ch. 54 (the closed Hindley–Milner island, a Volume I addendum). Chapters 1–27 and 35–52 have full prose, exercises, runnable scaffolds, and per-chapter solutions. Chapters 28–33 are pilot drafts (concept overview + worked example + 2–3 inline exercises); each can be promoted to a full chapter in a later session. Appendices A, B, C (auto-assembled from the per-chapter solutions files), D, E, and F are complete.

- Part I (1–5): Arithmetic; Variables and Naming; Functions; The Design Recipe; Conditionals and Booleans.
- Part II (6–9): Arrays as Lists; Recursion on Lists; Trees via Concepts; Mutual Recursion.
- Part III (10–12): Higher-Order Functions; Lambda; Local Definitions.
- Part IV (13–14): Mutually Recursive Data Designs; Generative Recursion.
- Part V (15–17): Accumulator-Style Recursion; Mutation; Streams.
- Part VI (18–22): Types and Data Modeling; Enums and Subranges as Invariants; Multi-Valued Logic; Logic Programming; The Meta-Circular Evaluator. Ch. 54 returns to types with `#language`

`axioma/hm` (closed Hindley–Milner island).

- Part VII (23–27): Linked Lists from Concepts; Stacks, Queues, Deques; Hash Tables and Dictionaries; Balanced Trees; Graphs as Reified Relationship Concepts.
- Part X pilots (28–33): The Graphs Landscape; Conceptual Spaces (Gärdenfors); Gödelization; Lojban-in-Axioma; NSM (Wierzbicka); Schank's Conceptual Dependency. Pilot-depth introductions to Axioma's signature knowledge-representation formalisms.
- Ch. 34: The MCP Bridge — Axioma tools inside Cascade's server.
- Parts VIII–XI (35–42): Big-O and Empirical Complexity; the Sorting Showcase; Graph Algorithms; State Machines; Event Loops; the Cascade Reactor; Stack Programming; Neuro-Symbolic Patterns.
- Volume III (43–52): Cascade Architecture; the KG Schema; the Ingest Pipeline; the Truth-Value Lifecycle; Reasoning Algorithms; Axioma Rule Firing; Writing Custom `.ax` Rules; the Agent Swarm; Reactor Internals; Operating Cascade at Scale. Ch. 53 adds the SQL surface. Ch. 54 is the closed Hindley–Milner island (read after Ch. 19).
- Appendices: A (Quick Syntax Reference), B (Design Recipe One-Pager), C (Solutions, auto-assembled), D (Where to Read Next), E (Introspection and Debugging), F (Cross-Language Translation).

Relationship to HtDP

This is an **original textbook inspired by HtDP**, not a translation or a chapter-by-chapter adaptation. It adopts the design recipe and data-driven progression, then develops its own Axioma examples, exercises, and knowledge-programming material.

HtDP itself is freely available online at https://htdp.org/ and is the canonical reference for the underlying pedagogy. Anyone reading this book seriously should also read the original.

License

The textbook prose is © 2026 Vlad Evenhar, available under the same license as the rest of the Axioma project. HtDP itself is © Felleisen et al.; we neither copy nor displace it.

Chapter 1 · Arithmetic

What this chapter is. Your first encounter with Axioma. By the end you can use it as a calculator, you know what kinds of numbers it handles, and you've seen your first *type error* — Axioma telling you that something doesn't fit.

A note on faithfulness to HtDP. The original *How to Design Programs* introduces several things in its first chapter that we'll defer here: an image library (circles, squares, overlays), strings, and the **design recipe** itself. We delay those to keep the first three chapters lower-friction. The full design recipe lands in Chapter 4 — early enough to drive the rest of the book.

1.1 The REPL as a calculator

A *calculator* lets you type `2 + 3` and shows you `5`. Axioma can do the same — and a lot more — through what's called the **REPL** (read – evaluate – print – loop). Open a terminal and start it:

```
$ axioma
Axioma programming language and environment v0.9
Calculemus! (Let us calculate) - Leibniz.
Type ':help' for commands, ':exit' to quit
axioma>
```

The `axioma>` prompt is asking what you'd like to compute. Try typing `2 + 3` and pressing Enter:

```
axioma> 2 + 3
5
axioma>
```

That's the whole loop: it **read** your line, **evaluated** the expression `2 + 3`, **printed** the result `5`, and **looped** back to the prompt for another. We're going to live in this REPL for the rest of the chapter.

A subtle thing about the prompt

Axioma's REPL can detect when an expression looks *incomplete* — for example, if you type a single line that has an open parenthesis but never closes it. When that happens, the prompt changes from `axioma>` to `...` to tell you "I'm waiting for more":

```
axioma> (2 + 3
...     ) * 4
20
axioma>
```

You can submit a half-finished line by hitting Enter on an empty `...` line. If you ever see `...` and don't remember why, that's the escape hatch: a blank line on the `...` prompt either submits what you have (if it's now complete) or returns to `axioma>` empty-handed.

For every REPL example in this book, we'll show the prompt that's asking, so you can match what's on your screen.

> **Beginner mode.** Throughout this book we'll usually invoke Axioma in the *beginner subset* — a restricted version that allows exactly one way of doing each thing, so you don't get distracted by alternatives:
>
> ```
> $ axioma --language axioma/beginner
> ```
>
> Or, equivalently, you can put `#language axioma/beginner` at the top of any script file. The REPL prompt looks the same, but Axioma will refuse some of its more advanced features. That's intentional — we'll unlock them chapter by chapter.

Two ways to see a value: `println` and the trailing dot

The REPL prints every result automatically. *Script files* don't — type `2 + 3` in a script and run it, and you'll see nothing on screen. Axioma gives you two ways to make a value visible:

```
println(2 + 3)        # the function call: prints 5
2 + 3 .               # the trailing-dot sigil: also prints 5
```

The trailing `.` (read it as "show me this") is Axioma's *inspect-print* operator. It works exactly the way the REPL does — evaluate the expression, then print the result — but inside a script file. Compare:

```
# Verbose script style
x: 5
y: 10
println(x)
println(y)
println(x + y)

# Same thing, terser
x: 5 .
y: 10 .
x + y .
```

Both print `5`, `10`, `15`. The dot version reads almost like a Prolog or Smalltalk dialect — every "sentence" ends with a period, and the period asks Axioma to show you what just happened.

The dot works at the top level of a script, and also **inside the bodies of loops, conditionals, and functions** — so it's particularly handy when you want to peek at intermediate values without cluttering your code with `println` calls:

```
total: 0
i: 1
while (i <= 5) [
  i .                          # show me i each iteration
  total = total + i
  i = i + 1
]
total .                        # show me the final total
```

That prints `1 2 3 4 5 15` — the loop variable for each iteration, then the final running total.

For most of this book we'll use `println(...)` because it's explicit and works the same way in every language you've used; the trailing dot is here so you can recognize it when you see it, and use it yourself once you find it more comfortable.

In the REPL transcripts in this book, binding acknowledgements and the `none` returned by output commands are omitted to keep attention on the value being discussed. Strings displayed as values keep their quotes; `println` and `printf` print the text itself. Error excerpts retain the meaning but may omit the source frame and hint.

Optional: save a program, then choose project settings

The REPL is enough for this chapter. When you want to keep a calculation, save it as an ordinary `.ax` file in your own directory. These examples assume the `axioma` executable is on your shell's `PATH`. Use `./axioma` when the executable is in the current directory.

For example, a file named `hello.ax` can contain:

```
println(2 + 3)
```

From that directory, use these terminal commands. Options go before the filename:

```
axioma hello.ax
axioma --typecheck hello.ax
axioma --typecheck --run hello.ax
```

The first command prints `5`. The second checks the file and reports `type check clean`, without executing `println`. The third checks first and prints `5` if there are no type errors. `axioma` by itself starts the REPL. No configuration file is needed for any of these commands.

As a project grows, you may want settings shared by its source files and editor. An optional `axioma.json` file provides them. For the small example above, you could deliberately add this configuration beside `hello.ax`:

```
{
  "version": 1,
  "entryPoints": ["hello.ax"]
}
```

`version` identifies the configuration format. `entryPoints` identifies the program to start when you select the project without naming a source file:

```
axioma --project .
axioma --project ./axioma.json --typecheck
axioma --project . --typecheck --run
```

The dot means the current directory. The first command reads `./axioma.json` and runs its sole entry, printing 5. The second checks that entry; the third checks and then runs. The flag accepts the directory or the configuration file's path. It requires an existing configuration and creates neither a project nor a source file.

An entry point is an ordinary `.ax` file. There is no required filename: `main.ax` is only a convention some programmers choose.

With zero or multiple entries, use `axioma --project . hello.ax` to select a source explicitly. The source must belong to the selected project; it need not be listed as an entry point.

Once a configuration exists, even `axioma hello.ax` finds the nearest one by searching from the source file's directory upward. That file supplies defaults for that source; the editor uses the same discovery. Nested configurations do not merge with their parents. With no configuration, normal defaults apply. The ordinary REPL does not discover these project settings.

Other optional settings choose the default language, checking options and disk import directories. Paths in the configuration are relative to that file. Explicit CLI checking flags override its defaults. A source `#language` directive keeps precedence over language defaults. For example, the beginner directive introduced earlier still selects the beginner subset.

A stricter checker setting does not by itself prevent an ordinary host program from running: its static findings remain advisory. Use `--typecheck --run` when you want type errors to prevent execution. Runtime errors remain possible; a clean static check is not a promise that every operation will succeed.

Project configuration is optional setup, not a prerequisite for learning Axioma. Adding one to an existing repository can change defaults for files below it, so make that choice deliberately. See the Manual's **Getting Started / Project configuration reference** for the complete fields and error cases. In the repository, the same scope is summarized in `docs/js-ts-adoption.md`.

1.2 Numbers come in two flavors

You can type whole numbers and Axioma understands them as **integers**:

```
axioma> 42
42

axioma> -7
-7

axioma> 1000000
1000000
```

Or numbers with a decimal point — those are **floats** (short for "floating-point"):

```
axioma> 3.14159
3.14159
axioma> -0.5
-0.5
```

Floats also accept **scientific notation**: write `1.5e6` to mean "1.5 times ten to the sixth power" (1,500,000), or `3.14e-2` to mean "3.14 times ten to the negative second" (0.0314):

```
axioma> 1.5e6
1.5e6
axioma> 1e9
1.0e9
axioma> 3.14e-2
0.0314
```

Notice that `3.14e-2` prints back as `0.0314` — scientific notation is just a way of *writing* a number, not a separate kind of number. Axioma reads `3.14e-2` and `0.0314` as the very same float.

(For *very* small or *very* large magnitudes Axioma keeps the scientific form when printing back — `3.4e-10` is easier to read than a string of ten zeros. The switch happens around one ten-thousandth: `0.0314` stays decimal, but `0.00000000034` prints as `3.4e-10`.)

You'll see floats whenever you write a number with a decimal point or an exponent — and whenever one enters a computation, because floats are contagious. (An all-integer division that comes out uneven gives you an exact *fraction* instead — `7 / 2` is the rational `7/2`, not a float; §1.5 tells that story.)

Controlling how a value prints — `printf` and `stringf`

`println` shows you a value the way Axioma chooses to. Most of the time that is exactly what you want. But sometimes you want to decide the layout yourself — for instance, to see `0.00000000034` written out in full instead of as `3.4e-10`. For that, Axioma gives you a *format string*.

`printf` ("print formatted") takes a format string followed by the values to drop into it. The format string contains **slots** — a `%` followed by a letter — and each slot is filled, in order, by one of the values:

```
axioma> printf("%d items cost $%.2f\n", 3, 7.5)
3 items cost $7.50
```

The common slots:

Slot	Fills with	Example	Prints
`%d`	an integer	`printf("%d", 42)`	`42`
`%f`	a float	`printf("%f", 1.5)`	`1.500000`
`%.2f`	a float, 2 decimal places	`printf("%.2f", 1.5)`	`1.50`
`%s`	a string	`printf("%s", "hi")`	`hi`
`%t`	a boolean	`printf("%t", true)`	`true`
`%%`	a literal percent sign	`printf("100%%")`	`100%`

Two things to remember. First, `printf` does **not** add a newline of its own — write `\n` in the format string wherever you want a line to end. Second, `%.Nf` lets you ask for exactly `N` digits after the decimal point. That is the answer to the puzzle from the last section:

```
axioma> printf("%.11f\n", 3.4e-10)
0.00000000034
```

The value never changed — `3.4e-10` and `0.00000000034` are the same float. `printf` just let us choose the long spelling. You can also pad a number to a fixed width, which is handy for lining up columns: `%05d` prints an integer in 5 columns, padded with zeros (`00042`).

`stringf` ("string formatted") takes the *same* format strings and the *same* slots, but instead of printing, it **hands the finished text back to you** as a value:

```
label: stringf("item-%d", 7)     # label is now the string "item-7"
println(label + "!")                  # prints: item-7!
```

Use `printf` when you just want to show something on screen; use `stringf` when you want to *keep* the formatted text — to store it in a variable, join it with another string, or return it from a function. They are the same tool, differing only in where the result goes.

If you come from Lua, Ruby, Java, or C#, the name you already know for this is `format` — and Axioma accepts it as an exact alias of `stringf`: `format("item-%d", 7)` also *returns* `"item-7"` (it never prints, just like `string.format` in Lua). Printing stays `printf`.

1.3 The four operators — plus two more

Axioma has the same operators as your calculator, with the same symbols:

Symbol	Meaning	Example
`+`	addition	`2 + 3 = 5`
`-`	subtraction	`10 - 4 = 6`
`*`	multiplication	`7 * 6 = 42`
`/`	division	`12 / 3 = 4`

Plus two that calculators don't always have:

Symbol	Meaning	Example
`%`	**modulo** — the remainder after division	`7 % 2 = 1`
`^`	**exponent** — raise to a power	`2 ^ 10 = 1024`

Try each of these in the REPL:

```
axioma> 7 % 2
1

axioma> 7 % 3
1

axioma> 100 % 7
```

```
2

axioma> 2 ^ 10
1024

axioma> 10 ^ 6
1000000
```

Modulo is how you check whether a number is even (`n % 2 == 0`) or how you wrap an hour around the clock (`hour % 24`). We'll use it a lot.

1.4 Order of operations

Mathematics gives multiplication and division priority over addition and subtraction. Axioma agrees:

```
axioma> 2 + 3 * 4
14

axioma> (2 + 3) * 4
20
```

In the first line, `3 * 4` happens first, giving `2 + 12 = 14`. In the second, the parentheses force the addition first, giving `5 * 4 = 20`.

The arithmetic subset used here (high → low):

1. Function calls and parentheses
2. `^` (exponent)
3. unary – (negation)
4. `*`, `/`, `%`
5. `+`, `-`

When in doubt, parenthesize. The broader table, including calls, access, logic, collections and special forms, is in Appendix A: Operator precedence and associativity.

> **Associativity.** Like most math textbooks, Axioma treats the exponent as *right*-associative: `2 ^3^2` means `2^(3^2) = 512`, not `(2^3)^2 = 64`. Every other arithmetic operator (`+`, `-`, `*`, `/`) is left-associative, as usual. If you want the left grouping, write `(2 ^ 3) ^ 2` explicitly. This appears in Exercise 1.3. Syntactic right-associativity chooses the default grouping; it does not say exponentiation is mathematically associative. Its two groupings above give different answers.
>
> **Negation vs. exponent.** Unary – binds *looser* than `^`, so `-2^2` means `-(2^2) = -4`, not `(-2)^2 = 4` — again the math-textbook reading. To square a negative base, parenthesize: `(-2)^2`.

1.5 Division has a surprise

Here's the first thing that might surprise you:

```
axioma> 7 / 2
```

```
7/2
```

`7 / 2` should be `3.5`, right? Axioma is telling us `7/2`. What happened?

When **both** operands are integers, Axioma keeps division **exact**: the answer is a *rational number* — a true fraction, not a rounded decimal. When the division comes out even, you get a plain integer back; when it doesn't, you get the exact fraction:

```
axioma> 6 / 2
3

axioma> 7 / 2
7/2

axioma> type(7 / 2)
Rational
```

Addition, subtraction, multiplication, and division of rational numbers remain exact when all operands are exact:

```
axioma> 1/3 + 1/6
1/2

axioma> 7/2 + 1/2
4

axioma> 7/2 == 3.5
true
```

This is the same *design choice* HtDP's teaching languages make: Racket's `(/ 7 2)` is the exact rational `7/2` too. Exact answers stay exact until *you* decide to leave the world of fractions. When you want the decimal, bring a float into the expression — or convert at the end:

```
axioma> 7.0 / 2
3.5

axioma> 7 / 2.0
3.5

axioma> float(7 / 2)
3.5
```

For these arithmetic operations, mixing a Float with an Integer or Rational chooses approximate floating-point arithmetic. Other operations have their own result rules: for example, `sqrt(2)` produces a Float even though its input is an Integer.

Asking for a Float explicitly: `fdiv`

Sometimes the required output type is part of the task: you need a floating-point result even when both inputs are exact integers. Use `fdiv`, either as a function or between its operands:

```
println(fdiv(1, 8))           # 0.125
println(1 fdiv 8)             # 0.125
println(type(fdiv(1, 8)))     # Float
println(1 / 8)               # 1/8
```

The ordinary slash keeps an exact integer or rational quotient when its operands are exact. `rdiv` is the word spelling of that slash: `1 rdiv 8` is `1/8`, the same AST as `1 / 8`. `fdiv` explicitly chooses a Float. `id iv` is the word spelling of floor `div` (same AST as `7 div 2` and `7 ÷ 2`). A zero divisor raises an error in either form. The interpreter supports `fdiv`; VM support is deferred and unsupported uses refuse. `rdiv` and `idiv` compile in the VM because they are `/` and `div`.

Asking for the floor quotient: `div` and ÷

To find how many whole groups fit, use `div`. Its glyph spelling is ÷:

```
println(7 div 2)              # 3
println(7 ÷ 2)                # 3
println(7.0 div 2.0)          # 3.0
println(-7 div 3)             # -3
println(-7.0 div 3.0)         # -3.0
```

Both spellings round the quotient **down**, toward negative infinity. For example, negative seven divided by three is between -3 and -2; its floor is -3. Integer inputs give an Integer; Float inputs give a whole-valued Float, which is why the third result prints as `3.0`.

`//` starts a **line comment** in Axioma. It is not an operator: `7 // 2` evaluates the expression `7` and ignores the rest of that line. Use `div` or ÷ when you mean floor division.

The remainder operator has aliases too: `%` and `mod`

You met `%` (modulo) in §1.3. It also has a keyword spelling, `mod`:

```
axioma> 100 % 7
2
axioma> 100 mod 7
2
axioma> 100 modulo 7
2
```

All three give 2. `modulo` is just a longer alias for `mod` — it exists because some people find the longhand more readable.

The pair `div` (or ÷) and `%` (or `mod`) are related by a useful identity: for any integers `a` and nonzero `b`,

```
(a div b) * b + (a % b) == a
```

That is, *quotient times divisor plus remainder equals dividend* — the same fact you learned in grade-school long division. Try it:

```
axioma> (100 div 7) * 7 + (100 % 7)
100
```

Two halves in one shot: `divmod`, `quotient`, `remainder`

Sometimes you want both the quotient and the remainder. You could write `100 div 7` and `100 % 7` separately, but that does the division twice. Axioma gives you a combined builtin:

```
axioma> divmod(100, 7)
(14, 2)
```

`divmod` returns a **tuple** — a pair of values bundled together. You can pull them apart with index access (`[1]` is the first item, `[2]` is the second):

```
axioma> r: divmod(100, 7)
axioma> r[1]
14
axioma> r[2]
2
```

If you only want one half, the prefix forms `quotient` and `remainder` exist and read as English:

```
axioma> quotient(100, 7)
14
axioma> remainder(100, 7)
2
```

These are exact synonyms for `100 div 7` and `100 % 7` respectively — pick the form that reads best in the sentence you're writing.

Extension: exact inputs and complex values

An exact input does not make every operation exact. Rational arithmetic such as `1/3 + 1/6` keeps a fraction; a transcendental function such as `sin` returns an approximation. Complex numbers extend the numeric domain: `im` names the imaginary unit, and multiplication remains explicit.

```
println(1/3 + 1/6)        # 1/2
println(sin(1/2))         # 0.479425538604203
println((1 + im)^2)       # 2.0 * im
println((1 + im)^2 is Complex)  # true
```

The printed `2.0 * im` is valid Axioma syntax. It records the Float component used by the complex calculation; it is not a new kind of imaginary unit. These examples distinguish three questions: what value was computed, what type represents it, and whether the result is exact. For a first reading, ordinary integers and fractions are enough; return here when a problem actually needs complex arithmetic.

1.6 Your first type error

Try this in the REPL:

```
axioma> "hello" / 2
ERROR at <stdin>:1:9: unknown operator: STRING / INTEGER
```

Axioma stopped you. The reason: division isn't defined between a string (the quoted text `"hello"`) and an integer (2). Axioma calls this an *error* because there's no sensible answer.

The error identifies an operation that is not defined for these inputs. Check the values and the operation before changing the program.

> **A subtle warning.** Axioma is permissive about *some* mixed-type arithmetic. `"hello" + 1` produces `"hello1"` (it converts the integer to a string and concatenates). `2 * "hi"` produces `"hihi"` (repetition, Python-style). These are *not* errors — they're features. Division is the one that has no string interpretation, so it's the one that errors. We'll see more of this in Chapter 2.

1.7 A worked example: area of a circle

You probably know the formula: a circle's area is π times the radius squared. Let's compute the area of a circle with radius 5.

Step 1: write down what you know. Radius = 5. $\pi \approx 3.14159$.

Step 2: translate to Axioma.

```
axioma> 3.14159 * 5 ^ 2
78.53975
```

That's it. Axioma evaluated `5 ^ 2` first (= 25), then multiplied by `3.14159`, giving `78.53975`. About 78.5 square units.

A small refinement. If you want a more precise answer, use more digits of π:

```
axioma> 3.141592653589793 * 5 ^ 2
78.53981633974483
```

> **A peek ahead.** Typing π's digits every time is tedious. In Chapter 2 you'll learn to give names to values (`pi: 3.14159`) so you can write `pi * 5 ^ 2` instead. In Chapter 3 you'll learn to package the whole computation into a function (`circle_area(5)`).

1.8 Exercises

Try each of these. After each, predict what Axioma will say *before* running it; then check. The predict-then-check habit is the most important habit you'll build this term.

Exercise 1.1

Compute the circumference of a circle with radius 7 (the formula is $2 \times \pi \times r$). Use 3.14159 for π.

Exercise 1.2

What does Axioma give you for `2 ^ 16`? What about `2 ^ 32`? What about `2 ^ 64`? Try them and see when (if ever) Axioma stops giving you exact answers.

Exercise 1.3

Predict each answer **before** typing it:

- `(2 + 3) * (4 - 1)`
- `10 - 2 * 3`
- `2 ^ 3 ^ 2` *(this one is sneaky)*
- `(2 ^ 3) ^ 2`
- `100 / 7`
- `100 % 7`
- `100.0 / 7`
- `100 div 7`
- `100 fdiv 7`
- `100 mod 7`
- `100.0 div 7.0`
- `-7.0 div 3.0` *(this one is also sneaky — recall the floor rule)*

Exercise 1.4

Modulo trick. For each of these, predict whether the answer is `0`, and check (`%` and `mod` are interchangeable — try both spellings):

- `12 % 3`
- `12 mod 4`
- `12 % 5`
- `2 ^ 10 mod 7`

What do all the *zero* results have in common with 12?

Exercise 1.5

Type-error exploration. Without looking at section 1.6, try to predict which of these will error and which won't:

- `"5" + 3`
- `5 + "3"`
- `5 - "3"`
- `5 * "3"`
- `5 / "3"`
- `"hi" * 4`
- `"hi" / 4`
- `"hi" - "bye"`

What rule (if any) can you guess about when Axioma errors and when it just gives you something?

Exercise 1.6 — open

Find an expression that uses **all four** arithmetic operators (`+`, `-`, `*`, `/`) AND modulo (`%`) AND an exponent (`^`), evaluates without error, and produces an answer between `0` and `100`. There are many possible answers — find one of yours.

(After you find one, find a second one that's structurally different from the first.)

Exercise 1.7 — grouping is not an algebraic law

Write explicit parentheses showing the default grouping, then predict the result: `20 - 6 - 3`, `24 / 4 / 2`, `2 ** 3 ** 2`, `-2 ^ 2`, and `(-2) ^ 2`. For the first two expressions, also compute the opposite grouping. Explain why "power parses to the right" does not mean that exponentiation is associative. Does a precedence table by itself tell you when a function prints a message?

Exercise 1.8 — optional return to this after Chapters 3–5 and 10

Use Appendix A to parenthesize `3 `add` 4 * 2` when `add(a, b) = a + b`, and compare it with `add(3, 4 * 2)`. For `double(x) = 2 * x` and `inc(x) = x + 1`, compare `(inc ∘ double)(3)` and `3 |> double |> inc`. Explain why `inc ∘ double(3)` is not the same syntax. Finally, explain why `false and (1 / 0)` does not divide by zero. Which fact is about precedence, and which is about evaluation? These are full-language forms; they are not prerequisites for the next beginner chapter.

Exercise 1.9 - optional files and project settings

Save the following calculation as `tally.ax` in your own exercise directory:

```
println(6 * 7)
```

Choose commands to run it without configuration, check it without running, and check it before running. Then describe an optional `axioma.json` that would make `axioma --project .` select `tally.ax`. What changes if you list two entry points? Explain whether you need to rename the file to `main.ax`, whether installation requires a configuration, and whether a stricter checker setting alone prevents an ordinary host run from executing.

1.9 Reflection

Why did we start with arithmetic? Two reasons.

One, every computer program — every spreadsheet formula, every video-game scoreboard, every weather model — is at its core *calculation on data*. Arithmetic is the smallest possible example of that. If you understand it, you understand the seed from which the rest grows.

Two, arithmetic lets us meet the **REPL** — the read-evaluate-print loop — without anything else getting in the way. The REPL is where you'll do most of your exploration in this book. Every chapter starts in the REPL; only after you've explored an idea interactively do we move to writing files.

In the next chapter we'll learn to **name** values you've computed, so a long expression becomes a short name and your work compounds.

Looking ahead — what you'll be able to do by Chapter 5:

- Define your own functions (`func`)
- Make decisions with `if … then … else …`

- Use the **Design Recipe** — the six-step process that turns "I need a program for X" into actual working code

By Chapter 9 you'll have built a small interpreter for arithmetic expressions, written in Axioma. The arithmetic you typed in Chapter 1 will literally be the data your Chapter 9 program processes.

What we used from Axioma

A one-line index for future lookup (the full reference is in the manual):

- The REPL, started by `axioma` or `axioma --language axioma/beginner`
- Integer literals: `42`, `-7`, `1000000`
- Float literals: `3.14159`, `1.5e6`
- Operators: `+`, `-`, `*`, `/` (or `rdiv`), `%`, `^`, `div` (or `÷`, `idiv`), `fdiv`, `mod` (or `modulo`, both alias `%`)
- Prefix builtins for integer division: `quotient(a, b)`, `remainder(a, b)`, `divmod(a, b) → (q, r)`
- Grouping with `( )`
- Formatted output: `printf(fmt, ...)` and `stringf(fmt, ...)` (alias `format`) with `%d` / `%f` / `%.Nf` / `%s` / `%t` / `%%` slots
- The first error format: `ERROR at <stdin>:line:col: …`

Solutions to selected exercises

Solutions to selected exercises: Chapter 1 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch01_solutions.md.)*

Chapter 2 · Variables and Naming

What this chapter is. Naming values. Once you can give a number (or a string, or any value) a name, you stop retyping it and start *thinking* with it.

2.1 Why name a value?

Chapter 1 ended with the area of a circle:

```
axioma> 3.141592653589793 * 5 ^ 2
78.53981633974483
```

That works, but two things bother us. First, the digits of π are tedious to retype. Second, the `5` is the *radius* — but nothing in the expression says that. A reader has to guess.

Both problems disappear when we **name** things:

```
axioma> pi: 3.141592653589793
axioma> radius: 5
axioma> pi * radius ^ 2
78.53981633974483
```

The expression now reads aloud the way we'd say it: *pi times radius squared*. Names are how programs become explanations.

2.2 Declaring with `:`

The declaration form is:

```
<name>: <expression>
```

Three things happen when Axioma sees this:

1. It evaluates the expression on the right.
2. It binds the resulting value to the name on the left.
3. From then on, that name *is* that value, until you rebind it.

```
axioma> x: 5
axioma> x
5
axioma> x + 3
8
axioma> y: x * 2
axioma> y
10
```

You can use names inside other declarations — that's how `y: x * 2` works. Axioma had already bound `x` to `5`, so the right-hand side reduced to `5 * 2 = 10` before binding `y`.

This `name: value` shape — read aloud as "name *gets* value" — is the preferred Axioma idiom. It's short, it scans well, and it puts the name first (which is what your eye looks for). You'll see it everywhere in this book.

Naming rules

- Names start with a letter or underscore.
- After the first character they can have letters, digits, or underscores.
- Names are case-sensitive: `radius`, `Radius`, and `RADIUS` are three different variables.
- By convention in this book (and in HtDP), **lowercase** is for values and **TitleCase** for Concepts. Since June 2026 this is a *convention*, not a hard rule: both `:` and = accept a name of any case — the right-hand side decides (a `concept { ... }` RHS makes a Concept; any other value makes a variable, so `B = {2,4,6}` is a perfectly good set). We keep the lowercase-for-values style throughout the book for readability.

These names are fine: `pi`, `radius`, `total_so_far`, `n`, `is_done`. These will error or surprise you: `2pi` (starts with a digit), `total-so-far` (the – is the subtraction operator).

2.3 The REPL vs. a script file

So far we've been typing one line at a time at the REPL. That's great for exploring, but a real program is more than one line. You'll want to put your code in a *file* and run it from your shell.

Create a file called `hello.ax`:

```
#language axioma/beginner
pi: 3.141592653589793
radius: 5
area: pi * radius ^ 2
println("Area for radius", radius, "=", area)
```

Run it:

```
$ axioma hello.ax
Area for radius 5 = 78.53981633974483
```

Three new things appeared:

1. **`#language axioma/beginner`** at the top of the file. This pins the file to the beginner subset. Like `--language axioma/beginner` from Chapter 1, but written *inside* the file so anyone running it gets the same restrictions.
2. **`println`** is a built-in that prints its arguments separated by spaces, then ends the line. Spaces inside a string remain part of that string. We use it because a script doesn't auto-print results the way the REPL does.
3. Strings — text in double quotes — get treated like any other value. You can pass them, name them,

and combine them. Most characters can appear directly (the source file is UTF-8), so `"hello"`, `"naïve"`, and `"∀ x. P(x)"` all just work. For the few characters that resist direct typing — line breaks, tabs, or rare codepoints — Axioma uses the universal escape forms `"\n"` (newline), `"\t"` (tab), `"\\"` (literal backslash), and `"\u{2203}"` (the Unicode codepoint U+2203, which is ∃). The full table lives in Appendix A. We won't need them in this chapter.

Everything else is what you already know.

2.4 A subtle point: reassignment

What happens if you bind the same name twice?

```
axioma> x: 5
axioma> x: 10
axioma> x
10
```

Axioma lets it happen — silently. `x: value` means "bind `x` to this value": it creates the name if it's new and updates it if it already exists. There is no separate declaration step, and no warning. The math-style spelling does exactly the same thing:

```
axioma> x: 5
axioma> x = 10
axioma> x
10
```

`x = 10` is a **synonym** of `x: 10` — the bind-or-update behavior is identical. It's there so a line that *reads* like an update (or like textbook mathematics, `B = {2, 4, 6}`) can be written that way.

Because rebinding is silent, the discipline has to live in *you*, not in the interpreter. In the beginner mindset, every name is bound exactly once and never changes; a name that gets rebound mid-program is almost always a bug-in-waiting — either it's really Chapter 16's topic (deliberate mutation) arriving early, or it's a second value that deserves a second name (`total_after_tax`, not a new `total`). For this chapter and the next several: one name, one value.

2.5 One form, by design

In some languages you choose between several binding keywords — `let pi = 3.14159` in ML, Scheme, or JavaScript; `pi := 3.14159` in Pascal, Ada, or Go; `val` / `var` in Scala or Kotlin. Axioma deliberately ships with **one** canonical *value* form:

```
pi: 3.14159
```

Three reasons:

- **It's the shortest of any binding form.** Two characters of overhead (`:`).
- **It puts the name first.** Your eye looks for the name; the form shouldn't get in the way.
- **It composes with type annotations.** `radius :: Integer: 5` reads naturally as "radius (of type Integer) gets 5." We don't use type annotations in Part I, but they show up in Part VI.

The textbook spelling `pi = 3.14159` is the **same** operator (find-or-update), kept so math examples can read unaltered. There is no separate "assignment only" `:=`. The immutable binding is `let NAME = …` (also spelled `val`), read exactly as in a proof: it *declares a fresh binding* that shadows an outer name, and the name then stays fixed. Its mutable twin is `var NAME = …`, and a program-wide constant is `const NAME = …` — tools for later chapters, not for Part I.

Binding model (map for later chapters)

Part I only needs `name: value`. The full model shows up when you write functions, loops, and closures. One table (full detail: *Axioma Manual* §3, "Binding model at a glance"):

Kind	Spelling	Idea
Value bind / update	`x: 5` or `x = 5`	Find-or-update in this function frame
Function definition	`f(x) = …`	Parenthesized head → definition, not a bare value bind
Binder	`foreach x in …`, parameters	Fresh name; shadows outer `x`
Fresh declaration, immutable	`let x = 5` or `val x = 5`	A **new** binding that shadows an outer `x`; never updates, and then stays fixed (the `let` of a proof)
Fresh declaration, mutable	`var x = 5` (`let mut` / `let mutable`)	`let` minus the fixedness — a fresh binding you may keep writing
Explicit local, mutable	`local x = 5`	Fresh mutable binding in this lexical scope; `var` semantics; typed bare forms are optional in §16.1b and §16.14
Deferred initialization	`let x`, `let x :: Integer`, or their `= _` forms	Declare now, fill once; reading early errors (Ch.16). `var`/`local` cells can be refilled; annotations are optional
Identity empty	`let xs :: Array = default`	Start as empty container — not for Integer zeros (Ch.16)
Outer write	`rebind x: …`	Update the nearest existing cell, searching across frames; never declares
Module write	`global x = …`	Julia's file / `module` body write; may declare; honors `::`
Immutable name	`const LIMIT = 10`	A program-level, top-level-only commitment: cannot reassign *or shadow* the name

For worked comparisons, closure captures, and the complete spelling table, see §16.14 · Make scope explicit with local.

So: beginners learn **one** form (`:`). The keywords mark directions the default doesn't take — `let` fixes fresh, `var` varies fresh, `rebind` writes the nearest outer cell, `global` writes this file (Julia) — (Chapter 16 / the manual) — and `:=` stays gone. The `_` / `default` forms are **not** needed until you deliberately leave a name unfilled or start a container empty; Chapter 16 explains why Integer is not silently `0`.

> **Old code may use other forms.** Earlier versions of Axioma accepted `let pi = 3.14159` and `pi := 3.14159` as synonyms of the canonical bind; both were retired in May 2026. `:=` stays a syntax error with a hint pointing at `x: value`. The `let` spelling returned in August 2026 with **different semantics**: `let pi = 3.14` now parses, but it *declares a fresh binding* (shadowing any outer `pi`) rather than updating one. For a top-level constant in old example code the result is the same value; inside functions and loops the difference is real — see the fresh-declaration row above and the manual's §3.
>
> The fix is mechanical: replace `let X = V` with `X: V`, and replace `X := V` with `X: V`.

For the rest of this book we use `name: value` almost exclusively. The `=` spelling appears in math-style lines and in later chapters; treat it as the same bind/update as `:`, not as a second language.

> **Persistence (advanced — skip on first read).** If you want a variable to *survive* between REPL sessions or between script runs — useful when you're slowly building up a knowledge base — the canonical form for that is `declare/persist x = 42` (using `=`, not `:`). This is the one place where `:` doesn't suffice, because the REBOL `:` binder deliberately has no refinement slot. Most code never needs persistence, so don't worry about it now — Part VI covers it.

2.6 A worked example: from radius to several quantities

Say we have a circle of radius 5. We want to compute:

- its diameter (`2 * radius`),
- its circumference (`2 * pi * radius`),
- its area (`pi * radius * radius`),
- the area of a slightly larger circle (radius 6).

Without names, that's four sprawling expressions. With names, it's clean:

```
#language axioma/beginner
pi: 3.14159
radius: 5
diameter: 2 * radius
circumference: 2 * pi * radius
area: pi * radius * radius
println("radius =", radius)
println("diameter =", diameter)
println("circumference =", circumference)
println("area =", area)
```

Run it. You get:

```
radius = 5
diameter = 10
```

```
circumference = 31.4159
area = 78.53975
```

> **Why not just `r: 5`?** It works — but silently steps on a built-in. Axioma starts every session with `r` already bound to a module (the bridge to the R statistical language), and `r: 5` simply *shadows* it: no warning, and the module is unreachable for the rest of your session. That's the flip side of §2.4's silent rebinding — nothing protects an existing name, not even a built-in, from being rebound. **Habit to build: pick names that say what they mean** (`radius`) rather than single-letter abbreviations. The full word also reads better when someone else (or future-you) opens the file.

Notice that we computed `area` using a name (`pi`) that we'd computed earlier. **Names compose.** This is the seed of the next chapter, where we'll learn to compose entire *computations*.

2.7 Seeing the value and its type

A whole-valued Float keeps a decimal marker when printed. The values `32` and `32.0` compare equal numerically, but their types differ:

```
println(32)                # 32
println(32.0)              # 32.0
println(type(32))          # Integer
println(type(32.0))        # Float
println(32 == 32.0)        # true
println(32.0 / 1.0)        # 32.0
```

Use `type(value)` when you want to inspect the type directly. Adding a fraction does not prove what the original type was: an Integer can also participate in a calculation whose result is a Float.

Float output keeps enough significant digits to recover the stored value when read back. It is not necessarily the short decimal you would write by hand. To present a rounded amount, use Chapter 1's `printf` or `stringf`; formatting the output does not change the stored value.

Extension: formatting a value for a reader

Use formatting when the *presentation* needs rounding. It does not change the stored number. Ordinary strings interpolate `${expression}`; an `f`-prefixed string uses `{expression}` and optional format specifications.

```
price: 12.345
customer: "Ada"
println(f"Total: {price:.2f}")   # Total: 12.35
println(f"{42:05d}")            # 00042
println(f"{{{customer}}}")      # {Ada}
println(rf"C:\reports\{customer}")  # C:\reports\Ada
println(price)                 # 12.345
```

The `r` in `rf` preserves backslashes; the `f` still evaluates fields. Double braces produce literal braces.

Expressions in fields run once, from left to right. Formatting is Axioma syntax, not a call into Python; not every Python format extension is accepted. When a colon is part of the expression itself, put that expression in parentheses.

Extension: Time is a relative amount of time

Axioma's `Time` follows a duration-like convention. Hours can exceed 23, overflow carries into the next unit, and negative values are useful. It is not a validator for a clock face. Two integer fields mean hours and minutes; three fields include seconds. The special two-field form with a fractional second field means minutes and seconds.

```
println(23:61)          # 24:01
println(23:59:61)       # 24:00:01
println(-1:30)          # -1:30
println(1:02.5)         # 0:01:02.5
println(1:30 - 0:40)    # 0:50
println(1:00 * 2.5)     # 2:30
println(0:01 / 0:03)    # 1/3
println(float(1:30))    # 5400.0
```

Adding a number to a Time adds seconds. A ratio of two Times is numeric; dividing a Time by a number produces a Time. Storage has nanosecond resolution, so this is not arbitrary-precision fractional time. `time(seconds)` constructs a Time explicitly. The separate datetime package offers `DateTime` and `Duration`; a Duration converts back with `time(duration_value)`. Calendar dates and time zones are separate questions from this relative-time arithmetic.

2.8 Exercises

Exercise 2.1

Set `r: 7` and compute the four quantities in §2.6: diameter, circumference, area, and the area of a circle one unit larger (radius 8). Print each with a clear label.

Exercise 2.2

Distance between two points. Bind `x1: 0`, `y1: 0`, `x2: 3`, `y2: 4`. The Pythagorean theorem says the distance is `sqrt((x2-x1)^2 + (y2-y1)^2)`. Compute it.

> **Hint.** Axioma has a built-in `sqrt`. Use it: `sqrt(25)` returns the Float `5.0`.
>
> **Predict first.** The points `(0,0)` and `(3,4)` form a famous right triangle. What should the answer be?

Exercise 2.3

Body Mass Index (BMI). `BMI = weight / height²`, where weight is in kg and height is in meters. Compute the BMI of someone who weighs 70 kg and is 1.75 m tall. The answer should be about 22.86.

Hint. You can write the division in one line (`bmi: weight / (height * height)`), or give the denominator a name first — the named version reads better:

```
height_squared: height * height
bmi: weight / height_squared
```

Exercise 2.4

Predict, then check. What does each print? (You'll need to think about types — see §2.7.)

```
a: 7
b: 2
println(a / b)
println(a + 0.0)
println((a + 0.0) / b)
```

Exercise 2.5

Silent rebinding. Type this at the REPL exactly:

```
axioma> n: 1
axioma> n: n + 1
```

What does `n` end up being? *Predict before checking.* Axioma does this without any warning — why might that be dangerous in a long program?

Exercise 2.6 — open

Pick a real-world calculation you've done in the last week — a tip on a restaurant bill, the area of a wall you want to paint, the time remaining on a charging laptop. Write it as a short script with well-named bindings, ending in a `println` of the answer.

2.9 Reflection

We did two things this chapter.

One, we gave values names. Names let us reuse work (the `pi` we bound once is the `pi` we used three times) and they let us *say what we mean* (the `radius` is named `radius`, not `5`).

Two, we moved from one-line REPL exploration to multi-line script files. The shape of all your programs from here on is the same: declare some inputs at the top, declare derived quantities below that, print the answer at the end.

In Chapter 3 we'll take the next step: when a calculation is *shaped like a recipe* — input goes in, answer comes out — we'll learn how to package the recipe into a **function**, so we can run it on many inputs without rewriting it.

Looking ahead — what you'll be able to do by Chapter 5:

- Define your own functions (`func`)
- Make decisions with `if … then … else …`
- Apply the **Design Recipe** to turn any problem statement into working code

What we used from Axioma

- `<name>: <expr>` — declaration (the single canonical form)
- `<name> = <new value>` — assignment (we'll come back to this in Ch.16)
- `declare/persist <name> = <expr>` — declaration marked for session saving (advanced; covered in Part VI)
- The `println` built-in
- Strings: `"text in double quotes"`
- `sqrt(x)` — square root (also: `abs`, `min`, `max`, in the manual)
- The `#language axioma/beginner` pragma at the top of a file
- Rebinding is silent: `x: v` and `x = v` both create-or-update a name

Solutions to selected exercises

Solutions to selected exercises: Chapter 2 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch02_solutions.md.)*

Deferred persistence boundary. `declare/persist` selects a save policy; it never forces the expression. Demand the value before saving. Session saves report and skip pending closures and contracted compound values; forced serializable values restore as immutable bindings. See the Manual's persistence section before relying on a deferred value surviving a session.

Chapter 2a · Loops

> **What this chapter is.** Chapter 2 taught you to name a value. This chapter teaches you to **do the same kind of work several times** — print a line five times, walk a short list of numbers — *before* Chapter 3 names a recipe as a function. The unit here is a *trip*: one pass of a body. You already know names, `println`, and scripts.
>
> HtDP waits until recursion (our Chapter 7) to process "many values." Axioma has loops as a first-class tool, so we teach them as soon as you can write a script. Recursion is still coming; it is a different way to think, not a replacement for this chapter.

2a.1 Why repeat a body?

Suppose you want to print a greeting three times. You *could* write the line three times:

```
#language axioma/beginner
println("hello")
println("hello")
println("hello")
```

```
hello
hello
hello
```

That works. It also does not scale: ten greetings is ten lines, and changing the message means changing every copy. A **loop** names the body once and says how many times, or over which values, to run it.

2a.2 Counted `repeat` and `loop`

The simplest loop is a **count**: run this body *n* times.

```
#language axioma/beginner
repeat 3 [
  println("hello")
]
```

```
hello
hello
hello
```

`loop` is the same keyword as `repeat`. Pick one spelling and stick with it in a file; this book uses `repeat`

for a count and `loop` when the body is the `… end` form, only so you see both.

```
#language axioma/beginner
loop 3
  println("hello")
end
```

```
hello
hello
hello
```

A header with **no `[`** whose next token sits on a **later line** runs until a matching `end`. Brackets and `end` are two spellings of one body. Same-line `repeat 3 [ … ]` and multiline `loop 3` / `end` cannot mean different things.

The count is not a variable. After `repeat 3 [ println("hello") ]` there is no leftover `i`. If you want a name for "which trip is this?", that is the next form.

2a.3 `for` — a name for each value

`for` walks a source and binds a **fresh** name on every trip:

```
#language axioma/beginner
for v in [10, 20, 30] [
  println(v)
]
```

```
10
20
30
```

`[10, 20, 30]` is a short list of values. Chapter 6 is the full story of arrays; here the list is just "these three numbers, in this order."

`foreach` is the same keyword as `for`. This book writes `for`.

The name `v` belongs to the loop. It does not leak afterwards, and it does not overwrite an outer name of the same spelling:

```
#language axioma/beginner
v: 99
for v in [1, 2] [
  println(v)
]
println("after:", v)
```

```
1
2
after: 99
```

That is the **binder** rule you will meet again for function parameters (Chapter 3) and for the generators in

Chapter 10. A loop variable is not an assignment to your `v`. It is a new `v` for the body, gone when the loop ends.

A compact way to walk the integers from 1 through 5 is a **range**:

```
#language axioma/beginner
for n in 1..5 [
  println(n)
]
```

```
1
2
3
4
5
```

`1..5` includes both ends. You do not need a counter you update by hand. Ranges as a topic — exclusive `..<`, a step `by 2`, open `1..` — wait until later; this chapter only needs "from here through there."

The `end` spelling works for `for` too:

```
#language axioma/beginner
total: 0
for n in 1..4
  total = total + n
end
println(total)
```

```
10
```

`total = total + n` is the find-or-update from §2.4: `total` is *your* name, so the body writes through to it. `n` is the loop's name. Accumulators work because a **body assignment** is not a binder.

2a.4 `while` — when the stop test is yours

`repeat 3` knows the trip count up front. `for n in 1..5` knows the source up front. Sometimes the stop condition is a test you re-check, and you do not have a list:

```
#language axioma/beginner
n: 1
while (n <= 3) [
  println(n)
  n = n + 1
]
println("after:", n)
```

```
1
2
3
after: 4
```

`while` binds **nothing**. The counter is yours and **survives** — that is why `after:` prints 4. A block-local `n` would never change the `n` in the test, and the loop would not stop. Chapter 16 returns to this as a design technique; here it is just the loop whose exit is a condition rather than a count or a list.

Parentheses around the test are optional. `while n <= 3 [ … ]` is the same loop.

If the test is already false, the body runs **zero** times:

```
#language axioma/beginner
n: 5
while (n < 0) [
  println("never")
]
println("done")
```

```
done
```

2a.5 Three constructs, not three spellings

Axioma has **three** loop constructs. Two of them have a second spelling.

Construct	Also written	What it walks	Who owns the name
`while`	—	a condition you re-test	you (binds nothing)
`for`	`foreach`	a source (`[…]`, `1..5`, …)	the loop (a binder)
`repeat`	`loop`	a count, a source, or a condition	optional; a count binds nothing

The beginner subset allows all three. What it still refuses are *other* spellings of nearby ideas (`lambda` for `func`, `match` for `if`, `print` for `println`). Do not treat `for` as a dialect of `while`.

The Manual's Loops section is the inventory for later forms: `until` (body first, then a test), labels (`'outer:`), combined `for a in A, b in B`, `break` / `continue`. This chapter does not need them.

2a.6 Looking ahead

Chapter 3 names a *computation* so the same formula is not copy-pasted. You can already loop; you cannot yet wrap "area of a circle" as a reusable recipe. After functions, Chapter 7 will process a list by *recursion* — stop, shrink, combine — which is the design-recipe tool for arbitrary-size data. Use a loop when you know the trips; use a function when you know the rule; use recursion when the data says how deep to go.

2a.7 Optional: a worded range header

The bounds can also be written with words. These two loops both print 2, 9 and 16. `by 7` is a positive step magnitude, and 20 is not reached exactly.

```
#language axioma/beginner
for i from 2 to 20 by 7 [println(i)]
for i from 2 to 20 by 7
  println(i)
end
```

Without `by`, the magnitude is 1. Swapping the bounds gives a countdown: `from 20 to 2 by 7` visits 20, 13, 6. Negative and zero magnitudes are errors. `foreach` accepts the same header and the same two bodies.

Exercises

Exercise 2a.1 — four ticks

Using `repeat` or `loop`, print the word `tick` four times, each on its own line.

Exercise 2a.2 — walk a list

Using `for`, print each value in `[2, 4, 6]`. Then print each value doubled (`v * 2`) on a second pass over the same list.

Exercise 2a.3 — a running total

Bind `total: 0`. Using `for n in 1..5`, add each `n` into `total`. Print `total`. (It should be 15.)

Exercise 2a.4 — countdown

Using `while`, print 5, then 4, then 3, then 2, then 1. Do not print `0`.

Exercise 2a.5 (open) — a table of squares

Print two columns, `n` and `n * n`, for `n` from 1 through 8. Either `for` or `while` is fine. Decide which name is the loop's and which name, if any, is yours.

Exercise 2a.6 — worded steps

Write the bracket and end versions of a loop from 20 to 2 by 7. Predict the output. Explain why 2 does not appear, then change the magnitude so that the endpoint is reached.

Chapter 3 · Functions

> **What this chapter is.** When a computation reads "input goes in, answer comes out", that computation is a **function**. Naming a function is even more powerful than naming a value: a value works once, but a function works on any input you give it.

3.1 Why give a *computation* a name?

Chapter 2 taught us to name values. Chapter 2a taught us to repeat a body. But look at this script:

```
#language axioma/beginner
pi: 3.14159
radius_a: 5
area_a: pi * radius_a * radius_a
radius_b: 7
area_b: pi * radius_b * radius_b
radius_c: 12
area_c: pi * radius_c * radius_c
```

We computed the area of three circles, and we copy-pasted the *formula* three times. That's a clear sign we should name the formula itself. Then we can use the name three times and write the formula exactly once.

The thing we want to name is a *recipe*: "given a radius, compute the area". In Axioma, that's a **function**.

3.2 Defining a function

The shape:

```
<name>: func(<parameter>) [ <body> ]
```

Read it left-to-right: "`name` gets the function that takes `parameter` and computes `body`."

```
#language axioma/beginner
pi: 3.14159
area: func(radius) [pi * radius * radius]
println(area(5))
println(area(7))
println(area(12))
```

Output:

```
78.53975
153.93791
452.38896
```

We defined `area` *once*. We *called* it three times. Each call passes a different `radius`. The function does the same thing in every case — it multiplies `pi` by that radius squared.

What just happened in `func(radius) [pi * radius * radius]`

- `func` — keyword saying "what follows is a function definition".
- `(radius)` — the parameter list. This one has a single parameter named `radius`. The parameter is a **placeholder** — it gets bound to whatever value is passed in.
- `[pi * radius * radius]` — the **body**, between square brackets. The body is an expression; its value is what the function returns.

When you write `area(5)`: 1. Axioma evaluates `5`. 2. It binds `radius` to `5` inside the function's body. 3. It evaluates the body — `pi * 5 * 5 = 78.53975`. 4. That value is the result of the call.

The `radius` binding only exists *inside the body*; once the function returns, it's gone. Functions have their own little namespace.

3.3 Functions with more than one parameter

Most useful functions take more than one input. The parameter list is comma-separated:

```
#language axioma/beginner
distance: func(x1, y1, x2, y2) [
  dx: x2 - x1
  dy: y2 - y1
  sqrt(dx ^ 2 + dy ^ 2)
]
println(distance(0, 0, 3, 4))
println(distance(1, 1, 4, 5))
```

Output:

```
5.0
5.0
```

Two things to notice.

One, the body is now *several lines*. When a body has multiple statements, only the value of the *last* one is the return value. The intermediate `dx: ...` and `dy: ...` are setup; the final `sqrt(...)` is the answer.

Two, rather than writing `sqrt((x2-x1)^2 + (y2-y1)^2)` in one line — which works — we extract `dx` and `dy` as names. Each line does one thing, and the intermediate quantities get meaningful names. That's just better style.

3.4 A worked example: Celsius to Fahrenheit

The formula: `F = C × 9/5 + 32`.

```
#language axioma/beginner
c_to_f: func(c) [
  scaled: c * 9.0 / 5.0
  scaled + 32.0
]
println(c_to_f(0))
println(c_to_f(100))
println(c_to_f(-40))
println(c_to_f(37))
```

Output:

```
32.0
212.0
-40.0
98.6
```

The `c * 9.0 / 5.0` uses **`9.0`** and **`5.0`** instead of 9 and 5 so the answer comes back as a *decimal*. Nothing would be *wrong* with `c * 9 / 5` — Axioma's division is exact, so `100 * 9 / 5` is `180` and `37 * 9 / 5` is the exact rational `333/5` (which *is* 66.6) — but a temperature that prints as `333/5` reads like a math worksheet, not a thermometer. A float literal anywhere in the expression keeps the whole result in decimal-land. Habit: for measurement-flavored functions, write the fraction as `9.0 / 5.0`.

The `-40` line is famous — it's the only temperature at which Celsius and Fahrenheit agree. Try predicting that *before* running.

3.5 Functions are values too

A function bound to a name is just another value. You can pass it around like any other:

```
#language axioma/beginner
double: func(x) [x * 2]
triple: func(x) [x * 3]
same_as_double: double
println(same_as_double(7))
```

Output:

```
14
```

The line `same_as_double: double` doesn't *call* `double` — it just binds another name to the same function. From then on, both names refer to the same recipe.

This is a small thing in Chapter 3. In **Chapter 10** it becomes the foundation of *higher-order functions* — functions that take functions as inputs and return them as outputs. That's what unlocks patterns like `map` and `filter`.

Looking ahead. In the full Axioma language there's also a shorter syntax for one-line functions:

```
double: lambda x => x * 2
```

The `lambda` form is **disabled in beginner mode**. We'll unlock it in Chapter 11. For now, `func(x) [body]` is the only form, and that's intentional — one tool, one job.

The textbook spelling. There is one more form, and unlike `lambda` it *does* work in beginner mode — because it is simply how mathematics writes a definition:

```
double(x) = 2 * x
```

That is exactly `double: func(x) [x * 2]` — same function, same everything, just written the way you would write it on paper. It also takes clause forms taught in §5.9: guards select among equations, and §18.6 develops patterns for constructor data:

```
fact(0) = 1                  # one clause per case …
fact(n) = n * fact(n - 1)    # … tried in order
sign(x) when x > 0 = 1       # a guard sits before the `=`
sign(x) = 0
```

We will keep writing `name: func(args) [body]` through this book. It is the form that makes "a function is a value you bind to a name" visible on the page, and that idea is doing real work in §3.5 and again in Chapter 10. Reach for the equation form when you are transcribing mathematics and want the page to match the source. Appendix A has the full rules.

3.6 Calling a function

Calling a function is so simple it's easy to miss. To call `f` with arguments `a` and `b`, write `f(a, b)`. The parentheses are required, and they go *immediately* after the function's name (no space — well, spaces are tolerated by the parser, but the convention is no space).

If you write `f` without parentheses, you're not calling it — you're referring to the function itself, as a value.

```
axioma> double: func(x) [x * 2]
axioma> double
func (x) [x * 2]
axioma> double(5)
10
```

The first `double` printed the *function object* (the recipe). The second printed `10` — the result of *applying* the recipe to `5`.

3.7 A peek at the Design Recipe

The full version is Chapter 4. Here's the spoiler: every function you write follows the same six steps, and **doing them in order** is the single biggest skill-builder in this book.

1. **Data definition.** What kind of input? What kind of output?
2. **Signature comment.** Write the *types* down. (`area: Float -> Float`)
3. **Examples.** Write `area(5) = 78.53975` *before* you write the code. Multiple examples.
4. **Template.** What's the shape of the body, given the shape of the input?
5. **Body.** Fill the template in.
6. **Tests.** Re-check your examples by running the function.

Watch this happen for our Celsius converter:

```
# Data:  Celsius temperature (Float)
# Signature:  c_to_f: Float -> Float
# Examples:
#   c_to_f(0)    = 32
#   c_to_f(100)  = 212
#   c_to_f(-40)  = -40
# Template:
#   c_to_f: func(c) [
#     ... c ...  # some expression in c
#   ]
# Body:
c_to_f: func(c) [
  c * 9.0 / 5.0 + 32.0
]
# Tests:
println(c_to_f(0))     # expect 32
println(c_to_f(100))   # expect 212
println(c_to_f(-40))   # expect -40
```

That's the recipe applied informally. Chapter 4 makes it the *primary* way you think about programming. The point isn't bureaucracy. The point is: most "bugs" are really *unfinished thinking*, and the recipe forces you to finish the thinking before the code.

Axioma even includes a guided wizard for the recipe. Try it:

```
$ axioma --recipe
```

It walks you through the six steps for whichever function you choose.

3.8 Exercises

Exercise 3.1

Write `circle_area` and call it on radii 1, 5, and 100. (Use `pi = 3.14159`.)

Exercise 3.2

Write `f_to_c` — Fahrenheit to Celsius. The formula is `C = (F − 32) × 5/9`. Test it on:

- `f_to_c(32)` should be `0`
- `f_to_c(212)` should be `100`
- `f_to_c(98.6)` should be about `37`

Tip. Same as §3.4: use `5.0/9.0` to keep the arithmetic in floats.

Exercise 3.3

Write `bmi(weight, height)` using the formula from Exercise 2.3. Then call it on three people:

- `bmi(70.0, 1.75)`
- `bmi(85.0, 1.80)`
- `bmi(55.0, 1.65)`

Exercise 3.4

Write a function `tip(bill, percent)` that takes a bill amount and a tip percentage (e.g., `20`) and returns the dollar amount of the tip. Test it on:

- `tip(50.0, 20)` should be `10`
- `tip(100.0, 15)` should be `15`
- `tip(42.50, 18)` should be about `7.65`

Exercise 3.5

Write a function `is_even(n)` that returns `true` if `n` is even and `false` otherwise. (Hint: `n % 2 == 0`.) Test it on a few integers. This is a sneak preview of Chapter 5.

Exercise 3.6 — open

Pick a calculation you do often (or imagine you might): converting a recipe between metric and imperial units, computing your monthly salary from an hourly rate, computing how many hours of streaming you can do per day on a metered internet plan. Whatever it is, package it as a function with sensible parameter names, and call it twice with different inputs.

Extension: defaults and named arguments

A default belongs to the function's interface: callers may omit that argument. Both the equation form and `func` can express defaults.

```
scale_number(x, factor = 2) = x * factor
println(scale_number(3))             # 6
println(scale_number(3, 4))          # 12
println(scale_number(3, factor: 5))   # 15
```

Choose a default that makes sense for the problem, and test both omission and an explicit value. A named argument uses `name: value` at the call.

Optional: receiver calls and returned functions

A value can be placed before a dot when calling an ordinary user function. The value becomes its first argument. This works alike for strings and numbers; the function stays in the current scope rather than belonging to the value. Save these optional examples in a separate full-language file.

```
decorate(value, suffix, ending) = str(value) + suffix + ending
println("Joan".decorate("!", ";"))    # Joan!;
println(5.decorate("!", ";"))         # 5!;
println(decorate(5, "!", ";"))        # 5!;
println("Joan".upper)                 # JOAN
println("Joan".uppercase())           # JOAN
println(16.sqrt())                    # 4.0
```

For a user function, `value.f(a, b)` means one call, `f(value, a, b)`. All supplied arguments take part in overload selection. If a unary definition and a two-argument definition share a name, the complete two-argument call selects the latter:

```
choose(x) = (n -> x + n)
choose(x, n) = x * n
println(5.choose(2))                  # 10
println(choose(5)(2))                 # 7
```

The second print makes **two calls**: `choose(5)` returns a function, then `(2)` invokes it. Earlier scalar dot calls used that two-call reading; code that needs it must now spell both calls explicitly. Bare `5.choose` still means `choose(5)`. Empty parentheses also perform only one call:

```
maker(x) = (n -> x + n)
add_five: 5.maker()
println(add_five(2))                  # 7
println(maker(5)(2))                  # 7
thunk(x) = func() [x + 1]
println(thunk(5)())                   # 6
```

Writing `5.maker(2)` attempts `maker(5, 2)` and raises an argument error. Putting parentheses around `(5.maker)` does not restore the old behavior; the parser discards grouping. Use the explicit function calls above or bind the returned function before invoking it. No failed call is retried using the other interpretation.

Exercise 3.7 (optional). Define `decorate(value, suffix, ending)` as above and predict its string and integer receiver outputs. Explain why the two `choose` calls give different results. Rewrite an old `5.maker(2)` call and an old `5.thunk()` call to preserve their original results, 7 and 6. Test both rewrites. Starter: `exercises/ch03/ex_3_7_receiver_calls.ax`.

After Chapter 6 introduces arrays, the same receiver-first rule applies:

```
xs: [1, 2, 3]
scale_values(values, factor = 2) = values .* factor
println(xs.scale_values(3))       # [3, 6, 9]
println(scale_values(xs, 3))      # [3, 6, 9]
println(xs.length)               # 3
println(xs.length())             # 3
```

This is a call rule, not a stored method added to each array. Builtin methods have a receiver-position table: `xs.map(f)` means `map(f, xs)`, while a user `scale_values` receives `xs` first. Existing properties take precedence; a callable field keeps its own call behavior. Registered text methods include `upper`/`uppercase`, `lower`/`lowercase`, `title`/`titlecase`, `capitalize`, `swapcase` and `trim`. A Character stays a Character through the case operations; trimming produces a String. Dictionaries, object methods and modules keep their own member lookup. Receiver calls use the interpreter; unsupported VM forms refuse. A dot does not request broadcasting: that is the separate `f.(xs)` form taught in Chapter 10.

Later reading: refinements

A **refinement** selects an optional behavior explicitly. This example uses the full language and can be saved as a separate file:

```
combine: func(a, b, /times amount) [
  if times then (a + b) * amount else a + b
]
println(combine(1, 2))                       # 3
println(combine/times(1, 2, amount: 10))      # 30
```

The declaration names the switch and its argument; an arbitrary suffix does not invent a behavior. User refinements remain an interpreter feature; the VM refuses unsupported refinement calls.

3.9 Reflection

A function bundles a *recipe*. We learned three things:

One, the syntax: `name: func(params) [body]`. The body is an expression; its value is what the function returns.

Two, multi-parameter functions; multi-line bodies; the fact that the last line of a multi-line body is the return value.

Three, functions are *values* — they can be bound to names just like numbers and strings. (We didn't *use* this much yet. Chapter 10 will return to it.)

In Chapter 4 we'll slow down and learn the **Design Recipe** — the six-step process that turns "I need a function for X" into actual working code, every time.

Looking ahead — what you'll be able to do by Chapter 5:

- Apply the **Design Recipe** end-to-end

- Make decisions inside a function with `if … then … else …`
- Build small functions that combine arithmetic and conditional logic

What we used from Axioma

- `<name>: func(<params>) [ <body> ]` — function definition
- `<name>(<args>)` — function call
- Multi-line function bodies with intermediate bindings
- The last expression of a body is the return value
- `sqrt(x)` — square root from the standard library
- `axioma --recipe` — the interactive Design Recipe wizard

Solutions to selected exercises

Solutions to selected exercises: Chapter 3 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch03_solutions.md.)*

Chapter 4 · The Design Recipe

What this chapter is. The single most important idea in this book. Not a feature of Axioma — a *habit* — that turns "I need a program for X" into a tested implementation. Six steps. Do them in order.

4.1 Why a recipe?

Most beginners — and a lot of professionals — sit down at a blank screen and start typing the body of a function. They have a vague sense of what it should do. They guess at the body. They run it. It fails. They edit. They run again. They debug. They edit. Eventually something works, sort of.

This is exhausting, and the resulting code is hard to trust. **Most "bugs" are really unfinished thinking** — the programmer never quite sat down and said exactly what the function takes and exactly what it returns. The recipe forces you to finish the thinking *before* you write the body.

Six steps. Each one builds on the previous one. With practice the sequence becomes a habit. It will not eliminate errors; it gives you a way to locate them and revise your design.

The six steps:

1. **Data definition.** What's the input? What's the output? Be precise about *types*.
2. **Signature.** A one-line comment of the function's type: `# name: InputType -> OutputType`.
3. **Examples.** Two or three concrete `name(input) = output` lines, *before* writing any code.
4. **Template.** What's the shape of the body, given the shape of the data?
5. **Body.** Fill the template in.
6. **Tests.** Re-check your examples by running them.

Stages 1–3 happen *before* you write any code. Stage 4 is mechanical. Stage 5 is "the actual work" — which is now small, because every other stage has prepared you. Stage 6 confirms.

4.2 The six stages, illustrated

We'll work the recipe for `f_to_c` (Fahrenheit to Celsius) from Chapter 3 — but this time, every step is written down.

Stage 1 — Data definition

> A Celsius (or Fahrenheit) temperature is a **Float**. Possibly negative (it gets cold).

That's it. One sentence. Sometimes the data definition is more complex (a list of integers, a tree of strings), but for now it's a single primitive type.

Stage 2 — Signature

Write the type of the function as a comment, in the form

```
# fname: InputType -> OutputType
```

```
# f_to_c: Float -> Float
```

For multi-parameter functions, the input types are space-separated:

```
# distance: Float Float Float Float -> Float
```

The arrow `->` is the universal "produces" symbol — input goes on the left, output on the right. It's borrowed from mathematicians' notation for functions (`f : ℝ → ℝ`).

Stage 3 — Examples

Write *concrete* input/output pairs. Two minimum. Three is better. Pick:

- one **easy** case (round numbers, obvious answer)
- one **edge** case (zero, negatives, boundary values)
- one **representative** case (something realistic)

```
# Examples:
#   f_to_c(32)   = 0       (freezing point — easy case)
#   f_to_c(212)  = 100     (boiling point — easy case)
#   f_to_c(98.6) = 37      (body temperature — representative)
#   f_to_c(-40)  = -40     (the famous fixed point — edge)
```

Notice: we wrote the *expected answer* for each one. Where did the answers come from? From the formula `C = (F - 32) × 5/9`, computed by hand or in our head. The point is to commit to what the answer *should* be — so when the function is wrong, we'll catch it.

> **Why this matters.** "Examples first" looks like extra work. It isn't. It saves time three times over. **First**, writing examples forces you to clarify what the function actually does — many vague specifications collapse here. **Second**, the examples become your tests in stage 6. **Third**, the *worked* examples often suggest the formula. If you couldn't compute `f_to_c(32) = 0` by hand, you probably don't yet know the formula well enough to write it as code.

Stage 4 — Template

The template is the **shape of the body**, determined by the shape of the data. For a function over a single primitive, the template is boring:

```
# Template:
# f_to_c: func(f) [
#   ... f ...
```

```
# ]
```

That's it. The body uses `f` somehow.

This stage feels silly for primitives. *Don't skip it anyway.* The template will *save your life* in Chapter 7, where the data is a list and the template has two cases (empty / non-empty), and in Chapter 8 where the data is a tree.

For now, the template just says "the body is an expression in `f`."

Stage 5 — Body

Now — and *only* now — write the actual code.

```
f_to_c: func(f) [
  (f - 32) * 5.0 / 9.0
]
```

We wrote `5.0/9.0` (not `5/9`) for float division, as in Chapter 3.

Stage 6 — Tests

Run the examples and check they match.

```
println(f_to_c(32))      # 0.0
println(f_to_c(212))     # 100.0
println(f_to_c(98.6))    # 37.0
println(f_to_c(-40))     # -40.0
```

Run the file. All four match. Done.

The full file, in order, looks like:

```
#language axioma/beginner
# Data: a Celsius or Fahrenheit temperature is a Float.
# Signature: f_to_c: Float -> Float
# Examples:
#   f_to_c(32)   = 0
#   f_to_c(212)  = 100
#   f_to_c(98.6) = 37
#   f_to_c(-40)  = -40
# Template:
#   f_to_c: func(f) [
#     ... f ...
#   ]
# Body:
f_to_c: func(f) [
  (f - 32) * 5.0 / 9.0
```

```
]
# Tests:
println(f_to_c(32))      # 0.0
println(f_to_c(212))     # 100.0
println(f_to_c(98.6))    # 37.0
println(f_to_c(-40))     # -40.0
```

This is the *shape* of every function file from here on. The comments aren't decoration — they're the recipe, written down, so a future reader can see your thinking. They also help you when you return to your own code a week later and have no idea what you were trying to do.

Stage 6, mechanized — doctests

Notice what Stage 6 made us do: re-type each Stage-3 example as a `println` and *eyeball* the output. Axioma can do the checking for you. Inside a `/**md … */` documentation block (or any `.md` file), open a fence with the info string `axioma doctest`, and every line of the form `expr  # → value` becomes a **checked assertion** — the examples *are* the tests, in the exact shape you wrote them in Stage 3:

```
f_to_c: func(f) [ (f - 32) * 5.0 / 9.0 ]
f_to_c(32)        # → 0.0
f_to_c(212)       # -> 100.0
f_to_c(98.6)      # → 37.0
f_to_c(-40)       # -> -40.0
f_to_c("cold")    # raises: unknown operator
```

Two spellings of the arrow are equivalent: the glyph `# →` (the convention used throughout this book) and the plain-keyboard ASCII `# ->` — both lines above are checked the same way. A line `expr  # raises: substr` asserts that the expression *errors* with a message containing `substr`, and any other line (like the `f_to_c` definition) is setup, run for effect. Run them with:

```
$ axioma --doctest myfile.ax     # ok: / not ok: per example, exit 1 on any miss
```

or the Playground's ✓ **Doctest** button. If a doctest fails, the recipe's diagnosis applies unchanged: the bug is in the body, the template, or the data definition — and now a machine tells you *which example* broke instead of you re-reading `println` output.

4.3 The recipe wizard

Axioma ships with an interactive wizard that walks you through the six stages. Try it:

```
$ axioma --recipe
```

It asks for your function name, then offers prompts for each stage in order. At each stage you type your answer; the wizard saves it and moves on. At the end it writes a starter file with everything in place — you only have to fill in the body.

The wizard is especially helpful when you're stuck. If you can't get past stage 3 (examples), you don't understand the problem yet. Walk away, think, come back. *The recipe is also a diagnostic.*

There are also `--recipe show` to list the five worked recipes built into the wizard, and `--recipe replay <name>` to step through one end-to-end. The five built-in recipes (chosen to match this book) are:

- `circle_area` — single-parameter arithmetic
- `f_to_c` — domain-specific formula
- `bmi` — multi-parameter arithmetic with intermediate names
- `is_even` — Boolean-returning function (preview of Chapter 5)
- `tip` — money math

Use them as crib sheets. The wizard never writes the body for you — it writes the *scaffolding* and asks you to finish.

4.4 The recipe applied: `tip_total`

Let's design a function that takes a bill amount and a tip percentage and returns the *total* you owe (bill + tip).

Stage 1 — Data definition

A bill amount and a tip percentage are both Floats. The result is a Float (the total to pay).

Stage 2 — Signature

```
# tip_total: Float Float -> Float
```

Stage 3 — Examples

Pick three cases:

```
# Examples:
#   tip_total(100.0, 0)    = 100      (zero tip — edge)
#   tip_total(100.0, 20)   = 120      (clean numbers — easy)
#   tip_total(42.50, 18)   = 50.15    (realistic — representative)
```

We computed `42.50 + 42.50 * 18 / 100 = 42.50 + 7.65 = 50.15` by hand.

Stage 4 — Template

```
# Template:
# tip_total: func(bill, percent) [
#   ... bill ... percent ...
# ]
```

The body uses both parameters somehow.

Stage 5 — Body

```
tip_total: func(bill, percent) [
  tip: bill * percent / 100
  bill + tip
]
```

Stage 6 — Tests

```
println(tip_total(100.0, 0))    # expect 100
println(tip_total(100.0, 20))   # expect 120
println(tip_total(42.50, 18))   # expect 50.15
```

Run it. All three match. Done.

> **Notice the pattern.** Stage 5 was *easy* because stages 1–3 already told us what the function had to do. The body has *zero* guesswork left in it. That's the payoff.

4.5 What to do when the recipe gets stuck

The recipe can fail at any stage. Each kind of failure is informative.

- **Stuck at Stage 1 (Data).** You don't yet understand what the function is supposed to do. *Walk away.* Talk to whoever assigned the problem. Don't write code yet.
- **Stuck at Stage 2 (Signature).** The data isn't a single primitive type — it's a list, or a record, or one of several alternatives. You don't have the tools yet (those land in Chapters 6–9). For Part I, every signature is `(some primitives) -> (some primitive)`.
- **Stuck at Stage 3 (Examples).** You can't compute the answer by hand. *That means you can't write the formula yet either.* Find the formula (in a textbook, in a calculator, on Wikipedia, by asking someone). The examples come second; the formula comes first.
- **Stuck at Stage 4 (Template).** Almost never for primitives. For lists and trees (Chapters 6+), this is where you'll spend real time.
- **Stuck at Stage 5 (Body).** Now you've finally hit the actual programming problem. With everything else nailed down, the body is *small* — usually a single expression.
- **Stuck at Stage 6 (Tests fail).** The body is wrong. Compare what the function did with what your example said it should do; the difference *is* the bug.

4.6 Exercises

For each of these, do **all six stages**. Don't skip stages even when they feel obvious. The point is to build the habit.

Exercise 4.1 — `triple`

Design a function that triples its argument. Inputs are integers.

Exercise 4.2 — `discount`

Design a function `discount(price, percent_off)` that returns the discounted price. For example, `discount(100.0, 25)` should be `75`.

Exercise 4.3 — `dollars_to_cents`

Design a function `dollars_to_cents(dollars)` that converts a dollar amount (Float) to integer cents. `dollars_to_cents(1.50)` should be `150`. *Hint:* think about what Axioma's `*` does to a Float, and whether you want the result as an Integer.

Exercise 4.4 — `rectangle_perimeter`

Design a function that computes the perimeter of a rectangle given its width and height. (Perimeter = `2 * (width + height)`.)

Exercise 4.5 — `is_positive`

Design a Boolean-returning function `is_positive(n)`. Returns `true` when `n > 0`, `false` otherwise. (This is a preview of Chapter 5.)

Exercise 4.6 — `open`

Pick a calculation you've done in the last week that wasn't already a function. Apply the recipe to it from scratch — *write down* all six stages, then run it.

4.7 Reflection

The Design Recipe is the most important habit in this book. Everything that follows — recursion, higher-order functions, mutual recursion, generative recursion — gets *easier* when you have the recipe, and *much harder* when you don't.

What the recipe *really* gives you is **structure for thinking**:

- **Data first** forces you to know what you're operating on.
- **Signature** forces you to commit to types.
- **Examples** force you to know the answer in concrete cases — which often reveals you don't yet understand the problem.
- **Template** prepares the *shape* of the code from the shape of the data.
- **Body** is now a small fill-in.
- **Tests** confirm.

By the end of Chapter 9 you'll have used the recipe on dozens of functions and stopped thinking of it as a chore. By the end of the book, you'll do it in your head, and you'll do it faster than the people who insist they "don't need to plan."

In Chapter 5 we'll add conditionals to your vocabulary — `if then else` — and we'll redo the recipe for functions that have to make decisions.

Looking ahead — what you'll be able to do by Chapter 5:

- Apply the Design Recipe even when the function has multiple *cases* (e.g., one answer when `n > 0`, another otherwise)
- Use Boolean values and Boolean-returning functions as building blocks

What we used from Axioma

- `# comment` for the signature, examples, and template
- `axioma --recipe` — the interactive Design Recipe wizard
- `axioma --recipe show` — list the five worked recipes
- `axioma --recipe replay <name>` — step through a built-in recipe end-to-end

Solutions to selected exercises

Solutions to selected exercises: Chapter 4 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch04_solutions.md.)*

Chapter 5 · Conditionals and Booleans

What this chapter is. How a function makes decisions. Up to here, our functions did the same thing for every input. Now we'll ask them to inspect the input and pick a branch.

5.1 Two values: `true` and `false`

There is a type called **Boolean** (named after the 19th-century logician George Boole). It has exactly two values:

```
axioma> true
true
axioma> false
false
```

That's the whole type. No other values are Booleans. (Some languages use `1` and `0`, or `T` and `nil`. Axioma uses words.)

You already produced Booleans without realizing in Chapters 1 and 3:

```
axioma> 5 > 3
true
axioma> 5 == 4
false
axioma> 7 % 2 == 0
false
```

Comparison operators *take* numbers (or other values) and *return* Booleans. The comparison `5 > 3` is asking "is 5 greater than 3?" — `true` or `false`.

5.2 Comparison operators

Operator	Meaning	Example
`==`	equal to	`5 == 5` → `true`
`!=`	not equal to	`5 != 4` → `true`
`<`	less than	`3 < 5` → `true`
`>`	greater than	`5 > 3` → `true`
`<=`	less than or equal	`5 <= 5` → `true`
`>=`	greater than or equal	`5 >= 6` → `false`

The double == is intentional. Single = is for *assignment* (as in `x: 5`); double == is for *comparison*. Most

beginners accidentally write = once when they meant ==, get a strange error, and learn the lesson.

```
axioma> 7 == 7
true
axioma> 7 != 8
true
axioma> 100 >= 100
true
axioma> 100 > 100
false
```

Comparisons work on numbers, strings and Booleans. Equality examples:

```
axioma> "apple" == "apple"
true
axioma> "apple" == "banana"
false
axioma> true == true
true
axioma> "apple" < "banana"
true
axioma> false < true
true
axioma> true <= false
false
axioma> min(true, false)
false
```

Strings compare lexicographically by character codes, with uppercase before lowercase.

Booleans order false before true. Comparing them with <= has implication's truth table: only a true left operand and a false right operand produce false. This is an order on Boolean values, not numeric conversion; comparing false with zero using < is an error. The collection operations `sort`, `min` and `max` use the same Boolean order. These comparisons work in both the evaluator and VM.

Predicate functions

A comparison produces a Boolean. So does a *predicate function* — a function whose whole job is to answer a yes/no question. By long Lisp/Scheme tradition (the one this book follows) their names end in a question mark, which Axioma reads as part of the name:

```
#language axioma/beginner
zero?(0)          # true        even?(4)      # true        odd?(3)   # true
positive?(5)      # true        negative?(-2) # true
empty?([])        # true        empty?("")    # true
```

`zero?(n)` is clearer than `n == 0`, and `even?(n)` says what you mean better than `n mod 2 == 0` — reach for the predicate when one exists. You'll use these constantly in conditionals and (Chapter 7) as the base-case test in recursion.

5.3 Boolean operators: `and`, `or`, `not`, `xor`

Compose Booleans into bigger Booleans with `and`, `or`, `not`, and `xor`.

Operator	Meaning	Example
`and`	both must be true	`true and false` → `false`
`or`	at least one must be true	`true or false` → `true`
`not`	flips true ↔ false	`not true` → `false`
`xor`	exactly one must be true	`true xor true` → `false`

```
axioma> true and true
true
axioma> true and false
false
axioma> false or true
true
axioma> not (5 > 3)
false
axioma> true xor false
true
axioma> true xor true
false
axioma> 1 < 2 && 2 < 3
true
axioma> true || false && false
true
axioma> false && (1 / 0 == 0)
false
axioma> true || (1 / 0 == 0)
true
```

Optional spellings. && and ∧ mean `and`; || and ∨ mean `or`. They preserve precedence (`and` binds tighter than `or`) and short-circuiting: Boolean `false && ...` and `true || ...` skip the right operand. These are exact aliases, including the multivalued logic rules taught later; they do not turn arbitrary values into Booleans or return an operand as a fallback. The book usually uses the words. Single & and | have other syntax roles.

Evaluation matters too. `xor` evaluates both operand expressions. So do `iff` (equivalence), `implies` (implication), NAND ⊼ and NOR ⊽, even when a Boolean result could already be known from the left. Use `and` / `or` for Boolean guards that must skip an unsafe right operand. For the complete rules, including truthiness guards, see Appendix A: Logical evaluation.

Watch the difference between `or` and `xor`. Plain `or` is *inclusive*: it is true when at least one side is true — including when *both* are. `xor` (*exclusive* or) is true when *exactly one* side is true — the "soup **or** salad, not both" reading that English often intends. When a menu, a coin flip, or a light switch is involved, `xor` is usually the operator you mean.

These read like English. *"Is n between 1 and 10?"* becomes:

```
axioma> n: 7
axioma> n >= 1 and n <= 10
```

`true`

"Is the year a leap year?" (we'll do this carefully in §5.7) becomes a slightly longer combination of `and` and `or`.

5.4 `if ... then ... else ...`

The main decision tool:

```
if <boolean-expression> then
  <expression-when-true>
else
  <expression-when-false>
```

Two flavors of use.

As an expression (returns a value)

```
axioma> n: 5
axioma> label: if n > 0 then "positive" else "non-positive"
axioma> label
"positive"
```

The whole `if ... then ... else ...` is an *expression* — it has a value. We can bind that value to a name, return it from a function, or pass it to `println`.

As a statement (decides what to do next)

```
n: 5
if n > 0 then
  println("n is positive")
else
  println("n is not positive")
```

Same form, but here we don't capture the result — we use the chosen branch *for its side effect* (the `println`). The whole construct is still technically an expression; we just don't use the value.

Compact `then` and the … `end` dual

The one-line spelling uses **`then`**, so `if`, `then`, and `else` read as the structure they encode. Some other languages write `if cond { … }` with braces and no `then`. Axioma's compact form keeps the word.

The **other** spelling is the one Chapter 2a already used for `loop` and `for`: a header with **no `then` and no `[`** whose next token sits on a **later line** runs until a matching `end`. It is the same `if` — same value, same branches — not a second construct.

```
#language axioma/beginner
n: 5
label: if n > 0
```

```
  "positive"
else
  "non-positive"
end
println(label)
if n > 0
  println("n is positive")
else
  println("n is not positive")
end
```

```
positive
n is positive
```

Pick per `if`. A compact `then` and an `end` body cannot mean different things. Beginner allows both. What it still refuses are *other* decision tools (`match`, `cond`, the ternary `? :`) — those wait until the optional section at the end of this chapter.

5.5 `else if` for multiple branches

Sometimes there are *more than two* cases. Chain `else if`:

```
#language axioma/beginner
letter_grade: func(score) [
  if score >= 90 then "A"
  else if score >= 80 then "B"
  else if score >= 70 then "C"
  else if score >= 60 then "D"
  else "F"
]
println(letter_grade(95))
println(letter_grade(85))
println(letter_grade(72))
println(letter_grade(50))
```

Output:

```
A
B
C
F
```

The cascade is read top-to-bottom: the *first* condition that's `true` wins. So `letter_grade(95)` doesn't accidentally trigger the `score >= 80` branch — once the first condition matches, the others are skipped.

> **Convention.** Lay out the conditions in *decreasing* order of threshold (90, 80, 70, 60), and the cases read like a table from top to bottom. You'd usually align the `else if`s for visual rhythm, as above.

The … `end` dual of a cascade uses **`elseif`** (one word) so one `end` closes the whole chain. `else if` (two words) is a nested `if` and each nested `if` needs its own closer if you write it that way. For an end-form

grade table, write `elseif`:

```
#language axioma/beginner
letter_grade_end: func(score) [
  if score >= 90
    "A"
  elseif score >= 80
    "B"
  elseif score >= 70
    "C"
  elseif score >= 60
    "D"
  else
    "F"
  end
]
println(letter_grade_end(95))
println(letter_grade_end(50))
```

```
A
F
```

5.6 A worked example: shipping cost

Suppose a shipping company charges by weight:

Weight (kg)	Cost
≤ 1	$5
1 to 5	$8
5 to 20	$15
> 20	$25

Apply the Design Recipe.

```
#language axioma/beginner
# Data: weight in kilograms (Float)
# Signature: shipping: Float -> Float
# Examples:
#   shipping(0.5)  = 5      (small)
#   shipping(3)    = 8      (medium)
#   shipping(15)   = 15     (large)
#   shipping(30)   = 25     (very large)
# Template:
# shipping: func(weight) [
#   if ... weight ... then ...
#   else if ... weight ... then ...
```

```
#   else if ... weight ... then ...
#   else ...
# ]
# Body:
shipping: func(weight) [
  if weight <= 1 then 5.0
  else if weight <= 5 then 8.0
  else if weight <= 20 then 15.0
  else 25.0
]
# Tests:
println(shipping(0.5))     # 5.0
println(shipping(3))       # 8.0
println(shipping(15))      # 15.0
println(shipping(30))      # 25.0
```

Notice that we test the **boundary values** — and you should choose extras like `shipping(1)` and `shipping(5)` to confirm which side of the boundary they're on. (Our <= chose "1 kg is in the small tier," "5 kg is in the medium tier.")

5.7 A worked example: leap year

Why do leap years exist at all? A calendar year is 365 days, but the Earth takes about 365.25 days to circle the Sun — so about every four years we add an extra day (February 29th) to let the calendar catch up. That was the old Julian calendar's whole rule: every fourth year. But a day every four years is a tiny bit *too much* catching up, so our modern Gregorian calendar skips a few. (The skips keep the average year at 365.2425 days — almost exactly the Earth's pace.)

Think of leap years as a club with three door rules, where **the last rule that applies to you wins**:

- **Rule 1 — you're in.** A year divisible by 4 (`year % 4 == 0`) is a leap year.
- **Rule 2 — the exception. Except when** it's also divisible by 100 — then it's back out. (Sorry, 1900.)
- **Rule 3 — the exception to the exception. Unless** it's *also* divisible by 400 — then it's back in after all! (Welcome, 2000.)

To judge a year, find the *last* rule that talks about it: 2024 only matches Rule 1, so it's a leap year. 1900 gets caught by Rule 2 — not a leap year. 2000 goes all the way to Rule 3 — a leap year. And 2023 matches no rule at all, so it's an ordinary year.

Translate the rule into Booleans:

```
#language axioma/beginner
# Data: a year (Integer)
# Signature: is_leap: Integer -> Boolean
# Examples:
#   is_leap(2000)  = true     (divisible by 400)
#   is_leap(2024)  = true     (div by 4, not by 100)
```

```
#   is_leap(1900)  = false    (div by 100, not by 400)
#   is_leap(2023)  = false    (not div by 4)
#   is_leap(2100)  = false    (div by 100, not by 400)
# Body:
is_leap: func(year) [
  (year % 4 == 0 and year % 100 != 0) or year % 400 == 0
]
# Tests:
println(is_leap(2000))     # true
println(is_leap(2024))     # true
println(is_leap(1900))     # false
println(is_leap(2023))     # false
println(is_leap(2100))     # false
```

The body reads as: "(divisible by 4 *and* not divisible by 100) *or* divisible by 400."

> **A teaching trick.** When the natural-English specification has the words *and*, *or*, *not*, you'll usually translate them directly to Axioma's `and`, `or`, `not`. The hard part is figuring out which English connectives are which. The phrase "except when" in the leap-year rule is *and not* — "is leap" *and* "not the century exception". And "unless it's also divisible by 400" opens a second route to qualifying — an *or*.

A sneak peek: the same shape, written as rules

Axioma has a whole second style of programming — *relations* and *rules* — that you'll meet properly in Chapter 21. It's a natural fit for rules-with-exceptions, so here's a taste. Take an everyday rule with the same shape as the leap-year rule: *you can play outside on a weekend — except when it's raining — unless you've packed your raincoat.*

```
relation weekend(d)
relation rainy(d)
relation has_raincoat(d)

assert weekend("saturday")       # a dry Saturday
assert weekend("sunday")         # a rainy Sunday – raincoat packed!
assert rainy("sunday")
assert has_raincoat("sunday")
assert rainy("monday")           # rainy Monday – a school day anyway

# Door 1: a weekend day with no rain
can_play(D) whenever weekend(D) and not rainy(D)
# Door 2: rain can't stop a raincoat
can_play(D) whenever weekend(D) and rainy(D) and has_raincoat(D)

println({D | D <- can_play(D)})  # {"saturday", "sunday"}
```

Read each `whenever` line as "…is true in every case where…" — the rule connective is literally the word `whenever`, so the code says exactly what the English says. (Axioma also accepts `if` here, and Prolog readers

can write `can_play(D) :- weekend(D) and not rainy(D)` — the same connective in Prolog dress.) There are two *doors* into the `can_play` club, and a day only needs one of them to be open. Notice how the English "except when it's raining" became `and not rainy(D)` on Door 1, and the "unless" clause became a door of its own. The leap-year rule fits the same two-door shape:

```
relation year(y)

assert year(1900)
assert year(2000)
assert year(2023)
assert year(2024)
assert year(2100)

# Door 1: divisible by 4 — except when divisible by 100
leap(Y) whenever year(Y) and Y % 4 == 0 and Y % 100 != 0
# Door 2: divisible by 400 — the exception to the exception
leap(Y) whenever year(Y) and Y % 400 == 0

println({Y | Y <- leap(Y)})      # {2000, 2024}
```

This is the same logic as `is_leap` above — "(divisible by 4 *and not* divisible by 100) *or* divisible by 400" — just wearing different clothes: the *or* became two separate doors, and the *and not* became a guard on Door 1.

5.8 Some patterns

A safe divider

```
#language axioma/beginner
safe_div: func(a, b) [
  if b == 0 then "undefined"
  else a / b
]
println(safe_div(10, 2))    # 5
println(safe_div(10, 0))    # undefined
```

Notice: `safe_div` returns *either* a number *or* a string. The signature is technically `Integer Integer -> Integer | String`. We'll formalize the "or" of types in Chapters 8 and 18 — for now, just notice that the function has two kinds of result.

Absolute value

```
#language axioma/beginner
absolute: func(n) [
  if n < 0 then -n
  else n
]
println(absolute(7))      # 7
println(absolute(-3))     # 3
println(absolute(0))      # 0
```

(Axioma also has a built-in `abs` — but writing your own version is a fine exercise.)

Three-way classification

```
#language axioma/beginner
sign_label: func(n) [
  if n > 0 then "positive"
  else if n < 0 then "negative"
  else "zero"
]
println(sign_label(5))     # positive
println(sign_label(-3))    # negative
println(sign_label(0))     # zero
```

> **A pitfall to avoid.** Some words in Axioma are *reserved* — `classify`, `match`, `axiom`, `postulate`, and a few others can't be used as variable names. If you try `classify: ...`, you'll get a "reserved keyword" error. The fix: rename. The convention in this book is to add a noun-like suffix: `sign_label`, `bucket_for`, `category_of`. Reading aloud-ability tends to improve as a side effect.

Optional spelling: the ternary expression

In the full language (a separate file without the beginner pragma), `condition ? yes : no` has the same purpose as `if condition then yes else no`. Only the selected branch is evaluated.

```
denominator: 0
answer: denominator == 0 ? none : 10 / denominator
println(answer)                  # none
println(true ? "yes" : "no")     # yes
```

The unselected division does not run. Use the longer `if` form when branches need explanation or several statements; nested ternaries can hide the decision structure you are trying to teach.

5.9 Guards on function clauses

The shipping calculation can also be written as a group of equations. A **guard** is the condition after `when`: the clause is eligible only when its parameter patterns match and that condition passes.

```
#language axioma/beginner
shipping_guard(weight) when weight <= 1 = 5.0
shipping_guard(weight) when weight <= 5 = 8.0
shipping_guard(weight) when weight <= 20 = 15.0
shipping_guard(weight) = 25.0
println(shipping_guard(0.5)) # 5.0
println(shipping_guard(3))   # 8.0
println(shipping_guard(15))  # 15.0
println(shipping_guard(30))  # 25.0
```

Trace `shipping_guard(3)`: the first clause binds `weight` to 3 but its guard is false, so its body is skipped. The second guard passes and returns 8.0. The later clauses are not tried, even though `3 <= 20` is also true. This is **ordered selection**, just like the `else if` cascade: the first eligible clause wins. Use Boolean comparisons for guards in these examples. More generally a falsey guard continues to the next clause; an error while evaluating a guard propagates instead of being treated as false.

The final unguarded variable clause is a **catch-all** for this arity. Keep it last. It could also be spelled `shipping_guard(weight) otherwise = 25.0`. Without an eligible clause, a call at a supported arity returns `none`:

```
#language axioma/beginner
positive_only(n) when n > 0 = n
println(positive_only(4))  # 4
println(positive_only(-4)) # none
```

That result is absence, not a successful numeric answer and not `om`. Calling with an unsupported number of arguments is an error, not this fallback. Choose an explicit last clause when your specification requires a result for every input. A `requires` precondition is different: it rejects a call that violates a declared contract. A guard selects a clause; it does not assert that all other inputs are invalid.

Why use clauses instead of `if` inside one body? Equations put each case's condition beside its result, which is useful for piecewise mathematics and small policies. Use an `if` body when several branches share local calculations or when it makes the control flow clearer. Start a guarded group with a guarded or patterned equation; ordinary all-variable equations on their own are simple function definitions, not a way to accumulate several default clauses.

Optional: the result-first equation spelling

Axioma also accepts Miranda-style guards. They have the same selection rules, including the same fallback; they are an alternate spelling, not a new policy:

```
#language axioma/beginner
sign_equation(n) = 1, if n > 0
                 = -1, if n < 0
                 = 0, otherwise
println(sign_equation(5))  # 1
println(sign_equation(-5)) # -1
println(sign_equation(0))  # 0
```

The guard follows the result after `, if`. Each continuation begins with = and reuses the preceding function head. Repeating the full head on each line also works. Use == for equality in a condition, not =. Choose one spelling for a group so readers can scan it consistently. Prefix `when` makes the condition visible first; result-first equations can resemble a mathematical piecewise definition. Both examples here work in beginner mode.

5.10 Optional full-language choice forms

The remaining examples use full Axioma: run them in a separate file without the beginner pragma. `cond` and pattern matching are deliberately excluded from beginner mode; function-clause guards in §5.9 are not. Optional material should expand the tools you can recognize without making them prerequisites for the next beginner chapter.

`cond`: a table of conditions

`cond` makes a multi-branch decision read as a table. The conditions need not all compare the same input. Here is a small classification:

```
sign_cond: func(n) [
    cond
    | n < 0 => "negative"
    | n == 0 => "zero"
    | else => "positive"
]
println(sign_cond(-3)) # negative
println(sign_cond(0))  # zero
println(sign_cond(3))  # positive
println(cond | false => 10) # none
```

Conditions are tried in order; only the chosen body runs. `else` supplies a last fallback; `otherwise` and `_` are alternate fallback spellings for cond. No passing condition and no fallback yields `none`. Use the same boundary tests as with `else if`: changing notation does not fix an incorrectly ordered rule. We write a bar before every arm for clarity; cond permits omitting its first bar.

`match`, `case` and `switch`: inspect one value

A matcher evaluates one input and tries **patterns** against it. A literal pattern recognizes a value; a lowercase variable pattern binds it; _ ignores it while matching anything. A `when` guard can then test

what the pattern bound:

```
let pair = (1, 2)
println(match pair with
    | (x, y) when x < y => x + y
    | _ => 0) # 3
println(case pair | (x, y) when x < y => x + y | _ => 0) # 3
println(switch pair | (x, y) when x < y => x + y | _ => 0) # 3
println(case "later" | "now" => 1) # none
```

The tuple pattern extracts two fields before testing `x < y`. If the pattern fails, or its guard is false, selection continues. The first accepted arm wins. Put an unguarded _ last when you want a fallback. An ordinary unmatched matcher returns `none`; it does not mean that every intended case was handled.

These three keywords use the same matcher. `match value with` introduces arms with an optional first bar; `case value` and `switch value` require the first bar and do not take `of`, `with` or `do`. Do not put braces around the arms. We use `match … with` as the teaching default; use case or switch when that spelling reads more naturally to your audience. `switch` is not C-style fallthrough. Chapter 18 develops constructor patterns, nested data, guards and coverage checks.

Choose the form by the question

Question	Useful form
Which of two expressions should run?	`if … then … else`
Which ordered condition holds?	`else if`, or full-language `cond`
Which equation applies to these arguments?	Function clauses with `when`
What shape does this one value have?	`match`, `case` or `switch`

A condition tests a proposition; a pattern also describes and extracts data. The form should expose the reason for the choice. Test false guards, overlapping conditions, boundary values and the last fallback—not just one successful arm. In the VM, cond uses the supported conditional path; match/case/switch explicitly refuse. Use the ordinary evaluator for the pattern examples.

Optional: a capture is a name, not an equality test

A structural pattern introduces each name once. `(x, x)` is an error, rather than a request to compare two fields. Name the fields separately and add a guard:

```
println(match (2, 2) with | (a, b) when a == b => true | _ => false) # true
println(match (2, 3) with | (a, b) when a == b => true | _ => false) # false
println(match (2, 3) with | (a, b) as pair => pair)                 # (2, 3)
```

The rule also applies to pattern parameters, rest names and aliases. In `(a, b) as pair`, all three names differ. _ can repeat. Alternatives may each bind the same names, but every successful path must supply the same set. A failed pattern leaves outer bindings intact. This does not undo effects from functions

used as extractors. Logical query variables, introduced in chapter 21, use a different contract that permits repeated variables to enforce equality.

Optional: one head and sequential guard steps

Use full Axioma for this optional section. A pipe group writes the function head once and puts each condition next to its result:

```
sign_pipe(n)
  | n > 0 = 1
  | n < 0 = -1
  | otherwise = 0
println([sign_pipe(-2), sign_pipe(0), sign_pipe(3)]) # [-1, 0, 1]
```

External pipes start separate lines and align deeper than the head. A result starts on its = line; open [there for a multiline body. Unbracketed nested arms indent deeper; parentheses or bracket bodies give explicit boundaries.

A guard can also compute a name for later steps and the result. <- matches one value, rather than iterating a collection. This example matches a pair, checks both components, computes a subtotal, and checks it before returning it:

```
small_order(value)
  | (price, quantity) <- value,
    price > 0, quantity > 0,
    let subtotal = price * quantity,
    subtotal < 100 = subtotal
  | otherwise = 0
println(small_order((4, 3)))  # 12
println(small_order((40, 3))) # 0
println(small_order("bad"))   # 0
```

The same chain may follow when, with the head repeated for another clause:

```
small_order_when(value)
  when (price, quantity) <- value,
       price > 0, quantity > 0,
       let subtotal = price * quantity,
       subtotal < 100
  = subtotal
small_order_when(value) otherwise = 0
println(small_order_when((4, 3))) # 12
```

Commas run steps once, left to right. A falsey condition or mismatching pattern tries the next branch; an error propagates. New names are local to that branch. `let` creates a fresh immutable binding, just as it does in a body. Once a chain succeeds, its body is final, even if it returns `none` or errors. Effects are not undone. An ordinary expression guard still uses Axioma truthiness. Match-family `when` guards accept these same steps. These forms are evaluator-only; the VM refuses them. Informational `--infer` remains

limited to its documented pattern fragment. Semicolons keep their statement meaning.

Optional: searching several sequence segments

Use full Axioma and the evaluator for this section. A single rest pattern describes one middle portion. With several portions, the input can have more than one decomposition. Keep that search explicit:

```
let split = sequence_pattern([{segment: "left"}, {segment: "right"}])
let all = sequence_match_all([1, 2], split)
println(all.matches) # [{left: [], right: [1, 2]}, {left: [1], right: [2]}, {left: [1,
 ↪  2], right: []}]
println(all.exhausted) # true
let middle = sequence_match_first([1, 2], split, {
  guard: func(b) [len(b.left) == 1],
  extract: func(b) [b.right]
})
println(middle.values)    # [[2]]
println(middle.complete)  # true
println(middle.exhausted) # false
let limited = sequence_match_all([1, 2], split, {matches: 2})
println(limited.status) # incomplete
println(limited.reason) # match_limit
```

Shorter segments are tried first, depth-first from left to right. Each part describes a literal (`{literal: 7}`), one captured item (`{capture: "item"}`), or a segment. Segment bounds default to `min: 0, max : 1024`. Names are ASCII identifier Strings; _ discards and may repeat, while other names must differ. Patterns match a whole Array or List; segment bindings are Arrays. Typed literals distinguish a Word from a String and an Integer from a Float.

The guard decides whether to accept the bindings. The extractor then computes a result; false and none are valid results. Both receive independent copies of the bindings. Prefer pure callbacks: changing an outer variable or doing I/O is an ordinary effect that search does not undo. The pattern itself is a snapshot with read-only `.parts` and `.captures` properties returning copies.

`complete` answers whether the requested operation finished. Finding the first answer completes that request; `exhausted: false` says that alternatives may remain. Only exhaustive search can establish `no _match`. An empty `incomplete` result means the search stopped before it could decide. The result also includes `matches`, parallel `values`, `status`, `reason`, and work `steps`. Without an extractor, values are independent binding copies.

Default limits are 10,000 matcher steps and 100 results in all mode; maxima are 100,000 steps and 1000 results. One visited matcher state costs one step; callback time is separate. All mode checks one extra accepted candidate before claiming `match_limit`, so exactly filling the cap can still be complete. Patterns allow at most 256 parts and subjects at most 1024 elements. Captures and extracted values must be finite inert data within the snapshot bounds (10,000 nodes, 64 levels). See the Manual's bounded search section for the complete domain. This API does not change ordinary `match` or logical query unification. Opaque patterns are not automatically session-serialized, and the VM explicitly refuses these search builtins.

Direct calls also give `--typecheck` the context to accept heterogeneous descriptor/subject literals and op-

tion records. Ordinary Array/Map bindings keep their usual typing rules; separately bound heterogeneous records need a schema annotation. Callback bodies and computed expressions are still checked.

Optional: reusable patterns inside structured data

Ordinary `match` handles fixed nested shapes. Use sequence search for several variable-length decompositions, composing reusable components:

```
let contents = seqpat ['contains, ..before, 'lamp, ..after]
let room = seqpat ['room, name, nested(contents)]
let r = sequence_match_all(
  ['room, 'study, ['contains, 'desk, 'lamp, 'book, 'lamp]], room)
println(len(r.matches)) # 2
println(len(r.matches[1].before)) # 1
println(len(r.matches[2].before)) # 3
println(r.exhausted) # true
```

Bare names capture one element; `..name` captures a segment. The quoted word `'lamp` is a literal. `nested(contents)` consumes one outer element and searches its whole Array/List contents. Writing `['contains, ..before, 'lamp, ..after]` inside the room pattern is the inline equivalent. Use `literal(expr)` to match a value computed from a surrounding binding, instead of capturing its name. These expressions run once when the pattern is constructed.

All capture names across components must differ, except discard `_`. Search returns bindings as data and does not rebind the caller. Nested alternatives compose in depth-first, left-to-right order under one shared work budget; guards run only after a complete outer match. An incomplete search still cannot prove absence. The full pattern allows 256 total parts and 32 pattern levels. Nested searches validate directly nested Array/List containers, each up to 1024 elements, before callbacks; finite snapshot limits also remain in force.

The equivalent descriptor is `{nested: component}` or `{nested: descriptors}`. The read-only `.parts` expands components to descriptor Arrays for safe reconstruction. Keep explicit Array/List nesting distinct from an automatic search of all descendants. This optional section requires full Axioma and the evaluator. `seqpat` uses a separated same-line opening bracket, with no end-form alternative. Existing ordinary matching and logical variables are unchanged.

5.11 Exercises

Apply the Design Recipe to each. Don't skip examples.

Exercise 5.1 — `max_of_two`

Write `max_of_two(a, b)` that returns the larger of two numbers. Examples: `max_of_two(3, 7) = 7`, `max_of_two(5, 5) = 5`, `max_of_two(-2, -8) = -2`.

Exercise 5.2 — `temperature_category`

Write `temperature_category(c)` (Celsius) that returns:

- `"freezing"` when `c <= 0`
- `"cold"` when `0 < c <= 15`
- `"comfortable"` when `15 < c <= 25`
- `"hot"` when `c > 25`

Exercise 5.3 — `is_in_range`

Write `is_in_range(n, low, high)` that returns `true` if `low <= n <= high`. Optionally write the two comparisons with && instead of `and`; check both boundaries and a value outside the range.

Exercise 5.4 — `tip_smart`

Modify Exercise 3.4's `tip` so that:

- if `bill < 10`, the tip is a flat `$2`
- otherwise, it's `percent` of the bill, as before.

Show three examples.

Exercise 5.5 — `quadrant`

Write `quadrant(x, y)` that returns `"I"`, `"II"`, `"III"`, or `"IV"` based on which quadrant of the Cartesian plane `(x, y)` is in. Quadrant I is both positive; Quadrant II is `x<0, y>0`; III is both negative; IV is `x>0, y<0`. (What about points *on* the axes — `x=0` or `y=0`? *Decide before writing code.* Document your decision in the examples.)

Exercise 5.6 — open

Pick a real-world decision you've made recently — what to wear given the weather, whether to bring an umbrella, whether to take a cab — and encode it as a function with `if/else if/else`. Make sure all branches are reachable (every combination of inputs lands in *some* branch).

Exercise 5.7 — guard order and boundaries

Rewrite temperature_category from Exercise 5.2 as guarded equations. Test -5, 0, 15, 25 and 30. Explain why the catch-all must be last. What changes if the comfortable-temperature clause is put before the freezing clause? Then remove the catch-all in a fresh file and find an input for which no clause applies.

Exercise 5.8 — optional cond policy

In full-language mode, rewrite the umbrella policy in Exercise 5.6 using cond. Test both overlapping conditions and the fallback. Explain how it agrees with the original else-if order. Show one unmatched cond without a fallback.

Exercise 5.9 — optional pair patterns

Use case to return “ascending”, “equal” or “descending” for a pair of numbers. Use a tuple pattern and when guards; add a fallback for a value that is not a pair. Test (1,2), (2,2), (3,2) and a String. Rewrite the header with match and switch, keeping the results unchanged.

Exercise 5.10 — alternate guarded equations

Write the temperature function in result-first `= body, if condition` form, using elided heads and `, otherwise`. Compare its boundaries with Exercise5.7. Explain the difference between a false guard, no matching clause and a guard that raises an error. A precondition rejecting a call is not another guard arm.

Exercise 5.11 — optional guard bindings

Extend `small_order` so that `subtotal == 100` succeeds. Test `(25,4)`, `(25,5)` and a non-pair. Then place `let subtotal = 999` in a later failing branch: explain why it cannot change another branch’s result or an outer binding.

Exercise 5.12 — optional capture equality

Write a matcher that returns true only when the two components of a pair agree. Test `(4, 4)` and `(4, 5)`. Explain why `(x, x)` and `(a, b) as a` are invalid, and why `(a, b) as pair` is valid. Use full-language mode for this exercise.

Exercise 5.13 — optional bounded sequence search

Divide `[1, 2, 3]` into two nonempty segments using `sequence_pattern`. Extract the two lengths for every match. Then cap results at one and explain why its status differs from an exhaustive search with no answers. What do `complete` and `exhausted` mean after a successful first-result search? Use full-language mode and the evaluator.

Exercise 5.14 — optional nested event search

Use a reusable `seqpat ['login, person]` component to find every login event inside `['events, [['ping], ['login, 'ada], ['ping], ['login, 'bob]]]`. Capture the earlier and later events. Print the number of matches and the length of each earlier prefix. Repeat with a result cap of one. Explain why search must not call that result exhaustive. Use the evaluator.

5.12 Reflection

The basic decision tools in this chapter:

Booleans — a tiny type with exactly two values — let us *capture the answer to a yes/no question*. Every comparison operator (<, >, ==, etc.) produces a Boolean.

`if ... then ... else ...` — the main decision form. The whole construct is an *expression* (returns a value) that picks between two alternatives. Chain it with `else if` when there are more than two

cases. The `… end` dual (no `then`, later-line body, a closing `end`) is the same `if`; `elseif` is the one-`end` cascade. Guards put ordered conditions on function clauses; the optional section adds cond tables and the match/case/switch family for inspecting data.

You now have everything you need to write Part I's worth of single-value functions. The next chapters open the door to *structures* — collections of values, like lists and trees — and that's where the language earns its keep.

In **Chapter 6** we'll meet **arrays** (the Axioma word for what HtDP calls *lists*) and start writing functions that work on collections, not just on single numbers.

Looking ahead — what you'll be able to do by Chapter 9:

- Define your own data shapes with Concepts
- Process arbitrarily-long lists with recursion
- Build a small *interpreter* for arithmetic expressions, written in Axioma, processing trees of `+`, `-`, `*`, `/` nodes

What we used from Axioma

- Boolean literals: `true`, `false`
- Comparison operators: `==`, `!=`, `<`, `>`, `<=`, `>=`
- Boolean operators: `and`, `or`, `not`, `xor`
- `if <bool> then <expr> else <expr>` as an expression
- The `if … end` dual, and `elseif` for a one-`end` cascade
- Multi-branch `else if` cascades and ordered function-clause guards
- Optional full-language `cond`, `match`, `case` and `switch`
- The list of reserved-keyword names that aren't usable as variables (we hit `classify`)

Solutions to selected exercises

Solutions to selected exercises: Chapter 5 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch05_solutions.md.)*

Chapter 6 · Arrays as Lists

What this chapter is. Up to here, every function we wrote took a *fixed* number of inputs: one temperature, two points, three arguments at most. Now we'll handle *many at once* — a list of grades, a sequence of measurements, all the words in a sentence. The data shape we'll use is called an **array**.

6.1 Why a new shape?

Suppose you want a function `sum_three(a, b, c)` that adds three numbers:

```
sum_three: func(a, b, c) [a + b + c]
println(sum_three(1, 2, 3))   # 6
```

Easy. But what about `sum_five`? Five parameters. `sum_ten`? Ten parameters. *Sum of however-many numbers the user gives you?* You can't write `sum_n_for_any_n` with this style — the parameter list has to be fixed at the time you write the function.

This is the wall that **arbitrary-size data** breaks through. We package "many things" into a single value — an array — and pass *one* parameter that happens to be the array.

```
total: func(numbers) [
  ...
]
println(total([1, 2, 3]))          # 6
println(total([1, 2, 3, 4, 5]))    # 15
println(total([42]))               # 42
println(total([]))                 # 0
```

That `...` body is what Chapter 7 will fill in. This chapter is about the *shape* — how to make arrays, how to look inside them, how to take them apart.

6.2 Creating an array

The literal form is comma-separated values inside square brackets:

```
axioma> [10, 20, 30]
[10, 20, 30]
axioma> [true, false, true]
[true, false, true]
```

```
axioma> ["red", "green", "blue"]
["red", "green", "blue"]
```

Axioma arrays are **heterogeneous** — different element types are allowed:

```
axioma> [1, "hello", true]
[1, "hello", true]
```

You usually *don't* mix types in practice (it makes the function that processes the array more painful), but ordinary evaluator execution permits it when no explicit element constraint has been attached. See the optional runtime annotation section below.

The empty array is `[]`:

```
axioma> []
[]
axioma> len([])
0
```

6.3 Looking inside: `len`, `first`, `rest`

Three primitive operations let us inspect an array. They're enough for everything in this part of the book.

Operation	What it does	Example
`len(arr)`	how many elements	`len([10, 20, 30])` → `3`
`first(arr)`	the first element	`first([10, 20, 30])` → `10`
`rest(arr)`	everything *except* the first	`rest([10, 20, 30])` → `[20, 30]`

```
axioma> nums: [10, 20, 30, 40]
axioma> len(nums)
4
axioma> first(nums)
10
axioma> rest(nums)
[20, 30, 40]
axioma> first(rest(nums))
20
axioma> head(nums)
10
axioma> tail(nums)
[20, 30, 40]
axioma> head(nums, 2)
[10, 20]
axioma> tail(nums, 2)
[30, 40]
```

The pair `first` / `rest` is the same idea as Lisp's `car` / `cdr` (if you've heard those terms). Axioma also offers `head` / `tail` and `hd` / `tl` as exact aliases, including the optional count. `tail` means everything after the first element, while `last` returns the final element. `head([])` raises an error; `tail([])` returns `[]`. The separate `ast_head` function inspects syntax values.

Following HtDP, we will use `cons`, `first`, and `rest` to teach recursion.

Notice that `first(nums)` returns the element *itself* (the integer `10`), while `rest(nums)` returns a *smaller array*. That asymmetry is exactly what makes recursion on arrays work — `rest(rest(...))` keeps shrinking the array until eventually it's empty.

6.4 Indexing: 1-based

You can also fetch elements by position, using square brackets:

```
axioma> nums[1]
10
axioma> nums[2]
20
axioma> nums[4]
40
```

Axioma indexes from 1, not from 0. This is the same convention as Lua, Julia, Pascal, COBOL, and standard mathematical notation. Most other languages (C, Python, JavaScript) use 0. If you're coming from one of those, the first few weeks will feel uncomfortable; after that it stops mattering.

Out-of-bounds access is an error, not a silent surprise:

```
axioma> nums[0]
ERROR: index out of bounds: 0 (indices start at 1)
axioma> nums[99]
ERROR: index out of bounds: 99
```

We'll prefer `first` / `rest` over `nums[i]` for most of Part II, because *recursion on the structure* — not on an index counter — is the design recipe's natural fit.

6.5 Putting arrays together

Two arrays glue together with `+`:

```
axioma> [1, 2, 3] + [4, 5]
[1, 2, 3, 4, 5]
axioma> [] + [99]
[99]
axioma> [10] + []
[10]
```

You can also use `push(arr, x)` to add one element at the end:

```
axioma> push([1, 2, 3], 4)
[1, 2, 3, 4]
```

`push` grows the original Array in place and returns that same Array. This is **mutation**, a topic we study in Chapter 16. If you want a new Array while preserving the input, use concatenation instead.

```
axioma> original: [1, 2, 3]
axioma> push(original, 99)
[1, 2, 3, 99]
axioma> original
```

```
[1, 2, 3, 99]
axioma> combined: original + [100]
axioma> original
[1, 2, 3, 99]
axioma> combined
[1, 2, 3, 99, 100]
```

6.6 Brackets do two jobs: arrays and blocks

In ordinary positions — bindings, arguments, +-chains — the single-element form `[x]` means exactly what you'd guess: a one-element array.

```
axioma> a: 5
axioma> single: [a]
axioma> single
[5]
axioma> type(single)
Array
axioma> [1] + [2, 3]
[1, 2, 3]
```

But brackets in Axioma do two jobs: they hold array literals *and* they delimit **blocks of code** — you've been using the second job all along, as function bodies (`func(x) [x * 2]`). In positions where the parser expects a block — an `if`/`else` **branch**, a function **body** — a bare `[x]` is read as a one-statement block whose value is `x`, not as an array (the block reading is what lets a branch hold several statements):

```
axioma> f: func(x) [ if x then ["yes"] else ["no"] ]
axioma> f(true)
"yes"
axioma> type(f(true))
String
```

The branch produced the *string* `"yes"`, because `["yes"]` there was a block. The spelling that always means "one-element array", in every position, is a **trailing comma**:

```
axioma> g: func(x) [ if x then ["yes",] else ["no",] ]
axioma> g(true)
["yes"]
axioma> type(g(true))
Array
```

`[x,]` is the canonical singleton spelling. (You may still meet the older workaround `push([], x)` in the wild — it works, but `[x,]` has replaced it.)

"In every position" is worth checking rather than taking on trust, because a function **body** is a block just as much as a branch is:

```
axioma> h: func(x) [x,]
axioma> h(3)
[3]
axioma> type(h(3))
Array
```

One gotcha while you are here. A **bare comma run** inside a body is *not* an array — it comes back as a

tuple:

```
axioma> k: func(x) [x, 1]
axioma> k(3)
(3, 1)
axioma> type(k(3))
Tuple
```

That trips people who expect `[x, 1]` to mean one thing everywhere. It doesn't, and the reason is the same block-vs-array split as above: a body is a sequence of **statements**, so its commas build a sequence of values, while an `if` branch and the right-hand side of a plain binding are **expression** positions, where commas separate array elements. (In a body that sequence is doing real work: it is Axioma's Pop-11-style postfix notation, where `func f() [2, 3, +]` returns `5`. With no operator in it, as here, it simply hands back the values.)

The fix is the one you already know. Add the trailing comma and a body gives you the array too:

```
axioma> m: func(x) [x, 1,]
axioma> m(3)
[3, 1]
```

So the rule to carry forward is short: **if you mean an array, write the trailing comma.** It is the one spelling that never changes meaning with position.

6.7 Worked example: top three

Suppose `scores` is an array of test scores. We want the *first three* — say, to highlight in a podium graphic.

1. Data. An array of at least three numbers.

```
scores: [88, 72, 95, 60, 81]
```

2. Signature.

```
# top_three: Array -> Array
```

3. Examples.

```
# top_three([88, 72, 95, 60, 81])  = [88, 72, 95]
# top_three([10, 20, 30])           = [10, 20, 30]
```

4. Template. Pluck three elements out by index.

5. Body.

```
top_three: func(arr) [
  [arr[1], arr[2], arr[3]]
]
```

6. Tests.

```
println(top_three([88, 72, 95, 60, 81]))    # [88, 72, 95]
println(top_three([10, 20, 30]))            # [10, 20, 30]
```

This works — but it has an obvious shortcoming. What if the array has fewer than three elements? Then `arr[3]` is out of bounds and the program crashes. A proper version would check `len(arr) >= 3` first and return something sensible otherwise. That's the kind of defensive coding Chapter 7 will set up cleanly.

6.8 The two cases: empty vs. non-empty

Every function that processes an array has to handle two cases:

1. The array is **empty**: `len(arr) == 0` (or equivalently `arr == []`).
2. The array is **non-empty**: there's at least one element to look at.

Why? Because `first(arr)` and `rest(arr)` *require* a non-empty array. If you call them on `[]` they fail. So the very first thing your function does must be: "Is the array empty?"

```
describe: func(arr) [
  if len(arr) == 0 then "empty"
  else "non-empty"
]
println(describe([]))           # empty
println(describe([1, 2, 3]))    # non-empty
```

That two-case structure is the **template** for almost every list-processing function in Part II. Chapter 7 will turn it into a recipe.

6.9 Reflection

We've widened the type system from *one value per input* to *a container with many values*. The cost is small — three new operations to memorize (`len`, `first`, `rest`) plus one syntax (brackets) — and the payoff is enormous: from now on, no function we write has a hard-coded number of inputs.

The pattern for Chapter 7 is already visible:

- *Empty array?* Return some base-case answer.
- *Non-empty?* Look at `first(arr)`, recur on `rest(arr)`, combine the two.

That's it. That's recursion on lists. The whole next chapter is worked examples of that one shape.

6.10 Ranges describe values; collect gathers them

A range stores a rule for producing values. An array stores the values. This matters when the sequence is large, or you only need its beginning.

```
println(collect(1..5))                # [1, 2, 3, 4, 5]
println(collect(5..1 by 2))           # [5, 3, 1]
println(collect(0.0..1.0 by 0.25))    # [0.0, 0.25, 0.5, 0.75, 1.0]
println(collect(0.0..<0.3 by 0.1))    # [0.0, 0.1, 0.2]
println(collect(5..1..2))             # []
println(first(1.., 4))               # [1, 2, 3, 4]
```

`..` includes an endpoint when the step reaches it; `..<` excludes it. With `by`, the step is a positive magnitude and the bounds determine direction. In `start..end..step`, the step is signed; a direction that does not approach the end gives an empty range. A zero step is an error. Float ranges use a rounding-aware sample grid, rather than promising exact decimal arithmetic. Ordinary scalar Float indexing remains distinct from slicing: an array can accept a whole-valued scalar index such as `1.0`, but a Float range is not an array slice index.

`collect(range)` returns an Array, as does `array(range)`. `collect` emphasizes consuming an iterable; when given a one-shot generator it exhausts it. Constructing or indexing a large range stays compact, while collecting it allocates storage and is subject to a materialization limit. An open-ended range cannot be collected in full: request a bounded prefix.

Try it. Predict the difference between `collect(1..8 by 3)` and `collect(1..<7 by 3)`, then run both. Which endpoint can appear?

6.11 Numerical extension: matrices are two-dimensional

A nested Array and a Matrix solve different problems. Use a Matrix when rows and columns are part of the mathematical meaning. Axioma preserves Integer and Rational matrix elements for exact arithmetic; it does not silently turn every entry into a Float. This optional preview includes a `for` loop: use a separate full-language file, without the beginner pragma.

```
m: matrix([[1, 2, 3], [4, 5, 6]])
println(shape(m))                  # [2, 3]
println(m[2, 3])                   # 6
println(m[1, 2] is Integer)        # true
println(m[:])                      # [1, 2, 3, 4, 5, 6]
println(shape(m[1, :]))            # [1, 3]
println(shape(m[:, 2]))            # [2, 1]
for cell in m [println(cell)]      # prints 1 through 6, in that order
fractions: matrix([[1/3, 1/6]])
println(fractions[1, 1] + fractions[1, 2])  # 1/2
println(float(m)[1, 1] is Float)    # true
```

Indices start at one. Linear access and direct iteration use **row-major** order: finish one row before starting the next. Row and column slices remain matrices, retaining their axes. Use `m[:]` when a flat Array is the intended result. `len(m)` is deliberately refused: ask `shape(m)` for dimensions, or `len(m[:])` when you specifically mean the count of flattened cells. The latter materializes an array.

Explicit `float(m)` opts into approximate cells for a numerical algorithm. An exact representation does not

imply that every linear-algebra routine has an exact implementation. Tensor operations and broadcasting build on the same need to specify axes; Chapter 10 introduces that next layer.

Try it. Build a two-by-two matrix containing `1/3`. Compare the type of that cell before and after `float(m)`, and explain why the conversion is a choice rather than a formatting change.

Exercises

> Reminder: every solution file in this chapter should start with `#language axioma/beginner` so you stay in the safe subset.

Exercise 6.1 — `count_zeros`

Write a function `count_zeros(arr)` that returns how many elements of `arr` equal `0`. Use `if`, comparison, and the operations from this chapter. **Don't** use recursion yet — write a version that works for arrays of length 0, 1, 2, or 3 only, by explicit case-analysis on `len(arr)`.

Test cases:

- `count_zeros([])` → `0`
- `count_zeros([0])` → `1`
- `count_zeros([5])` → `0`
- `count_zeros([0, 0, 5])` → `2`

Exercise 6.2 — `is_short`

Write `is_short(arr)` that returns `true` when `arr` has fewer than 3 elements, `false` otherwise.

- `is_short([])` → `true`
- `is_short([1, 2])` → `true`
- `is_short([1, 2, 3])` → `false`

Exercise 6.3 — `swap_first_last`

Given an array of *exactly* 3 elements, return a new array with the first and last swapped.

- `swap_first_last([1, 2, 3])` → `[3, 2, 1]`
- `swap_first_last(["a", "b", "c"])` → `["c", "b", "a"]`

(Use indexing. Don't worry about the general case.)

Exercise 6.4 — `second_or_zero`

Return the second element of the array, or `0` if the array has fewer than 2 elements.

- `second_or_zero([])` → `0`
- `second_or_zero([10])` → `0`
- `second_or_zero([10, 20])` → `20`
- `second_or_zero([10, 20, 30])` → `20`

Exercise 6.5 — `glue_with_separator`

Given two arrays `a` and `b`, return a single array that has all of `a`, then a separator element, then all of `b`. The separator value is up to you (use a string like `"-"`).

- `glue("abc", [1, 2], [3, 4])` … design your own signature.

(This exercise is intentionally a little open. Pick a separator, write down the signature first, then implement.)

Exercise 6.6 (open) — three things to know about an array

Pick *any* array, say `[14, 7, 22, 7, 100, 7]`. Write three functions that each compute one statistic from it:

- the first element
- the last element
- the length

The point is to wire up three tiny functions and have them all operate on the same array. Bonus: write a fourth function `describe(arr)` that returns a `String` summarizing all three.

Optional: an Array with an explicit element contract

This section uses the full language. `Array of Integer` checks the initial values and keeps that constraint on the shared Array. An alias sees valid changes and cannot insert a String. Even aliases captured before annotation must obey it.

```
scores :: Array of Integer: [1, 2]
other_scores: scores
other_scores[1] = 3
push(other_scores, 4)
println(scores)                      # [3, 2, 4]
```

The next example intentionally fails in the ordinary evaluator:

```
scores :: Array of Integer: [1, 2]
other_scores: scores
push(other_scores, "bad")             # runtime error; scores stays [1, 2]
```

`extend` and slice replacement check the whole batch before changing the Array. Nested `Array of (Array of Integer)` annotations constrain inner Arrays as well; a broader annotation does not erase an earlier constraint. Arrays without an explicit element annotation remain heterogeneous at runtime. Static warnings, `--typecheck`, and `--infer` are separate from this runtime guarantee. Full VM parity is deferred: applicable element annotations explicitly refuse under `--vm`. This strengthens the former initial-only runtime check for annotated Arrays.

Exercise 6.7 — aliases and rejected growth (optional)

In the valid example, predict `scores` after `extend(other_scores, [5, 6])`. Then explain why `extend(other_scores, [5, "bad"])` must change neither Array contents nor any incoming nested contracts. Would an unannotated Array accept a String in ordinary evaluator execution?

Next chapter — recursion turns "the first element plus a smaller problem" into a complete algorithm.

Solutions to selected exercises: Chapter 6 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch06_solutions.md.)*

Chapter 7 · Recursion

What this chapter is. The most important chapter in Part II. Recursion is not a trick and it is not "loops wearing a costume." It is the way you write a function whose input can be *any size*, once you can name a smallest case and a strictly smaller case. Master the two-case pattern here and the rest of the book — trees, mutually recursive data, generative algorithms, even logic rules — is the same idea on a new shape.

7.1 A definition that uses the thing being defined

A **to-do list** is either empty, or it is one item plus a to-do list. That sentence is legal. It is also slightly startling the first time you take it seriously, because the word "to-do list" appears on both sides. That is not a bug in the sentence. It is the sentence telling you the *shape of the data*.

Open a to-do list.

- If there is nothing on it, you are done.
- If there is something on it, do the first item, then you are left with — a to-do list. Smaller. Same kind of thing.

A function that *processes* a to-do list has the same two sentences inside it. That is recursion: a function that solves a problem by solving a strictly smaller problem of the **same kind**, then combining. The empty list is how the process stops. The smaller list is how it continues.

Three questions, in this order, write every recursive function you will need for the next several chapters:

1. **When do I stop?** (the **base case** — the smallest input)
2. **What is the smaller problem?** (the **recursive case** — a strictly smaller input of the same type)
3. **How do I combine this piece with the answer to the smaller problem?** (the **combining rule**)

Write the base case first. Always. A recursive function with no base case is a picture frame that contains a picture frame that contains a picture frame, forever. Axioma will not hang your machine; it will stop you with a clear error. But the function is still wrong. The base case is the off switch.

A useful mental habit, borrowed from the way experienced functional programmers actually work: **assume the function already works for every smaller input**, and only write what *this* call has to add. You are not "thinking about all the calls at once." You are writing one step, plus the promise that the same step works one size down.

7.2 How Axioma spells a recursive function

The beginner form is the same `func` you have been writing since Chapter 3. There is no extra keyword. If the body mentions the name being defined, the function is recursive.

```
#language axioma/beginner
sum_list: func(arr) [
  if len(arr) == 0 then 0
  else first(arr) + sum_list(rest(arr))
]
println(sum_list([1, 2, 3]))   # 6
```

Three things that look like they should work, and one of them does not.

Spelling	Recurses?	Notes
`sum_list: func(arr) [ … sum_list(…) … ]`	yes	Canonical. Use this.
`sum_list = (arr) => [ … sum_list(…) … ]`	yes	Same idea; arrow form. Chapter 11.
`func sum_list([]) [0]` plus a second clause	yes	Pattern clauses; see §7.12.
`let sum_list = func(arr) [ … sum_list(…) … ]`	**no**	`let` is a fresh cell. The body closes over the *old* environment, which does not yet contain `sum_list`.

Python, JavaScript, and Java all let a function call itself by name with no extra keyword — `def f(…): … f(…)` , `function f(…) { f(…); }`, `int f(…) { return f(…); }`. Axioma's `func` and `=` work the same way. `let` is the exception: it is a fresh cell whose body closes over the environment from *before* the binding, so `let f = func() [f()]` does **not** see `f`. There is no `rec` keyword to opt in. Use `:` or `=` (find-or-update) or a `func name` clause when the body has to call itself.

Bodies are square brackets, `[ … ]`. Curly braces `{ … }` are sets and dictionaries, not function bodies. That is a different pitfall, and it is just as loud.

Axioma indexes from **1**, not from 0. `s[1]` is the first character of a string; `s[len(s)]` and `s[-1]` are both the last. Slices are **inclusive** on both ends: `s[2: n - 1]` drops the first character and the last. That is the same convention as Julia (`s[2:end-1]`) and Lua (`s:sub(2, #s-1)`). Python (`s[1:-1]`) and JavaScript (`s.slice(1, s.length - 1)`) are 0-based and drop the end index. Write the Axioma bounds on paper once before you type them.

7.3 The pattern, once

Every function that processes an array follows the same template. It is the to-do-list sentence, compiled:

```
f: func(arr) [
  if len(arr) == 0 then
    <some base-case answer>
  else
    <combine first(arr) with f(rest(arr))>
]
```

Two branches: **empty** (the base case) and **non-empty** (the recursive step). The empty branch returns a value *immediately* — nothing left to recur on. The non-empty branch peels off the first element, recurs on the rest, and combines.

That is it. The rest of this chapter instantiates that template — for sum, count, last, membership, max, palindromes, early-stop scans, and a helper that walks by index. Each time, the only things that change are "what to return when empty" and "how to combine this piece with the recursive result."

This is what *How to Design Programs* calls **structural recursion**: the recursion follows the shape of the data. The data is "either empty, or a first element plus a rest"; the function is "either return a base value, or combine first with the recursive result on rest." You do not invent the decomposition. The data definition already did.

7.4 First worked example: `sum_list`

Problem. Given an array of numbers, return their sum.

1. Data. An array of numbers (which may be empty).

2. Signature.

```
# sum_list: Array -> Number
```

3. Examples.

```
# sum_list([])          = 0
# sum_list([42])        = 42
# sum_list([1, 2, 3])   = 6
# sum_list([10, -3, 5]) = 12
```

The empty example is not optional. It *is* the base case, written as a test.

4. Template. The empty/non-empty split:

```
sum_list: func(arr) [
  if len(arr) == 0 then
    ...
  else
    ... first(arr) ... sum_list(rest(arr)) ...
]
```

5. Body. Fill in the holes:

- *Empty:* the sum of no numbers is `0`.
- *Non-empty:* `first(arr)` is one number, `sum_list(rest(arr))` is the sum of the rest. Add them.

```
#language axioma/beginner
sum_list: func(arr) [
  if len(arr) == 0 then 0
  else first(arr) + sum_list(rest(arr))
]

println(sum_list([]))           # 0
println(sum_list([42]))         # 42
println(sum_list([1, 2, 3]))    # 6
println(sum_list([10, -3, 5]))  # 12
```

All four match. Move on with confidence.

How it actually runs

Trace `sum_list([1, 2, 3])` by substituting, the way you substitute in algebra. Do not try to hold the whole call stack in your head as a movie. Replace one call at a time:

```
sum_list([1, 2, 3])
  = 1 + sum_list([2, 3])
  = 1 + (2 + sum_list([3]))
  = 1 + (2 + (3 + sum_list([])))
  = 1 + (2 + (3 + 0))
  = 1 + (2 + 3)
  = 1 + 5
  = 6
```

Each call peels one element. After three peels the array is empty — the base case fires — and the pending additions run on the way back. Those pending `1 +`, `2 +`, `3 +` are real work waiting on the stack. We will come back to them in §7.11.

If a recursive function you wrote is "almost right," this kind of trace is the fastest debugger you have. Pick the smallest example that fails. Substitute by hand until the substitution disagrees with what you meant. The bug is on that line.

7.5 Second worked example: `count_list`

Problem. Given an array, return the number of elements. (`len` already does this — we reimplement it so the *pattern* is the only thing in view.)

```
#language axioma/beginner
# count_list: Array -> Integer
# count_list([])          = 0
# count_list([42])        = 1
```

```
# count_list([1, 2, 3])   = 3

count_list: func(arr) [
  if len(arr) == 0 then 0
  else 1 + count_list(rest(arr))
]

println(count_list([]))         # 0
println(count_list([42]))       # 1
println(count_list([1, 2, 3]))  # 3
```

The body changed from `first(arr) + recur` to `1 + recur`. We did not look at the element itself — we just counted it. The *shape* of the function is identical to `sum_list`.

That is the whole skill. Name the combining rule; the template does the rest.

7.6 Third worked example: `last_element`

Problem. Return the last element of an array. The empty case has no last element. Use `none` — Axioma's absence value, the one you get from a total read that found nothing. (`om` is a different bottom, an undetermined value. They are not interchangeable. Empty-array "there isn't one" is `none`.)

```
#language axioma/beginner
# last_element: Array -> Any-or-none
# last_element([])          = none
# last_element([42])        = 42
# last_element([1, 2, 3])   = 3

last_element: func(arr) [
  if len(arr) == 0 then none
  else if len(arr) == 1 then first(arr)
  else last_element(rest(arr))
]

println(last_element([]))          # none
println(last_element([42]))        # 42
println(last_element([1, 2, 3]))   # 3
```

This version has *three* branches:

- empty → `none`
- exactly one element → that element
- otherwise → recur on the rest

The two-case template grew a third case because the data has *two flavors of non-empty* — singleton and longer. That is a normal extension when the problem demands it. The recursive call still receives a strictly smaller array, so we still terminate.

(Axioma also has a built-in `last(arr)`. The exercise is in seeing the pattern, not in replacing the primitive.)

7.7 Fourth worked example: `has_value`

Problem. Does the array contain a given value?

(`contains` is a reserved word — a builtin that tests membership on several collection types. We cannot bind that name, so the recursive version gets its own.)

```
#language axioma/beginner
# has_value: Array Any -> Boolean
# has_value([], 5)             = false
# has_value([1, 2, 3], 2)      = true
# has_value([1, 2, 3], 99)     = false
# has_value(["a", "b"], "b")   = true

has_value: func(arr, target) [
  if len(arr) == 0 then false
  else if first(arr) == target then true
  else has_value(rest(arr), target)
]

println(has_value([], 5))             # false
println(has_value([1, 2, 3], 2))      # true
println(has_value([1, 2, 3], 99))     # false
println(has_value(["a", "b"], "b"))   # true
```

Look at the structure:

- *Empty* → `false` (nothing here matches anything).
- *First element matches* → `true`. **Stop.** Do not look at the rest.
- *Otherwise* → recurse on the rest, return whatever that says.

That middle branch is **early exit**. Recursion is not obliged to visit every element. `map` and `reduce` (Chapter 10) always do. When the problem is "walk until you know the answer," recursion — or a loop with `break`, later — is the honest shape. `has_value([1, 2, 3], 1)` never even looks at 2 or 3.

Boolean-returning functions on lists almost always have this shape: an empty-array default, a test on the front, and a recur-on-the-rest for everything else.

7.8 Fifth worked example: `largest`

Problem. Return the largest number in an array. Like `last_element`, the empty case has no sensible numeric answer — `none`.

```
#language axioma/beginner
# largest: Array -> Number-or-none
# largest([])              = none
# largest([7])             = 7
# largest([1, 99, 3])      = 99
# largest([-5, -10, -2])   = -2

largest: func(arr) [
  if len(arr) == 0 then none
  else if len(arr) == 1 then first(arr)
  else [
    rest_largest: largest(rest(arr))
    if first(arr) > rest_largest then first(arr) else rest_largest
  ]
]

println(largest([]))              # none
println(largest([7]))             # 7
println(largest([1, 99, 3]))      # 99
println(largest([-5, -10, -2]))   # -2
```

Three new tricks here:

1. **Local binding inside a function body.** `rest_largest: largest(rest(arr))` names the recursive result so we can refer to it twice without calling `largest` twice (which would re-compute the whole tail). Chapter 12 will make this a habit; for now it is a peek at *local definitions*.
2. **The block expression `[ ... ]` as an `else` branch.** When an `else` arm has more than one thing in it (here: a local binding *and* an `if`-expression), wrap them in `[ ... ]`. Same block syntax as a function body — a sequence whose value is the last expression. Without the block, the parser does not know where the multi-line `else` is supposed to end.
3. **The two-arm comparison.** Once we have `first(arr)` and `rest_largest`, the answer is whichever is larger. A simple `if`. Chapter 5; now tucked into a list-recursion.

A combining rule you will meet constantly: `max(this, recur)`. `min`, `and`, `or`, `+`, `+` on arrays — all the same skeleton.

7.9 Strings: a palindrome

Lists are not the only thing that shrinks. A **string** is either short enough to be obviously the same forwards and backwards, or it is a first character, a last character, and a strictly shorter string in the middle.

Problem. Is `s` a palindrome — the same backwards as forwards? `civic` is. `cynic` is not. `radar` is. A single letter is. The empty string is (vacuously: there is nothing to mismatch).

Walk it the way you actually check a word on paper:

- `radar` — first and last match (`r`/`r`). Look at `ada`.

- `ada` — first and last match (`a`/`a`). Look at `d`.
- `d` — one letter. Yes.

And the failing case:

- `cynic` — first and last match (`c`/`c`). Look at `yni`.
- `yni` — `y` and `i` differ. No. Stop. Do not look at `n`.

That *is* the three questions:

1. Stop when the string has length 0 or 1 — those are palindromes.
2. The smaller problem is the string with the first and last characters dropped.
3. Combine: if the ends differ, the answer is `false`; if they match, the answer *is* the answer for the middle.

```
#language axioma/beginner
# is_palindrome: String -> Boolean
# is_palindrome("")          = true
# is_palindrome("a")         = true
# is_palindrome("civic")     = true
# is_palindrome("cynic")     = false
# is_palindrome("radar")     = true
# is_palindrome("runner")    = false

is_palindrome: func(s) [
  n: length(s)
  if n <= 1 then true
  else if s[1] != s[n] then false
  else is_palindrome(s[2: n - 1])
]

println(is_palindrome(""))         # true
println(is_palindrome("a"))        # true
println(is_palindrome("civic"))    # true
println(is_palindrome("cynic"))    # false
println(is_palindrome("radar"))    # true
println(is_palindrome("runner"))   # false
```

Index arithmetic, slowly, because this is where a 0-based habit (Python, JavaScript, Java, C) disagrees with Axioma's 1-based inclusive slices.

- Axioma strings are **1-indexed.** `s[1]` is the first character; `s[n]` is the last. (`s[-1]` is also the last. Negative indices count from the end, the way a single `xs[-1]` already did in Chapter 6.)
- The slice `s[2: n - 1]` is **inclusive** on both ends. For `civic` (`n` is 5) that is `s[2: 4]`, which is `"ivi"`. Then `"v"`. Then we stop. Julia writes the same window `s[2:end-1]`; Lua writes `s:sub(2, #s-1)`.
- Python's `s[1:-1]` and JavaScript's `s.slice(1, s.length - 1)` land on the same characters — `"ivi"` — but the numbers are 0-based and the end is *exclusive*. Copy those numbers into Axioma and you drop the wrong characters. A slice that overshoots becomes `""`, empty is a palindrome, and `"cynic"` answers `true`: a wrong answer at exit 0.

`rest` does **not** work on strings. `first("radar")` happens to return `"r"`, but `rest("radar")` is an error: `rest` wants an array, tuple, set, or list. On a string, drop the first character with `s[2:]` (or `s[2..]`), and drop both ends with `s[2: n - 1]` or the equivalent `s[2..-2]`.

A logged trace

When the substitution-by-hand feels too quiet, print the argument as you go. This is the same function with a narrator:

```
#language axioma/beginner
is_palindrome_logged: func(s) [
  n: length(s)
  println("seeing if '" + s + "'")
  if n <= 1 then [
    println("length " + n + " — yes")
    true
  ] else if s[1] != s[n] then [
    println("mismatch " + s[1] + " vs " + s[n] + " — no")
    false
  ] else
    is_palindrome_logged(s[2: n - 1])
]

is_palindrome_logged("civic")
is_palindrome_logged("cynic")
```

```
seeing if 'civic'
seeing if 'ivi'
seeing if 'v'
length 1 — yes
true
seeing if 'cynic'
seeing if 'yni'
mismatch y vs i — no
false
```

Take the `println`s out when you are done. They are a diagnostic, not part of the answer.

Same algorithm, no copying

Each slice builds a new string. For a 50,000-character palindrome that is a lot of copying for a question that only needed two characters per step. The fix is a **helper** that carries two indices and never allocates a middle:

```
#language axioma/beginner
is_palindrome2: func(s) [
  go: func(start, finish) [
    if start >= finish then true
    else if s[start] != s[finish] then false
```

```
    else go(start + 1, finish - 1)
  ]
  go(1, length(s))
]

println(is_palindrome2("civic"))      # true
println(is_palindrome2("cynic"))      # false
println(is_palindrome2("redivider"))  # true
```

The public function takes one argument. The inner `go` takes the moving bounds. Callers never see `start` and `finish`. That wrapper-plus-helper shape is how you keep a simple interface when the recursion needs extra state. Chapter 12 treats helpers as a design step; you just used one because the problem asked for it.

Notice that `go` is bound with `:` inside the body, so it *can* call itself. `let go = func(start, finish) [ … go(…) … ]` would not.

(There is also the one-liner `s == reverse(s)`. Use it in production. We are not practicing `reverse` right now; we are practicing *shrinking an input from both ends*.)

7.10 Stop in the middle: `sum_until_negative`

`map` and `reduce` visit every element. Some problems should not. Sum the numbers in an array **up to but not including** the first negative. If there is no negative, sum all of them. If the first element is already negative, the sum is `0`.

Two base cases: empty array, and "the front is negative." The recursive case adds the front and continues.

```
#language axioma/beginner
# sum_until_negative: Array Number -> Number
# sum_until_negative([], 0)              = 0
# sum_until_negative([-1, 2, 3], 0)      = 0
# sum_until_negative([1, 2, 3], 0)       = 6
# sum_until_negative([1, 2, 3, -9, 10], 0) = 6

sum_until_negative: func(items, total) [
  if len(items) == 0 then total
  else if first(items) < 0 then total
  else sum_until_negative(rest(items), total + first(items))
]

println(sum_until_negative([], 0))                # 0
println(sum_until_negative([-1, 2, 3], 0))        # 0
println(sum_until_negative([1, 2, 3], 0))         # 6
println(sum_until_negative([1, 2, 3, -9, 10], 0)) # 6
```

The running `total` is an **accumulator**: the answer so far, carried forward, so that when we stop we

already have it. The recursive call *is* the last thing the function does — there is no pending + waiting for the return. That is the shape Chapter 15 names and exploits. You do not need the name yet. You need the picture: work happens *on the way down*, and the base case returns the finished answer.

Why not `reduce`? Because `reduce` has no polite way to say "stop now, ignore the rest." You can fake it with a flag inside the combining function, and then you have written a worse recursive function. When the algorithm is "walk until a condition," write the walk.

7.11 Pending work, tail calls, and the depth guard

Look back at `sum_list`:

```
sum_list([1, 2, 3])
  = 1 + sum_list([2, 3])
  = 1 + (2 + sum_list([3]))
  = 1 + (2 + (3 + 0))
```

The + is **pending** during the recursive call. The interpreter has to remember "when that returns, add 1." For three elements that is harmless. For three million it is three million pending additions.

The interpreter guards nested evaluation. This build reports a limit of 50,000 active Eval frames (`recursion_limit()` reports the number). Non-tail recursion of a long array trips a catchable error:

```
ERROR: recursion limit exceeded (max Eval depth 50000): possible
infinite recursion or excessively deep nesting
```

Use a small bounded example to study the result:

```
axioma> make_range: func(n) [ [i | i <- [1..n]] ]
axioma> sum_list(make_range(100))
5050
```

The maximum safe input also depends on the function and available resources; a source-level recursive call can use several Eval frames. Avoid treating the limit as a supported array size. Chapter 15 rewrites the same function so the recursive call is in **tail position** — nothing pending after it — and Axioma can reuse stack space for supported tail calls. The work and allocation costs of the body still matter.

A tiny pair makes the difference visible. Repeat a string `n` times.

Tail version — the recursive call *is* the answer:

```
#language axioma/beginner
repeat_str: func(s, acc, n) [
  if n == 0 then acc
  else repeat_str(s, acc + s, n - 1)
]
println(repeat_str("go", "", 3))   # gogogo
```

Trace: `("go", "", 3)` → `("go", "go", 2)` → `("go", "gogo", 1)` → `("go", "gogogo", 0)` → `"gogogo"`. No pending concatenations. The accumulator *is* the result.

Non-tail version — concatenation waits for the recursive result:

```
#language axioma/beginner
repeat_str2: func(s, n) [
  if n == 0 then ""
  else s + repeat_str2(s, n - 1)
]
println(repeat_str2("go", 3))      # gogogo
```

Trace: to know `"go" + repeat_str2("go", 2)` we must first know `repeat_str2("go", 2)`, which must first know `repeat_str2("go", 1)`, which must first know `repeat_str2("go", 0)` = `""`, and only then do the concatenations wind back: `"go"`, `"gogo"`, `"gogogo"`. Same answer. Pending work on every frame.

Most textbooks open recursion with factorial, which is the non-tail shape (`n * fact(n - 1)`). It matches one mathematical definition and it makes recursion *feel* like a magic trick you have to unwind. Palindromes and `repeat_str` finish at the bottom: once you hit the base case, you have the answer. That is the process you want in your hands first. Factorial's winding-back is real, and you just saw it in `sum_list` and `repeat_str2`. It is not the only process, and it is not the one Axioma can run at unbounded depth.

For this chapter: do not worry about the ceiling. Coursework lists are small. Know that it exists, that tail-position self-recursion (and mutual tail-recursion) reuse a frame, and that Chapter 15 is where we start writing the tail form on purpose.

7.12 Other spellings you will see

The template with `if len(arr) == 0` is the one to write until it is muscle memory. Axioma also lets the *parameter pattern* carry the empty/non-empty split:

```
#language axioma/beginner
func sum_list3([]) [0]
func sum_list3([h | t]) [h + sum_list3(t)]
println(sum_list3([1, 2, 3, 4]))   # 10
```

Two clauses, one name. The first clause *is* the base case. The second names the head `h` and the tail `t` without calling `first` and `rest`. It is the same function. Read it when you see it; write the `if` form until the recipe is automatic.

The same `[h | t]` syntax **constructs** on the way back out. Doubling every element:

```
#language axioma/beginner
double_each: func(arr) [
  if len(arr) == 0 then []
  else [2 * first(arr) | double_each(rest(arr))]
]
println(double_each([1, 2, 3]))   # [2, 4, 6]
```

`[head | tail]` on the right-hand side means "this element, then that array." It is how you build an

array of the same length as the input. `push` appends at the *end*, so `push(double_each(rest(arr)), 2 * first(arr))` would reverse the result if you are not paying attention. Put the new head on the left.

Helpers that walk by **index** (1-based) are the other common spelling. Collect the positions of the short words:

```
#language axioma/beginner
keep_indices: func(arr, pred) [
  helper: func(position, acc) [
    if position > len(arr) then acc
    else if pred(arr[position]) then
      helper(position + 1, acc + [position])
    else
      helper(position + 1, acc)
  ]
  helper(1, [])
]
is_short: func(s) [length(s) < 6]
words: ["cow", "aardvark", "squirrel", "fish", "snake", "capybara"]
println(keep_indices(words, is_short))   # [1, 4, 5]
```

`cow`, `fish`, `snake` sit at positions 1, 4, and 5 — not 0, 3, 4. Anyone arriving from a 0-based language (Python, JavaScript, Java, C) will write `helper(0, [])` once. The error is `index out of bounds: 0`. Start at 1; stop when `position > len(arr)`.

7.13 A pitfall catalog

These are the failures that show up in real code reviews. Each one is a wrong answer or a loud error, not a style preference.

What you wrote	What goes wrong	The fix
No base case, or a base case the input never hits	`recursion limit exceeded`	Write the smallest input first. Check that every recursive call is *strictly smaller* by a measure you can name (length, `n - 1`, `finish - start`).
`is_palindrome(str)` inside `is_palindrome(s)`	Infinite recursion: you passed the original, not the rest	Recur on `s[2: n - 1]` (or `rest(arr)`, or `n - 1`). The argument must change.
`let f = func() [f()]`	The body cannot see `f`	Use `f: func() [f()]` or `f = …`. There is no `rec` keyword.
`let rec f = …`	`SyntaxError: unexpected token '='`	There is no `rec` keyword. `func` is already recursive.
`{ … }` as a function body	`{}` is the empty set; a block is `[ … ]`	Square brackets.

What you wrote	What goes wrong	The fix
`s[0]`, `helper(0, …)`	`index out of bounds: 0`	Indices start at 1. The last character is `s[n]` or `s[-1]`.
`s[1: n - 1]` meaning "drop both ends"	You kept the first character	Drop-both-ends is `s[2: n - 1]` or `s[2..-2]`. Inclusive slices.
`rest(s)` on a string	Error: `rest` is not for strings	`s[2:]` drops the first character.
`largest(rest(arr))` written twice	Correct, but exponential in a tree and twice-as-slow on a list	Bind once: `r: largest(rest(arr))`.
`reverse(rest(arr)) + [first(arr)]` on a long array	Quadratic copies (+ copies arrays)	Fine for this chapter. Chapter 15's accumulator makes it linear. Or call the builtin `reverse`.
Recursing on the whole input "to be safe"	The input does not shrink	If you cannot point at a smaller value, it is not a recursive step.
Using recursion because you have heard it is "more functional," on a full-list `map`	A worse `map`	Chapter 10. `map` / `filter` / `reduce` / `{x | x <- xs}` when you really do visit every element.

A recursive function is wrong in exactly two ways: it does not stop, or it stops with the wrong combination. The trace in §7.4 distinguishes them. If the substitution never reaches a base case, it does not stop. If it reaches a base case and the values coming back are wrong, the combining rule is wrong.

7.14 When *not* to recurse

Recursion is the right tool when:

- the data is an arbitrary-size list, tree, or nest, and
- the decomposition is "this piece plus a smaller one of the same kind," and
- you may need to stop before the end.

It is the wrong first tool when:

- You want to transform *every* element the same way. That is `map` (Chapter 10), or a comprehension `{2 * x | x <- arr}`.
- You want to keep some elements and drop others, looking at each once. That is `filter`.
- You want one running total over the whole collection, no early stop. That is `reduce`.
- You are counting with a local variable and a `while`. That still works (Chapter 16). Recursion is not a moral improvement on a loop. It is a better fit for recursive *data*.

Industry practice is blunt about this. Linear scans over arrays, in production systems, are loops or library HOFs. Recursion earns its keep on **trees, nested documents, grammars, and "walk until"** — the shapes where the data definition is already recursive. Chapter 8 is trees. That is where people who were unimpressed by `sum_list` suddenly need this chapter.

7.15 Reflection — the *one* skill you are acquiring

Look back at the bodies:

```
sum_list:            first(arr) + sum_list(rest(arr))
count_list:          1 + count_list(rest(arr))
last_element:        [base] / first(arr) / last_element(rest(arr))
has_value:           first(arr) == target / has_value(rest(arr), target)
largest:             max(first(arr), largest(rest(arr)))
is_palindrome:       ends match, then is_palindrome(middle)
sum_until_negative:  stop, or total + first and continue
```

What varies is the *combining rule* (and where you are allowed to stop). What stays the same is the *shape*: smallest case, smaller case, combine. If you can name those three, you can write the function.

The three questions again, because they are the whole chapter:

1. When do I stop?
2. What is the smaller problem?
3. How do I combine this piece with that answer?

Chapter 8 asks the same three questions of a binary tree. The data definition changes. The skill does not.

Exercises

Exercise 7.1 — `count_zeros` (the recursive version)

Rewrite Exercise 6.1 to handle arrays of *any* length using recursion. Signature: `Array -> Integer`.

- `count_zeros([])` → `0`
- `count_zeros([0])` → `1`
- `count_zeros([1, 0, 2, 0, 0])` → `3`
- `count_zeros([1, 2, 3])` → `0`

Exercise 7.2 — `smallest`

By analogy with `largest`. Empty array returns `none`; otherwise the minimum value.

- `smallest([])` → `none`
- `smallest([7])` → `7`
- `smallest([3, 1, 4, 1, 5, 9])` → `1`

Exercise 7.3 — `sum_of_squares`

Sum the *squares* of the elements.

- `sum_of_squares([])` → `0`
- `sum_of_squares([3])` → `9`
- `sum_of_squares([1, 2, 3])` → `14`

Exercise 7.4 — `count_above`

How many elements are strictly greater than `threshold`?

- `count_above([], 10)` → `0`
- `count_above([5, 12, 8, 99], 10)` → `2`
- `count_above([1, 2, 3], 100)` → `0`

Exercise 7.5 — `double_each`

Return a new array with every element doubled. (This produces a *new array* of the same length — the recursive result is an array, not a number.)

- `double_each([])` → `[]`
- `double_each([1])` → `[2]`
- `double_each([1, 2, 3])` → `[2, 4, 6]`

Hint: `[2 * first(arr) | double_each(rest(arr))]` — the **cons** form, which puts the new head on the front of the recursive result. It reads the same way `[h | t]` reads when you destructure, and because the whole thing is one bracket there is no singleton to spell and no block-vs-array question to worry about (contrast §6.6). Mind the order: the doubled head goes on the *left*. `push(...)` appends at the end, so the push-based spelling needs the arguments the other way around: `push` will not preserve order if you build head-first.)

Exercise 7.6 (open) — `reverse`

Return the array in reverse order.

- `reverse_arr([])` → `[]`
- `reverse_arr([42])` → `[42]`
- `reverse_arr([1, 2, 3])` → `[3, 2, 1]`

Two design choices to make:

1. *Where* does `first(arr)` go in the recursive answer — at the front or at the end?
2. Which operation glues it on — `+`? `push`? the cons form?

Pick one, write down examples, then implement. (A linear-time version waits for Chapter 15. A correct quadratic version is a complete answer to this exercise.)

Exercise 7.7 — `is_palindrome`

Write `is_palindrome(s)` for strings, using the shrink-both- ends recursion from §7.9. Empty and single-character strings are palindromes. Do not call `reverse`.

- `is_palindrome("")` → `true`
- `is_palindrome("a")` → `true`
- `is_palindrome("civic")` → `true`
- `is_palindrome("cynic")` → `false`
- `is_palindrome("redivider")` → `true`
- `is_palindrome("runner")` → `false`

Then write `is_palindrome2` that uses a nested helper with two indices and never slices. Both must agree on the examples above. (Do not name the helper `check` — that word is reserved. `go` is fine.)

Exercise 7.8 — `take_while` and `drop_while`

`take_while(arr, pred)` returns the **longest prefix** of `arr` whose elements all satisfy `pred`, and stops at the first failure — even if a later element would have passed. `drop_while(arr, pred)` returns whatever `take_while` did not take: the rest of the array starting at that first failure. If every element passes, `take_while` returns the whole array and `drop_while` returns `[]`.

```
data: [2, 6, 42, 5, 7, 20, 3]
is_even: func(n) [n % 2 == 0]
# take_while(data, is_even)  → [2, 6, 42]
# drop_while(data, is_even)  → [5, 7, 20, 3]
```

`20` is even and it is *not* in the `take_while` result. The walk stopped at `5`.

Next chapter — the same three questions applied to a new data shape: binary trees, built from Concepts.

Solutions to selected exercises: Chapter 7 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch07_solutions.md.)*

Chapter 8 · Trees via Concepts

What this chapter is. A list is a flat sequence: every element is followed by zero or one "next." A tree is *branching*: every node can have several children. We'll model trees in Axioma using **Concepts** — Axioma's record/structure mechanism — and apply the same structural-recursion recipe from Chapter 7 to a new data shape.

8.1 Why a new shape?

A list of grades is a fine shape for "all the test scores in order." But for a *family tree*, or a *file system*, or the **arithmetic expression** `(1 + 2) * (3 - 4)`, a sequence isn't enough — each node has multiple sub-structures. We need a shape where one node can hold *both* a left and a right.

We'll build that shape now. The data type we'll define here — binary tree, with a value plus a left and right child — is the running example. In Chapter 9 we'll generalize.

8.2 Introducing Concepts

Axioma's mechanism for "a thing with several named fields" is the **Concept**. Concepts are Axioma's analog of:

Other language	Mechanism
Racket / Scheme	`(define-struct posn (x y))`
Haskell / ML	`data Node = Node Int Tree Tree`
C / Go	`struct Node { int v; Node *l; Node *r; }`
Python	`class Node: ...` (or `@dataclass`)
Java	`class Node { int v; Node l; Node r; }`

In Axioma, the syntax is natural-language style:

```
concept Node           # declare Node as a kind of Concept
Node has value: 0          # Node has a field called 'value', default 0
Node has left: none        # ... and 'left', default none
Node has right: none       # ... and 'right', default none
```

Four lines, four sentences. `concept Node` declares the type; `has` clauses declare fields with default values. The defaults matter — if you don't give a value when constructing, the field takes the default.

8.3 Making instances

To make an actual node, use the indefinite-article form `a Node`:

```
axioma> n5: a Node {value: 5}
axioma> n5.value
5
axioma> n5.left
none
axioma> n5.right
none
```

We provided only `value`; the other two defaulted to `none`. To make a node with children, pass them explicitly:

```
leaf4: a Node {value: 4}
leaf7: a Node {value: 7}
n5: a Node {value: 5, left: leaf4, right: leaf7}
```

That builds this little tree:

```
      5
     / \
    4   7
```

We can read fields with dot notation:

```
axioma> n5.value
5
axioma> n5.left.value
4
axioma> n5.right.value
7
```

(Chapter 1 of the Axioma manual covers Concepts in more depth. Here we use just enough for the tree shape.)

8.4 The tree shape, formally

> A **Tree** is one of: - `none` — the empty tree - a `Node` instance with a `value`, a `left` (which is a Tree), and a `right` (which is a Tree)

That two-case definition is the *data definition* — and once again it has two cases. Empty/non-empty for lists in Chapter 7. Empty/ node for trees here. The recipe says: one branch in every function for each case.

8.5 Worked example: `tree_size`

Problem. Count how many nodes are in a tree.

1. Data. A Tree as defined above.

2. Signature.

```
# tree_size: Tree -> Integer
```

3. Examples.

```
# tree_size(none)                       = 0
# tree_size(a Node {value: 1})     = 1
# tree_size(  5
#            / \
#           4   7
#         )                               = 3
```

4. Template. Two cases (empty vs. Node) and *two* recursive calls in the non-empty branch — one for each child.

```
tree_size: func(t) [
  if t == none then
    ...
  else
    ... tree_size(t.left) ... tree_size(t.right) ...
]
```

5. Body.

- *Empty:* zero nodes.
- *Node:* this one node plus all the nodes in the left subtree plus all the nodes in the right subtree.

```
tree_size: func(t) [
  if t == none then 0
  else 1 + tree_size(t.left) + tree_size(t.right)
]
```

6. Tests.

```
leaf4: a Node {value: 4}
leaf7: a Node {value: 7}
n5: a Node {value: 5, left: leaf4, right: leaf7}

println(tree_size(none))    # 0
println(tree_size(leaf4))   # 1
println(tree_size(n5))      # 3
```

Notice the shape: same two-branch `if`, same "do something, then combine recursive results" — but now we have *two* recursive calls instead of one. Trees are structurally richer than lists, and the recursion mirrors that.

8.6 Worked example: `tree_depth`

Problem. Return the depth of the deepest leaf below the root. A leaf-only tree (one node) has depth 1. The empty tree has depth 0.

```
# tree_depth: Tree -> Integer

tree_depth: func(t) [
  if t == none then 0
  else [
    dl: tree_depth(t.left)
    dr: tree_depth(t.right)
    1 + (if dl > dr then dl else dr)
  ]
]

println(tree_depth(none))   # 0
println(tree_depth(leaf4))  # 1
println(tree_depth(n5))     # 2
```

We use a `[ ... ]` block in the `else` branch (per Chapter 7's trick) to bind the two recursive results to names, then take the larger one and add 1 for this node.

8.7 Worked example: `tree_contains`

Problem. Does the tree contain a node with this value?

```
# tree_contains: Tree Any -> Boolean

tree_contains: func(t, target) [
  if t == none then false
  else if t.value == target then true
  else tree_contains(t.left, target) or tree_contains(t.right, target)
]

println(tree_contains(n5, 7))    # true
println(tree_contains(n5, 99))   # false
println(tree_contains(none, 1))  # false
```

The three branches: empty (false), match (true), neither (check both subtrees and `or` the results). Compare to the list `contains` in §7.5 — same shape, with an extra `or` because we now have *two* sub-problems instead of one.

8.8 Worked example: `mirror`

Problem. Produce a tree that's a horizontal mirror image of the input — left becomes right, recursively, at every node.

```
# mirror: Tree -> Tree

mirror: func(t) [
  if t == none then none
  else a Node {value: t.value, left: mirror(t.right), right: mirror(t.left)}
]
```

The empty tree mirrors to itself (still empty). A non-empty node mirrors to a *new* node with the same `value` but with the left and right children *swapped* (and each recursively mirrored).

```
m: mirror(n5)
println(m.left.value)    # 7  (was on the right)
println(m.right.value)   # 4  (was on the left)
```

This is the first function we've written that *constructs* a new tree (as opposed to computing a number or boolean from a tree). The shape is the same — two-branch recursion — but the recursive combiner builds a new `a Node` instead of doing arithmetic.

8.9 The pattern

Just like for lists in Chapter 7, *every* tree-processing function follows the same template:

```
f: func(t) [
  if t == none then
    <base-case answer>
  else
    <combine t.value with f(t.left) and f(t.right)>
]
```

What changes from function to function:

- **What to return for `none`** — `0` for size/sum, `false` for contains, `none` for mirror, `[]` for traversal.
- **How to combine `t.value`, `f(t.left)`, and `f(t.right)`** — add them, `or` them, build a new node, append the traversals.

That's the whole pattern. Internalize it and you can write any tree-processing function in one sitting.

8.10 Reflection

We just transferred a habit — structural recursion — from one data shape (list) to another (tree). The skill generalizes: every recursive data type comes with a matching recursion template. Chapter 9 makes the

generalization official: *mutually recursive* data shapes, where two (or more) Concepts refer to each other.

The Concept system is doing more than syntactic sugar here. By naming a record type (`Node`), we get a vocabulary for our data that the rest of the program can rely on. Compare with HtDP's `define-struct posn (x y)` — Axioma's `Concept` is the same idea with natural-language syntax and a few extra features (defaults, inheritance) we'll meet in Chapter 18.

Exercises

You'll need this preamble at the top of every solution:

```
#language axioma/beginner
concept Node
Node has value: 0
Node has left: none
Node has right: none
```

(And probably a small fixed tree to test against.)

Exercise 8.1 — `tree_sum`

Return the sum of all `value` fields in the tree.

- `tree_sum(none)` → `0`
- `tree_sum(a Node {value: 7})` → `7`
- For the tree

```
      3
     / \
    1   5
       / \
      4   7
```

  `tree_sum` → `20`.

Exercise 8.2 — `count_leaves`

A *leaf* is a non-`none` node whose `left` and `right` are both `none`. Count the leaves.

- `count_leaves(none)` → `0`
- `count_leaves(a Node {value: 7})` → `1`
- `count_leaves` on the picture above → `3` (leaves are 1, 4, 7).

Exercise 8.3 — `tree_max`

Return the largest `value` in the tree, or `none` for an empty tree.

Hint: for the empty subtree case *inside* the recursion, you need to return a value that the comparison will lose to. One easy trick: a separate helper `tree_max_helper(t, best)` that carries a running maximum. Or have `tree_max` itself handle empty/leaf/ internal as three branches.

- `tree_max(none)` → `none`
- `tree_max` on the picture above → `7`.

Exercise 8.4 — `tree_min`

By analogy with `tree_max`. Smallest `value`, or `none` for empty.

Exercise 8.5 — `inorder`

Return an array of `values` in *inorder* — left subtree, then this node, then right subtree.

Hint: cons this node onto the right subtree, then glue the left one in front — `inorder(t.left) + [t.value | inorder(t.right)]`. The cons form carries the singleton for you, so there is no trailing-comma spelling to remember. (The three-way concatenation `inorder(t.left) + [t.value,] + inorder(t.right)` also works; the trailing comma makes the Array intention explicit. A bare `[t.value]` also works mid-chain; the block reading matters at the start of a branch or function body (§6.6).)

- `inorder(none)` → `[]`
- `inorder` on the picture above → `[1, 3, 4, 5, 7]`.

Exercise 8.6 (open) — `tree_height_balanced`

Return `true` if for *every* node in the tree, the depths of the left and right subtrees differ by at most 1.

This is harder than it sounds. Spend time on the data definition and examples *before* writing code. Consider what "balanced" means for `none`, for a leaf, for a node whose subtrees are themselves balanced but of different depths.

Next chapter — what if `Node` referenced `Tree` and `Tree` referenced `Node`? Mutually recursive data shapes, and the mutually recursive functions that process them.

Solutions to selected exercises: Chapter 8 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch08_solutions.md.)*

Chapter 9 · Mutual Recursion

> **What this chapter is.** Sometimes two data shapes refer to each other. A *Person* belongs to a *Household*; a *Household* contains *Persons*. Or: an *Expression* is either a literal *or* an operator with sub-expressions. When the data shapes cross- reference, the *functions* that process them have to do the same — they become **mutually recursive**.

9.1 The data shape: two Concepts that refer to each other

Up to here every recursive data type referred only to *itself* — a tree's children are trees, an array's tail is an array. Now imagine modeling a small household:

> A **Person** has a name and (sometimes) a household they belong to.
>
> A **Household** has an address and an array of people.

The arrow goes both ways: Person → Household → Person. That's **mutual recursion** at the data level.

In Axioma:

```
concept Person
Person has name: ""
Person has household: none

concept Household

Household has address: ""
Household has members: []
```

Now we can build a small instance:

```
h: a Household {address: "123 Main", members: [a Person {name: "Alice"}, a Person
 ↪  {name: "Bob"}]}
alice: a Person {name: "Alice", household: h}
bob: a Person {name: "Bob", household: h}
```

The household contains member records whose `household` slot remains `none`. The separate `alice` and `bob` records point to that household. These are finite views of the same people, not a cyclic object graph. Keeping the representation finite makes the stopping case explicit; Chapter 27 handles graphs using separate vertices, edges, and visited sets.

Now we can walk the data either direction:

```
axioma> alice.name
"Alice"
axioma> alice.household.address
"123 Main"
axioma> alice.household.members[1].name
"Alice"
axioma> alice.household.members[2].name
"Bob"
```

9.2 First worked example: counting household members

First, two cooperating functions:

```
# person_household_size: Person -> Integer
# household_size: Household -> Integer

person_household_size: func(p) [
  if p == none then 0
  else household_size(p.household)
]

household_size: func(h) [
  if h == none then 0
  else len(h.members)
]

println(person_household_size(alice))   # 2
println(person_household_size(bob))     # 2
println(household_size(h))              # 2
```

These functions follow one link, but are **not mutually recursive**: `person_household_size` calls `household_size`, which returns without calling back. Mutual recursion needs a cycle in the *call graph*, not a cycle in the data. The next example demonstrates that distinction.

> **A note on call-order.** `person_household_size` calls `household_size`, but we *defined* `person_household_size` first. In Axioma that works because function bodies are looked up at *call* time, not at definition time. By the time we call `person_household_size(alice)`, both functions exist.

9.3 Second worked example: arithmetic expressions

Time for a richer example. We'll model arithmetic expressions like `(1 + 2) * (3 - 4)`. The data definition is:

> An **Expression** is one of: - a **Literal** number, or - an **Operator** application — a *kind* (`"+"`, `"-"`, `"*"`, `"/"`) plus a left Expression and a right Expression.

That's *not* purely mutual recursion in the textbook sense (Operator points to Expression, but Literal

doesn't) — it's **variant recursion**: one type with two flavors, and one flavor recurses. We'll use two Concepts to model the two flavors.

```
concept Lit
Lit has value: 0

concept Op

Op has kind: "+"
Op has lhs: none
Op has rhs: none
```

To build `(1 + 2) * (3 - 4)`:

```
e1: a Lit {value: 1}
e2: a Lit {value: 2}
plus: an Op {kind: "+", lhs: e1, rhs: e2}

e3: a Lit {value: 3}
e4: a Lit {value: 4}
minus: an Op {kind: "-", lhs: e3, rhs: e4}

expr: an Op {kind: "*", lhs: plus, rhs: minus}
```

Now the **evaluator**. The data has two flavors; the evaluator has two branches:

```
eval_expr: func(e) [
  if e is Lit then e.value
  else if e is Op then eval_op(e)
  else none
]

eval_op: func(e) [
  l: eval_expr(e.lhs)
  r: eval_expr(e.rhs)
  if e.kind == "+" then l + r
  else if e.kind == "-" then l - r
  else if e.kind == "*" then l * r
  else if e.kind == "/" then l / r
  else none
]

println(eval_expr(e1))      # 1
println(eval_expr(plus))    # 3
println(eval_expr(minus))   # -1
println(eval_expr(expr))    # -3
```

Two new ideas at work:

1. **The `is` test.** `e is Lit` returns `true` if `e` was built with `a Lit`. We dispatch on the *kind* of object, not on a field. (Note that `is` only *tests* — declaring the concept is `concept Person`, as in Chapter 9's data definitions.)
2. **Two functions that call each other.** `eval_expr` dispatches an operator to `eval_op`; `eval_op` calls `eval_expr` on each child. This is mutual recursion. Every cycle moves to smaller sub-expressions, and a literal stops the recursion. The same evaluator could be written as a single recursive function, but splitting dispatch from operator evaluation exposes the call graph.

This is, in miniature, an **interpreter** — a program that takes data describing a computation and runs that computation. Chapter 22 builds it out into a full meta-circular evaluator for Axioma itself.

9.4 The shape

When data shapes A and B point at each other, the functions that process them point at each other too:

```
process_a: func(a) [
  ... process_b(a.field_of_type_B) ...
]
process_b: func(b) [
  ... process_a(b.field_of_type_A) ...
]
```

Each function gets one branch per *shape* of its argument, and the recursive calls hop between types as needed.

When data has *variants* (one type with several flavors), one function with one branch per variant does the job — *or* you split into one function per variant. Either is fine.

9.5 Reflection

We started Part II with arrays — the simplest arbitrary-size data shape. We ended with mutual recursion — the most general. The through-line is the **design recipe** unchanged:

1. Write down the data definition.
2. From the data definition, derive the function template — one branch per case.
3. Fill in the branches by combining the recursive results.

That recipe has now been applied to four different data shapes: fixed-size atoms (Part I), arrays (Chapter 7), trees (Chapter 8), and mutually-recursive concept graphs (this chapter). Every new data shape you ever meet — JSON, ASTs, file systems, scene graphs — you process the same way.

You're done with Part II. The rest of the book is about **abstraction** (factoring out the parts of these patterns that *don't* change so you can reuse them), **stateful programming** (when mutation is the right tool), and **Axioma-specific extensions** (types and data modeling, enums, multi-valued logic, logic programming, and the meta-circular evaluator).

Exercises

Exercise 9.1 — `name_in_household`

Given a Person, return an array of the *names* of all members of their household.

- For `alice`: `["Alice", "Bob"]`
- For a person whose `household` is `none`: `[]`

Exercise 9.2 — `count_total_people`

Given an array of Households, return the total number of people across all of them.

- `count_total_people([])` → `0`
- `count_total_people([h1, h2])` → sum of `len(h1.members) + len(h2.members)`

Exercise 9.3 — `expr_depth`

Add an `expr_depth(e)` function that returns the depth of the expression tree. A `Lit` has depth 1; an `Op` has depth `1 + max(depth(lhs), depth(rhs))`.

- `expr_depth(a Lit {value: 7})` → `1`
- `expr_depth((1 + 2))` → `2`
- `expr_depth((1 + 2) * (3 - 4))` → `3`

Exercise 9.4 — `count_ops`

Count the number of `Op` nodes in an expression (i.e., the number of binary operators).

- `count_ops(a Lit {value: 7})` → `0`
- `count_ops((1 + 2))` → `1`
- `count_ops((1 + 2) * (3 - 4))` → `3`

Exercise 9.5 — `add_one_to_lits`

Build a *new* expression in which every `Lit` has its `value` increased by 1, but the operator structure is unchanged.

- `add_one_to_lits(a Lit {value: 7})` → `a Lit {value: 8}`
- `add_one_to_lits((1 + 2))` → `(2 + 3)`

Hint: this is the "build a new tree" pattern from §8.8 applied to the two-variant Expression shape.

Exercise 9.6 (open) — extend the evaluator

Add support for a new operator of your choice — perhaps `"%"` (modulo), `"^"` (exponent), or `"max"` (the larger of two sub-results). What does the data definition gain? What changes in the evaluator?

Or — a harder twist — add a *unary* operator like `"neg"` (numeric negation) by introducing a third Concept `UnaryOp` and updating the evaluator to dispatch on three flavors instead of two.

End of Part II. Part III turns these patterns into reusable abstractions: `map`, `filter`, `reduce`, lambda functions, and local definitions.

Solutions to selected exercises: Chapter 9 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch09_solutions.md.)*

Chapter 10 · Higher-Order Functions

What this chapter is. Part II ended with five recursive functions that all *look the same*. This chapter names what's common, hands you three pre-written tools — `map`, `filter`, `reduce` — and shifts your job from "write the recursion" to "describe the transformation." Broadcasting then extends a scalar rule to compatible collections, and fusion connects those rules without intermediate arrays.

10.1 The repetition you already noticed

At the end of Chapter 7 we listed the bodies of `sum_list`, `count_list`, `largest`, `contains`, and `last_element` side by side. They all had the same shape — peel off `first(arr)`, recur on `rest(arr)`, combine. Only the *combining rule* changed.

In Chapter 7 §7.8 I called that **structural recursion**. Now we're going to take the next step: pull the *recursion itself* out into a reusable function, and pass in the combining rule as a *parameter*.

That parameter is another function. A function whose argument is a function is called a **higher-order function** — and writing them is the single biggest leap in your power as a programmer since you learned `if`.

10.2 First example — `map`

The pattern. "Take every element of an array, do *something* to it, return a new array of the results."

We saw this in §7.5 exercise `double_each`: take `[1, 2, 3]`, double each one, get `[2, 4, 6]`. We could also halve each one, or square each one, or convert each one to a string. The *something* changes; the *for-every-element* part doesn't.

The for-every-element part has a name: `map`.

```
#language axioma/beginner
nums: [1, 2, 3, 4, 5]
double: func(x) [x * 2]
println(map(double, nums))    # [2, 4, 6, 8, 10]
```

Read that aloud: "`map` the function `double` over the array `nums`." `map` is a function that takes *two* arguments — a function and an array — and returns a new array.

The function we pass in doesn't have to be defined first; we can write it inline:

```
#language axioma/beginner
println(map(func(x) [x * x], [1, 2, 3, 4, 5]))
# [1, 4, 9, 16, 25]
```

Why this is worth learning. Before `map`, `double_each` was seven lines of recursion. After `map`, it's one line. The recursion is *still happening* — `map` does it internally — but you don't have to write it yourself.

10.3 Second example — `filter`

The pattern. "Keep the elements that satisfy a test; throw the rest away."

Exercise 7.4 was `count_above(arr, threshold)` — count how many elements exceed a threshold. A close cousin: *list* the elements that exceed the threshold. The test changes (>10, even, prime, non-empty…); the keep-the-passing-ones part doesn't.

The keep-the-passing-ones part has a name: `filter`.

```
#language axioma/beginner
nums: [3, 1, 4, 1, 5, 9, 2, 6]
is_big: func(x) [x > 4]
println(filter(is_big, nums))     # [5, 9, 6]
println(filter(func(x) [x % 2 == 0], nums))   # [4, 2, 6]
```

The function you pass to `filter` must return a Boolean — `true` to keep the element, `false` to drop it. By convention such a function is called a **predicate**, and predicate names usually read like a yes/no question: `is_big`, `is_even`, `is_negative`.

10.4 Third example — `reduce`

The pattern. "Walk through the array, combining elements with a running total."

`sum_list` did this with `+`. `largest` did it with `max`. We could imagine `product_list` with `*`, `concat_all` with `+` on strings, `all_true` with `and`, and so on. The combining operation changes; the walk-and-accumulate part doesn't.

The walk-and-accumulate part has a name: `reduce`.

```
#language axioma/beginner
nums: [1, 2, 3, 4, 5]
add: func(acc, x) [acc + x]
times: func(acc, x) [acc * x]
println(reduce(add, 0, nums))     # 15   (1+2+3+4+5)
println(reduce(times, 1, nums))   # 120  (1*2*3*4*5)
```

`reduce` takes *three* arguments: the combining function, a starting value (the **accumulator** — what the running total should be before any element is seen), and the array.

The combining function takes two arguments: the running total so far, and the next element. It returns

the new running total.

Two common pitfalls:

- **Wrong starting value.** Starting `reduce(add, 0, ...)` gives the right sum because 0 is the *identity* for addition (adding 0 doesn't change a number). For `*` the identity is 1, not 0. For string concatenation it's `""`. For `max` it's the smallest possible value (which is awkward — see below).
- **Wrong combining direction.** Some functions don't matter (`+`, `*`, `max`). Others do — try `reduce(func(a, b) [a - b], 0, [1, 2, 3])`: you get `-6`, not `-4` or `4`, and which one you wanted depends on whether you're folding from the left or the right.

`reduce` in Axioma folds from the *left*: it computes `((0 + 1) + 2) + 3`.

When the direction matters, you can say it explicitly. Axioma has the Scheme-faithful folds `foldl` and `foldr` (with `fold_left` / `fold_right` spellings too):

```
#language axioma/beginner
foldl(func(acc, x) [acc - x], 0, [1, 2, 3])   # -6  - ((0-1)-2)-3, same as reduce
foldr(func(x, acc) [x - acc], 0, [1, 2, 3])   # 2   - 1-(2-(3-0))
```

`foldl` calls its function `fn(acc, elem)` (the accumulator first, like `reduce`); `foldr` calls `fn(elem, acc)` and works from the right end inward. Two more relatives in the same family: `append_map(fn, xs)` maps a collection-returning function and concatenates the results (Scheme's `append-map`; also spelled `flatmap` / `mapcat`), and `for_each(fn, xs)` runs `fn` for its side effects and returns nothing.

10.5 Worked example — `mean` in three lines

Problem. Compute the average of an array of numbers.

1. Data. A non-empty array of numbers.

2. Signature.

```
# mean: Array -> Number
```

3. Examples.

```
# mean([4])              = 4
# mean([1, 2, 3])        = 2
# mean([10, 20, 30, 40])  = 25
```

4-5. Template / body. Sum the array, divide by length:

```
#language axioma/beginner
mean: func(arr) [
  reduce(func(a, b) [a + b], 0, arr) / len(arr)
]
```

6. Tests.

```
println(mean([4]))              # 4
println(mean([1, 2, 3]))        # 2
println(mean([10, 20, 30, 40]))  # 25
```

Three lines. Compare with the equivalent recursive version (probably ten lines, with a helper for sum). The recursion is hidden inside `reduce`; we just told it *what to do with each element*.

10.6 Combining HOFs — the pipeline shape

The real power emerges when you chain them. Here's "average of the even numbers larger than 10":

```
#language axioma/beginner
nums: [3, 11, 14, 17, 22, 8, 25, 30]
above_10: filter(func(x) [x > 10], nums)
evens: filter(func(x) [x % 2 == 0], above_10)
total: reduce(func(a, b) [a + b], 0, evens)
answer: total / len(evens)
println(answer)    # 22  (14 + 22 + 30 = 66, 66 / 3 = 22)
```

Each line is one transformation. The variables (`above_10`, `evens`, `total`) name the *intermediate result* — they're not strictly necessary, but they document the steps.

In Chapter 12 we'll learn how to make those intermediate names **local** to a single computation, so they don't clutter the top level. For now, just appreciate that each step is a one-liner.

There is a second way to build a pipeline: instead of naming each intermediate *result*, name the *transformation* and glue the pieces together into a single reusable function. That is **composition**, and it gets its own treatment in §10.10.

10.7 Writing your own higher-order functions

`map`, `filter`, and `reduce` are not magic. They're recursive functions you could have written yourself. Here is `map` in full — same structural-recursion template you saw in Chapter 7:

```
#language axioma/beginner
my_map: func(f, arr) [
  if len(arr) == 0 then []
  else [f(first(arr)) | my_map(f, rest(arr))]
]
println(my_map(func(x) [x * 2], [1, 2, 3]))   # [2, 4, 6]
```

The cons expression `[f(first(arr)) | my_map(f, rest(arr))]` places the transformed first value before the recursively transformed tail. An equivalent concatenation is `[f(first(arr)),] + my_map(f, rest(arr))`. The comma in that alternative distinguishes the one-element Array from a branch

block (§6.6).

Two HtDP-classic HOFs not built into Axioma are `any?` and `all?`: "does *any* element satisfy this predicate?" and "do *all* elements satisfy it?" Defining them is a great exercise in HOF composition (and in fact you'll do exactly that as Exercise 10.5 below):

```
#language axioma/beginner
any_of: func(pred, arr) [
  if len(arr) == 0 then false
  else if pred(first(arr)) then true
  else any_of(pred, rest(arr))
]
println(any_of(func(x) [x < 0], [3, 1, -1, 4]))   # true
println(any_of(func(x) [x < 0], [3, 1, 4]))       # false
```

10.8 The shift, named

In Part II your task was: "tell the computer *step by step* how to walk through the data." In this chapter your task became: "tell the computer *what to do with each element* — it'll handle the walking."

That shift has a name in computer science: **declarative** vs. **imperative**. Imperative code says *how*; declarative code says *what*. Higher-order functions don't make declarative code mandatory — they just make it easy.

You haven't lost any expressive power. The recursive version is *still there* (you wrote `my_map` in §10.7). What you've gained is the ability to *name and reuse* the common shape.

10.9 Aside — sets and comprehensions

There's a sibling notation worth flagging before Part IV. Axioma has a *second* collection type alongside arrays: the **set**. A set is written with curly braces; it has no order and no duplicates:

```
arr: [1, 2, 3, 2, 1]    # array — order matters, duplicates kept
println(arr)            # [1, 2, 3, 2, 1]

s: {1, 2, 3, 2, 1}      # set   — unordered, duplicates removed
println(s)              # {1, 2, 3}
```

Use an array when *order* matters or *repetition* is meaningful (a sequence of test scores, a queue of pending tasks). Use a set when you only care *which* values are present (the set of users that logged in today, the words that appear in a document).

Sets have their own operators: `union`, `intersection`, `difference`, and the membership test `in`:

```
a: {1, 2, 3}
b: {3, 4, 5}
println(a union b)         # {1, 2, 3, 4, 5}
```

```
println(a intersection b)  # {3}
println(a difference b)    # {1, 2}
println(2 in a)            # true
println(7 in a)            # false
```

(For the mathematically inclined: `∪`, `∩`, and `∈` work as synonyms.)

Comprehensions

Both arrays and sets have a **comprehension** form. The shape `[ expr | name <- source ]` builds a new array by stepping `name` through `source` and collecting the value of `expr`:

```
sqs: [x * x | x <- [1, 2, 3, 4]]
println(sqs)               # [1, 4, 9, 16]
```

Swap the brackets for braces and you get a set comprehension — same shape, deduplicated result:

```
sqs_set: {x * x | x <- [1, 2, 3, 4]}
println(sqs_set)           # {1, 4, 9, 16}  (a set)
```

A comprehension can carry filters too — extra clauses after the generator. Each filter is a Boolean expression; only elements that satisfy *all* filters survive:

```
evens_squared: [x * x | x <- [1, 2, 3, 4, 5], x % 2 == 0]
println(evens_squared)     # [4, 16]
```

That's the same as `map(func(x) [x * x], filter(func(x) [x % 2 == 0], [1, 2, 3, 4, 5]))` — and many programmers find the comprehension easier to read.

If you learned set-builder notation from a math text, you can write it the way the book printed it: the colon separator and membership-style binder both work — `{x : x ∈ u, x > 1}` means exactly `{x | x <- u, x > 1}`. (When `x` is already bound, `x in s` keeps its membership-test meaning, so filters like `{x | x <- s, x in t}` still read as intersection.)

Why we mention it now

Comprehensions are a **declarative** shorthand for the `filter`-then-`map` patterns you just learned. They appear heavily in Part VI — Chapter 21's logic-programming queries *are* set comprehensions over the fact base:

```
# Preview from Ch.21: who are John's grandchildren?
grandkids: {Z | Y <- parent("John", Y), Z <- parent(Y, Z)}
```

You don't need to internalise comprehensions to finish Part III — `map`/`filter`/`reduce` cover the same territory and read slightly differently. But know that the notation exists, and that *every set comprehension can be rewritten with `filter` and `map`*, and vice versa. They're the same idea, with different ergonomic

trade-offs.

Python syntax — the same comprehension, two faces

If you've used Python, the shape `[x*2 for x in xs if x > 0]` will feel native. Axioma accepts it as-is — the `for`/`in`/`if` keywords are aliases for `<-`/`,` in the comprehension clause list, and the pipe `|` becomes optional when you use them:

```
# Axioma pipe form (mathematical, set-builder-flavored)
doubled: [x * 2 | x <- [1, 2, 3, 4, 5], x > 0]

# Python pipe-less form — same result, paste-from-Python friendly
doubled: [x * 2 for x in [1, 2, 3, 4, 5] if x > 0]
```

Both produce `[2, 4, 6, 8, 10]`. The two forms can be mixed in the same program — pick whichever reads better for the expression at hand. Mathematical-leaning code tends toward the pipe form; data-processing code tends toward the `for`/`if` form. There is no semantic difference.

The same dual surface applies to set comprehensions:

```
xs: [1, 2, 3, 4, 5]
{x * 2 | x <- xs, x > 0}            # Axioma pipe form
{x * 2 for x in xs if x > 0}        # Python pipe-less form
```

Walrus bindings — compute once, reuse

Sometimes you want to *compute a value*, *test it*, and then *use it in the output expression* — all in one comprehension. Python 3.8 introduced the walrus operator `:=` for this; Axioma uses the same `:` it uses everywhere else for binding:

```
xs: [1, 2, 3, 4, 5]

# Walrus binding inside a comprehension clause — same `:` as ordinary binding
{y | x <- xs, y: x * x, y > 4}     # → {9, 16, 25}
```

The binding is **iteration-local**: `y` doesn't leak outside the comprehension. (Python's walrus *does* leak, which has surprised a generation of Python programmers — Axioma deliberately diverges here.)

When do you reach for a walrus? When the same expression appears in both the filter *and* the output:

```
# A small, pure calculation used in both forms
f: func(x) [x - 2]
# Awkward — `f(x)` may run twice for a retained element
{f(x) | x <- xs, f(x) > 0}

# Better — `f(x)` runs once, named `y`
{y | x <- xs, y: f(x), y > 0}
```

Dict comprehensions — building hashes

Axioma's hashes use curly braces with `key: value` pairs (you'll meet them properly in Chapter 14). Comprehensions can build them too — the only change is that the output expression is a key-value pair instead of a single value:

```
xs: [1, 2, 3, 4, 5]

# Axioma pipe form
squares: {x: x * x | x <- xs}
# {1:1, 2:4, 3:9, 4:16, 5:25}

# Python pipe-less form
squares: {x: x * x for x in xs}
# same result
```

Filters and walrus bindings work as expected:

```
big_squares: {x: y | x <- xs, y: x * x, y > 4}
# {3:9, 4:16, 5:25}
```

The result is a hash; you index it with `squares[3]` (which is `9`) or `squares.3`.

Lazy generators — infinite sources, bounded consumption

When you wrap a comprehension in **parens** instead of brackets or braces, Axioma builds a **generator**: a lazy iterator that produces elements one at a time on demand, instead of eagerly materialising them all up front.

```
xs: [1, 2, 3, 4, 5]

# Eager — builds the full array immediately
arr: [x * 2 | x <- xs]            # [2, 4, 6, 8, 10]

# Lazy — builds a generator that produces 2, then 4, then …
gen: (x * 2 | x <- xs)            # <generator>
gen: (x * 2 for x in xs)          # Python form works too
```

You consume a generator with these four built-ins:

Builtin	Effect
`force(gen)`	Pull everything into an array.
`gen_take(n, gen)`	Pull the first `n` elements; rest stay in the gen.
`gen_drop(n, gen)`	Advance past `n` elements (mutates, returns gen).
`gen_next(gen)`	Pull one element; returns `Ω` when exhausted.

Why does laziness matter? Because the source can be **infinite**. Axioma exposes a few classical infinite sets via the `infinite_set` builtin — `"naturals"`, `"integers"`, `"primes"`, `"evens"`, `"odds"`, `"fibonacci"`. Wrapped in a lazy comprehension they become pull-driven streams:

```
# Naturals: 1, 2, 3, 4, … (genuinely infinite)
nats: (n | n <- infinite_set("naturals"))

# Take just the first 10 — runs in constant time, no overflow
first_ten: gen_take(10, nats)     # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Eager comprehension over an infinite source would never finish. Lazy generators give you the syntactic convenience of comprehensions for problems that *cannot* be solved eagerly.

A common pattern: filter expensively, take a few:

```
# Find the first 5 prime numbers > 100 (without computing them all)
big_primes: (p | p <- infinite_set("primes"), p > 100)
first_five: gen_take(5, big_primes)
# [101, 103, 107, 109, 113]
```

> **Note.** The bare names `naturals`, `integers`, `primes` in Axioma refer to *finite samples* (the first 100 naturals, etc.) — handy for quick exploration but not infinite. Reach for `infinite_set("…")` when you want the genuinely-unbounded form.

Lazy generators carry the full clause surface

A lazy generator is not restricted to one source and one filter. Everything you can write in a bracket comprehension works inside the parens too — multiple generators, walrus bindings, tuple destructuring:

```
# Multiple generators — a lazy Cartesian product
pairs: ((x, y) | x <- [1, 2], y <- [10, 20])
force(pairs)        # [(1,10), (1,20), (2,10), (2,20)]

# Walrus binding inside the lazy clause list
squares: (s | x <- [1, 2, 3, 4], s: x * x)
force(squares)      # [1, 4, 9, 16]

# First-generator tuple destructure
labelled: (n + len(s) | (n, s) <- [(1, "a"), (2, "b")])
```

The chain stays lazy all the way through. `gen_take(2, pairs)` pulls only as much of the product as it needs — the outer loop advances exactly when the inner one is exhausted, never sooner. A two-generator stream over two infinite sources is still perfectly safe to *take* from.

The source of *any* generator clause can be a **concept extent** or a **relational query**, not just a list — the same shapes you met for eager comprehensions:

```
concept Planet
Planet has name: ""
earth: a Planet {name: "Earth"}
gen: (p.name | p <- Planet)
println(force(gen))                    # ["Earth"]

relation planet_parent(adult, child)
planet_parent("John", "Mary")
kids: (c | c <- planet_parent("John", c))
println(force(kids))                   # ["Mary"]
```

Row caps — `limit` and `offset`

Two clauses borrowed from SQL cap the output of *any* comprehension, eager or lazy. `limit N` keeps at most `N` rows; `offset N` skips the first `N`:

```
[x | x <- naturals, limit 5]             # [1, 2, 3, 4, 5]
[x | x <- naturals, offset 6, limit 3]   # [7, 8, 9]

# On a lazy generator, limit bounds total pulls — so a
# capped stream over an infinite source terminates:
g: (n | n <- infinite_set("naturals"), limit 4)
force(g)                                 # [1, 2, 3, 4]
```

`limit` and `offset` are *soft* keywords — they are still ordinary identifiers you can use as variable or field names everywhere else; the parser only reads them as clause keywords when they open a comprehension clause.

Where Axioma goes beyond Python

Python's comprehensions are excellent at what they do. Axioma covers the same surface (you can paste Python comprehensions directly into Axioma code) and then keeps going. Here are the features that *only* Axioma has:

Feature	Python	Axioma
List comprehensions	[done]	[done]
Set comprehensions	[done]	[done]
Dict comprehensions	[done]	[done]
Filters with `if`	[done]	[done]
Multiple generators (Cartesian)	[done]	[done]
Walrus binding	[done]	[done] (×3 forms)
Lazy generator expressions	[done]	[done] (full clause surface)
`limit` / `offset` row caps	[no]	[done]

Feature	Python	Axioma
Mathematical pipe form `|`	[no]	[done]
Iteration-local walrus scope	leaks	[done]
Multi-filter folds with `and`	[done]	[done]
Bare-concept iterables `<- C`	[no]	[done]
`is`-comprehensions	[no]	[done]
Relational fact queries	[no]	[done]
Cross-relation unification	[no]	[done]
Variable-chain unification	[no]	[done]
Tag-filter `@theorem`/`@axiom`	[no]	[done]
Implicit `_1`, `_2` binders	[no]	[done]

The last six rows are the interesting ones — they're features that are not built into Python's comprehensions. Python libraries can provide related facilities; Axioma integrates them into its language. They're previewed in Chapter 21 (logic programming) and put to serious work in Chapters 24, 36, and 42. Here is a small, self-contained preview:

```
concept Country
Country has name: ""
example_country: a Country {name: "Exampleland"}
println({c.name | c <- Country})       # {"Exampleland"}

relation preview_parent(child, adult)
relation preview_grandparent(child, gp)
preview_parent("Alice", "Mary")
preview_parent("Mary", "John")
preview_grandparent(C,G) whenever preview_parent(C,P) and preview_parent(P,G)
println({X | preview_parent(X,Y) and preview_grandparent(X,Z)}) # {"Alice"}
println({X @theorem | X <- preview_grandparent(X, _)})         # {"Alice"}
```

Concepts receive a fuller treatment in Chapter 18. But the comprehension syntax you've learned here will carry you the whole way: the same `{expr | source, condition}` shape works over arrays, sets, concept extents, fact stores, and lazy streams without changing surface.

> **Aside — comprehensions in other languages.** SQL has comprehensions in disguise (`SELECT ... FROM ... WHERE ...`), Haskell has them with multiple generators and guards, F# has them as *sequence expressions*, LINQ has them in C# and VB. Each language tweaks the syntax (`yield`, `from`/`select`, `for`/`if`), but the underlying idea — *describe what you want, not how to build it* — is the same everywhere. Axioma's contribution is making the *same notation* work over both data collections and a logical fact base.

The same question, many ways

A useful exercise: pick one simple question and ask how many surface forms answer it. Take *"is there a person named Mike in this set?"*. The collection is

```
concept Person
Person has name: ""
mike, alice: a Person
mike.name: "Mike"
alice.name: "Alice"
persons: {mike, alice}
```

and the predicate is `x.name == "Mike"`. Here are **seventeen working forms** that all return `true`:

```
# --- Quantifier family ---
exists x in persons | x.name == "Mike"              # English-natural
∃      x in persons | x.name == "Mike"              # Unicode

# --- Set-comprehension family (matching set is non-empty) ---
len({x | x <- persons, x.name == "Mike"}) > 0       # cardinality test
{x | x <- persons, x.name == "Mike"} != {}          # non-empty test
not ({x | x <- persons, x.name == "Mike"} == {})    # De Morgan flip
len({p for p in persons if p.name == "Mike"}) > 0   # Python pipe-less
{p for p in persons if p.name == "Mike"} != {}      # Python pipe-less

# --- List-comprehension family (indicator sum > 0) ---
sum([1 | p <- persons, p.name == "Mike"]) > 0       # probability-style
sum([1 for p in persons if p.name == "Mike"]) > 0   # Python
len([1 for p in persons if p.name == "Mike"]) > 0   # Python count

# --- Higher-order ---
len(filter(func(p) [p.name == "Mike"], persons)) > 0
reduce(func(acc, p) [acc or p.name == "Mike"], false, persons)

# --- Walrus binding (compute name once) ---
{x | x <- persons, n: x.name, n == "Mike"} != {}

# --- Lazy generator ---
g: (x | x <- persons, x.name == "Mike")
len(force(g)) > 0                                   # materialize then check
len(gen_take(1, (x | x <- persons, x.name == "Mike"))) > 0   # SHORT-CIRCUIT

# --- Hash destructure (different source shape) ---
hashes: [{name: "Mike"}, {name: "Alice"}]
len({n | {name: n} <- hashes, n == "Mike"}) > 0

# --- Negated universal — ∃x.P(x) ≡ ¬∀x.¬P(x) ---
not (forall x in persons | x.name != "Mike")
```

Plus the related-but-distinct **membership** test, which presupposes you already have the witness:

```
mike in persons                                     # "is mike one of them?"
```

Why do all these exist? Because each one fits a different **reader's mental model**. Math readers parse the

pipe form naturally. Python programmers see `for x in xs if P(x)` as familiar territory. SQL writers think in `having`/`limit`. Functional programmers reach for `reduce`. Probability and measure-theory readers see the indicator sum. Logic programmers want the <– from Prolog.

These forms express the same question, but do not all use the same execution strategy. Choose the clearest form, then consider how much of the input it consumes and whether it builds an intermediate collection.

Short-circuit note. Quantifier forms (`exists`/∃) and `gen_take(1, lazy_gen)` stop at the first match. Eager comprehension forms scan the whole collection. For large inputs and cheap predicates, the short-circuit forms can be orders of magnitude faster — at zero readability cost.

This pattern (one question, many surface forms) repeats throughout Axioma: classification (`is`, ancestor walk, intensional class membership), property access (dot, possessive, hash bracket), concept declaration (block, `has`, inheritance, `defines`). The textbook section §10.9 just made it visible for comprehensions; you'll meet the same shape elsewhere.

10.10 Composition — making the pipeline itself a value

Back in §10.6 you built a pipeline out of named intermediate results:

```
above_10: filter(func(x) [x > 10], nums)
evens:    filter(func(x) [x % 2 == 0], above_10)
```

Each name holds a *value* — the result of one step. That works, but notice what you cannot do with it: you cannot hand the pipeline to `map`, store it in a dict, or run it on a different list tomorrow. The pipeline exists only while those variables do.

Composition fixes that by naming the *transformation* instead of the result.

The idea

Two functions, wired end to end. The output of the first becomes the input of the second, and the pair behaves as a single new function:

```
#language axioma/beginner
add_one:  func(x) [x + 1]
times_ten: func(x) [x * 10]

both: add_one >> times_ten     # a NEW function — nothing has run yet
println(both(3))               # 40   (3 + 1 = 4, then 4 * 10 = 40)
```

Read `>>` as "and then". `add_one >> times_ten` is "add one, and then multiply by ten". Nothing is computed until you call it.

The crucial part: `both` is an ordinary value. You can map it, store it, pass it, or compose it again.

```
println(map(both, [1, 2, 3]))      # [20, 30, 40]
```

Point-free — the argument you never name

Compare these two definitions of the same thing:

```
#language axioma/beginner
strip_it:  func(s) [trim(s)]
shrink_it: func(s) [lower(s)]

# pointed — names the argument s
normalize_pointed: func(s) [ lower(trim(s)) ]

# point-free — never mentions s at all
normalize: strip_it >> shrink_it

println(normalize("  Hello  "))          # hello
println(normalize_pointed("  Hello  "))  # hello
```

Both compute the same thing. The difference is *structure*: in the point-free version the pipeline is assembled from named stages, so each stage can be tested and reused on its own. In the pointed version the stages are buried inside a nest of calls.

The name “point-free” comes from mathematics, where an argument is called a *point*. Point-free means argument-free.

Two directions, and why both exist

Axioma writes composition four ways. They differ only in reading order:

```
#language axioma/beginner
f: func(x) [x + 1]
g: func(x) [x * 10]

println(compose(f, g)(3))   # 40   named form: "f, then g"
println((f >> g)(3))        # 40   infix:      "f and then g"
println((g << f)(3))        # 40   infix:      "g after f"
println((g ∘ f)(3))         # 40   glyph:      "g after f"
```

The first two read **left to right**, like a shell pipeline or a recipe. The last two read **right to left**, the way mathematics has written composition for over a century: `g ∘ f` is spoken “g after f”, and the function nearest the argument runs first.

Both directions are correct; they are the same operation read from opposite ends. Axioma ships both rather than forcing you to translate in your head — but notice the argument order flips:

`f >> g` is the same function as `g << f`

If you only ever want one, use >>. It reads in the order things happen, which is what most people want

most of the time.

> ∘ is typed as `\circ` in most editors, but you never have to. `<<` is the same operator in plain ASCII, and `` `circ `` works too.

Composition is closed

The result of composing is itself composable, without limit:

```
#language axioma/beginner
f: func(x) [x + 1]
g: func(x) [x * 10]
h: func(x) [x - 2]

step1: f >> g          # a function
step2: step1 >> h      # composing a composition
println(step2(3))      # 38
```

This is why composition scales where nesting does not. `h(g(f(x)))` grows a parenthesis every time; `f >> g >> h` grows a word.

Building a pipeline from a list

Because `compose()` with no arguments is the *identity* — the function that returns its input unchanged — a list of stages can be folded into a single function, whatever its length:

```
#language axioma/beginner
f: func(x) [x + 1]
g: func(x) [x * 10]

stages: [f, g]
built: reduce(func(acc, s) [compose(acc, s)], compose(), stages)
println(built(3))       # 40

empty: reduce(func(acc, s) [compose(acc, s)], compose(), [])
println(empty(3))       # 3   — no stages, so nothing happens
```

That last line is the point of having an identity: the empty pipeline is not an error, it is the pipeline that does nothing.

When a chain isn't enough — `converge`

Composition builds a *chain*: one value in, one value out, one stage at a time. Some computations need a *branch* instead. A mean, for instance, needs the **sum** and the **count** of the same list, divided:

```
#language axioma/beginner
total:    func(xs) [reduce(func(a, b) [a + b], 0, xs)]
```

```
count_of: func(xs) [len(xs)]
divide:   func(a, b) [a / b]

mean: converge(divide, [total, count_of])
println(mean([2, 4, 6, 8]))     # 5
```

`converge(combiner, [f, g])` feeds the **same** input to `f` and to `g`, then hands both results to `combiner`. Read it as "fork, then rejoin". Without it you would have to name the list just to mention it twice.

Comparing by a key — on

The other shape a chain can't express: applying a preparation step to *both* arguments of a two-argument function.

```
#language axioma/beginner
length_of: func(s) [len(s)]
longer: on(func(a, b) [a > b], length_of)

println(longer("hello", "hi"))    # true   — compares by length
```

`on(cmp, key)(a, b)` is `cmp(key(a), key(b))`. Composition cannot do this: `cmp ∘ key` would feed one result to a function expecting two.

One gotcha — calls bind tighter

```
(f >> g)(3)     # 40  — compose, then call
f >> g(3)       # ERROR — reads as f >> (g(3)), composing with a number
```

Parenthesise the composition before calling it, exactly as you would on paper.

|> is not the same thing

You will see `|>` used for pipelines too, and it is worth keeping them apart:

```
3 |> f |> g        # 40  — a computation. There is a 3 in it.
f >> g             # a FUNCTION. Reusable, mappable, storable.
```

`|>` pushes a *value* through stages, right now. `>>` builds a *function* out of stages, for later. Same reading direction, different product. Use `|>` when you have the value in hand; use `>>` when you want to keep the pipeline.

Summary

Want	Write
a function that does A then B	`A >> B`
the same, mathematical order	`B << A` or `B ∘ A`
the same, as a call	`compose(A, B)`
run a value through stages now	`x |> A |> B`
a do-nothing function	`compose()`
fork and rejoin	`converge(combine, [A, B])`
compare by a key	`on(cmp, key)`

The full treatment — including where composition came from and how Axioma compares with other languages — is in `resources/docs/claude/FUNCTION_COMPOSITION.md`.

10.11 Broadcasting: one scalar rule, many values

Suppose you already know how to double *one* number. How do you apply that rule to every number in an array? Section 10.2 used `map`. **Broadcasting** gives an ordinary function an elementwise call form, written with a dot immediately before the parentheses:

```
#language axioma/beginner
double: func(x) [2 * x]
nums: [1, 2, 3]
println(double.(nums))          # [2, 4, 6]
println(map(double, nums))      # [2, 4, 6]
println(nums.map(double))       # [2, 4, 6]
```

`double` still describes one number. The call `double.(nums)` applies it at each position and builds an array of answers. The input array is unchanged. For this single-array example, all three expressions give the same result. `nums.map(double)` is a collection-method spelling of `map(double, nums)`; its dot is followed by a **name**, whereas a broadcast dot is followed by **parentheses**. Neither operation means "change `nums`."

The important extension beyond this example is that broadcasting can coordinate **several arguments** and reuse a scalar across all positions. It is useful for numerical sampling, unit conversion, and transformations of records or strings too.

Worked example: calibrating measurements

A sensor produces readings. To calibrate one reading, multiply it by a gain, then add an offset. Follow the design recipe:

1. **Data:** a reading, gain, and offset are numbers.
2. **Signature:** `calibrate : Number × Number × Number → Number`.
3. **Examples:** `(2, 3, 1)` should produce `7`; `(0, 3, 1)` should produce `1`.
4. **Body:** translate the rule directly.

5. **Tests:** check individual readings before broadcasting.

```
#language axioma/beginner
calibrate: func(reading, gain, offset) [reading * gain + offset]
expect("one reading", calibrate(2, 3, 1), 7)
expect("zero reading", calibrate(0, 3, 1), 1)

readings: [0, 1, 2]
println(calibrate.(readings, 3, 1))       # [1, 4, 7]
println(calibrate.(readings, [2, 3, 4], 1)) # [1, 4, 9]
println(calibrate.(readings, [3], 1))    # [1, 4, 7]
println(calibrate.([], 3, 1))            # []
```

In the first call, the gain and offset are **scalars**: each reading uses the same values. The second call supplies one gain per reading. The third supplies a **singleton** array, an array of length one; its one element is reused at every position. Axioma does not first allocate a repeated array such as `[3, 3, 3]` to do this. The final result is a new array.

For one-dimensional inputs, lengths must match or one must be one. Two arrays of lengths three and two do **not** get silently truncated, cycled, or padded. The call fails with a shape error before any elementwise call begins. An empty compatible result makes no calls to the scalar function.

For an ordinary function-call spelling, write `broadcast(calibrate, readings, 3, 1)`. It computes the same values here. We will shortly see why its evaluation boundary differs from nested dotted syntax.

What counts as one element?

A broadcast input that is an Array, Tuple, List, or finite Range has one axis. Strings and dictionaries are scalar values. A nested array is still an array of *elements*; broadcasting does not recursively flatten it or treat its inner arrays as axes:

```
#language axioma/beginner
println(length.(["red", "blue"]))       # [3, 4]
println(length.([[1, 2], [3]]))         # [2, 1]
println(length.("red"))                 # 3
println(abs.(-2..2))                    # [2, 1, 0, 1, 2]
```

The second expression calls `length` on each inner array. The third has only a scalar argument, so it calls `length` once and returns a scalar. You choose the level at which a function works by choosing what its arguments contain.

Broadcasting eagerly creates its result, with a limit of one million output elements. For the unbounded generators from §10.9, first select a finite portion and collect it; broadcasting is not a way to consume an infinite sequence.

Arithmetic with a dot

For familiar arithmetic, dotted operators provide a compact form:

```
#language axioma/beginner
nums: [1, 2, 3]
println(nums .+ 10)             # [11, 12, 13]
println(nums .* 2)              # [2, 4, 6]
println(nums .^ 2)              # [1, 4, 9]
println(nums .+ [10, 20, 30])    # [11, 22, 33]
```

Keep the dot: ordinary array + concatenates, as Chapter 6 explained. Here `.+` adds numbers at corresponding positions.

An important current distinction: dotted arithmetic on Arrays supports numeric scalars and equal-length arrays, but not singleton-array expansion. With `add: func(a, b) [a + b]`, `add.([1, 2, 3], [10])` works; `[1, 2, 3] .+ [10]` fails. Function broadcasting has the more general shape rules. Do not assume that replacing a dotted function with a dotted operator preserves every accepted shape.

Fusion: nested dots share a walk

Suppose you want to square every number and then increment it:

```
#language axioma/beginner
square: func(x) [x * x]
increment: func(x) [x + 1]
nums: [1, 2, 3]
println(increment.(square.(nums)))      # [2, 5, 10]
println(increment.(nums .^ 2))          # [2, 5, 10]
```

These connected dotted expressions use **loop fusion**. For the first element, Axioma computes `square(1)` and then `increment(1)`; for the second, `square(2)` and then `increment(4)`; and so on. Only the final result array is created. There is no intermediate array `[1, 4, 9]` between the two functions.

Fusion saves intermediate collections and their allocation work. It does not promise that every expression runs faster; measure when speed matters. It also does not turn a reduction such as `sum(increment.(square.(nums)))` into that same loop: the dotted part produces its array, and `sum` then consumes it.

Bindings and ordinary calls provide an explicit way to finish one stage before starting the next:

```
#language axioma/beginner
square: func(x) [x * x]
increment: func(x) [x + 1]
nums: [1, 2, 3]
squares: square.(nums)
println(increment.(squares))              # [2, 5, 10]
println(increment.(broadcast(square, nums))) # [2, 5, 10]
println(nums |> square.(_) |> increment.(_)) # [2, 5, 10]
```

All three forms materialize the squared array before incrementing it. Ordinary pipe stages are boundaries too. The underscore is the pipe's argument placeholder: it receives the **whole array**, then the dotted call applies its function element by element. Without a placeholder, the pipe adds the value as the last

argument, just as in §10.10. It does not search inside nested calls for a placeholder.

Why evaluation order belongs in the contract

For pure functions such as `square`, fusion and separate stages give the same values. Functions can also print, draw random numbers, or update state. Then the order is observable. Predict the output before running this example:

```
#language axioma/beginner
inside: func(x) [println("inside", x); x + 10]
outside: func(x) [println("outside", x); x * 2]
println(outside.(inside.([1, 2])))
# inside 1
# outside 11
# inside 2
# outside 12
# [22, 24]
```

If you instead bind `middle: inside.([1, 2])` and then evaluate `outside.(middle)`, both `inside` calls finish first. Use that explicit intermediate when your intent is “finish all of stage one, then start stage two.”

Source expressions and function expressions are evaluated once. The scalar operations inside a fused expression run at each final output position. An inner operation on a singleton can therefore run more than once as its axis expands; its result is not automatically cached. Shape errors are caught before those scalar calls. A callback error stops evaluation without retry or rollback of effects that already happened.

Looking ahead: matrices and tensors

The same mechanism supports values with explicit dimensions:

```
#language axioma/beginner
add: func(a, b) [a + b]
row: matrix([[1, 2]])
column: matrix([[10], [20]])
answer: add.(row, column)
println(shape(answer))          # [2, 2]
println(answer[:])              # [11, 12, 21, 22]
println(add.(answer, [100, 200])[:]) # [111, 212, 121, 222]
```

Here the shapes `[1, 2]` and `[2, 1]` expand to `[2, 2]`. Axes align **from the right**: an array of length two matches the final axis, the columns in this example. The walk is row-major, agreeing with matrix linear indexing. The nested arrays passed to `matrix` are construction data; it is the resulting Matrix, not an ordinary nested array, that has two broadcast axes.

A Matrix input produces a Matrix result; a Tensor input takes precedence and produces a Tensor. Ordinary sequences produce an Array. Matrix cells preserve exact Integer and Rational values. Tensor storage remains floating-point and rejects silent precision loss. Only the final container is constructed in a

fused chain, so intermediate scalar values are not forced through an intermediate Matrix or Tensor conversion. These numeric-container details matter when choosing a materialization boundary; the manual's "Function broadcasting" section is the reference for them.

Execution mode: these examples use the interpreter, including `axioma/beginner`. Function broadcasting and connected dotted expressions requiring fusion are not implemented in the VM; the VM explicitly rejects them. This does not remove its support for standalone dotted arithmetic.

Choose the tool that describes the task. Use `map` for a simple transformation of one sequence, broadcasting to combine scalar rules across compatible shapes, and `reduce` to combine many values into one. Connected dots fuse; ordinary calls, bindings, and pipe stages let you preserve a deliberate boundary.

10.12 Optional: symbolic names for binary functions

A binary function can also have a symbolic spelling. Define and test the ordinary function first, then declare the operator at file top level:

```
minInt :: Integer -> Integer -> Integer
minInt(x, y) = if x > y then y else x
operator (&&&) = minInt
println(8 &&& 3)                  # 3
println((&&&)(8, 3))              # 3
println(reduce((&&&), 100, [8, 3, 9])) # 3
```

The signature above curries its two parameters with arrows. You may also list them with commas — `minInt :: Integer, Integer -> Integer` — which is the same type written the way the definition head `minInt(x, y)` already reads. Arrows say more when currying is the point (`minInt(8)` is a one-argument function either way); commas say more when the arguments go together. Parentheses around a comma list mean something else again: they build one tuple-shaped parameter, not two.

`8 &&& 3` becomes `minInt(8, 3)`. Parentheses around the symbol produce the ordinary function value, so `reduce` uses the same argument checks, contracts and dispatch. The signature belongs to `minInt`; an operator is not a type declaration. Arguments have the target function's usual semantics: this declaration does not create a short-circuiting conditional.

The short form groups left at multiplication precedence, just like a backtick call. Static metadata can choose another existing precedence band:

```
subtract(x, y) = x - y
operator (⊛) = subtract with
    precedence: sum
    associativity: right
end
println(20 ⊛ 5 ⊛ 2)       # 17: 20 - (5 - 2)
println((20 ⊛ 5) ⊛ 2)     # 13: (20 - 5) - 2
```

`⊛` is an unreserved mathematical glyph. The allowed bands, from loose to tight, are `pipe`, `fallback`,

`implies`, `or`, `and`, `equality`, `comparison`, `range`, `sum`, `product`, and `power`. Only grouping changes; choosing `and` or `fallback` does not give a function special evaluation rules. Associativity is `left`, `right`, or `none`. `none` requires parentheses when another operator occupies the same band in a chain. Mixed custom operators with different associativity at one band also need parentheses.

This example is intentionally rejected:

```
subtract(x, y) = x - y
operator (⊛) = subtract with
    associativity: none
end
20 ⊛ 5 ⊛ 2       # SyntaxError: ambiguous operator chain
```

Use ASCII runs drawn from `!%&*+-/<=>?^|~`, or unreserved Unicode mathematical symbols. Letters, digits, emoji identifiers, quotes, dots, colons and brackets do not belong to an operator name. Existing complete operators and reserved glyphs keep their meanings; comment and graph delimiters cannot occur in a custom spelling. Declare before use. With no matching declaration, existing `2+-3` still means `2 + (-3)`. Use spaces when combining operators with existing literal or identifier suffixes; the declared spelling is selected at an ordinary token boundary.

Each source file declares its own syntax, including a file that imports its target function (`operator (&&& ) = Helpers.minInt`). Imports do not install operators. Declarations inside functions, nested modules and runtime blocks are refused. In the native REPL, declarations survive later inputs. Repeating the same symbol, target and metadata with `:source` or `:refresh` is accepted; changing that mapping requires a fresh session or a different symbol. Updating the target function's body uses ordinary rebinding rules. A parse failure installs nothing; a runtime failure before a declaration leaves that declaration unavailable to later inputs. Saved images do not persist the syntax table: source the declarations again after loading.

Direct calls through these operators work in the evaluator and, for a supported target, the VM. Higher-order helpers retain their existing VM limits: for example, `apply` cannot call a VM closure. Guards, contracts, macros and other evaluator-only targets do not gain VM support. Custom unary and postfix operators are outside this binary feature; binary sections are covered in the optional lesson below.

Exercises

Exercise 10.1 — `triple_each`

Use `map` to triple every element of an array. One line.

- `triple_each([])` → `[]`
- `triple_each([1, 2, 3])` → `[3, 6, 9]`

Exercise 10.2 — `keep_positive`

Use `filter` to return only the positive elements (> 0).

- `keep_positive([])` → `[]`
- `keep_positive([1, -2, 3, 0])` → `[1, 3]`

Note: 0 is *not* positive.

Exercise 10.3 — `product`

Use `reduce` to multiply all elements of an array. Watch the starting value — what's the identity for $*$?

- `product([])` → `1`
- `product([4])` → `4`
- `product([2, 3, 5])` → `30`

Exercise 10.4 — `sum_of_squares` with HOFs

Re-do Exercise 7.3, but this time use `map` *and* `reduce` in combination. The structure is "square each, then sum." Two function calls, no `if` in sight.

- `sum_of_squares([])` → `0`
- `sum_of_squares([3])` → `9`
- `sum_of_squares([1, 2, 3])` → `14`

Exercise 10.5 — `all_of`

Define `all_of(pred, arr)` — `true` exactly when *every* element satisfies `pred`. (The dual of `any_of` from §10.7.)

- `all_of(func(x) [x > 0], [])` → `true` (vacuously!)
- `all_of(func(x) [x > 0], [1, 2, 3])` → `true`
- `all_of(func(x) [x > 0], [1, -1, 3])` → `false`

Yes, the empty case really is `true`. Think about what "every element of nothing satisfies P" should mean — there are zero violators, so the claim holds. This is called **vacuous truth** and shows up in mathematical logic too.

Exercise 10.6 (open) — your own `count_if`

Write `count_if(pred, arr)` — how many elements satisfy `pred`?

- `count_if(func(x) [x > 10], [])` → `0`
- `count_if(func(x) [x > 10], [3, 12, 8, 99])` → `2`

Two designs are possible:

1. Write it as a direct recursion (mirror `count_above` from Exercise 7.4 but take the predicate as a parameter).
2. Write it as a one-liner: `filter` then `len`.

Implement *both*. Compare the two solutions side by side. Which one is easier to read? Easier to *change* if the requirements shift slightly? You'll meet this kind of trade-off — direct recursion vs. HOF composition — many times. Neither one always wins.

Exercise 10.7 — comprehension warm-up

Rewrite each of these three functions as a one-line comprehension. No `map`, no `filter`, no recursion — just the comprehension surface.

- `doubles_of([1, 2, 3])` → `[2, 4, 6]`
- `positives_of([1, -2, 3, -4])` → `[1, 3]`
- `squared_positives([1, -2, 3, -4])` → `[1, 9]`

For the third one, write it *two ways*: first with nested `filter`/`map`, then as a comprehension with a filter clause. Time how long each took you to write. Read both back a week from now — which one will be easier to remember?

Exercise 10.8 — Python and Axioma side by side

Each of the three Python comprehensions below has an Axioma equivalent. Write each Axioma form in *two* styles: pipe form and Python pipe-less form. Both should produce the same result.

```
# Python
[x ** 2 for x in range(1, 6)]
{c.upper() for c in "hello"}
{w: len(w) for w in ["red", "green", "blue"]}
```

For the first one in Axioma, use `1..5` as the range. For the second, iterate over `"hello"` and apply `upper(c)` to each character. An explicit array `["h", "e", "l", "l", "o"]` works too.

Bonus: write the first one a *third* way, using `map` over the range. Compare all three. Which reads best?

Exercise 10.9 — walrus to the rescue

The function `cube_root(n)` is expensive — say it takes a millisecond per call. You want all integers in `1..20` whose cube root is greater than 2, paired with that cube root.

A naive comprehension:

```
{(n, cube_root(n)) | n <- 1..20, cube_root(n) > 2}
```

…calls `cube_root` for every candidate and again for each selected result. Rewrite it using a walrus binding (`r: cube_root(n)`) so `cube_root(n)` runs only once per candidate.

Exercise 10.10 — your first dict comprehension

You have an array of names. Build a hash mapping each name to its length:

- `name_lengths(["Anna", "Bob"])` → `{"Anna": 4, "Bob": 3}`
- `name_lengths([])` → an empty hash. It *prints* as `{}`, but don't write it that way in source — a bare `{}` is the empty *set*; the empty-hash literal is `dict()`.

Write it as a dict comprehension — both the pipe form and the Python form. Then write the *same* function using `reduce` over an empty hash (seed the fold with `dict()`, not `{}`), accumulating one key per iteration.

Which version is shorter? Which is easier to *explain* to someone who hasn't seen comprehensions before?

Exercise 10.11 — laziness saves the day

`infinite_set("naturals")` is the genuinely-infinite sequence 1, 2, 3, 4, ... You cannot materialise it — that would never finish. But you *can* build a lazy generator over it and take a finite prefix:

- Produce the first 7 even naturals.
- Produce the first 7 naturals whose square is greater than 100.
- Produce the first 3 naturals divisible by both 3 and 5.

For each task, write the comprehension as `(expr | n <- infinite_set("naturals"), filter)` and then call `gen_take(7, gen)` or `gen_take(3, gen)` on the result. Notice that the generator does not get pulled at all until you call `gen_take` — that's the laziness paying off.

Exercise 10.12 (open) — what the comparison table doesn't say

Open the comparison table in §10.9 above. Pick one row whose Python column is [no] — a feature Python doesn't have. Write a one-paragraph explanation of *why* Python doesn't have it.

You will need to think about what Python's data model *is* and what Axioma's data model adds on top. Some rows are easier than others — the "Mathematical pipe form" row is a pure syntax choice; the "Relational fact queries" row reflects a deeper language design decision.

There is no single correct answer. The exercise is to articulate the trade-off in your own words.

Exercise 10.13 - calibration across several arguments

Write `calibrate(reading, gain, offset)` from §10.11, then use a single dotted call for each task. Keep the function itself scalar.

- Calibrate `[0, 1, 2]` with gain `3` and offset `1`: `[1, 4, 7]`.
- Use gains `[2, 3, 4]` instead: `[1, 4, 9]`.
- Use the singleton gain `[3]`: `[1, 4, 7]`.
- Calibrate an empty input with scalar gain and offset: `[]`.

Explain why gains `[2, 3]` cannot be matched to three readings. Should the function run on any reading before that shape error?

Starter: `exercises/ch10/ex_10_13_broadcast_calibration.ax`.

Exercise 10.14 - square, increment, then total

Given `square: func(x) [x * x]` and `increment: func(x) [x + 1]`, write two versions that produce `[2, 5, 10]` from `[1, 2, 3]`:

1. One connected dotted expression.
2. Two explicit stages using an intermediate binding.

Then use `reduce` to total the first result, giving `17`. Which expressions allocate the squared array? Does adding the ordinary `reduce` call fuse the reduction into the dotted walk?

Starter: `exercises/ch10/ex_10_14_broadcast_fusion.ax`.

Exercise 10.15 (open) - when the same values are not enough

Use the printing `inside` and `outside` functions from §10.11. Predict, then run, all three forms:

- `outside.(inside.([1, 2]))`
- `outside.(broadcast(inside, [1, 2]))`
- `[1, 2] |> inside.(_) |> outside.(_)`

Record the printed order and the final array separately. Explain which version you would choose if every inner operation must finish before any outer operation starts. What would you do if an expensive inner operation on a singleton should run once and its result should then be reused across a larger output?

Starter: `exercises/ch10/ex_10_15_broadcast_order.ax`.

Exercise 10.16 - one function, two spellings

Define `smaller(x, y)`, then give it the operator spelling &&&. Use both the infix form and `(symbol)` as the function passed to `reduce` to find the minimum of `[8, 3, 9]`, starting from `100`. Explain why a symbolic declaration does not avoid evaluating an erroneous argument. For subtraction, compare `20 ⊛ 5 ⊛ 2` with left and right grouping; state what `associativity: none` requires. This optional exercise uses the full language, following §10.12.

Optional: recursive flattening or one-level concatenation

`flatten` recursively descends through nested Arrays, preserving non-Array leaves such as tuples and strings. `flat_map` maps a function and concatenates one level of Array/Tuple/Set results. With `identity`, it provides one-level concatenation rather than recursive flattening:

```
println(flatten([[[1]], [[2]]]))             # [1, 2]
println(flat_map(identity, [[[1]], [[2]]]))  # [[1], [2]]
```

These functions keep their existing runtime behavior; the recursive wording corrects the old one-level description in `flatten` help metadata.

Optional: preserving a caller's type

A broad annotation such as `Number` checks membership in a family. Sometimes a library needs a relationship: two inputs must have the same type, and the result must preserve that type. Axioma already applies this rule to repeated direct constructor parameters. Generic functions bring it to the call boundary.

```
same_choice: func[T](left :: T, right :: T) :: T [left]
println(same_choice(10, 20))
println(same_choice("red", "blue"))
println(error?(try(same_choice(10, 20.0))))
```

```
let choose_integer = same_choice(10)
println(choose_integer(30))
println(error?(try(choose_integer("red"))))
```

The output is `10`, `red`, `true`, `10`, and `true`, one per line. Each complete call chooses its own T. The partial application remembers Integer; supplying a String later violates that choice. Type equality here does not mean equal values: 10 and 20 have the same type.

A return annotation `:: T` also prevents an implementation from returning a String after accepting an Integer. It does not prove that the result equals the input or that the algorithm is correct. Use `[A, B]` when inputs should have independent types. Use `[T of Number]` when T must satisfy a numeric bound, without implicit numeric promotion. Several bounds join with &, and a type argument must satisfy each: `[T of Showable & Numeric]` accepts `21` and refuses `"s"`, which is Showable but not Numeric (Chapter 18 introduces &).

The aliases `fn`, `fun`, and `function` share the anonymous syntax. Named forms put `[T]` after the name, as in `echo[T](x :: T) :: T = x`. A detached signature `echo[T] :: T -> T` can precede any existing lambda spelling. Guarded clauses and pipe groups use one detached signature above the group.

This optional full-language feature checks contracts at call boundaries. `--typecheck` catches certain conflicts, while dynamic calls are checked at runtime. A direct T slot chooses an Array's outer type without constraining its contents. `Array of T` and `List of T` check element relationships; `(A, B, C)` checks tuple components independently. These structures can nest, and bounds such as `T of Number` apply at their leaves. Empty Arrays and Lists supply no element evidence. Generic callbacks share their input/output choices with the enclosing signature. Named ADT annotations use `Box of T` over existing direct parameter fields. Supported VM bodies execute generic functions; ADTs and clause groups still refuse in that backend. Optional §54.10 shows the stricter HM body proof for unbounded generic signatures; bounds and defaults stay host features. Generic recursive calls currently use ordinary recursion to retain return checks.

Exercise 10.18 — one relationship, two independent parameters

Write `keep_right` using `[A, B]`, returning the second argument as B. Partially apply it to an Integer first argument, then call that partial function once with a String and once with an Integer. Explain why both calls should succeed, and how replacing `[A, B]` by a shared `[T]` changes the contract.

Optional: deferred computations and fresh bindings

A normal binding computes its right-hand side now. `declare` creates a fresh, immutable binding that computes its value on the first read. Successful reads share the cached result; failed attempts can retry. Effects of a failed attempt are not rolled back. A later fresh declaration shadows the name without changing a cell captured by earlier code.

```
let seed = 10
declare answer :: Float = seed + 1
let seed = 20
println(answer)                    # 11.0
println(type(answer))              # Float
```

The annotation constrains the eventual value and converts it before caching. An immutable binding cannot be reassigned, but an Array returned from it still supports element mutation. `[42]` remains an Array in a deferred declaration.

A declaration group is recursive: all its names are visible to every RHS in the group. This differs from eager `let`, whose initializer sees the preceding scope. A later fresh declaration does not become visible retroactively.

```
declare left, right = right + 1, 41
println(left)                      # 42
declare ones = stream_cons(1, lazy ones)
println(first(ones, 4))             # [1, 1, 1, 1]
```

Demanding `declare x = x + 1` would revisit an active computation and report a cyclic-demand error. Producing a stream node before asking for its tail makes the second example productive.

Explicit `lazy` produces a deferred value that can be passed or stored without running it. `force` requests the value. A `/lazy` parameter requests its argument when the body reads the parameter; its type describes the eventual value.

```
let pending = lazy (6 * 7)
println(force(pending))            # 42
lazy_twice(body/lazy :: Float) = body + body
println(lazy_twice(21))            # 42.0
```

A mutable handle can instead use `var pending = lazy ...` and later assign a new explicit thunk. `memoize(f)` caches by argument values; generators produce successive items on demand. Neither changes the binding contract of `declare`. `define` records knowledge definitions and remains distinct. These deferred binding/expression/parameter forms are evaluator-only; the VM refuses them.

Exercise 10.19 — demand and capture

In the first example, explain why the result is 11.0 despite the later seed of 20. Replace the first seed with a `var` and the second declaration with an assignment. Predict the result before running it. Then read the answer once before that assignment and predict the two reads. Explain why a typed hole (`let x :: Integer = _`) stores no deferred computation.

Optional: generic List operators and sections

A detached signature belongs to the target function. The operator uses that same contract; it does not need a second signature. ++ has no built-in meaning, but an explicit declaration may now give it a binary

meaning:

```
join_lists[T] :: List of T -> List of T -> List of T
join_lists(xs, ys) = xs + ys
operator (++) = join_lists with
    precedence: sum
    associativity: right
    doc: "Concatenate two Lists with the same element type."
end
println(`[1] ++ `[2] ++ `[3])              # `[1, 2, 3]
println((`[1] ++)(`[2]))                   # `[1, 2]
println((++ `[2])(`[1]))                   # `[1, 2]
println(join_lists(`[], `[2]))             # `[2]
println(error?(try(join_lists(`[1], `["s"])))) # true
```

Without a declaration, ++ still reports the existing increment diagnostic. This supersedes only its previous blanket reservation: no increment operation or built-in concatenation spelling is introduced. `--` remains reserved.

Custom sections `(left OP)` and `(OP right)` are unary functions. Like existing built-in sections, they evaluate the fixed operand on each invocation, then call the declared target with operands in their original order. `(OP)` remains the ordinary target value; `(OP)(left)` uses its ordinary partial application. These are binary sections, not custom unary or postfix operators.

`doc("++")` reports the active declaration's target, precedence, associativity, current target signature and operator documentation (falling back to the target), without calling it. An unbound target is reported as such. Editor hover and completion also include a locally available target signature; imported signatures may be unavailable.

The metadata slots are `precedence`, `associativity`, and `doc`. Fixity values are a fixed parser vocabulary: both `sum` and `"sum"` are accepted, but a variable holding that text is not evaluated. These labels are not runtime enum values. `doc` accepts a literal String, including triple-quoted and raw strings; expressions and interpolated strings are rejected.

`doc("++")` prints the operator's documentation and returns `none`. An omitted, empty, or whitespace-only `doc` falls back to the target function's documentation. Operator documentation does not change `doc("join_lists")`, and it is available even before the target is bound. A compatible redeclaration replaces the doc text (omitting it restores the fallback); changes to the target or fixity still conflict. Editor hover and completion show the declared documentation.

Unannotated Lists remain heterogeneous. Explicit `List of T` recursively checks its elements, including nested Array/List/product structures. Empty Lists supply no evidence for T. Two empty inputs can return an empty List; they cannot justify a nonempty result whose element type is an unresolved T. Partial calls retain established types and recheck captured values, including mutable Arrays nested inside Lists. A bare T still checks only its outer runtime type.

Generic List contracts execute in the VM when their bodies use supported operations. Custom sections follow their ordinary target. `--typecheck` checks supported List evidence; the optional HM lens models List types, literals and its supported higher-order collection operations. A skip remains outside its proof.

Exercise 10.20 — typed concatenation and sections

Using the declaration above, compare `join_lists(`[], `[])`, `join_lists(`[1], `["s"])`, and `join_lists(`[], `["s"])`. Explain when T is chosen. Write a section that appends `[3]` as a List element, and distinguish it from a section that concatenates the List `[3]`. Add a `doc` slot describing the type contract, and explain what happens to `doc("++")` when you remove that slot.

Optional: sections and unary signs

The missing operand is the argument. `(OP right)` means `x => x OP right`; `(left OP)` means `x => left OP x`. These are one-argument functions:

```
println((+ 1)(41))                  # 42
println((- 1)(5))                   # 4
println((1 -)(5))                   # -4
println((* 2)(21))                  # 42
println((/ 2)(10))                  # 5
println((+ "A")("B"))               # BA
println(("A" +)("B"))               # AB
println(map((- 1), [1, 2, 3]))       # [0, 1, 2]
```

Numeric addition and multiplication agree across the two sides. Subtraction, division, powers, comparisons and String concatenation preserve the written operand order and can give different answers.

For a leading +, −, or ∗, separation **after** the operator selects a section. A glued operand retains the unary reading:

```
println((+1))                      # 1
println((-1))                      # -1
println(( +1))                     # 1
println(( + 1)(41))                 # 42
let source_value = 9
let reference = &source_value
println((*reference))              # 9
```

Tabs, newlines and comments also separate the tokens. Bare prefix forms outside a group (`+ 1`, `- 1`, `*reference`) and prefix calls (`+(1)`, `-(1)`, `*(reference)`) retain their meanings. `(+)`, `(-)`, and `(*)` remain operator values. Signed tuples and ascriptions retain their grammar; `(- (1 :: Integer))` explicitly groups a typed section operand.

This replaces the earlier rule that `(+ 1)` and `(- 1)` were numbers and `(* p)` was dereferencing. Remove the space after the operator when migrating an old unary expression. Axioma's formatter preserves the distinction. The section's written operand is evaluated on each invocation, in its lexical environment. Direct section calls and `map` work in the evaluator and VM; other higher-order helpers keep their existing VM limitations.

Exercise 10.21 — which operand is missing?

Predict `(- 2)(10)`, `(2 -)(10)`, `(^ 2)(3)`, and `(2 ^)(3)`. Explain why `(+1)` is a number and `(+ 1)` is a function. Write a section that appends an exclamation mark to a String, and map it over an Array of Strings. Explain why moving the fixed String to the other side changes where the mark appears.

Next chapter — we drop the requirement to *name* every function. Anonymous functions, written `lambda x => x + 1`, let us inline tiny one-off computations without the ceremony of `func(x) [x + 1]`. It's a small syntactic shift with surprisingly large stylistic consequences.

Solutions to selected exercises: Chapter 10 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch10_solutions.md.)*

Optional: collection and clause-function consistency

The fold family also reads persistent Lists. This changes the accepted collection type, not the direction: `foldl` puts the accumulator first and `foldr` puts the element first. `sum`, `product`, `min`, `max`, and `member`/`contains` read Lists too. Operations such as `zip`, `unique`, and `index_of` still require an explicit conversion to an Array.

```
let linked_numbers = `[1, 2, 3]
println(foldl(func(acc, x) [acc - x], 0, linked_numbers)) # -6
println(foldr(func(x, acc) [x - acc], 0, linked_numbers)) # 2
println(max(linked_numbers))                              # 3
println(member(linked_numbers, 2))                        # true
```

A function defined by pattern clauses can partially apply below its narrowest arity, just like an ordinary function. The held arguments are checked together with the remaining arguments when a body will run. Preconditions, postconditions, and termination measures still apply to that complete call. Creating a partial function alone does not run its body or preconditions.

```
func guarded_sum(0, y) [y]
func guarded_sum(x, y) [x + y]
contract guarded_sum { requires: x >= 0, ensures: result == x + y }
let add_two = guarded_sum(2)
println(add_two(3))                                    # 5
println(@try(guarded_sum(-1)(3)) == Error)              # true
println(@try(partial(guarded_sum, -1)(3)) == Error)     # true
```

These clausal functions are evaluator-only; the VM refuses them. Builtins use explicit `partial`, as in `partial(map, (+ 1))`. Exercise 10.18's independent versus shared generic parameters keep the same meaning when the function is written with clauses.

A name binds once within a function head or within each alternative of an `or` pattern. Use separate names and an equality guard for a repeated value. An uppercase parameter such as `X` follows the same rule unless it names a constructor; that distinction is checked when the function is defined.

Strings remain distinct from Lists, but their accessors are consistent: `rest`/`tl` return a String after dropping Unicode characters. `implode` accepts String or Character elements in an Array, Tuple, or List. Convert with `chars` when a mapping function needs individual string elements.

```
println(rest("aλé"))                                    # λé
println(implode(`['a', 'λ', 'é']))                       # aλé
```

Optional: contracts for callbacks and returned functions

A detached signature can describe a function that returns another function. Keep parentheses around that result arrow. Axioma's `Integer, Integer -> Integer` and `Integer -> Integer -> Integer` both describe two parameter slots; `(Integer, Integer) -> Integer` describes one tuple slot. A nested result arrow adds a distinct call boundary whose argument and result remain checked.

```
nested_add :: Integer -> (Integer -> Integer)
nested_add: lambda x y => x + y
println(nested_add(2)(3))
println(annotations(nested_add(2)))

mapping[A, B] :: (A -> B), List of A -> List of B
mapping(f, xs) = map(f, xs)
println(mapping(n => str(n), `[1, 2]))

empty_values[T] :: () -> List of T
empty_values() = (`[])
println(len(empty_values()))
```

The output is `5`, `Integer -> Integer`, `["1", "2"]`, and `0`. Calling `nested_add(2)(1.5)` fails the inner Integer contract. Mapping fixes A from the elements and B from the callback's declared signature or actual results. An empty collection supplies no element evidence. Returning an empty List is valid for every T; returning `1` from a function promising `() -> T` is not universally valid.

These examples run in both backends. Generic callback elements can also live in Arrays, Maps, immutable Lists/tuples and specialized ADTs. Mutable callback slots receive temporary guards during annotated calls; cleanup preserves user writes, and extracted callbacks retain their checked values. Keyed Sets/Bags remain unsupported for callable adaptation. Concrete annotated Arrays enforce callable contracts on existing slots and checked writes. Host ADT annotations such as `Box of T` inspect existing parameter fields. Immutable typed ADTs and `match` receive the static proof in §54.11; ADTs remain outside the VM. Existing module/export interfaces receive closed HM checking in §54.13. Reusable module signatures and opaque ascription are introduced in §54.14; §54.15 adds type equations and references to existing types; §54.16 adds fixed-type value requirements. Higher-rank polymorphism, functors and sharing constraints are separate language features.

Exercise 10.22 — composing two relationships

Write `compose_value[A, B, C]` with detached parameter types `(B -> C)`, `(A -> B)`, and `A`, returning C. Apply it to `str`, a function that adds one, and 2. Then explain why a shared `[T]` signature requiring every input and output to be T cannot describe this Integer-to-String composition.

Optional: named signatures and nested data fields

A transparent type alias may name the entire detached arrow signature. The alias resolves where the function is defined; its argument and result are checked on calls. Parenthesized tuple parameters still mean one argument.

```
type IntEndo = Integer -> Integer
increment :: IntEndo
increment(x) = x + 1
println(increment(4))

data Batch[T of Number] = Batched(Array of T)
println(Batched([1, 2]))
```

The output is `5` and `Batched([1, 2])`. The constructor uses one T for every element; bounds apply to each leaf. `Batched([1, "bad"])` is rejected. Direct and nested fields share choices; empty collections provide no evidence. Recursive ADT fields work, while cyclic structural values refuse safely. Construction choices remain erased, so later shared mutations are rechecked at applicable call boundaries without rollback. The ADT example uses the evaluator; the named function signature also runs in the host VM. Neither extension enlarges the closed HM proof to all host-language constructs.

Exercise 10.22 keeps its explicit generic binder: naming a concrete arrow alias does not introduce higher-rank or parameterized type aliases.

Optional: shared containers of callbacks

```
invoke_first[T] :: Array of (T -> T), T -> T
invoke_first(fs, x) = fs[1](x)
callbacks: [x => x]
println(invoke_first(callbacks, 7))
println(invoke_first(callbacks, "hello"))

invoke_named[A, B] :: Map of String to (A -> B), A -> B
invoke_named(fs, x) = fs.convert(x)
println(invoke_named({convert: x => str(x)}, 12))
```

The output is `7`, `hello`, and `12`, in the evaluator and host VM. The two Array calls make independent choices for T. During an annotated call, aliases share checked callable slots, and insertions/replacements receive the same contract. Cleanup runs on return or error and preserves user mutations. An extracted callback retains its checked value after cleanup. Outside an active call, generic containers keep boundary

checking; they do not acquire permanent restrictions from that call's type variables. Concrete Array annotations continue to enforce persistent write contracts.

Exercise 10.23 — preserving a checked mutation

Write `add_callback[T]` taking `Array of (T -> T)` and T, returning T. Append an identity callback, then invoke that new callback with the supplied value. Show that the caller sees the appended slot after the function returns. Explain why a callback returning a String would violate the contract when the supplied value is an Integer.

Optional: generic union alternatives

A union can include a signature's type variables. Concrete alternatives are tried first, then alternatives containing variables in their written order. A successful concrete alternative leaves the variables free for other inputs.

```
fallback_value[T] :: (T | String), T -> T
fallback_value(candidate, value) = value
println(fallback_value("missing", 7))
residual: fallback_value("missing")
println(residual(true))
println(residual(8))

choose_pair[T] :: ((T, Integer) | (String, T)), T -> T
choose_pair(pair, value) = value
println(choose_pair(("missing", true), false))
```

The output is `7`, `true`, `8`, and `false`, in the evaluator and host VM. `"missing"` uses the concrete String alternative, so it does not fix T to String. The partial function can therefore choose Boolean and Integer on its independent calls. In `choose_pair`, the first alternative tentatively chooses String for T, but fails because the second tuple slot is Boolean. That trial's choice is discarded; the second alternative chooses Boolean for T.

A successful alternative is not reopened if a later argument conflicts. For `(T | U), T, U -> T`, the first generic alternative is T: passing `1, "s", 1` fails the shared T requirement, even though choosing U for the first argument could have made that call fit. Use a concrete alternative or separate functions when the distinction matters.

Unions also compose with Array/List/Set/Bag/Map leaves, products, specialized ADT fields, callback arguments and callback results. For example, `Array of ((T -> T) | None)` permits missing callback slots while checking present callbacks. Existing temporary mutable-container guards still apply. Callable alternatives are chosen using their known signatures and arity; selection never executes a callback. Equally applicable generic alternatives use written order, including opaque callbacks; invocation errors do not retry another alternative. Generic bounds still apply to every chosen T leaf.

An empty input may leave T unresolved. A result `T | None` may then return `none`, but cannot invent an Integer for T. Callback outputs can supply fresh output-type evidence as in earlier examples. Static

checking diagnoses known conflicts and accepts mixed collection literals under explicit union element annotations at generic call sites. Uncertain runtime values remain checked on calls. Generic unions are outside the closed HM proof, and existing VM refusals for ADTs and other unsupported body forms remain.

Exercise 10.24 — a missing value without a type choice

Write `keep_optional[T]` taking `Array of (T | None)` and T, returning T. Return the second argument. Call it with `[none, 1]` and 2, then partially apply it to `[none]` and use the partial with both String and Integer values. Explain why `none` does not choose T and why `[1]` would fix T to Integer.

Optional: spreading values into a call

A collection can hold the positional arguments for a function. Prefix it with `...` inside the call to pass its elements as separate arguments:

```
weighted: func(a, b, scale = 1) [(a + b) * scale]
values: [2, 3]
println(weighted(...values))
println(weighted(...values, scale: 10))
println(weighted(1, ...`[2], scale: 4))
println(weighted(...(4, 5)))
```

The output is `5`, `50`, `12`, and `9`. Array, List and Tuple sources are accepted. Empty sources supply no arguments, and several spreads may occur in one call. A nested collection stays one argument. Sets, Dictionaries, Strings, Ranges, Generators and Sequences require an explicit finite conversion first; a Dictionary does not supply labels through positional `...`. The optional named-spreading section below gives its explicit form. Ordinary `apply(f, args)` remains useful when the callable and its argument collection are both data.

Each supplied expression runs once in source order. A spread copies its outer elements immediately; later mutations do not change the arguments already collected, but nested objects keep their shared identity. Positional arguments fill fixed parameters from the left regardless of where labels occur in source. A duplicate label or a positional/label collision errors. Missing defaults run in declaration order inside the call frame; variadic tails receive an Array. Named calls honor the same early returns and result annotations as positional calls.

Call spreading currently runs in the evaluator. The VM refuses it, and closed HM inference does not prove its dynamic arity. The host checker visits source expressions and diagnoses known invalid source types; expanded arity and argument types remain runtime checks. Broadcast, macro, lazy-parameter and syntax/environment builtin calls do not accept spreading in this first version.

Exercise 10.25 — a spread snapshots the outer collection

Predict both the collected arguments and the final value of `items`. Explain why the two spreads can contribute different numbers of arguments.

```
items: [1]
extend_items: func() [push(items, 2); 0]
pack: func(...values) [values]
println(pack(...items, extend_items(), ...items))
println(items)
```

Optional: named relationships and checked input

A parameterized transparent alias can name a recurring function relationship:

```
type Endo[T] = T -> T
type IntEndo = Endo of Integer
bump :: IntEndo: n => n + 1
println(bump(41))
```

The output is `42`. `Endo of Integer` substitutes Integer for T; it creates no new value constructor. Its runtime argument and result contracts still apply. This host-language extension uses existing `of` applications. It does not add parameterized aliases to the closed HM language. Wrong arity, unapplied aliases in annotations and cyclic aliases are errors.

When receiving data, separate validating today's value from attaching a contract that governs future writes:

```
type Batch = {count :: Integer}
batch: checked({count: 2, label: "morning"}, Batch)
println(batch.count)
println(batch.label)
println(type(checked(3, Float)))
```

The output is `2`, `morning`, and `Integer`. `checked` preserves extra fields, object identity and numeric kinds. It does not attach the schema to future writes; `::` supplies that ongoing contract. Invalid fields are runtime errors. The optional `--strict-external` checker asks for validation or an explicit guard before typed use of decoded JSON and foreign eval results. It is a local check, not a proof across arbitrary unannotated functions.

The host checker also follows primitive `is` tests through Boolean conjunction, disjunction and early returns. Unknown calls and writes conservatively invalidate facts. These static conveniences leave runtime checking in place. They borrow TypeScript's useful flow discipline while retaining Axioma's runtime semantics.

Exercise 10.26 — an alias and a data boundary

Define `Endo[T]` and specialize it to String. Bind a function that appends `"!"` and print its result on `"ready"`. Validate a dictionary with required Integer field `count` and extra String field `label`, then print the extra field. Explain why validation preserves that field and why `checked(3, Float)` still has runtime type Integer. Would validation alone restrict later dictionary writes?

Optional: spreading named arguments

The colon distinguishes named expansion from positional expansion: `f(...args, ...: options)`. The source of `...: options` must be a Dictionary whose String keys name parameters. Keep a space after the colon: `...:options` already means positional spreading of the get-word `:options`.

```
type ReportOptions = {path :: String, overwrite :: Boolean}
options :: ReportOptions = {"path": "report.csv", "overwrite": false}
report: func(value, path, overwrite = false) [
    println(path)
    println(value)
    println(overwrite)
]
report(42, ...: options)
```

The output is `report.csv`, `42`, and `false`, on separate lines. The schema lets the checker describe a stored dictionary whose fields have different types. Inline option dictionaries can also contain different value types; nested collections keep their normal checks. Expanded labels and parameter contracts are checked at runtime. This form runs in the evaluator; the VM and closed HM refuse it, as they do positional call spreading.

Each spread expression runs once in written call order and snapshots its outer bindings. Later mutation of the source cannot replace a value already supplied; nested mutable values remain shared. Dictionary entry evaluation itself keeps its existing rules. `dict()` supplies no labels, whereas `{}` is a Set and refuses. Unknown names and duplicate assignments error. For example, supplying `overwrite` through both `options` and an explicit label errors; their order does not choose an override. Ordinary explicit-label checker gaps remain as documented in the call reference.

Exercise 10.27 — named spread snapshots and collisions

Create options with Integer field `a` set to 1. Call a two-parameter function with a named spread first, followed by an expression that changes options.a to 9 and supplies b = 2. Predict the two bound values and the final options.a. Then explain why supplying a explicitly as well must error.

Optional: one function, several contracts

Some functions work on more than one kind of input and keep a different promise for each. A single arrow cannot say that: `Integer | String -> Integer | String` accepts either input, but it would also allow an Integer in and a String out. An intersection of arrows, written with &, keeps each promise separately:

```
echo :: (Integer -> Integer) & (String -> String)
echo(x) = x
println(echo(5))
println(echo("hi"))
println(try(echo(true)) is Error)
```

The output is `5`, `hi`, and `true`. At each call, an arm applies when the argument satisfies its parameter. `true` satisfies neither arm, so the call is refused before the body runs: the caller made the mistake. Otherwise the body runs once, and its result must satisfy the result of every arm that applied. The arms never choose a body: this is still one function with one body. To run different code for different types, use the guarded clauses of §5.9 or a `match`.

A body that breaks a promise is caught at the result:

```
shout :: (Integer -> Integer) & (String -> String)
shout(x) = if x is String then upper(x) else str(x)
println(shout("hi"))
println(try(shout(3)) is Error)
```

The output is `HI` and `true`. The call `shout(3)` meets the Integer arm, so its result must be an Integer. `str(3)` is a String, so this time the function is at fault, and the error is a return type error. In this version, every arm takes the same number of parameters, and none of them takes a function. Intersections of arrows also run under `--vm`, but these examples use `try`, which `--vm` does not compile, so they run in the evaluator.

Exercise 10.28 — a promise per input

Write `double_up` with the signature `(Integer -> Integer) & (String -> String)`. It doubles a number and repeats a string twice. Call it on `21` and `"ab"`. Show that `double_up(1.5)` fails, and say whether the caller or the function is at fault. Then explain why the signature `Integer | String -> Integer | String` would accept a body that returns `str(x)` for every input.

Chapter 11 · Lambda

What this chapter is. A small syntactic upgrade. We stop giving a name to every one-off function. The result reads better, especially inside `map`/`filter`/`reduce` calls.

Heads up: this is the one chapter where we *leave* the `axioma/beginner` subset. The new `lambda x => expr` form is not allowed in beginner mode — and that's deliberate, because the beginner subset wants you to *practise* naming functions first. Now that you can, we relax the rule.

11.1 Why anonymous?

Look back at the examples from Chapter 10. They're full of disposable functions:

```
#language axioma/all
nums: [3, 1, 4, 1, 5, 9, 2, 6]
is_big: func(x) [x > 4]
println(filter(is_big, nums))
```

The name `is_big` is used at one call site. Naming it is optional: `filter(func(x) [x > 4], nums)` already passes an anonymous function. The lambda form is a shorter spelling for the same idea.

Most languages — Python, JavaScript, Scheme, Haskell, anything in the ML family — let you write the function *inline* without naming it. Axioma's name for the unnamed-function syntax is **lambda**, after the Greek letter from Alonzo Church's 1936 λ-calculus.

```
nums: [3, 1, 4, 1, 5, 9, 2, 6]
println(filter(lambda x => x > 4, nums))   # [5, 9, 6]
```

Same result, no intermediate variable. The `lambda x => x > 4` *is* the function — it has no name, but you can pass it anyway.

Use `#language axioma/all` for this chapter. If you combine snippets into one file, remove an earlier beginner pragma rather than expecting a blank line or a new code fence to change the language mode. Beginner mode forbids `lambda` (it nudges you toward naming). Once you graduate from naming-every-function (as you have, by finishing Chapter 10), you're allowed to drop the pragma and use lambda.

11.2 Syntax — three forms

Axioma's `lambda` syntax has three forms:

```
# One argument, no parens needed:
lambda x => x * 2

# Multiple arguments, parens required:
lambda (x, y) => x + y

# Zero arguments — empty parens:
lambda () => "hello"
```

For comparison, here are the equivalent `func` forms (which *do* work in beginner mode):

```
func(x) [x * 2]
func(x, y) [x + y]
func() ["hello"]
```

Pick whichever reads better in context. The general rule:

- **For inline one-off functions inside HOF calls, lambda reads better** — `map(lambda x => x * 2, ...)` is shorter and doesn't visually nest brackets.
- **For functions you'll reference more than once, name them with `func`** — `area: func(r) [3.14159 * r * r]` and then call `area(5)`, `area(10)`. The name is the documentation.

11.3 Worked example — `sort_by_key`

Problem. Sort a list of records (think: an array of `{name, age}` hash literals) by a key the caller specifies.

Axioma has sorting builtins (`sort`, `sort_by`) — but building the machinery ourselves is the point of this section: we'll implement insertion sort, then parameterize it on the comparison.

1. Data. An array of comparable items (numbers, strings, or hashes with a numeric/string field).

2. Signature.

```
# sort_by:  (Func[A, A -> Boolean], Array[A]) -> Array[A]
```

The first parameter is a *function* — specifically, a two-argument function that returns `true` if its first argument should come *before* its second.

3. Examples.

```
# sort_by(lambda (a, b) => a < b, [3, 1, 4, 1, 5])
#   = [1, 1, 3, 4, 5]
# sort_by(lambda (a, b) => a > b, [3, 1, 4, 1, 5])
#   = [5, 4, 3, 1, 1]    (reversed)
```

4–5. Template / body. Insertion sort: walk the input, inserting each item into its sorted position in the accumulator.

```
insert_sorted: func(less, x, sorted) [
  if len(sorted) == 0 then [x,]
  else if less(x, first(sorted)) then [x | sorted]
  else [first(sorted) | insert_sorted(less, x, rest(sorted))]
]

sort_by: func(less, arr) [
  if len(arr) == 0 then []
  else insert_sorted(less, first(arr), sort_by(less, rest(arr)))
]
```

(The trailing-comma singletons `[x,]` are the §6.6 spelling — these sit at the start of branches, where a bare `[x]` would be read as a block.)

(We call our helper `insert_sorted`, not `insert`. `insert` is a reserved built-in name in Axioma — it inserts facts into the logic-programming knowledge base. Trying to redefine it here would silently shadow that builtin. The convention "if it errors with `must be a string relation name`, your name clashes with a builtin" is one of those papercuts every Axioma programmer learns once.)

6. Tests.

```
println(sort_by(lambda (a, b) => a < b, [3, 1, 4, 1, 5]))
# [1, 1, 3, 4, 5]
println(sort_by(lambda (a, b) => a > b, [3, 1, 4, 1, 5]))
# [5, 4, 3, 1, 1]
```

That's *the whole point* of accepting `less` as an argument: one implementation, two behaviours. Want descending? Flip the comparison. Want sort-by-second-coordinate? Pass `lambda (a, b) => a[2] < b[2]`. Want sort-by-string-length? Pass `lambda (a, b) => len(a) < len(b)`. No rewrite needed.

11.4 When *not* to use lambda

Lambda isn't universally better. Three cases where naming wins:

Reuse. If the function is used more than once, name it. `is_positive: func(x) [x > 0]` and then `filter(is_positive, arr1)` *and* `filter(is_positive, arr2)`. The name documents intent; copy-pasting the lambda hides the duplication.

Complexity. If the body has more than one line, name it. A five-line lambda buried inside a `reduce` call is unreadable. Pull it out:

```
# DON'T:
reduce(lambda (acc, x) => [if x > 100 then acc + x else acc], 0, nums)

# DO:
add_if_big: func(acc, x) [
  if x > 100 then acc + x else acc
]
```

```
reduce(add_if_big, 0, nums)
```

(The first version is also a syntax trap — putting an `if` in a lambda body without brackets can confuse the parser. Named `func` bodies have explicit `[...]` and don't suffer this.)

Recursion. A lambda can't easily refer to itself by name (there *is* no name). If your function needs to call itself, give it a name with a binding:

```
# Won't work — lambda has no name to recurse with:
# map(lambda n => if n <= 1 then 1 else n * RECURSE(n-1), [1,2,3,4,5])

# Works:
fact: func(n) [ if n <= 1 then 1 else n * fact(n-1) ]
map(fact, [1, 2, 3, 4, 5])    # [1, 2, 6, 24, 120]
```

11.5 Lambda + closure — what's actually going on

A lambda doesn't only carry its *code* — it also carries any variables from the enclosing scope. That packaged-together pair of code and captured environment is called a **closure**.

```
make_multiplier: func(k) [
  lambda x => x * k
]
times7: make_multiplier(7)
times11: make_multiplier(11)
println(times7(3))     # 21
println(times11(3))    # 33
println(times7(3))     # 21  (still 7 — closures don't leak)
```

Each call to `make_multiplier` produces a *new* closure that remembers its own `k`. They don't interfere. This is one of the deepest ideas in programming: a function can carry data with it, *without* being a class with instance variables.

We'll use closures in Chapter 12 for cleaner local helpers, and Chapter 17 (Streams) leans on them to build lazy infinite sequences.

11.6 Reflection — lambda is small but pivotal

You haven't gained any new computational power in this chapter. Every lambda is equivalent to a `func` bound to a name. What you've gained is *fluency*. Code that uses HOFs is now short enough to take in at a glance:

```
# Adult ages from a list of people, sorted descending
people: [{age: 17}, {age: 32}, {age: 21}]
adult_ages: map(lambda p => p.age,
```

```
                filter(lambda p => p.age >= 18, people))
println(sort_by(lambda (a, b) => a > b, adult_ages))  # [32, 21]
```

Read that aloud — "sort, filtering for adults, of the ages of people." The structure of the prose matches the structure of the code, line by line. That's the prize.

Exercises

Exercise 11.1 — Rewrite Chapter 10 with lambda

Take your solution to Exercise 10.1 (`triple_each`) and rewrite the inline function as a `lambda`. Same for 10.2 and 10.3. Don't introduce any named functions.

Exercise 11.2 — `negate_all`

Use `map` and a lambda to negate every element of a list.

- `negate_all([])` → `[]`
- `negate_all([1, -2, 3])` → `[-1, 2, -3]`

Exercise 11.3 — `split_by`

Write `split_by(pred, arr)` that returns a *tuple* `(yes_arr, no_arr)`: the elements that satisfy `pred` and the elements that don't. (One pass through the array — don't call `filter` twice.)

- `split_by(lambda x => x > 0, [])` → `([], [])`
- `split_by(lambda x => x > 0, [1,-2,3,-4])` → `([1, 3], [-2, -4])`

Hint: recur on `rest(arr)` and let the recursive call return the pair `(yes_arr, no_arr)`; then decide which half `first(arr)` belongs in. Tuple syntax: `(a, b)`; tuple access is 1-based, like arrays: `t[1]`, `t[2]`.

(We call it `split_by` instead of the conventional name `partition` because `partition` is a reserved keyword in Axioma — it builds disjoint concept hierarchies. Same papercut as `insert` in §11.3.)

Exercise 11.4 — build `compose` yourself

Axioma already ships `compose`, plus the `>>` / `<<` / `∘` operators (see §10.10). Build one anyway — under a different name, so you don't shadow the builtin — because writing it is how you see that composition is *just* a lambda returning a lambda.

Define `after(f, g)` as a function that returns a *new function*: "first apply `f`, then apply `g`."

- `after(lambda x => x + 1, lambda x => x * 2)(5)` → `12` (5 + 1 = 6, then 6 * 2 = 12)

Implement `after` as a one-line lambda whose body is a lambda.

Then check your work against the builtin — they should agree, because they are the same function:

```
add1: lambda x => x + 1
dbl:  lambda x => x * 2

after(add1, dbl)(5)          # your version
compose(add1, dbl)(5)        # the builtin — 12
(add1 >> dbl)(5)             # the operator — 12
```

Note the direction. `compose(f, g)` and `f >> g` apply `f` **first**, reading left to right. The mathematician's `g ∘ f` also applies `f` first, but writes `g` on the left — so `f >> g` and `g << f` are the same function with the arguments swapped. Getting that flip right is most of the exercise.

Exercise 11.5 — `sort_descending` in one line

Reuse the `sort_by` from §11.3. Define `sort_descending = sort_by(lambda (a, b) => a > b, ...)`. Try to write it as a function that takes only an array (i.e., apply `sort_by` partially):

- `sort_descending([3, 1, 4, 1, 5])` → `[5, 4, 3, 1, 1]`

Hint: `sort_descending: lambda arr => sort_by(... , arr)`.

Exercise 11.6 (open) — your own `pipe`

Many functional languages have a "pipe" operator (in F# it's `|>`, in Elixir it's the same). It's syntactic sugar for "feed the result of the left expression into the function on the right." Define a function `pipe(x, fs)` that takes a value `x` and an array of functions, and applies them left-to-right:

```
pipe(5, [lambda x => x + 1,
         lambda x => x * 2,
         lambda x => x - 3])
   = (((5 + 1) * 2) - 3)
   = 9
```

Implement this with `reduce`. Then write a small test that chains four or five lambdas — and *try* to read the result aloud. Does pipeline-style code feel different from nested function calls?

Next chapter — we still have one cleanup left. Code that uses HOFs heavily tends to accumulate tiny helper functions at the top of every file. Chapter 12 shows how to **localize** them: define them *inside* the function that uses them, where they're hidden from the rest of the program.

Solutions to selected exercises: Chapter 11 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch11_solutions.md.)*

Chapter 12 · Local Definitions

> **What this chapter is.** A scope-management upgrade. We move helper functions and intermediate values *inside* the function that uses them. The change is small, but it ends Part III with code that is genuinely *modular* — each function carries its own private workshop.
>
> Back in `axioma/beginner` mode for this chapter — locals are beginner-safe.

12.1 The problem

At the end of Chapter 10, our pipeline looked like this:

```
#language axioma/beginner
nums: [3, 11, 14, 17, 22, 8, 25, 30]
above_10: filter(func(x) [x > 10], nums)
evens: filter(func(x) [x % 2 == 0], above_10)
total: reduce(func(a, b) [a + b], 0, evens)
answer: total / len(evens)
println(answer)
```

Three of those bindings — `above_10`, `evens`, `total` — are *intermediate steps* of one computation. They have no use outside of computing `answer`, yet they live at the top level, visible to every other piece of code that loads this file.

That's a smell. The HtDP design rule says: **the scope of a name should be no larger than its use.** Names that are only needed inside one function should live inside that function.

12.2 The fix — local bindings inside a function body

You've actually been using locals informally since Chapter 7; this chapter just names the pattern. A function body in Axioma can have its own local bindings, *separated by semicolons*, with the final expression being the function's return value:

```
#language axioma/beginner
pipeline: func(nums) [
  above_10: filter(func(x) [x > 10], nums);
  evens: filter(func(x) [x % 2 == 0], above_10);
  total: reduce(func(a, b) [a + b], 0, evens);
  total / len(evens)
]
```

```
println(pipeline([3, 11, 14, 17, 22, 8, 25, 30]))   # 22
```

`above_10`, `evens`, `total` are *invisible* outside of `pipeline`. Try to reference them at the top level and you get an "undefined word" error. They're born when `pipeline` is called, used to compute the answer, and discarded when `pipeline` returns.

That's **lexical scoping**: the *region of the source code* where a name is visible. Locals are visible only inside the brackets they're declared in.

12.3 Worked example — BMI revisited

Recall the BMI exercise from Chapter 3:

```
# Naive version — exposes h_squared at top level
h_squared: 1.75 * 1.75    # used only inside bmi!
bmi: func(weight, height) [ weight / (height * height) ]
```

The `h_squared` precomputation only made sense as part of computing `bmi`. With locals, we hide it:

```
#language axioma/beginner
bmi: func(weight_kg, height_m) [
  h_squared: height_m * height_m;
  weight_kg / h_squared
]
println(bmi(70, 1.75))    # 22.857...
```

Now `h_squared` is computed once per call (not redundantly twice as `height_m * height_m * weight_kg`-style code would do), and it doesn't pollute the top level.

This is HtDP's *third* design recipe payoff (after "examples catch errors early" and "data shape drives template"): **locals let you give meaningful names to intermediate results without paying a global namespace cost.**

12.4 Worked example — average plus standard deviation

Now we're computing *two* statistics, and they share work. The population standard deviation is the square root of the mean squared distance from the mean. The local binding computes that mean just once. Both functions require a non-empty numeric array:

```
#language axioma/beginner
mean: func(arr) [
  reduce(func(a, b) [a + b], 0.0, arr) / len(arr)
]
stddev: func(arr) [
  m: mean(arr);
```

```
  diffs: map(func(x) [(x - m) * (x - m)], arr);
  variance: reduce(func(a, b) [a + b], 0.0, diffs) / len(arr);
  sqrt(variance)
]
println(mean([4, 8, 6, 5, 3, 7]))     # 5.5
println(stddev([4, 8, 6, 5, 3, 7]))   # 1.707825127659933
```

(Why `0.0` instead of `0` as the `reduce` seed? Display, not correctness: with `0` the arithmetic stays exact, and the mean comes out as the exact rational `11/2` rather than `5.5` — the right value, in fraction form. Seeding with `0.0` keeps the running total a float, so the statistics print as the decimals you expect from numeric code.)

Two things to notice:

1. **`m` is the local mean** — captured by the `map` lambda inside `stddev`. That lambda *closes over* `m` (Chapter 11's closure idea applied to a local). Without closures, we'd have to pass `m` as a parameter to a helper, and the `map` API doesn't support that.
2. **`diffs` and `variance` are local** — they exist for the duration of one `stddev` call and then vanish. If you load this file and ask the REPL for `diffs`, you'll get "undefined word."

12.5 Local *functions*

So far the local definitions have been values (numbers, arrays). But `func(...)` is itself a value — so you can define local functions, and that's where this technique really earns its keep:

```
#language axioma/beginner
max_in_each_row: func(matrix) [
  row_max: func(row) [
    if len(row) == 0 then 0
    else if len(row) == 1 then first(row)
    else [
      rest_max: row_max(rest(row));
      if first(row) > rest_max then first(row) else rest_max
    ]
  ];
  map(row_max, matrix)
]
println(max_in_each_row([[1, 4, 2], [9, 3, 5], [7, 8, 0]]))
# [4, 9, 8]
```

`row_max` is a private helper. It's recursive (it calls itself), and it lives entirely inside `max_in_each_row`. No other code can use it, no other code can shadow it, and the name "row_max" doesn't appear at the top level.

This is the same pattern HtDP calls a **local helper**. In a larger program you'd have many such helpers — each function keeps its own toolbox.

12.6 What if I want to share?

Sometimes a helper is genuinely useful in more than one place. In that case, pull it back out to the top level — but give it a *clearly general* name and a comment, so it doesn't look like a leftover scaffold:

```
#language axioma/beginner
# Reusable: returns the maximum of a non-empty numeric array.
max_of: func(arr) [
  if len(arr) == 0 then none
  else if len(arr) == 1 then first(arr)
  else [
    r: max_of(rest(arr));
    if first(arr) > r then first(arr) else r
  ]
]

max_in_each_row: func(matrix) [
  map(max_of, matrix)
]
```

The rule is: **start local, promote when there's a second caller.** Don't pre-emptively make everything top-level "in case someone uses it later." That clutters the namespace.

12.7 Block expressions — the bracket syntax for "do these in

order, return the last one"

You've been using `[...]` brackets to delimit function bodies, and you may have noticed they're more flexible than that. Inside any expression context, `[stmt; stmt; expr]` runs the statements in order and returns the final expression's value.

We used this in §12.5's `row_max` helper:

```
else [
  rest_max: row_max(rest(row));
  if first(row) > rest_max then first(row) else rest_max
]
```

The whole `[...]` is the `else` branch. It first computes `rest_max`, then evaluates the `if` to decide which value to return. Without the brackets, you couldn't have a local binding inside an `if` arm.

This bracket form has a name: a **block expression**. It's the local-scope equivalent of REPL multi-line input.

12.8 Reflection — the modularity story

Look back at Parts I, II, and III as a whole:

- **Part I** taught you to compute single values.
- **Part II** taught you to compute over collections.
- **Part III** taught you to *abstract* the computation — parameterize it (HOFs), inline the one-offs (lambdas), hide the intermediate steps (locals).

Each step *reduced what the rest of your program has to know about a function's internals.* By the end of Chapter 12, a function can present a simple input-output contract on the outside while doing arbitrarily complex helper work inside.

That's modularity. It's the single most important property of a large codebase: each piece should be replaceable without the rest having to change. HOFs + closures + locals are the tools that make modularity possible.

Part IV puts this to work on harder data — mutually recursive records, generative recursion (the algorithmic kind, not the data-shape kind). Bring all three tools.

Exercises

Exercise 12.1 — `bmi_category`

Combine BMI with a category label ("underweight", "normal", "overweight", "obese"). The thresholds are 18.5, 25, 30. Make the BMI computation a *local* — the caller shouldn't be able to see it.

- `bmi_category(50, 1.75)` → `"underweight"` (BMI ≈ 16.3)
- `bmi_category(70, 1.75)` → `"normal"` (BMI ≈ 22.9)
- `bmi_category(85, 1.75)` → `"overweight"` (BMI ≈ 27.8)
- `bmi_category(100, 1.75)` → `"obese"` (BMI ≈ 32.7)

Exercise 12.2 — `triangle_area` via Heron's formula

Heron: with sides `a, b, c` and semi-perimeter `s = (a+b+c)/2`, the area is `sqrt(s*(s-a)*(s-b)*(s-c))`. Compute `s` once as a local; reference it three times in the body.

- `triangle_area(3, 4, 5)` → `6.0` (the classic 3-4-5 right triangle)
- `triangle_area(5, 5, 6)` → `12.0`

Exercise 12.3 — `running_average` with a local helper

Given an array `[a, b, c, d, …]`, return the array of running averages: `[a, (a+b)/2, (a+b+c)/3, …]`. Use a local helper function that takes an index `i` and computes the average of the first `i` elements.

- `running_average([])` → `[]`
- `running_average([4])` → `[4]`
- `running_average([2, 4, 6])` → `[2, 3, 4]`

Hint: `running_average([2, 4, 6])` should call the local helper three times: at i=1, i=2, i=3.

Exercise 12.4 — refactor your pipeline

Take the pipeline from §10.6 (average of even numbers > 10). Wrap it in a function `mean_of_big_evens(nums)` whose body uses locals for every intermediate value. The top-level code should have nothing but `mean_of_big_evens: ...` and a few `println` tests.

Exercise 12.5 — `count_words_per_line`

Given an array of strings (each string is a "line"), return an array of integers (the number of words per line, where a word is a maximal non-space run). Use a *local* helper function `word_count(s)`. (Hint: `split(s, " ")` splits a string on spaces; filter out empty strings to handle multiple spaces.)

- `count_words_per_line([])` → `[]`
- `count_words_per_line(["hello world", "the", "a b c d"])` → `[2, 1, 4]`

Exercise 12.6 (open) — closures as a substitute for state

Write `make_counter()` that returns a *function*. Each time you call that function, it should return the next integer (1, 2, 3, ...). Hint: think about what the closure captures.

```
c: make_counter()
c()    # 1
c()    # 2
c()    # 3
d: make_counter()
d()    # 1     (independent counter)
c()    # 4     (c's count continues)
```

Two questions for after you have it working:

1. Is your counter *pure* (does it return the same answer on the same input every time)? Why or why not?
2. The beginner *discipline* says every name is bound once and never changes. Which line of `make_counter` breaks that rule, and why can't a counter work without breaking it? (Axioma will run the code either way — the discipline is a habit the book teaches, not a parser restriction.)

Welcome to the edge of Part IV — this is the first place where "a function is a black box" stops being entirely true. Functions that close over mutable state remember things between calls. We revisit this in Chapter 16 (Mutation).

End of Part III. You can now: parameterize over functions (Ch.10), inline anonymous computations (Ch.11), hide intermediate state (Ch.12). Part IV puts these to work on *intertwined* data — networks, expression trees, anything where the data definition has cross-references.

Solutions to selected exercises: Chapter 12 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch12_solutions.md.)*

Chapter 13 · Mutually Recursive Data Designs

> **What this chapter is.** Chapter 9 introduced **functions** that call each other across two data shapes (Person/Household, Lit/Op). This chapter zooms out: what if the *data definitions* themselves cross-reference, and the cross-reference *is the point*? We meet **reified relationships** — turning the edge between two things into a *thing of its own* — and we use them to model the first networks in this book: a social graph, a road map, a bill-of-materials.

13.1 From "two referring concepts" to "edges are concepts too"

The Person/Household pair from Chapter 9 already crossed the divide once: a Person knew its Household and a Household knew its Persons. That's enough when **a relationship has nothing extra to say** — "this person belongs to this household" is just *the fact of belonging*, no metadata attached.

But real domains are messier. A social network's edge isn't "these two are friends," it's "these two have been friends since 2019, on a scale of acquaintance-to-best-friend, primarily through *work*." A road's edge isn't "city A connects to city B," it's "city A to B is 95 miles, costs $4 in tolls, takes 90 minutes in rush hour." A bill-of-materials edge isn't "this assembly uses this part," it's "this assembly uses *4 of* this part, sourced from supplier X, at $2.50 each."

When the *edge has attributes*, you need a concept for the edge. We'll call this pattern **reifying the relationship** — making the relationship a first-class object instead of just an implicit pointer.

> *Reify* (Latin *res*, "thing") — "make a thing of." A relationship goes from being implicit (a field pointing at another object) to explicit (its own object with its own fields).

13.2 Worked example — a small friendship graph

We start with people and friendships. The friendship records *how long* and *how close*; it has fields, so it's an object.

```
#language axioma/beginner
concept Person
Person has name: ""
Person has age: 0

concept Friendship
```

```
Friendship has source: none
Friendship has target: none
Friendship has years: 0
Friendship has closeness: 0   # 1=acquaintance, 5=best friend
```

We have *three* concepts: two "endpoint" concepts (just Person for now — friendship is symmetric) and one "edge" concept (Friendship). The edge points at both endpoints via `source` and `target`.

> **Why two fields, not one?** A symmetric relation (`friendship`) might seem like it should have a single `{partyA, partyB}` set. We use `source`/`target` because Axioma's Concepts don't have built-in set-typed slots — and because in practice, even "symmetric" relationships have a direction in storage (someone initiated the friendship, the data was entered in some order). For asymmetric edges (a follows-on-social-media relation, a parent-of relation, *road from A to B*), source/target is the natural fit.

Build a small graph:

```
alice: a Person {name: "Alice", age: 30}
bob:   a Person {name: "Bob",   age: 25}
carol: a Person {name: "Carol", age: 35}
dave:  a Person {name: "Dave",  age: 28}

f1: a Friendship {source: alice, target: bob,   years: 5, closeness: 4}
f2: a Friendship {source: alice, target: carol, years: 2, closeness: 3}
f3: a Friendship {source: bob,   target: carol, years: 3, closeness: 5}
f4: a Friendship {source: carol, target: dave,  years: 1, closeness: 2}

people:      [alice, bob, carol, dave]
friendships: [f1, f2, f3, f4]
```

Now the questions. Each is one helper function — and the helpers *all use the same recipe*: walk the array of friendships, filter or transform, return the answer.

Friends of a given person

A friendship `f` involves `p` if `f.source == p` or `f.target == p`:

```
friendships_of: func(p) [
  filter(func(f) [f.source == p or f.target == p], friendships)
]
println(len(friendships_of(alice)))   # 2 (f1, f2)
println(len(friendships_of(carol)))   # 3 (f2, f3, f4)
```

The *other* person in a friendship

Given a person `p` and a friendship `f` they're in, the *other* party is `f.target` if `p` is the source, otherwise `f.source`:

```
other_party: func(p, f) [
  if f.source == p then f.target else f.source
]
println(other_party(alice, f1).name)   # Bob
println(other_party(bob, f1).name)     # Alice
```

Direct friends of a person (just names)

```
direct_friends: func(p) [
  map(func(f) [other_party(p, f).name], friendships_of(p))
]
println(direct_friends(alice))   # ["Bob", "Carol"]
println(direct_friends(carol))   # ["Alice", "Bob", "Dave"]
```

Three small, well-named helpers and we already have a queryable graph. The composition `friendships_of → other_party → name` is the same recipe from §10.6 — pipeline of stages — applied to a new data shape.

13.3 Worked example — a road map

A second pattern: directed edges with numeric weights.

```
#language axioma/beginner
concept City
City has name: ""

concept Road

Road has src: none
Road has dst: none
Road has miles: 0

nyc: a City {name: "NYC"}
phl: a City {name: "Philadelphia"}
dc:  a City {name: "DC"}
bos: a City {name: "Boston"}

roads: [
  a Road {src: nyc, dst: phl, miles: 95},
  a Road {src: phl, dst: dc,  miles: 140},
  a Road {src: nyc, dst: bos, miles: 215},
  a Road {src: dc,  dst: nyc, miles: 225}    # one-way back
]
```

Notice the field names: `src` and `dst` — *not* `source` and `target`. Both `source` and `from` collide with reserved or heavily-overloaded identifiers in some contexts (the words `from` and `to` are reserved keywords

in Axioma's `from..to` range syntax). Naming edge endpoints is a place where readability and keyword-avoidance tug at each other; conventions like `src`/`dst` or `head`/`tail` are battle-tested compromises.

Outgoing roads from a city

```
outgoing: func(c) [
  filter(func(r) [r.src == c], roads)
]
println(len(outgoing(nyc)))   # 2 (phl, bos)
println(len(outgoing(dc)))    # 1 (nyc)
println(len(outgoing(bos)))   # 0
```

Direct neighbors

```
neighbors: func(c) [
  map(func(r) [r.dst], outgoing(c))
]
println(map(func(c) [c.name], neighbors(nyc)))   # ["Philadelphia", "Boston"]
```

Reachability — *one* step or *many*?

This is where structural recursion on the *list of roads* meets a **search through the graph**. A reachability check is intuitively "is there a path from `src` to `dst` using any number of hops?". The natural recursion isn't on a list — it's *on the graph*, and the graph can have cycles, so we need a termination story (the subject of Ch.14).

A short-but-honest version uses a *fuel parameter* — a hop budget that decrements each step. When fuel reaches zero, we give up. This is **bounded search**, not full reachability, but it's correct for graphs with paths shorter than the bound.

```
reach_via: func(s, d, fuel) [
  if s == d then true
  else if fuel == 0 then false
  else [
    nbrs: neighbors(s);
    any_reach(nbrs, d, fuel - 1)
  ]
]
any_reach: func(arr, d, fuel) [
  if len(arr) == 0 then false
  else if reach_via(first(arr), d, fuel) then true
  else any_reach(rest(arr), d, fuel)
]
println(reach_via(nyc, dc,  10))   # true  (nyc -> phl -> dc)
println(reach_via(bos, nyc, 10))   # false (bos is a dead end)
println(reach_via(dc,  bos, 10))   # true  (dc -> nyc -> bos)
```

Two mutually recursive functions: `reach_via` walks one step forward, then asks `any_reach` to check the neighbors; `any_reach` walks the neighbor list calling `reach_via` on each. The pattern is exactly the Person/Household pattern from §9.2 — *but the data isn't a tree, it's a graph*. The fuel parameter is what keeps the recursion well-founded; without it a cycle (`nyc → bos → nyc → bos …`) would loop forever.

Chapter 14 will name this kind of recursion — *generative*, because the recursive subproblem isn't a piece of the input but something we *make up* (the smaller fuel budget) — and develop the termination argument formally.

13.4 Worked example — bills of materials

Third pattern: a hierarchical decomposition where some pieces are *atomic* (a screw, a wire) and others are *composite* (an assembly that contains other parts, which may themselves be assemblies). The data shape:

> A **Part** is one of:
>
> - a **BasicPart** with a name and a weight, or
> - an **Assembly** with a name and an array of Parts.

This is exactly the **variant recursion** pattern from §9.3 — one logical type (Part) with two flavors. The Assembly's `parts` field is an array of Parts, which can include other Assemblies, so the recursion goes down arbitrarily deep.

```
#language axioma/beginner
concept BasicPart
BasicPart has name: ""
BasicPart has weight: 0

concept Assembly

Assembly has name: ""
Assembly has parts: []

wheel: an Assembly {
  name: "wheel",
  parts: [
    a BasicPart {name: "rim",    weight: 500},
    a BasicPart {name: "tire",   weight: 700},
    a BasicPart {name: "spokes", weight: 200}
  ]
}
frame: a BasicPart {name: "frame", weight: 3000}
bike: an Assembly {
  name: "bicycle",
  parts: [frame, wheel, wheel]
}
```

The function: total weight of a Part. Two recursions hopping between each other — `weight_of` dispatches

on the variant; `sum_weights` walks the parts array.

```
weight_of: func(p) [
  if p is BasicPart then p.weight
  else if p is Assembly then sum_weights(p.parts)
  else 0
]
sum_weights: func(arr) [
  if len(arr) == 0 then 0
  else weight_of(first(arr)) + sum_weights(rest(arr))
]
println(weight_of(frame))    # 3000
println(weight_of(wheel))    # 500 + 700 + 200 = 1400
println(weight_of(bike))     # 3000 + 1400 + 1400 = 5800
```

This is the *same* mutual-recursion shape as the family-tree and expression-tree examples from Chapter 9, but with one twist: the recursion goes through an **array of Parts**, not through a fixed left/right pair. The pattern *array of recursive things* + *list recursion* + *variant dispatch* compose cleanly because each piece handles exactly one concern.

13.5 The four-quadrant view

We can now see the design space:

	Edges have no metadata	Edges have metadata
Tree (no cycles)	Ch.8 — direct field pointers	Ch.13 §13.4 — Bill of Materials
Graph (possibly cyclic)	Pointer-set per node	Ch.13 §13.2-3 — reified Friendship/Road

The textbook climbed the table column by column:

- **Top-left** (Ch.8): each Node has direct child pointers. No edge attributes, no cycles. Recursion is straightforwardly structural.
- **Bottom-right** (Ch.13 §13.2): cycles possible, edges carry attributes. We **reify the edges** as their own concept and store an array of edges. Recursion is *bounded* by a fuel parameter to avoid cycles.
- **Top-right** (§13.4): hierarchical decomposition with attributed leaves. Same as top-left, except one variant (Assembly) recurses on its `parts` array.

The bottom-left is rare in practice (if your edges have no attributes, you might as well attach them to nodes), but appears in classical graph theory.

13.6 The design recipe — adapted to mutually recursive data

The recipe stays the same; the *data definition* gets a new clause.

1. **Data definition**: When concept A points at concept B and vice versa, write *both* definitions and underline the cross-references. If an edge has metadata, give it its own concept.
2. **Function template**: For each concept A, write a function that branches on A's possible shapes (atoms vs. structures). Wherever the data points at B, the function calls B's processor — even if you haven't written it yet. *Bidirectional* reference begets *bidirectional* function calls.
3. **Body**: Fill in each branch. Recurse through cross-references exactly the way the data structure points. If the graph has cycles, introduce a fuel parameter or a visited-set accumulator (Ch.15).
4. **Tests**: At least one test per shape × per concept. A network with three nodes is enough to exercise most branches.

This is the recipe HtDP calls **mutually-recursive data**. It's the last leg of the structural-recursion ladder. Chapter 14 moves to the *other* kind of recursion — algorithmic, not structural — and Chapter 15 brings in accumulators to handle graphs cleanly.

13.7 Reflection — what we've done

Look back across Parts II–IV:

- Part II: arrays, then trees, then mutually recursive *function* pairs.
- Part III: higher-order parameterization, lambdas, locals.
- This chapter: **mutually recursive *data* designs** — modeling networks of related things in a first-class way.

Every domain you'll encounter for the rest of your career — file systems, scene graphs, dependency graphs, AST passes, spreadsheet calc engines, query plans, routing tables, social networks, organizational charts, scientific entity-relation diagrams — is one of these shapes. The recipe doesn't change.

The big chapter ahead — Ch.14 — is your first encounter with recursion that *isn't* about walking a data shape. Quicksort doesn't have a "list" to recur on in the usual sense; it partitions and recurs on its own outputs. Mergesort splits the list down the middle, again not "rest of list." Binary search halves the search range. These algorithms generate their subproblems out of thin air, and you have to *argue* that the subproblems are smaller — the **termination argument** — instead of getting that argument for free from the data shape.

Exercises

Exercise 13.1 — `mutual_friends`

Given two People `p` and `q`, return an array of names of people who are friends with *both* `p` and `q`. (Build on `direct_friends` from §13.2.)

- `mutual_friends(alice, bob)` → `["Carol"]`
- `mutual_friends(alice, dave)` → `["Carol"]`
- `mutual_friends(carol, alice)` → `["Bob"]`

Exercise 13.2 — `friendship_years_total`

Sum the `years` field across all friendships involving a given person. (Hint: this is `friendships_of(p)` from §13.2, then reduce over `years`.)

- `friendship_years_total(alice)` → 7 (5 + 2)
- `friendship_years_total(carol)` → 6 (2 + 3 + 1)
- `friendship_years_total(dave)` → 1

Exercise 13.3 — `total_route_miles`

Given an *array of cities* representing a planned route (`[nyc, phl, dc, nyc]`), sum the miles on each consecutive leg. Use the `roads` array from §13.3 to look up each leg's distance. If a leg has no direct road, return `-1` for the whole route.

- `total_route_miles([nyc, phl])` → 95
- `total_route_miles([nyc, phl, dc])` → 235
- `total_route_miles([nyc, phl, dc, nyc])` → 460
- `total_route_miles([bos, nyc])` → -1 (no bos→nyc road)

Exercise 13.4 — `count_parts`

For a Part, return the total number of *BasicPart* leaves (ignoring intermediate Assemblies).

- `count_parts(frame)` → 1
- `count_parts(wheel)` → 3
- `count_parts(bike)` → `1 + 3 + 3 = 7`

Exercise 13.5 — `flatten_parts`

Return a flat array of every BasicPart inside a Part, in left-to-right order.

- `flatten_parts(frame)` → `[frame]`
- `flatten_parts(wheel)` → `[rim, tire, spokes]` (3 BasicParts)
- `len(flatten_parts(bike))` → 7

Then verify: `sum_of_weights(flatten_parts(bike))` equals `weight_of(bike)`. (Pick any name for the helper.)

Exercise 13.6 (open) — `route_exists`

Use the road network from §13.3 to write a `route_exists(src, dst)` function that returns `true` if a route exists from `src` to `dst` (any number of hops). Adapt the `reach_via` / `any_reach` pair from the chapter; you'll need to pick a fuel budget.

- `route_exists(nyc, dc)` → `true`
- `route_exists(bos, dc)` → `false`
- `route_exists(dc, bos)` → `true`

Then think: what would go wrong if you forgot the fuel parameter? What would go wrong if you set fuel too small? This is the *new* pedagogical hurdle of Chapter 14 — convincing yourself a recursion terminates when the data shape isn't giving you the answer for free.

End of Chapter 13. Chapter 14 takes the leap from *structural* recursion (the kind you've been doing — recurring on pieces of the input) to *generative* recursion (recurring on problems you make up). Quicksort, mergesort, and binary search await.

Solutions to selected exercises: Chapter 13 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch13_solutions.md.)*

Chapter 14 · Generative Recursion

What this chapter is. Every recursion you've written so far has been *structural* — the recursive call worked on a *piece of the input* (the tail of a list, a subtree, the other half of a mutually recursive pair). The data shape *told you* what the subproblem was. This chapter changes that. In **generative** recursion, the subproblem is *something you make up*: the smaller half of a sorted partition, a halved search range, a reduced numeric quantity. The data doesn't tell you what to do; you tell *yourself* what smaller problem to solve next.

The new pedagogical hurdle is the **termination argument**. With structural recursion, termination came for free: every recursive call shrank the input, and the empty input was the base case. With generative recursion, you have to *show* that the subproblem is smaller — by some measure, by some amount, reliably — or your program loops forever.

14.1 Structural vs. generative — the central distinction

Compare two recursions you already know how to write:

```
# Structural — sum a list
sum_list: func(arr) [
  if len(arr) == 0 then 0
  else first(arr) + sum_list(rest(arr))
]
```

The recursive call is `sum_list(rest(arr))`. The argument is a *piece of* `arr` — specifically, all but the first element. There's a one-to-one correspondence between the data shape (the list) and the recursion (the recursive call). The base case (empty list) is *the smallest possible piece of the data shape*.

Now look at this:

```
# Generative — Euclid's GCD
gcd: func(a, b) [
  if b == 0 then a
  else gcd(b, a % b)
]
```

The recursive call is `gcd(b, a % b)`. Neither `b` nor `a % b` is "a piece of" `a` or `b` in any structural sense. They're *derived* — newly computed from `a` and `b` by a formula. The base case (`b == 0`) is *a property of the numbers*, not a piece of any data structure.

That's the difference. **Structural recursion**: the recursion is guided by the input's shape. **Generative**

recursion: the recursion is guided by an algorithmic idea, and the input gets *transformed* into the subproblem.

Why does it matter? Two reasons:

1. **Templates work differently.** Structural recursion has a template you can derive from the data definition (Ch.4). Generative recursion doesn't — every algorithm is its own creative act.
2. **Termination isn't free.** Structural recursion always terminates because the input shrinks toward the base case. Generative recursion can loop forever if the "make a smaller problem" step doesn't actually make it smaller.

We'll see four worked examples. Each one introduces a *new* algorithmic idea, and each one needs its own termination argument.

14.2 First worked example — Euclid's GCD

We started with this one. The full setup:

```
#language axioma/beginner
gcd: func(a, b) [
  if b == 0 then a
  else gcd(b, a % b)
]
println(gcd(48, 18))    # 6
println(gcd(7, 13))     # 1
println(gcd(100, 75))   # 25
println(gcd(15, 0))     # 15
```

The algorithmic idea: any common divisor of `a` and `b` is also a divisor of `a % b` (the remainder when `a` is divided by `b`). And any common divisor of `b` and `a % b` is also a divisor of `a`. So `gcd(a, b) = gcd(b, a % b)` — *the GCD is preserved under the "swap and take remainder" transformation*. We can keep doing it until one operand reaches `0`; then the answer is the other.

The termination argument:

- The second argument (`b`) is a non-negative integer.
- Each recursive call has `a % b < b` (the remainder is strictly less than the divisor).
- So the second argument *strictly decreases* with each call.
- Strictly decreasing non-negative integers must reach `0` in finitely many steps.

That's the shape of every generative-recursion termination argument: pick a quantity that decreases, show it can't go on forever. For GCD the quantity is `b`; for the next example, it'll be the size of an array.

14.3 Second worked example — quicksort

The most famous generative recursion in computer science.

The algorithmic idea: to sort an array, pick any element (the "pivot"). Everything less than the pivot goes in the left pile; everything greater-or-equal goes in the right pile. Recursively sort each pile. Splice

them back together: `sorted_left + [pivot] + sorted_right`.

```
#language axioma/beginner
qsort: func(arr) [
  if len(arr) == 0 then []
  else if len(arr) == 1 then arr
  else [
    p: first(arr);
    tail: rest(arr);
    smaller: filter(func(x) [x < p],  tail);
    bigger: filter(func(x) [x >= p], tail);
    qsort(smaller) + [p | qsort(bigger)]
  ]
]
println(qsort([]))                      # []
println(qsort([42]))                    # [42]
println(qsort([3, 1, 4, 1, 5, 9, 2, 6])) # [1, 1, 2, 3, 4, 5, 6, 9]
```

Where's the generative step? `smaller` and `bigger` are *new arrays we made up from `tail`*. They aren't pieces of the input in the structural sense — we computed them by filtering. We even chose the partition strategy (split on the *first* element; we could have chosen the middle, the median, a random element). Every choice changes the *shape* of the recursion.

The termination argument:

- Both `smaller` and `bigger` are subsets of `tail`.
- `tail` has length `len(arr) - 1`, so each of `smaller` and `bigger` has length *at most* `len(arr) - 1`.
- The total work happens on arrays of *strictly decreasing* length.
- The base cases (`len == 0` and `len == 1`) are reached in at most `len(arr)` levels of recursion.

The argument relies on `smaller` and `bigger` being **strictly smaller** than `arr`. The choice to use `tail` (not `arr`) for filtering is what makes that strict. Filter on `arr` itself — including the pivot in the left or right pile — and you risk an infinite loop. (Worst case: all elements equal the pivot, so every level recurses on the same array.)

Note on the `[p,]` glue. The trailing comma is doing real work: `[p,]` is the unambiguous one-element array (§6.6), so it splices a single value between the two sorted halves. (You may also meet the older spelling `push([], p)` in existing code — it builds the same singleton.)

Worst case vs. average case — a preview

Quicksort's recursion depth depends on the pivot. With a well-chosen pivot (close to the median), each recursive call roughly halves the array, and the algorithm runs in $O(n \log n)$ time. With a poorly-chosen pivot (always the smallest or largest element of the remaining range), one side is empty and the other side is everything but the pivot — $O(n^2)$. Choosing the pivot well is one of those Real Engineering Problems that an introductory port doesn't dwell on, but it's good to know it exists.

14.4 Third worked example — mergesort

A different generative idea. Instead of partitioning by *value*, split by *position*: the first half goes one way, the second half the other. Sort each half recursively. Then merge.

```
#language axioma/beginner

# take_n / drop_n — `take` is a reserved name, hence the suffix
take_n: func(arr, n) [
  if n == 0 then []
  else if len(arr) == 0 then []
  else push([], first(arr)) + take_n(rest(arr), n - 1)
]
drop_n: func(arr, n) [
  if n == 0 then arr
  else if len(arr) == 0 then []
  else drop_n(rest(arr), n - 1)
]

# Merge two sorted arrays
merge: func(a, b) [
  if len(a) == 0 then b
  else if len(b) == 0 then a
  else [
    fa: first(a);
    fb: first(b);
    if fa <= fb then push([], fa) + merge(rest(a), b)
    else push([], fb) + merge(a, rest(b))
  ]
]

# The driver
msort: func(arr) [
  n: len(arr);
  if n <= 1 then arr
  else [
    mid: n div 2;
    left: msort(take_n(arr, mid));
    right: msort(drop_n(arr, mid));
    merge(left, right)
  ]
]
println(msort([]))                    # []
println(msort([42]))                  # [42]
println(msort([3, 1, 4, 1, 5, 9, 2, 6])) # [1, 1, 2, 3, 4, 5, 6, 9]
```

The algorithmic idea: divide the array in half (by position), sort each half, then merge the sorted halves in linear time.

Two generative steps:

1. `take_n(arr, mid)` and `drop_n(arr, mid)` — *position-based* splits. We *generate* the two subarrays from the input.
2. `merge(left, right)` is itself *structural* on the two subarrays (it's a list-recursion on both at once) — so mergesort *uses* generative recursion at the top level but relies on a structural helper inside. Mixing the two patterns is common.

The termination argument for msort:

- `n div 2` rounds down (integer-part division — plain `/` would give the exact fraction `n/2` for odd `n`, which is no use as a length). For `n >= 2`, `mid = n div 2` is at least `1` and at most `n - 1`.
- So `take_n(arr, mid)` has length `mid >= 1`, strictly less than `n`.
- And `drop_n(arr, mid)` has length `n - mid >= 1`, strictly less than `n`.
- Recursion depth is bounded by `log2(n)`: roughly `log2(8) = 3` for an 8-element array, `log2(1024) = 10` for 1024 elements.

The termination argument for merge is structural: each recursive call drops the first element of either `a` or `b`. The combined length `len(a) + len(b)` strictly decreases. Reaches zero when both arrays are empty.

Comparing quicksort and mergesort

Property	Quicksort	Mergesort
Split strategy	by *value* (partition around pivot)	by *position* (halve in the middle)
Combine strategy	concatenate (trivial)	merge (linear scan)
Best case time	O(n log n)	O(n log n)
Worst case time	$O(n^2)$	O(n log n)
Extra space	in-place possible	needs auxiliary array
Stability	not stable	stable

Both are "divide-and-conquer." Both split, recurse, combine. They just make *different choices* at each step. Real-world standard libraries pick mergesort (or hybrid mergesort/quicksort variants like Timsort) for stability and worst-case predictability; quicksort is famously the fastest *on average* with a tuned pivot.

14.5 Fourth worked example — binary search

The other classic generative recursion. Search a sorted array for a target. At each step, look at the middle; if it matches, return its index; if the target is smaller, recurse on the left half; if larger, recurse on the right half.

```
#language axioma/beginner
bsearch_in: func(arr, target, lo, hi) [
  if lo > hi then 0
  else [
    mid: (lo + hi) div 2;
    v: arr[mid];
```

```
    if v == target then mid
    else if target < v then bsearch_in(arr, target, lo, mid - 1)
    else                    bsearch_in(arr, target, mid + 1, hi)
  ]
]

bsearch: func(arr, target) [
  bsearch_in(arr, target, 1, len(arr))
]

nums: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
println(bsearch(nums, 7))    # 4
println(bsearch(nums, 1))    # 1
println(bsearch(nums, 19))   # 10
println(bsearch(nums, 4))    # 0  (not found)
println(bsearch(nums, 0))    # 0
println(bsearch(nums, 20))   # 0
println(bsearch([], 5))      # 0
```

The algorithmic idea: at each step, the target either *is* the middle element, or it's in the left half, or it's in the right half. Each step eliminates half the remaining range.

The generative step: `lo..mid-1` and `mid+1..hi` are *derived* ranges. They aren't structural pieces of `arr` — they're *coordinate transformations* on the search bounds.

The termination argument:

- The search range size is `hi - lo + 1`. Initially it's `len(arr)`.
- After one recursive call, the range size is roughly `(hi - lo + 1) / 2`, strictly less.
- The range shrinks to `0` (when `lo > hi`) in at most `log2(len(arr)) + 1` steps. That's about `7` steps for a 100-element array, `10` for 1000, `20` for a million.

Notice the convention: we return `0` to mean "not found." That's because Axioma arrays are 1-indexed, so `0` is never a valid index. (If we used `-1`, callers would have to remember the sentinel; `0` is more natural in this dialect.)

14.6 The general shape of a generative-recursive function

Comparing the four:

```
generative_func: func(problem) [
  if is_trivially_solvable(problem) then solve_directly(problem)
  else [
    smaller_problem: generate_subproblem(problem);
    answer: generative_func(smaller_problem);
    combine(problem, answer)
  ]
]
```

You need to provide:

1. **A trivial case** — "the answer is obvious here." For GCD, when `b == 0`. For quicksort, length 0 or

1. For binary search, when `lo > hi`. (For mergesort, also length 0 or 1.)
2. **A subproblem generator** — "given the current problem, make a smaller one." For GCD, `(b, a % b)`. For quicksort, `smaller` and `bigger`. For binary search, the half-range that could contain the target.
3. **A combiner** — "given the subproblem's answer, build the answer for the whole problem." For GCD, the answer just *is* the subproblem's answer. For quicksort, concatenate left + pivot
   - right. For binary search, the recursive call directly returns the answer.
4. **A termination argument** — "this subproblem is strictly smaller, by some measure, by some positive amount." Without this argument, you have a possibly-infinite loop, not an algorithm.

The first three are how you *write* the function. The fourth is how you *trust* it.

14.7 When generative recursion goes wrong — infinite loops

The classic mistake: a "smaller" subproblem that isn't.

```
# WRONG: this loops forever
qsort_bad: func(arr) [
  if len(arr) == 0 then []
  else if len(arr) == 1 then arr
  else [
    p: first(arr);
    # BUG: filtering arr, not tail. Pivot stays in one of the partitions.
    smaller: filter(func(x) [x < p],  arr);
    bigger: filter(func(x) [x >= p], arr);
    qsort_bad(smaller) + push([], p) + qsort_bad(bigger)
  ]
]
```

If `arr = [3, 1, 2]`, then `p = 3`, `smaller = [1, 2]` (good, shrinks), `bigger = [3]`. Recursing on `[3]` hits the `len == 1` base case. Fine.

But if `arr = [3, 3, 3]`, then `p = 3`, `smaller = []`, `bigger = [3, 3, 3]`. We recur on `[3, 3, 3]` — *the same array we started with*. Infinite loop. (In Axioma, the stack runs out and the process aborts.)

The fix is the *correct* version from §14.3: filter on `tail = rest(arr)` instead of on `arr`. That guarantees both partitions are strictly smaller than the input — the termination argument requires it.

Always write the termination argument *before* you trust a generative recursion. If you can't sketch the strictly-decreasing measure, you don't yet have a working algorithm.

14.8 Reflection

Structural recursion and generative recursion are the two pillars of recursive programming.

- **Structural recursion** is the workhorse — 95% of all recursive functions are list-walks, tree-walks, mutual recursions on related concepts. The data definition gives you the template; termination is automatic.

- **Generative recursion** is the toolkit for *algorithms* — sorting, searching, optimization, divide-and-conquer numerical methods, fractal generation. You design the subproblem from scratch; you owe a termination argument.

Most of Part V (accumulators, mutation, streams) revisits recursion from a third angle — *iterative* recursion, where the recursive call replaces a loop. Part VI's Chapter 22 (the meta-circular evaluator) brings all three together in a single program that interprets itself.

Welcome to the end of Part IV. You can now design programs over:

- Fixed-size data (Part I)
- Arbitrary-size structural data (Part II)
- Abstracted-over data (Part III)
- Intertwined data and algorithmic recursion (Part IV)

That's the structural-recursion-as-pedagogy half of HtDP, ported end-to-end. The rest of the book is about *state*, *streams*, and the Axioma-specific superpowers (concepts, MVL, logic programming, metacircularity).

Exercises

Exercise 14.1 — `power`

Implement integer exponentiation using generative recursion. The naive structural recursion is `power(b, n) = b * power(b, n-1)` (with `power(b, 0) = 1`); that's n multiplications. The generative *fast power* uses the identity:

- if `n` is even, `power(b, n) = power(b * b, n / 2)`
- if `n` is odd, `power(b, n) = b * power(b, n - 1)`

That's `O(log n)` multiplications.

- `power(2, 10)` → `1024`
- `power(3, 0)` → `1`
- `power(2, 16)` → `65536`
- `power(5, 3)` → `125`

Termination question (write this out as a comment in your solution): by what measure does the recursion shrink? Why is that measure guaranteed to reach zero?

Exercise 14.2 — `nat_log2`

`nat_log2(n)` returns the largest integer `k` such that `2^k <= n`. Equivalently, the number of times you can halve `n` before reaching `1` (or below). Use generative recursion.

- `nat_log2(1)` → `0`
- `nat_log2(2)` → `1`
- `nat_log2(7)` → `2` ($2^2 = 4$, still ≤ 7; $2^3 = 8$, too big)
- `nat_log2(1024)` → `10`

Termination question: why is the recursion well-founded?

Exercise 14.3 — `hanoi`

The Tower of Hanoi: three pegs (`"A"`, `"B"`, `"C"`), `n` disks on the source peg in size order (largest at bottom). Move all disks to the destination peg using the third as via, never placing a larger disk on a smaller one. Return an array of `(from_peg, to_peg)` moves in execution order.

```
hanoi: func(n, src, dst, via) [
  if n == 0 then []
  else hanoi(n - 1, src, via, dst)
       + push([], (src, dst))
       + hanoi(n - 1, via, dst, src)
]
```

The above is the canonical solution — verify it:

- `hanoi(1, "A", "C", "B")` → `[("A", "C")]`
- `hanoi(2, "A", "C", "B")` → `[("A", "B"), ("A", "C"), ("B", "C")]`
- `len(hanoi(3, "A", "C", "B"))` → `7`
- `len(hanoi(5, "A", "C", "B"))` → `31`

Then *explain in a comment* (a) why this is generative recursion (what's the subproblem?) and (b) the termination argument.

Exercise 14.4 — `bsearch_count`

Modify `bsearch` from §14.5 to also return the number of comparisons made. Return a tuple `(index, comparisons)`. For a 1000-element array, `comparisons` should be at most `10` (`log2(1000) ≈ 9.97`).

- `bsearch_count([1,3,5,7,9], 5)` → `(3, 1)` (mid hit on first try)
- `bsearch_count([1,3,5,7,9], 1)` → `(1, ?)` (you compute)
- `bsearch_count([1,3,5,7,9], 11)` → `(0, ?)` (not found; you compute)

Exercise 14.5 — `merge_three`

Generalize `merge` from §14.4 to merge *three* sorted arrays at once. (Hint: one approach is `merge(a, merge(b, c))`. The *native* three-way version is harder — pick the smallest of three first-elements at each step.) Implement the native version and verify against the chained version on a few examples.

- `merge_three([1, 4, 7], [2, 5, 8], [3, 6, 9])` → `[1, 2, 3, 4, 5, 6, 7, 8, 9]`
- `merge_three([], [1, 2], [3, 4])` → `[1, 2, 3, 4]`
- `merge_three([], [], [])` → `[]`

Exercise 14.6 (open) — `is_perfect_square`

Write `is_perfect_square(n)` that returns `true` if `n` is a perfect square (`0, 1, 4, 9, 16, …`), `false` otherwise. Use generative recursion: a binary-search-style narrowing of the range `[0, n]` looking for `k` such that `k * k == n`.

- `is_perfect_square(0)` → `true`
- `is_perfect_square(1)` → `true`
- `is_perfect_square(16)` → `true`
- `is_perfect_square(17)` → `false`
- `is_perfect_square(144)` → `true`
- `is_perfect_square(143)` → `false`

Then think:

1. What's the search range, and why does halving it terminate?
2. Why is the simpler `sqrt(n) * sqrt(n) == n` approach *unreliable* for large `n`? (Hint: floating-point.)

End of Part IV. Part V brings *accumulators* — a way to restructure recursive functions so they use constant stack space — *mutation*, and a first look at *streams*. The recipe continues to apply; the techniques get a little less pure.

Solutions to selected exercises: Chapter 14 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch14_solutions.md.)*

Chapter 15 · Accumulator-Style Recursion

What this chapter is. A *style* of recursion, not a new data shape. You've been writing **natural recursion** since Chapter 7 — recur on the tail, combine the result with the head, return. This chapter introduces **accumulator recursion** — carry the running answer *forward* in an extra parameter, so the work happens *before* the recursive call instead of after. The same problems get a different structure, and the new structure has two big advantages: it uses less stack space (when the host supports tail calls) and it builds up the answer in a single forward pass.

15.1 The two shapes side-by-side

Chapter 7's `sum_list`:

```
#language axioma/beginner
sum_list: func(arr) [
  if len(arr) == 0 then 0
  else first(arr) + sum_list(rest(arr))
]
```

The recursive call `sum_list(rest(arr))` happens, returns, and *then* we add `first(arr)` to its result. The addition is **pending** during the whole recursive descent — the interpreter remembers, in each stack frame, "after the recursive call comes back, add the head."

For a 5-element array, the call stack looks like:

```
sum_list([1,2,3,4,5])
  1 + sum_list([2,3,4,5])
        2 + sum_list([3,4,5])
              3 + sum_list([4,5])
                    4 + sum_list([5])
                          5 + sum_list([])
                                0          (base case)
                          5 + 0   = 5
                    4 + 5         = 9
              3 + 9               = 12
        2 + 12                    = 14
  1 + 14                          = 15
```

Five pending additions live on the stack at the deepest point. For a 5-million-element array, that's five million pending additions — and on Axioma's runtime, non-tail recursion like this stops with a clean recursion-limit error at a depth in the low tens of thousands (the interpreter guards nested evaluation at

50,000 steps, and each recursion level costs a few of them). Real Racket and many other Lisps optimize away the stack usage when the shape is tail-recursive — and so does Axioma: a self-recursive tail call reuses its frame and runs at any depth.

Now the **accumulator** version:

```
#language axioma/beginner
sum_acc: func(arr, acc) [
  if len(arr) == 0 then acc
  else sum_acc(rest(arr), acc + first(arr))
]
sum_list_acc: func(arr) [ sum_acc(arr, 0) ]

println(sum_list_acc([1, 2, 3, 4, 5]))   # 15
```

Two functions: an outer wrapper that supplies the initial accumulator (`0`), and an inner worker that carries the running total. The call stack looks like:

```
sum_list_acc([1,2,3,4,5])
  sum_acc([1,2,3,4,5], 0)
    sum_acc([2,3,4,5], 1)
      sum_acc([3,4,5], 3)
        sum_acc([4,5], 6)
          sum_acc([5], 10)
            sum_acc([], 15)
              15
            15
          15
        15
      15
    15
  15
```

Each call's pending work is *nothing* — it just returns whatever the next call returns. That's the **tail call** position: the recursive call is the very last thing the function does.

The accumulator now carries *all* the computational state. The stack carries only the call chain.

15.2 The recipe for converting natural → accumulator

Take any natural recursion that combines results with +, ∗, list-cons, or any other associative operation:

```
natural_func: func(input) [
  if is_base(input) then base_value
  else combine(head(input), natural_func(rest(input)))
]
```

Mechanically rewrite it as:

```
acc_func: func(input, acc) [
  if is_base(input) then acc
  else acc_func(rest(input), combine(acc, head(input)))
]
```

```
wrapper: func(input) [ acc_func(input, base_value) ]
```

Three steps:

1. Add an `acc` parameter to the worker.
2. Initialize it to whatever `base_value` was in the natural version.
3. At each step, *combine the head into the accumulator* before recursing — instead of combining *after* recursing.

The combine operation has to be *associative* for this to give the same answer in either order (because the natural version combines right-to-left and the accumulator combines left-to-right). Addition, multiplication, concatenation, set union, `min`/`max` — all associative, all safe to convert. Subtraction, division, and "first non-zero element" are not associative; they need extra care.

15.3 Worked example — reverse a list, two ways

Reversing is the textbook example, because the two styles produce *visibly different* recursive shapes.

Natural reverse

```
#language axioma/beginner
rev_nat: func(arr) [
  if len(arr) == 0 then []
  else rev_nat(rest(arr)) + push([], first(arr))
]
println(rev_nat([1, 2, 3, 4, 5]))   # [5, 4, 3, 2, 1]
```

The recursive call returns the reverse of `rest(arr)`. Then we append the head *at the end* — because the head was the *first* element of the original, it must end up *last* in the reversal.

What's the cost? The `+` operation here is *array concatenation*, which in Axioma is `O(n)` (it walks the left array to copy it). At each of `n` levels, we do `O(n)` work. Total cost: `O(n²)`. For a 1000-element list, that's a million operations — still fast, but you can feel it for 10000.

Accumulator reverse

```
#language axioma/beginner
rev_acc: func(arr, acc) [
  if len(arr) == 0 then acc
  else rev_acc(rest(arr), push([], first(arr)) + acc)
]
rev_fast: func(arr) [ rev_acc(arr, []) ]
println(rev_fast([1, 2, 3, 4, 5]))   # [5, 4, 3, 2, 1]
```

We *prepend* each head to a *growing-at-the-front* accumulator. After the first step the accumulator is `[1]`; after the second, `[2, 1]`; after the fifth, `[5, 4, 3, 2, 1]` — already reversed. Each step does *constant* work (`push([], first) + acc` is one prepend); total cost is `O(n)`.

A useful mental image. Natural reverse is "build the answer from the back forward, one level of recursion per element." Accumulator reverse is "walk forward through the input, prepending each element to a growing reversed prefix." The accumulator reverse builds *the answer itself* as it goes; the natural reverse builds the *plan* for the answer and only assembles it on the way back up.

15.4 Worked example — factorial, both ways

Factorial is the classical introduction to recursion. With one input (an integer) it's the simplest possible accumulator target.

```
#language axioma/beginner
fact_nat: func(n) [
  if n == 0 then 1
  else n * fact_nat(n - 1)
]
println(fact_nat(10))   # 3628800

fact_acc: func(n, acc) [
  if n == 0 then acc
  else fact_acc(n - 1, n * acc)
]
fact_fast: func(n) [ fact_acc(n, 1) ]
println(fact_fast(10))  # 3628800
```

Both give 3628800. The natural version stacks `n * (n - 1) * (n - 2) * ... * (2 * (1 * 1))` to compute right-to-left; the accumulator version computes `((((1 * n) * (n-1)) * (n-2)) * ... * 1)` left-to-right. Since multiplication is *commutative and associative*, the two products are equal.

The big input numbers expose the stack issue — and the payoff:

```
fact_nat(100000)   → recursion limit exceeded (max Eval depth 50000)
                     — a clean, catchable error: every pending `n *`
                     holds its frame open, and 100,000 of them blow
                     past the interpreter's non-tail ceiling
fact_fast(100000)  → runs. The recursive call is in tail position, so
                     Axioma reuses the frame — and the answer comes
                     back as an exact 456,574-digit integer.
```

That's the accumulator payoff made concrete. The natural version *must* remember 100,000 pending multiplications; the accumulator version carries the answer-so-far in an argument, so nothing is pending and the frame can be reused. Axioma's self-tail-call optimization gives `fact_acc` exactly the R5RS-Scheme behavior this chapter is teaching: constant stack space at any depth. The ~50,000-frame ceiling still exists, but it applies only to recursion that *isn't* a self-tail-call — the natural version above, or mutual recursion between two functions (the optimization is per-function).

A second reason to prefer accumulator style. When you can think of your computation as "walk forward, remember the answer so far, finish when the input runs out," your code often reads more like *a loop*. That's why accumulators are the bridge from functional to

imperative thinking — Chapter 16 will show how `while` loops are just accumulator recursions in disguise.

15.5 Worked example — running max of an array

A pattern with a non-trivial combiner. Find the largest element. Naive accumulator: track the maximum-so-far.

```
#language axioma/beginner
# Helper: max of two
maxi: func(a, b) [ if a > b then a else b ]

# Accumulator version — stable for any non-empty array
max_acc: func(arr, acc) [
  if len(arr) == 0 then acc
  else max_acc(rest(arr), maxi(acc, first(arr)))
]
max_of: func(arr) [
  if len(arr) == 0 then none
  else max_acc(rest(arr), first(arr))
]
println(max_of([3, 1, 4, 1, 5, 9, 2, 6]))   # 9
println(max_of([42]))                        # 42
println(max_of([]))                          # none
```

The empty-array case needs the wrapper because *there's no sensible accumulator seed for "the maximum of an empty list."* We can't seed with `0` (what if the array's values are all negative?), and we can't seed with `-infinity` because Axioma doesn't expose that. So the wrapper handles empty-array specially and otherwise seeds the accumulator with the *first* element, recurring over the rest.

Two design patterns to notice:

1. **The wrapper hides the worker's two-argument shape.** The user calls `max_of(arr)`; only `max_of` knows about the accumulator. This is the same wrapper-pattern from §15.1–4 but with an *interesting* (non-base) seed.
2. **The "seed is `first(arr)`" trick.** When there's no meaningful zero for the operation, the first element of the input doubles as the initial accumulator and is consumed by the wrapper's `rest`-then-recur step.

15.6 Multiple accumulators

Sometimes one accumulator isn't enough. To compute the **mean** of an array in a single pass, you need both the running sum *and* the running count:

```
#language axioma/beginner
mean_acc: func(arr, sum, count) [
```

```
  if len(arr) == 0 then sum / count
  else mean_acc(rest(arr), sum + first(arr), count + 1)
]
mean_of: func(arr) [
  if len(arr) == 0 then 0
  else mean_acc(arr, 0.0, 0)
]
println(mean_of([1.0, 2.0, 3.0, 4.0, 5.0]))   # 3.0
println(mean_of([4.0, 8.0, 6.0, 5.0, 3.0, 7.0]))  # 5.5
```

Two accumulators (`sum` and `count`) carried in parallel. The combination of accumulators turns out to be the single-most-common shape in algorithm design: most "streaming statistics" (mean, variance, min, max, median approximation, count-distinct) are accumulator recursions with 2–4 carried values. Real numerical libraries spell this "online algorithm" or "single-pass statistic"; you've just written one in nine lines.

The `0.0` seed for the sum is the same float-promotion trick from Ch.12 — without it the statistics stay in exact arithmetic, so an uneven mean comes back as a fraction (`16 / 5` is the rational `16/5`, not `3.2`). Right value, wrong outfit for a streaming-stats report.

15.7 When accumulators don't help

Some recursions can't be easily expressed accumulator-style:

- **Tree walks.** A tree-recursive function has *two* recursive calls per internal node (one for the left subtree, one for the right). At least one of them can't be in tail position. You can carry an accumulator through one half, but the other half's pending computation has to live on the stack.
- **Mutual recursion** with non-trivial combiners (Ch.13's reachability example) sometimes needs an *explicit visited set* — really an accumulator, but threaded through *both* mutually recursive functions.
- **Generative recursion** that doesn't have a natural "accumulate the answer so far" structure — for instance, binary search returns either the answer or fails, with no partial answers to accumulate.

The rule of thumb: **if the natural version writes `combine(head, recur(rest))` with an associative `combine`, the accumulator version exists and is mechanical.** If the recursive structure is fundamentally tree-shaped or generative, the accumulator may not help — and that's fine. You've still got natural recursion for those cases.

15.8 Reflection — three idioms for the same computation

For "sum the elements of a list," we now have:

1. **Natural recursion** (Ch.7) — clear, straightforward, uses stack space proportional to list length.
2. **`reduce` HOF** (Ch.10) — abstract, declarative, hides the recursion entirely. Internally tail-recursive.
3. **Accumulator recursion** (this chapter) — explicit, single forward pass, would use constant stack space under full TCO.

The accumulator version is *implementation-shaped*: it's how `reduce` is implemented under the hood. The natural version is *specification-shaped*: it reads as "the sum of a list is the head plus the sum of the

rest." Both are correct; they emphasize different things.

By Part V's end, we'll have a *fourth* idiom — explicit mutation with a `while` loop (Ch.16) — that abandons recursion entirely and just uses a mutable counter. Each shift trades some property for another (clarity for speed, abstraction for control). Mature programmers reach for whichever shape fits the problem.

Exercises

Exercise 15.1 — `sum_acc`

Write `sum_acc(arr, acc)` and wrap it as `sum(arr)`. Verify it agrees with the natural `sum_nat(arr)` on three small inputs.

- `sum([])` → `0`
- `sum([7])` → `7`
- `sum([1, 2, 3, 4, 5])` → `15`

Exercise 15.2 — `prod_acc`

Same pattern, multiplication. Seed the accumulator with `1`.

- `prod([])` → `1`
- `prod([7])` → `7`
- `prod([2, 3, 4])` → `24`
- `prod([1, 2, 3, 4, 5])` → `120`

Why does seeding with `0` not work?

Exercise 15.3 — `len_acc`

Re-derive `len` (which Axioma already provides as a builtin) in accumulator style. Seed with `0`, increment by `1` per element. Compare your version's output to `len`'s.

- `my_len([])` → `0`
- `my_len([a, b, c])` → `3`

Exercise 15.4 — `count_evens_acc`

Given an array of integers, return the count of even numbers. Use a single accumulator.

- `count_evens([])` → `0`
- `count_evens([1, 3, 5])` → `0`
- `count_evens([2, 4, 6])` → `3`
- `count_evens([1, 2, 3, 4, 5, 6])` → `3`

Exercise 15.5 — `sum_and_count`

Walk an array once and return a tuple `(sum, count)`. Use *two* accumulators carried in parallel. Then verify `mean = sum / count` agrees with §15.6's `mean_of`.

- `sum_and_count([])` → `(0, 0)`
- `sum_and_count([10, 20, 30])` → `(60, 3)`
- `sum_and_count([1.0, 2.0, 3.0, 4.0])` → `(10.0, 4)`

Exercise 15.6 (open) — `running_sums`

Given `[a, b, c, d, …]`, return an array of running totals `[a, a+b, a+b+c, a+b+c+d, …]`. Use accumulator-style recursion that carries the *current running total* and *builds the output array as it walks*.

- `running_sums([])` → `[]`
- `running_sums([7])` → `[7]`
- `running_sums([1, 2, 3, 4])` → `[1, 3, 6, 10]`

Hint: you'll need two accumulators — the running total *and* the output array being built. This is the natural lead-in to Chapter 16's `while`-loop, which uses the same two-counter pattern with mutation instead of recursion.

End of Chapter 15. Chapter 16 takes the accumulator idea one step further: instead of *passing* the accumulator as a function argument, **mutate** a local variable inside a loop. Same computation, different mechanism, different ergonomics.

Solutions to selected exercises: Chapter 15 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch15_solutions.md.)*

Chapter 16 · Mutation

What this chapter is. The other half of the functional/imperative divide. Up to here we've treated variables as **names for values** — once bound, a name's value never changes. This chapter introduces **mutation**: a name can be *re-bound* to a different value over time. Combined with `while`-loops, mutation gives you the *iterative* style of programming familiar from C, Java, Python, JavaScript — the same one your non-Axioma friends learn from day one.

Mutation is a sharp tool. It lets you write a constant-stack loop where natural recursion would overflow, build state machines whose whole job is to *change*, and model objects with private memory. It also lets you create bugs that no amount of unit testing will find. Used carefully it's indispensable; used carelessly it's how production systems turn unmaintainable. We'll do both.

16.1 Reassignment — the new thing

You've seen bindings before, in Chapter 2:

```
#language axioma/beginner
x: 5
println(x)   # 5
```

What's new is the **bare assignment** form — same name, *no* declaration colon:

```
#language axioma/beginner
x: 5
x = 10
println(x)   # 10
```

`x = 10` is *not* a re-declaration; it's an *update* of the existing binding. At file scope the name `x` still refers to the same storage cell, now holding `10` instead of `5`. Inside a function, the search stops at that function frame; the next subsection explains explicit outer writes.

Three rules:

1. `x: value` (REBOL bind) is the canonical declaration — it introduces the variable, and updates it if it already exists.
2. `x = value` is a *synonym*: since June 2026 it also finds or updates — it declares the name if absent and updates it if present, for any name and case. (That is what lets textbook math like `B = {2, 4,`

6} read verbatim.)

3. So both operators work whether or not the name already exists. By convention, reach for `:` when you mean "introduce this" and = when you mean "update this" (or for math/textbook style) — but mechanically they are interchangeable.

(The `x: 5` then `x = x + 1` pattern parsed cleanly in beginner mode all along — Ch.12's `make_counter` quietly used it. We're just calling it out by name now.)

16.1a Crossing a function frame — `rebind` and `global`

A body assignment (`total = total + x` inside a function) **stops at that function**. It will not update a name that lives outside. Two keywords spell the reach, and they are **not** the same write:

Spelling	Where it lands	Declares if missing?
`rebind total = total + x`	the **nearest** existing cell	no — a typo is an error
`global total = total + x`	**this file** (or this nested `module` body)	yes, if the RHS needs no prior read

On a file-level accumulator they agree — that cell is both nearest and the module. They diverge when an enclosing function has the same name: `rebind` updates the enclosing local; `global` skips it (Julia). A `::` on the module cell is honored either way (`global counted = counted + 1` after `counted :: Intege r = 0` converts or refuses like any other write). Full detail: *Axioma Manual* §3.

16.1b Holes (_), empty containers (`default`), and zeros

Sometimes you want a name **before** you know its first real value — TypeScript's `let x: number;` then `x = 5`, or Go's `var n int` which silently starts at `0`. Axioma separates three ideas that those languages often blur.

Three intents

Intent	Spelling	Can you read it immediately?
Hole — fill in later	`let n :: Integer = _`	No — reading errors
Empty container	`let xs :: Array = default`	Yes — it is `[]`
You mean zero / empty string	`let n :: Integer = 0`	Yes — you wrote `0`

The hole: `let name :: Type = _`

The hole and `default` examples are host Axioma, so they declare `#language axioma/all`. Bare `#lang uage axioma` (no `/subset`) is not a dialect name — the checker lists the eight known ones and refuses the file.

```
#language axioma/all
var lifespan :: Integer | "ongoing" | "uncertain" = _
# println(lifespan)   # ERROR: uninitialized
lifespan = 89
println(lifespan)     # 89
lifespan = "ongoing"
println(lifespan)     # ongoing
```

Rules worth memorizing:

1. Use a **fresh named binding spelling**; a **`::` type** is optional. The marker **`= _`** is also optional (see the bare forms below). A `let` hole takes exactly **one** write — the fill — and is immutable from then on; a slot you mean to keep refilling (like `lifespan` above) is a `var` hole.
2. The hole is **not** `none`, **not** `om`, and **not** `0`. It is *binding state*: "this name is not ready."
3. First successful = (or `:`) assignment fills the same cell (and, for a `let` hole, seals it).
4. Under `--typecheck`, a direct read needs a fill on every path that reaches it. A returning branch does not reach a later read; a skipped short-circuit operand does not supply a fill. Loop bodies and conditions are checked, and a post-test `repeat` checks its body before its condition. Runtime still refuses an early value read, including `:name` (get-word).

The type need not have a default value. A hole names a cell waiting for a value; it does not create a partly constructed value. This works for function annotations and user concepts too:

```
let callback :: Function = _
callback = func(n) [n + 1]
println(callback(2))          # 3
concept Person
let person :: Person = _
person = a Person {}
println(person is Person)     # true
var amount :: Float = _
amount: 3
println(amount is Float)      # true
```

Records, variants, schemas, container annotations, and aliases use the same principle. The fill checks the cell's declared type and performs its normal conversions, such as Integer to Float. `val` follows `let`; `local`, `let mut` and `let mutable` follow `var`. A fresh shadow has its own contract, while an earlier closure retains the original cell and contract.

The static checker cannot model every valid runtime annotation precisely. It accepts these explicit annotations and still tracks a direct read before assignment; a clean check does not prove that an arbitrary later function or concept value fits. Runtime checks remain authoritative. Defining a closure does not execute captured reads or writes. Body-local holes are checked independently; simple known calls also check captured-cell readiness, while unknown calls remain a runtime responsibility. Under `--vm`, immutable holes, concept/record declarations, and persistent Array element contracts still produce explicit refusals.

Source-sensitive checks are separate from the stored type: a schema annotation on a fresh dictionary literal rejects extra keys, whereas an ordinary assignment may accept them. Delaying initialization does

not change that assignment rule. Prefer an immediate value when one is available; a hole is useful when computing the first value must happen later, not as a substitute for deciding what value the program means.

Optional: follow the paths that reach a read

A declaration and a fill are separate events. To check a read, follow each possible path to that read and ask whether it filled the **same cell**.

```
function choose_result(cancel)
    local result :: Integer
    if cancel then [return 0] else [result = 10]
    result
end
println(choose_result(true))    # 0
println(choose_result(false))   # 10
```

Both calls pass `--typecheck`. The returning arm never reaches `result` at the end, so only the other arm needs to fill it. Code after an unconditional `return`, `break`, or `continue` contributes no initialization facts. Ordinary type and annotation checks still apply to that unreachable syntax.

A post-test loop reaches its condition after its body:

```
local result :: Integer
repeat [result = 1] until result == 1
println(result)                 # 1; also passes --typecheck
```

A `continue` reaches the condition too. Moving it before `result = 1` leaves an unreadable hole at the condition. A `break` bypasses the condition; every break path that reaches a later read must supply its own fill. A literal `while true` has no zero-iteration exit, so a fill followed by `break` can establish readiness. An ordinary pre-test loop may run zero times.

Conditional evaluation matters just as much as explicit branches:

```
local result :: Integer
ignored = false and [result = 1]
println(result)                 # ERROR: uninitialized; --typecheck flags it
```

The Boolean `false` skips the right operand. Writing `true or [...]` skips it too. An unknown left operand means the checker must retain a possible skip path. Fallback operators (`??`, `???`, `otherwise`) also have conditional writes. Multivalued logical operands retain their evaluator semantics; the checker does not replace them with Boolean truth values.

Get-word fetches a value without invoking it. It still needs a value to exist:

```
local result :: Integer
println(:result)                # ERROR: uninitialized; --typecheck flags it
```

By contrast, `p = &result` takes the address of the cell and is allowed before its fill. Reading `*p` needs an initialized value. The checker follows simple reference aliases, so `*p = 1` fills that same cell. A fresh shadow does not redirect an earlier reference.

A match joins the paths from its arms. An unguarded catch-all, both Boolean cases, or all constructors with irrefutable fields can establish coverage; a guarded or refutable pattern leaves a possible no-write path. Pattern and loop-variable bindings shadow outer cells rather than filling them.

Optional: evidence for a later fill

The checker can use a known positive repetition count or a known nonempty collection. The body must still fill the cell on every path that reaches the later read; a `continue` or `break` before the assignment breaks that proof. Both examples below pass `--typecheck`:

```
local repeated :: Integer
let count = 1
repeat count [repeated = 7]
local selected :: Integer
let values = [9]
foreach value in values [selected = value]
println((repeated, selected))   # (7, 9)
```

A reference retains cell identity after a fresh declaration changes what the name denotes. Here the indirect write fills the earlier cell:

```
local answer :: Integer
let address = &answer
local answer = 20
*address = 10
println((*address, answer))    # (10, 20); passes --typecheck
```

A known call contributes effects only when invoked. `rebind` deliberately selects the captured cell; ordinary `answer = 42` inside `fill` would create a function-local binding and leave the outer hole empty.

```
local answer :: Integer
fill = func() [rebind answer = 42]
fill()
println(answer)                # 42; passes --typecheck
```

Errors separate paths too. The first assignment below finishes before the division fails. In the second example the catch handler supplies the fill.

```
local completed :: Integer
try [completed = 3; 1 / 0]
local recovered :: Integer
try
    1 / 0
```

```
catch problem
    recovered = 5
end
println((completed, recovered)) # (3, 5); passes --typecheck
```

Moving `1 / 0` before `completed = 3` leaves a hole, so the checker flags the later read and runtime refuses it. A failing type conversion likewise does not count as a fill. `attempt`, `otherwise` and finally blocks obey the same principle: retain writes that completed on each path reaching the read.

The analysis remains bounded. It recognizes literal evidence and simple bindings, direct aliases and simple known function bodies; it never runs user code to discover effects. Unknown calls, recursion, overloaded clauses, partial applications, named-plus-rest calls, lazy/refinement/structural-pattern parameters, function-local `global` marks, arbitrary contracts and loop-carried alias targets remain conservative boundaries. Mutable collection contents can change through aliases, so an opaque call invalidates their nonemptiness evidence. A clean check is not a complete safety proof. The same rules apply to all six fresh binding spellings and typed or untyped bare declarations and explicit `= _` forms. The runnable `local_stabilization.ax` showcase in `tests/axioma/showcase/` combines imperative, functional, reference and knowledge-representation examples; VM expansion is deferred.

Optional: what the checker can establish

The runtime rule remains simple: do not read a cell until a value has been successfully assigned. Checker precision means recognizing more evidence for that rule and explaining uncertainty accurately.

Diagnostic	Meaning	Consequence
Definitely uninitialized	The tracked cell has no value at the read	Fix the missing fill
May be used before assignment	Some paths may reach the read without a value	Fill every reaching path or simplify the proof
Analysis incomplete	A call or operation exceeds the supported analysis	Inspect its effects; runtime checks still apply

Both read diagnostics stop `--typecheck`. An incomplete-analysis note alone is informational; a later read may also produce a possible-read diagnostic. Normal execution warns about checker errors and still enforces runtime guards. A clean check is useful evidence, but it does not certify arbitrary program behavior.

Default arguments now contribute evidence when the function is called. Named arguments replace their defaults; a supplied argument prevents its default expression from running. These calls both pass `--typecheck`:

```
local answer
fill = func(n = 42) [rebind answer = n]
fill()
println(answer)                # 42
```

```
fill(n: 43)
println(answer)                # 43
```

Defaults run in parameter order and can use earlier converted parameters. Positional rest arguments supply a known Array, while exact Integer addition, subtraction, multiplication and comparisons can establish simple conditions and counts. Here the count is one and the rest Array has one element:

```
local repeated
repeat (2 - 1) [repeated = 7]
local selected
select = func(...values) [foreach value in values [rebind selected = value]]
select(9)
println((repeated, selected))  # (7, 9); passes --typecheck
```

A default is still a computation that can fail. This intentional failure is reported by `--typecheck`, and runtime also refuses the final read:

```
local answer
fill = func(n = 1 / 0) [rebind answer = n]
try [fill()]
println(answer)                # Error: uninitialized
```

The error happens before the body writes `answer`. If the default first fills that cell and then fails, the completed fill survives the caught error.

Repeated calls can reuse an effect summary for matching arguments and captured cell facts. It records normal completion and error exits separately. A changed alias or a newly created escaping cell requires fresh analysis. This reuse does not run a function, cache its runtime result, or change when defaults execute.

Some valid programs remain outside this proof. Lazy parameters are one example:

```
local answer
fill = func(n/lazy) [rebind answer = n]
fill(42)
println(answer)                # 42 with --no-typecheck
```

Here the checker reports an incomplete call analysis and a possible early read, so `--typecheck` stops. The runtime does execute the body and fill the cell. Understanding that difference is preferable to inventing a default value merely to silence a warning. The runnable `tests/axioma/showcase/checker_precision.ax` collects supported cases. Recursion, dispatch, deeper alias reasoning and VM expansion remain separate work.

Optional: bare declarations and the cell they name

`local result :: Integer` means exactly `local result :: Integer = _`. The shorter form declares a type and reserves a cell; it does not supply a value. The same choice is available across the fresh named

binding family:

One successful fill, then immutable	First fill, then mutable
`let x :: T`	`var x :: T`, `local x :: T`
`val x :: T`	`let mut x :: T`
	`let mutable x :: T`

```
local result :: Integer
if true then [result = 10] else [result = 20]
println(result)                # 10
let callback :: Function
callback = func(n) [n + 1]
println(callback(2))           # 3
var count :: Integer, label :: String
count = 2
label = "ready"
println((count, label))        # (2, "ready")
```

Each name in a bare batch may have its own type: `local label, count :: Integer` constrains only `count`. Bare patterns remain refused because destructuring needs an existing value. A newline or statement terminator ends a declaration, except that a following = continues an initializer even across a newline. `const`, `declare`, and `rebind` keep their existing forms; detached `x :: T` remains an annotation rather than this fresh-cell declaration.

Follow the cell, not just its name. Assignment fills the existing cell; a fresh declaration creates another cell. The following is an intentional error:

```
local result :: Integer
if true then [local result = 10] else [local result = 20]
println(result)                # ERROR: outer result is still uninitialized
```

Both branches initialize their own short-lived shadow. The checker diagnoses the final read. Replace the two inner `local result` declarations with ordinary `result` assignments, as in the first example, to fill the outer cell instead.

Visibility is sequential, with no hoisting: earlier expressions and closures keep the cells visible when they were evaluated. Creating a closure that reads a hole does not read it yet:

```
local answer :: Integer
peek = func() [answer]
answer = 42
println(peek())                # 42
```

Calling this known `peek()` before the fill is diagnosed by the checker and fails at runtime. Defining it does neither read nor fill `answer`. Unknown callees remain a runtime boundary. Prefer an immediate value when available; use the bare spelling or = _ when a later computation must supply the first value. Both teach deferred initialization without inventing a default.

Optional: initialization and type are separate choices

An untyped bare declaration reserves a cell without deciding its value type:

```
local answer
answer = 42
answer = "ready"
println(answer)                # ready
let once = _
once = 7
println(once)                  # 7
```

`local answer` aliases `local answer = _`; neither fixes its type. Add `:: Integer` to constrain later values. `let once` aliases `let once = _`; immutability forbids a second fill. All six fresh named forms follow these rules. Untyped holes currently refuse under `--vm`; use the evaluator.

A failed computation does not supply the first value:

```
let answer
try [answer = 1 / 0]
answer = "recovered"
println(answer)                # recovered
```

Reading `answer` between the failed computation and its successful fill would error. The same is true after a rejected typed fill. In contrast, assigning `none` or `om` succeeds: absence and undetermined knowledge are real values, not an uninitialized cell. Taking an address and capturing a cell in a closure do not read it; using its stored value does.

Parameters and patterns introduce their own bindings. They do not fill an outer hole that happens to have the same name. A declaration batch evaluates all right-hand sides in the preceding scope before introducing its new cells. A lazy `declare` group follows another rule: its computations can refer to all names in that group. These are intentional differences in binding behavior.

An immutable binding also does not freeze a mutable object stored in it. The repository's `binding_contracts.ax` showcase exercises these distinctions, along with module exports, references and deferred computations. A logical variable introduced by a query or quantifier is another role: do not model it as an unreadable imperative cell or substitute `om` for a missed assignment.

Empty containers: `let name :: Type = default`

```
#language axioma/all
let xs :: Array = default
println(len(xs))    # 0 — safe; xs is a real empty array
```

`default` is allowed only for types that already have an honest **identity empty** — the same group as nullary constructors:

`Array`, `Set`, `Tuple`, `Dictionary`, `Bytes`, `Bag`, `Stack`, `Graph`.

Why Integer is not `default → 0`

```
let n :: Integer = default   # ERROR
let n :: Integer = 0         # Ok — you asked for zero
let n :: Integer = _         # Ok — hole; assign before use
```

Zero is ordinary data. If the language filled every "unset" integer with `0`, a forgotten assignment would look like a correct program that multiplies by zero or compares equal to zero — a silent wrong answer. Axioma's rule matches its constructor surface: **containers may start empty; scalars do not get silent zeros.** You may always write `= 0`, `= ""`, or `= false` when that is what you mean.

Bottoms are still something else

```
let a = none                 # absence (a value)
let b = om                   # undetermined (a value); a == b is false
let c :: Integer = _         # hole (not a value)
let d :: Array = default     # empty array (a value)
```

Use _ only for "not ready yet," `default` only for "start empty container," and `none`/`om` when you mean one of Axioma's two bottoms.

Full reference: *Axioma Manual* § Fresh declarations with `let` (Uninitialized slots and identity defaults); everyday patterns in *Axioma Pragmatics*.

16.2 The `while` loop

Mutation pairs with iteration. Axioma's `while` loop runs a body as long as a condition is true:

```
#language axioma/beginner
i: 1
total: 0
while (i <= 10) [
  total = total + i;
  i = i + 1
]
println(total)   # 55
println(i)       # 11
```

The same body can end with a keyword instead of brackets. Put the body on the next line and finish it with `end`:

```
#language axioma/beginner
i: 1
total: 0
while i <= 10
```

```
  total = total + i
  i = i + 1
end
println(total)   # 55
println(i)       # 11
```

Both spellings run the same loop. Indentation makes the body readable; the closing `end` determines where it stops.

This is identical to:

```
#language axioma/beginner
sum_acc: func(i, total, limit) [
  if i > limit then total
  else sum_acc(i + 1, total + i, limit)
]
sum_to: func(limit) [ sum_acc(1, 0, limit) ]
println(sum_to(10))   # 55
```

The `while` loop is the accumulator recursion from Chapter 15 *written as iteration*. Three pieces map across:

Recursion	Loop
accumulator parameter	mutable local
recursive call with +1	`i = i + 1` and re-test
base-case return	exit when condition false

Every `while` loop has an equivalent accumulator-recursion; every accumulator-recursion has an equivalent `while` loop. Which one you reach for is a *style* choice: an accumulator recursion whose call is in tail position runs in constant stack space (Ch. 15's self-tail-call optimization), just like the loop.

> **Beginner-mode pragma.** The `axioma/beginner` subset *allows* both `x: …` and `x = newval`. The early chapters of this book stuck to *no mutation* as a teaching choice — every chapter through 15 worked without ever using persistent state as a design technique. That discipline was for clarity, not language enforcement. Starting with this chapter, mutation is on the table.

16.3 First worked example — counters with private state

A counter is the smallest interesting use of mutation. We met it already in Ch.12's open exercise:

```
#language axioma/beginner
make_counter: func() [
  n: 0;
  func() [
    rebind n = n + 1;
    n
```

```
  ]
]
c: make_counter()
v1: c()
v2: c()
v3: c()
println(v1)   # 1
println(v2)   # 2
println(v3)   # 3

# A second counter has its own n
second_counter: make_counter()
w1: second_counter()
println(w1)   # 1
v4: c()
println(v4)   # 4    (c's count continues independently)
```

The local `n` is captured by the inner `func() [...]`. When the outer `make_counter` returns, *the inner function still holds a reference to n*. Mutating `n` inside the inner function mutates *that captured cell*. Every call to `make_counter()` produces a *new* `n` cell, so different counters don't share state.

This is the **closure with mutable state** pattern. It's how JavaScript implements every object that has private fields, how Smalltalk does instance variables, how Lua does objects without an `object` keyword. The combination — *function that closes over a mutable local* — *is* an object in everything-but-name.

16.4 Second worked example — a state machine

A **state machine** is a system whose behavior depends on *which state it's currently in*. The canonical example: a turnstile. Two states (*locked*, *unlocked*) and two events (*coin*, *push*) with this transition table:

Current state	Event	Next state	Side effect
locked	coin	unlocked	—
locked	push	locked	— (refused entry)
unlocked	coin	unlocked	— (no-op)
unlocked	push	locked	increment count

In code:

```
#language axioma/beginner
state: "locked"
people: 0

coin: func() [
  if state == "locked" then [rebind state = "unlocked"]
  else [rebind state = state]
]
```

```
push_t: func() [
  if state == "unlocked" then [
    rebind state = "locked";
    rebind people = people + 1
  ]
  else [rebind state = state]
]

coin();   push_t()      # unlock, walk through  → 1 person
coin();   push_t()      # unlock, walk through  → 2
push_t()                # refused (already locked) → still 2
coin();   push_t()      # unlock, walk through  → 3
println(state)    # locked
println(people)   # 3
```

Two mutable variables (`state`, `people`) hold the machine's *memory*. Two functions (`coin`, `push_t`) read the current memory, decide what to do, and update.

> **Why the `else [state = state]` no-op?** Axioma's `if` is an *expression* — every branch must produce a value. When we have nothing meaningful to do in the `else`, a no-op assignment satisfies the parser. A real-language alternative would be a separate `when cond [...]` statement (no else branch), but that's an intermediate-tier feature outside the beginner subset.

State machines are everywhere: UI widgets (a button is *hover/pressed/disabled*), network protocols (`SYN-SENT`, `ESTABLISHED`, `CLOSE-WAIT` in TCP), parsers (lexer states for a tokenizer), game logic (player *alive/dead/respawning*), business rules (an order is *placed/paid/shipped/delivered*). Two mutable variables and a small function per event is enough for surprisingly complex behavior.

16.5 Third worked example — collecting results into an array

A practical mutation pattern: instead of building a list via recursion, *append to a growing array* in a loop.

```
#language axioma/beginner
# Build the array [1*1, 2*2, …, n*n] using a mutating accumulator
squares_to: func(n) [
  i: 1;
  acc: [];
  while (i <= n) [
    acc = acc + [i * i,];
    i = i + 1
  ];
  acc
]
println(squares_to(5))   # [1, 4, 9, 16, 25]
```

`acc` starts empty and grows by one element per iteration. The `[i * i,]` builds a one-element array (the trailing-comma singleton from §6.6); `acc + …` concatenates.

A cost note. Each `acc = acc + [v,]` does an `O(n)` array concatenation — `+` always builds a new array — so the loop totals $O(n^2)$. The true mutating grow is the bare statement `push(acc, i * i)`: it extends `acc` in place at amortized `O(1)`, and so does the functional shape (`map(func(i) [i * i], range(1, n))`, `O(n)` overall). For small `n` all three are fast. The lesson: *mutation doesn't automatically make code faster.* The cost is in the operations you choose, not the imperative-vs-functional surface.

16.6 When mutation is the right tool

Three cases where reaching for mutation is the *clearer* choice, not just an optimization:

1. **State machines** (§16.4). The whole *point* is that memory changes over time. Threading the state through a stateless accumulator is possible but contorted; explicit `state = newstate` *reads* like the spec ("on event X, transition from A to B").
2. **Long-running loops over external input** — reading a file, polling a sensor, processing a network connection. You can't pre-compute the input; you *receive* it incrementally. A loop with mutable variables matches the streaming structure.
3. **Performance-critical inner loops** where the constant-factor cost of allocating new closure accumulators is measurable. (Rarely a concern at this level; comes up in numerical code and game engines.)

16.7 When mutation is the wrong tool

Three cases where mutation makes code worse:

1. **Pure computations.** Computing the area of a triangle? Use a fresh binding. There's no state to track; nothing changes over time. Reaching for `bind; assign; assign` here just adds noise. Functional purity makes pure code easy to read.
2. **Shared mutable state in concurrent code.** Multiple threads / coroutines / actors reading and writing the same mutable variable is the *#1 source of bugs in real-world systems*. Axioma doesn't (yet) expose concurrency primitives, but when it does, the rule will be: avoid sharing mutable state if you possibly can.
3. **"Save typing by mutating in place."** Re-assigning a variable to mean a different thing inside the same function is *cheap to write and expensive to read*. Reviewers (including future-you) have to track the variable's value at every line. Use a new binding with a meaningful name; the variable cost is small, the clarity gain is large.

The rule of thumb: **mutation is a tool for modeling time and change**. If the *concept* you're modeling changes over time (a counter, a state machine, a UI widget, a game character's HP), mutation is right. If the concept is *timeless* (a math function, a data transformation, a predicate), a binding is right.

16.8 Reflection — three regimes of variable

Surveying everything we've taught about variables:

1. **Pure binding** (Chapters 1–14): `x: value` — names a value; the value never changes through the lifetime of the binding. Most of the textbook.
2. **Accumulator parameter** (Chapter 15): the carried-along "current value" of a recursion. Looks pure (no reassignment), but the *role* is the same as a mutable counter — the value evolves through the computation.
3. **Mutable variable** (this chapter): `x = newvalue` — the binding stays, but the value it holds changes over time.

Each is correct in its place. Pure binding is the default because it's the easiest to reason about. Accumulator recursion is the bridge: it gives you the *evolving-value* feel without leaving the functional discipline. Mutation is the imperative tool: when you genuinely need a value to change, just *change it*.

The HtDP discipline — "data definition drives function template; never mutate; build new things instead of modifying existing things" — buys you safety. Mutation trades some of that safety for power. Real-world programmers use both, and *knowing when* is half of being a good programmer.

16.9 Places and values: Array vs List

The regimes of §16.8 come to a point in Axioma's two sequence types, because they split exactly along the divide this chapter is about.

An **Array is a place.** It has *identity over time*: two names can refer to the same array, and a mutation through either name is visible through both. Places are what imperatives act on, so an array answers command sentences:

```
place: [1, 2, 3]
same_place: place   # same_place names the SAME place
place push 4        # a command: "grow, in place"
println(same_place) # [1, 2, 3, 4] — same_place saw it
```

A **List is a value.** It has *no* identity over time — like the number 5, it simply is its content. You never change a value; you compute new values from it:

```
l: `[1, 2, 3]
m: cons(0, l)       # a NEW value; l is untouched
println(l)          # `[1, 2, 3]
println(m)          # `[0, 1, 2, 3]
```

`cons` is O(1) precisely *because* nothing ever mutates: `m` simply points at `l` as its shared tail, and that sharing is safe only while cells never change. This is also why a list *refuses* command sentences — `l push 3` is an error that points you back at `cons`. If lists could be commanded, growing one would silently grow every list sharing its spine.

The rule, which the language itself enforces: **a place takes imperatives; a value takes expressions.** Reach for the Array when your program is about change over time — this chapter's territory. Reach for the List when you want Part I–IV's guarantees back while still growing data: sharing without fear,

old versions surviving for free, and equational reading (`m == cons(0, l)` stays true forever, because nothing can edit either side).

16.10 Editing an array without replacing the binding

Array mutation changes the shared container. Rebinding a name changes which value that name denotes. Test aliases to see the difference.

```
items: [10, 20, 30]
shared_items: items
removed: delete items[2]
println(removed)                  # 20
println(shared_items)                    # [10, 30]
items[1:1] = [1, 2]
extend(items, [40, 50])
println(items)                    # [1, 2, 30, 40, 50]
println(shift(items))              # 1
unshift(items, 0)
println(splice!(items, 2))         # 2
println(items)                    # [0, 30, 40, 50]
```

`delete` returns the removed value; `remove_at` returns the edited array. Slice replacement may change length. Out-of-bounds element writes do not grow an array: use an insertion or extension operation. `reverse(a)` makes a copy; `reverse!(a)` changes the array. Mutation verbs such as `push` already mutate and do not gain a `push!` alias. The name `splice!` avoids conflicting with metaprogramming's `splice`.

16.11 Product loops and explicit exits

Use the full language for this extension: remove the beginner pragma from the file or start a separate example. Several generators in one header walk a Cartesian product. The rightmost generator varies fastest; a later generator can depend on an earlier one.

```
visits: []
for row in 1..2, col in 1..3 [push(visits, (row, col))]
println(visits)     # [(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3)]

selected: []
'search: for row in 1..3 [
  for col in 1..3 [
    if row == 2 and col == 2 then break 'search
    push(selected, (row, col))
  ]
]
println(selected)   # [(1, 1), (1, 2), (1, 3), (2, 1)]
```

`for` and `loop` with `end`

In the full language, `for` visits each value in its source and `loop` can repeat a body a given number of times. Their bodies can also use newlines and `end`, just like the `while` example in §16.2:

```
total: 0
for n in 1..4
  total = total + n
end
println(total)   # 10

loop 2
  println("again")
end
# prints again on two lines
```

The range `1..4` includes both endpoints. `repeat` is another spelling of `loop`. Nested loops each need their own `end`; a combined header needs one. Choose brackets or `end` independently for each construct. The beginner subset already allows `for` and `loop` (Chapter 2a). This section is the product header and labelled exit, which 2a deferred. No `do` is required.

A combined header is **one loop**: its unlabelled `break` exits the product. Two nested loop statements have separate exits; an unlabelled `break` exits only the innermost. A label makes the intended target explicit. `continue 'search` advances the labelled loop to its next iteration. Matrix iteration already walks cells; use row/column index loops when the axes themselves are part of the task.

16.12 Handling an error and finishing cleanup

`try(expr)` captures an expression's failure as an Error value. The multiline `try/catch/finally/end` form adds handlers and cleanup.

```
finished: false
result: try
  1 / 0
catch problem is DivByZero
  "cannot divide by zero"
finally
  rebind finished = true
end
println(result)                  # cannot divide by zero
println(finished)                # true
```

The selected handler supplies the result. Cleanup runs on success and failure; its normal value does not replace the result. A cleanup failure can replace a pending result, so cleanup needs care too. `rebind` here explicitly updates the outer cell. An unmatched captured error remains an Error value; use `raise(problem)` when it should propagate to a caller. Selected catches test the Error's type, not a substring of its message. This end-form remains interpreter-only; the VM refuses it.

16.13 Optional: closed mutable records

Chapter 18 · Types and Data Modeling presents the full declaration family and how to choose a representation.

A programming record can have a fixed set of typed fields while allowing their values to change. Use `struct mutable` in the type declaration. This is separate from the mutability of the binding: `let cursor` keeps the name fixed, while `cursor.x` can change. Two separately constructed mutable records have distinct identity and compare unequal; an alias refers to the same record.

```
type PointCursor = struct mutable
    x :: Float
    y :: Float
end
let cursor = PointCursor(1.0, 2.0)
let alias_cursor = cursor
alias_cursor.x = 3.0
println(cursor.x)
let moved = {cursor with y: 4.0}
println(moved.y)
println(cursor.y)
println(cursor == moved)
```

This prints `3.0`, `4.0`, `2.0`, and `false`. `cursor.x = "three"` refuses without changing the field; `cursor.z = 1.0` refuses because `z` is undeclared. A field reference `&cursor.x` enforces the same checks. Index writes are not a record API. The functional update makes a separate record and shares untouched nested values.

`copy(cursor)` also makes an independent record, and the copy keeps the field checks. Because a mutable record's identity is not its contents, ordering verbs such as `sort`, `min` and `max` refuse it, as `<` does; order by a field instead, for example `sort_by(func(c) [c.x], cursors)`. `PointCursor has z: 1.0` refuses: the field list is fixed by the declaration.

Omit the modifier for an immutable record. Its fields cannot be replaced, though nested mutable values keep their own behavior. Braces give the equivalent compact form `type PointCursor = struct mutable {x:: Float, y:: Float}`. Optional spellings are `record` for `struct` and `mut` for `mutable`, after either kind marker: `struct mut`, `record mutable`, and `record mut` all declare mutable structs. They change neither the closed field contract nor equality and identity; both brace and `end` bodies still work. See §18.5's optional spelling aliases and Exercise 18.14 for checked examples. Neither record kind creates domain entities, registers a concept extent, or supports domain inheritance. Use the existing `concept Person` for that model.

The unified family also accepts `type Shape = variant Circle(radius:: Float) | Origin`, `type Direction = enum North, South`, and `type Person = concept`. Their short `data`, `enum`, and `concept` declarations still work. Dictionary schemas remain structural: `type Row = {x:: Integer}` accepts matching dictionaries. The explicit `schema` marker opens an end body. A concept end body opens with `with`, as in `concept Person with` followed by `name: ""` and `end`; a bare concept declaration never requires an end. Aliases, tuple products, unions and opaque wrappers keep their existing syntax. These records use the evaluator; the VM explicitly refuses them. The static checker validates declarations,

known constructor types and the struct writes it can see; runtime checks govern actual writes.

16.14 Optional: make scope explicit with `local`

Axioma is a teaching language with several spellings for related ideas. The useful question is not just "which keyword do I recognize?" but **which binding will this statement select, and may that binding change?** This section brings the forms from Chapter 2, Chapter 12, and this chapter into one comparison. Use full host Axioma for these examples; the optional lesson goes beyond the beginner subset.

An initialized, fresh, mutable declaration

`local` emphasizes **where** a new binding lives; `var` emphasizes **that its value can vary**. Initialized declarations under either spelling create the same fresh mutable cell:

```
local count :: Integer = 0
count += 1
println(count)                # 1
```

A cell is the storage associated with one binding. Two bindings can have the same name and still have different cells. `local` is optional: ordinary Axioma assignments already respect function boundaries. Use it when you specifically want a **new binding**, including when another binding has the same name.

Assignment versus fresh declaration

```
function local_scope_demo()
  score = 10
  if true
    local score = 20
    score += 1
    println(score)
  end
  score
end
println(local_scope_demo())
```

This prints `21`, then `10`. The branch gets its own writable `score`; after the branch, that shadow is out of scope and the function still has its original `score = 10`. Removing `local` makes the same program print `21`, then `21`: ordinary assignment finds and updates the existing cell within the function. Exercise 16.8 compares the two programs and an immutable variation.

A loop-body `local` is fresh on every iteration. At file/module level, it creates a binding in that unit; **it does not mean private or prevent exporting the name**. Being out of scope also does not mean a cell is destroyed: a closure can keep it alive, as we will see below.

The main binding forms

Spelling	Which binding?	May it change?
`x: value` or `x = value`	Find an existing binding within this function frame; otherwise introduce one here	Yes, if its existing contract permits writes
`let x = value`	Always a fresh binding	No
`val x = value`	Same as `let` for named declarations	No
`var x = value`	Always a fresh binding	Yes
`let mut x = value`	Same as `var`	Yes
`let mutable x = value`	Same as `var`	Yes
`local x = value`	Always a fresh binding in this scope	Yes; typed holes allow a later first fill
`rebind x = value`	Nearest existing binding, searching across function boundaries; error if absent	Only if the selected binding permits writes
`global x = value`	Current file/module binding; may introduce it	Subject to that binding's contract
`global x`	Direct subsequent reads and assignments in this function frame to the module binding	Does not initialize a value itself
`const LIMIT = value`	Top-level constant name	Neither reassignment nor shadowing

`rebind` and initialized `global` also accept `:` instead of `=`. Fresh keyword declarations and `const` require `=`. For ordinary value binding, `:` and `=` are interchangeable; there is no separate `:=` operator. A parenthesized head, such as `double(x) = 2 * x`, defines a function instead.

`let` versus `const`. A `let` binding cannot be changed, but a fresh binding can shadow it. A `const` name cannot be shadowed. Fixing a binding is also distinct from deeply freezing its value: §16.13's `let curso r` names a mutable record whose fields can change.

Aliases and teaching intent. `val` emphasizes a fixed value; `var`, `let mut`, and `let mutable` emphasize mutation; `local` emphasizes lexical scope. These spellings share binding rules, so learning an alias does not require learning another memory model.

`local`, `rebind`, and `global` select different places

Run this program first with ordinary assignment inside `inner`:

```
x = 10
function outer()
  x = 20
  function inner()
```

```
    x = 30
    x
  end
  result = inner()
  (result, x)
end
println(outer())
println(x)
```

It prints `(30, 20)`, then `10`. Run each substitution below as a separate program, starting again with module `x = 10` and outer-function `x = 20`:

Statement inside `inner`	`outer()` prints	Module `x` afterward
`x = 30`	`(30, 20)`	`10`
`local x = 30`	`(30, 20)`	`10`
`rebind x = 30`	`(30, 30)`	`10`
`global x = 30`	`(30, 20)`	`30`

Ordinary assignment and `local` agree here because `inner` has no existing local `x`. The branch example shows when they differ. `rebind` searches from the current scope outward, so an existing nearer cell wins; it does not always skip to an outer function. `global` explicitly chooses the module cell. Exercise 16.9 and its solution run all four cases.

An active local declaration and `global` cannot select the same name in one function frame:

```
function conflicting_scope()
  global conflict_value
  local conflict_value = 1
end
conflicting_scope()            # ERROR: already declared global
```

The reverse order is also rejected. This protection includes `let`, `var`, their aliases, and pattern declarations. Nested **functions** have independent frames; a nested block does not. The check concerns active declarations during execution, not a whole-function scan of branches that have not run.

Fresh bindings preserve closure captures

Each declaration creates another binding; it does not overwrite the previous one:

```
local x = 1
peek = func() [x]
local x = x + 1
x += 10
println(peek())                # 1
println(x)                     # 12
```

The initializer reads the old `x` before the new binding exists. `peek` retains the old binding, while subsequent statements use the new one. Repeated `let` and `var` declarations have the same fresh-binding behavior; `let` additionally refuses later writes to the cell. Compare this with Chapter 12's closure model and §16.3's counter, which deliberately updates its captured cell with `rebind`.

Annotations, multiple names, and patterns

These are variations on a binding form, rather than additional scope policies:

```
local ratio :: Float = 3
ratio += 1
println(ratio)                # 4.0
local a, b = 1, 2
local a, b = b, a
println((a, b))               # (2, 1)
local (left, right) = (3, 4)
left += 1
println((left, right))        # (4, 4)
```

Annotations retain the existing conversion and later-write checks. Multiple initializers evaluate before the new bindings are installed, so the second multi-declaration reads both previous values before introducing either shadow.

Pattern form	Behavior
`(a, b): pair` or `(a, b) = pair`	Destructure using ordinary find-or-update rules
`let (a, b) = pair`	Fresh immutable bindings
`var (a, b) = pair`	Fresh mutable bindings
`local (a, b) = pair`	Fresh mutable bindings
`let mut (a, b) = pair`	Same as `var`; `let mutable` also works

`local` also accepts the existing array and dictionary patterns. `val` retains its pattern restriction: use `let` for immutable pattern declarations.

Initialization, holes, and defaults

`local` shares `var`'s initialization choices. Compare §16.1b:

```
local ready = 0                 # Initialized mutable binding
local empty :: Array = default  # A real empty array
local later :: Integer          # Uninitialized mutable slot; same as = _
let once :: Integer = _         # One successful fill is allowed
later = 7
once = 8
println((ready, len(empty), later, once))   # (0, 0, 7, 8)
```

Refused form	Reason or replacement
`local x`	Supply an initializer or an explicit `:: Type`
`local x: 1`	Fresh declarations require =
`local x += 1`	Declare first, then use `x += 1`
`local mut x = 1`	`local` already makes a mutable cell

A typed `default` is allowed for supported container identities; it does not invent scalar defaults such as zero. `none` and `om` remain actual values, unlike an uninitialized hole. Reading `later` before its first assignment would fail; assigning `once` a second time would fail.

The other binding-related families

Some forms address evaluation timing or introduce names as part of another construct. They answer different questions from the scope/mutability table:

Family	Relationship to `local`
`declare x = expression`	First read evaluates and caches the expression; a `local` initializer evaluates immediately, while a typed hole has no computation to run
`declare/persist`, `declare/transient`	Add session-persistence choices to deferred bindings
`define x = expression`	Value form is deferred and memoized; other `define` forms introduce definitions or knowledge claims
`where` sections	Supporting bindings are written after the result but execute before it
Parameters, loop variables, comprehension generators, match captures	Already introduce scoped bindings; no `local` is needed
Pattern guards, quantifier variables, relation logic variables	Bind according to matching, quantification, or unification rules
Function, type, concept, module declarations	Introduce named language entities under their own definition rules

For example, a deferred binding computes once even when read twice:

```
calls = 0
tick() = [
  rebind calls = calls + 1
  calls
]
declare delayed = tick()
define other = tick()
println(calls)                # 0: neither initializer has run
println(delayed)              # 1: first computation
println(delayed)              # 1: cached
```

```
println(other)                # 2: second computation
println(other)                # 2: cached
println(calls)                # 2
```

`where` changes presentation and execution order. A comprehension's intermediate binding is already iteration-local:

```
calc_change(owed, paid) = [
  max(change, 0)
  where change: paid - owed
]
println(calc_change(10, 25))    # 15
println([y | x <- [1, 2, 3], y: x * 2])   # [2, 4, 6]
```

The comprehension binding clause uses `:`; neither `y := expression` nor `let y = expression` is its syntax. These examples name values in their own contexts without changing the enclosing program's same-named bindings.

Under the hood: one shared binding mechanism

This optional implementation view connects the lesson to Chapter 22's interpreter. The lexer recognizes reserved keyword `local`; an ordinary identifier with that spelling must be escaped as `$local`. POP-11 word lists still treat `[local x]` as the two strings `"local"` and `"x"`.

The parser builds the existing `ast.LetStatement` with `Fresh: true` and `Mutable: true`, retaining the original keyword for diagnostics. The evaluator runs initializers in the old environment, creates a new lexical environment layer, installs the bindings there, and lets subsequent statements use that layer. This gives shadowing while preserving earlier closures. Scope checks look through active fresh-binding layers to detect local/global conflicts. Help, highlighting, and completion recognize the spelling as well.

There is no whole-function scan that retroactively makes a name local: the declaration takes effect when execution reaches it. The spelling supports a Julia-oriented scope lesson without importing Julia's whole-scope analysis or REPL soft-scope rules. Axioma's ordinary writes still stop at function frames.

Execution boundary. Run the nested-scope examples in the evaluator. Simple initialized locals and supported mutable typed holes work under `--vm`, but fresh bindings in nested blocks produce an explicit compilation refusal. `local` also inherits the existing fresh-binding restrictions for imported-file top level and `trace` blocks. The examples here do not claim full VM parity or whole-program static scope analysis.

For a teaching sequence, start with `:/=`, introduce `let/var` to separate fresh declaration from mutation, and then compare `local`, `global`, and `rebind` when discussing scope. The aliases have distinct teaching uses while sharing a consistent set of binding rules.

16.15 Optional: scoped knowledge-base transactions

The error forms in §16.12 let us decide where to recover. When several related knowledge-base changes must succeed together, `with transaction [ body ]` also decides which changes to keep. Readers

who have not yet studied relations can return here after Chapter 21; Appendix E.10 lists the mutation operations.

```
relation batch_fact(x)
result: with transaction [
    insert("batch_fact", "kept")
    42
]
println(result)                          # 42
println(len({ X | X <- batch_fact(X) }))      # 1

failed: try(with transaction [
    insert("batch_fact", "discarded")
    1 / 0
])
println(error?(failed))                  # true
println(len({ X | X <- batch_fact(X) }))      # 1
```

The first scope commits and returns its last value. The second rolls back before the surrounding `try` captures the error. Keep `with transaction [` on one source line; its bracketed body can span lines. An empty body returns `none`. The body has ordinary enclosed-block scope. A `return`, `break`, or `continue` commits before continuing its normal enclosing function or loop control flow; it does not just leave the transaction body.

Only an escaping error causes rollback. Catching an error inside the body turns it into a value and permits commit. **Only journaled knowledge-base state is restored.** Mutable Arrays and other ordinary program effects remain:

```
relation scoped_fact(x)
events: []
failed: try(with transaction [
    insert("scoped_fact", "discarded")
    push(events, "attempt")
    1 / 0
])
println(error?(failed))                  # true
println(len({ X | X <- scoped_fact(X) }))     # 0
println(events)                         # ["attempt"]

captured: with transaction [
    insert("scoped_fact", "kept")
    try(1 / 0)
]
println(error?(captured))                # true
println(len({ X | X <- scoped_fact(X) }))     # 1
```

The journal covers facts, their journaled metadata, and unification bindings. It does not cover variable assignment, collection mutation, output, or files. Commit keeps those knowledge-base changes in the current session; it does not promise a durable disk write.

Transactions do not nest. The scope owns its journal: explicit `transaction_begin`, `transaction_commit`, and `transaction_rollback` calls inside it refuse before changing transaction state, even through a helper. If that refusal is caught inside the body, the original journal remains usable.

This is a synchronous full-interpreter feature. It cannot run inside a `produce` producer, and producer construction, suspension, or demand while the scope is active refuse. Eager `generate` is supported. The VM and knowledge-core subset refuse the form. It has no resource binder, multiple-resource header, other resource kind, or `end` variant. The word `transaction` remains available as an ordinary name elsewhere.

Exercises

Exercise 16.1 — `sum_with_loop`

Re-implement `sum_list` from Ch.7 using a `while`-loop with two mutable variables (`i` for index, `total` for running sum). Verify it matches §15.1's recursive `sum_list` on three inputs.

- `sum_with_loop([])` → `0`
- `sum_with_loop([7])` → `7`
- `sum_with_loop([1, 2, 3, 4, 5])` → `15`

Exercise 16.2 — `make_bank_account`

Write `make_bank_account(start_balance)` that returns a function. Each call to that function takes an amount and applies a deposit (positive) or withdrawal (negative) to the account, returning the new balance. Withdrawals that would overdraw the account return the unchanged balance instead.

```
acc: make_bank_account(100)
v1: acc(50)     # 150
v2: acc(-30)    # 120
v3: acc(-200)   # 120  (refused, would overdraw)
v4: acc(0)      # 120  (query balance)
```

Exercise 16.3 — `traffic_light`

Model a 3-state traffic light: `"red"` → `"green"` → `"yellow"` → `"red"` → … . Write `make_traffic_light()` that returns a function. Each call advances to the next state and returns the *new* state.

```
light: make_traffic_light()
println(light())   # green   (started red, advanced)
println(light())   # yellow
println(light())   # red
println(light())   # green
```

Hint: a local `state` variable plus an `if/else if/else` cascade in the inner function.

Exercise 16.4 — `count_passing`

A turnstile records how many people walk through. Use the §16.4 turnstile but expose `passing_count()` as a function that returns the current count *without* changing the state.

- `coin(); push(); passing_count()` → `1`
- `coin(); push(); coin(); push(); passing_count()` → `3`
- `push(); passing_count()` → `0` (no coin yet, refused)

Exercise 16.5 — `find_first_with_loop`

Given an array and a predicate, return the first element for which the predicate is true, or `none` if none qualifies. Use a `while` loop with a mutable `result` variable.

- `find_first_with_loop([1, 3, 5, 7], func(x) [x > 4])` → `5`
- `find_first_with_loop([1, 3, 5], func(x) [x > 100])` → `none`
- `find_first_with_loop([], func(x) [true])` → `none`

Compare your loop version against the recursive `find_first` from earlier chapters. Which is easier to read? Which is easier to verify correct?

Exercise 16.6 (open) — Conway's game-of-life cell

Model a *single* cell in Conway's Game of Life. The cell has a mutable state (`true` = alive, `false` = dead). Each generation, the cell takes an integer argument `neighbors` (count of living neighbors) and updates its state by the classical rules:

- live cell with 2 or 3 neighbors → stays alive
- live cell with anything else → dies
- dead cell with exactly 3 neighbors → comes alive
- dead cell otherwise → stays dead

Write `make_cell(initial_alive)` returning a function `step` that takes `neighbors`, mutates the cell, and returns the new state.

```
c: make_cell(true)
println(c(2))   # true  (lives)
println(c(1))   # false (dies)
println(c(3))   # true  (born again)
println(c(0))   # false (dies)
```

Then think: how would you orchestrate many cells in a grid? What's the *minimum* state each cell needs? The answers shape how you'd build a full game-of-life simulator.

Exercise 16.7 — an alias and a functional copy

Using PointCursor above, construct an alias and a functional copy with an updated x. Change the alias's y. Predict which of the three records observes that change, and explain why the copy has different identity even if all fields later agree.

Exercise 16.8 — local scope and mutable cells

Predict the two printed values in `local_scope_demo` above, then remove `local` and predict them again. Explain why both versions allow `score += 1`. Finally replace `local` with `let`: which statement fails,

and why?

Exercise 16.9 — choose the destination

Run the four substitutions in §16.14's `outer`/`inner` example as separate programs. Explain why ordinary assignment and `local` agree there, and why `rebind` and `global` disagree. Then put a `local x = 25` before `rebind x = 30` in `inner`: which cell will `rebind` update?

Exercise 16.10 — follow the captured cell

In §16.14's `peek` example, predict the output if you replace the second `local x = x + 1` with ordinary `x = x + 1`. Explain why a closure captures a binding rather than a frozen copy of its value. Then replace that second `local` with `let`: which later statement fails?

Exercise 16.11 — a hole has a type, not a default

In the deferred callback example in §16.1b, explain why no empty Function value is needed. Which operation fails if you read `callback` before its fill? Which fails if you assign it a second function after its fill? Change the declaration so a second function is allowed, while assigning an integer still fails.

End of Chapter 16. Chapter 17 closes Part V with a brief look at **streams** — values that *produce* their elements one at a time, on demand, without ever materializing the whole sequence. Streams and mutation together let you model genuinely *infinite* data.

Solutions to selected exercises: Chapter 16 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch16_solutions.md.)*

Exercise 16.12 — fill or shadow?

Declare `local result :: Integer`, then use an `if`/`else` to supply either 10 or 20 and print it after the branch. Explain why adding `local` to each branch assignment breaks the program. Create a closure reading `result` before the branch, but call it afterwards. Does defining it perform the read? What happens if you call it before the branch? Finally, replace the outer declaration with `let result :: Integer` and explain which later writes are allowed.

Exercise 16.13 — which paths need a fill?

In `choose_result`, replace `return 0` with the value `0`. Explain why the later read now needs another fill. Repair it without changing the declaration. Then move a `continue` before the assignment in the post-test example: which read becomes unsafe? Finally, explain why taking `&result` and fetching `:result` have different requirements before initialization.

Exercise 16.14 — justify the evidence

In the four examples under "evidence for a later fill," explain what proves the destination cell has a value. Change the count to zero, move the indirect write before the fresh shadow, omit `fill()`, and move the division before the first assignment, one change at a time. Which reads fail, and which still succeed? Finally replace `local` with `let`: which examples need more than one successful fill of the same cell?

Exercise 16.15 — four independent questions

Declare an untyped mutable cell, fill it with 42, then replace it with a string. Explain what changes if you add `:: Integer`, or use `let` instead of `local`. Move a read before the first assignment, then replace that first assignment with one that fails. Finally, explain why `none`, `om`, and an empty Array are readable values, and why an immutable binding can still hold a mutable Array.

Exercise 16.16 — evidence and uncertainty

Change `fill()` to `fill(n: 7)` in the failing-default example: why is it safe? Then make the default fill `answer` before dividing by zero. What survives the catch? Explain how an incomplete-call note, a possible-read diagnostic, and a definite-read diagnostic differ.

Exercise 16.17 — decide what a transaction keeps

Write a function that inserts a relation fact inside `with transaction` and then returns early. Verify both its result and the surviving fact. Next insert a second fact, append to an outer Array, and cause an error: catch it outside the scope and explain which effects survive. Move the catch inside the body and explain why the fact now remains. Finally, predict what happens when a helper calls `transaction_commit()` inside the scope, and explain why this scope cannot surround producer suspension. Use the full interpreter.

Chapter 17 · Streams

What this chapter is. A *small* introduction to a *big* topic: lazy sequences. A **stream** is a sequence that *generates* its elements on demand — you can take the first ten naturals without materializing all the naturals. Streams let you express genuinely infinite data: the natural numbers, the prime numbers, the Fibonacci sequence — as values you can pass around and compose, even though no finite memory can hold them. The cost is a small amount of ceremony: every stream element is wrapped in a *thunk* (a zero-argument function that, when called, produces the next piece).

Axioma does have built-in lazy forms — the generator expression `(x | x <- src, …)`, driven by `force`, `gen_take`, and `gen_next`; and `lazy e`, which is the *thunk itself* as a first-class value (§17.8), plus the replayable built-in `Sequence` (§17.10) — but this chapter builds streams *by hand* out of pairs and closures, exactly how SICP and HtDP build them out of cons cells. Seeing the mechanism is the teaching purpose; in production code, reach for the built-ins.

17.1 The shape — pair of (head, thunk)

A stream is a *2-tuple*:

```
stream = (head, tail_thunk)
       =  ( the_first_element,
            a_function_that_when_called_returns_the_rest_stream )
```

The crucial idea: the *rest* isn't stored — it's a *recipe* for producing the rest. Calling the tail thunk *evaluates the recipe* and yields another `(head, tail_thunk)` pair. No matter how long the conceptual sequence is, only *two pieces of data* live in memory at any moment: the current head and a single thunk.

In Axioma:

```
#language axioma/beginner

# A stream of natural numbers starting from k
nats_from: func(k) [
  (k, func() [nats_from(k + 1)])
]
```

`nats_from(1)` returns `(1, <function>)`. The function, when called, returns `(2, <function>)`. The pattern continues indefinitely — the *recipe* for "the next natural" is captured in the closure; no actual numbers beyond the current head are stored.

17.2 Operations on streams

Three operations are enough to do interesting work.

`stream_head` and `stream_tail`

```
#language axioma/beginner

stream_head: func(s) [ s[1] ]
stream_tail: func(s) [ s[2]() ]
```

`stream_head` reads the first element directly; `stream_tail` *calls* the thunk to get the next stream. Tuple indexing is 1-based (Ch.6).

`take_stream(n, s)` — extract n elements as an array

```
#language axioma/beginner

take_stream: func(n, s) [
  if n == 0 then []
  else [
    h: stream_head(s);
    t: stream_tail(s);
    push([], h) + take_stream(n - 1, t)
  ]
]
```

A recursion that walks n levels of the stream, peeling off heads as it goes. The stream itself is *unbounded*, but we stop after n peels.

```
nats_from: func(k) [ (k, func() [nats_from(k + 1)]) ]
stream_head: func(s) [ s[1] ]
stream_tail: func(s) [ s[2]() ]
take_stream: func(n, s) [
  if n == 0 then []
  else [
    h: stream_head(s);
    t: stream_tail(s);
    push([], h) + take_stream(n - 1, t)
  ]
]
println(take_stream(10, nats_from(1)))    # [1, 2, ..., 10]
println(take_stream(5, nats_from(100)))   # [100, 101, ..., 104]
```

`nats_from` is *defined* to be infinite. `take_stream` is the "sampling" tool that lets us look at any finite prefix.

17.3 Map and filter on streams

Streams support the same algebra as arrays — `map`, `filter`, `reduce`. But the operations work *lazily*: applying `map` to an infinite stream produces another infinite stream; no work happens until somebody asks for elements.

```
#language axioma/beginner

map_stream: func(f, s) [
  (f(stream_head(s)), func() [map_stream(f, stream_tail(s))])
]

filter_stream: func(pred, s) [
  h: stream_head(s);
  if pred(h) then (h, func() [filter_stream(pred, stream_tail(s))])
  else filter_stream(pred, stream_tail(s))
]
```

Notice the *shape*. `map_stream` returns a new pair: a new head (`f` applied to the original head) and a *new* thunk that recursively maps when called. `filter_stream` does the same, but if the head doesn't satisfy the predicate, it keeps advancing through the source stream until it finds one that does (then returns *that* element as the new head).

Compose them:

```
nats: nats_from(1)
squares: map_stream(func(x) [x * x], nats)
println(take_stream(10, squares))   # [1, 4, 9, ..., 100]

# Evens via filter
evens: filter_stream(func(x) [x % 2 == 0], nats)
println(take_stream(10, evens))     # [2, 4, ..., 20]

# Compose: squares of evens
even_squares: map_stream(func(x) [x * x], evens)
println(take_stream(5, even_squares))   # [4, 16, 36, 64, 100]
```

Three infinite streams (`nats`, `squares`, `evens`, `even_squares`) compose with operations you've seen before (`map`, `filter`). The lazy structure makes *infinite* sequences first-class — they fit in named variables, get passed to functions, get transformed.

17.4 Fibonacci via streams — the classic

The poster child for lazy sequences. The Fibonacci sequence is *defined* by a two-state recurrence (each term is the sum of the previous two). A stream-based version writes the definition directly:

```
#language axioma/beginner

fib_pair: func(a, b) [
  (a, func() [fib_pair(b, a + b)])
]
fibs: fib_pair(0, 1)

println(take_stream(10, fibs))   # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
println(take_stream(20, fibs))
```

`fib_pair(a, b)` says "the next Fibonacci is `a`; after that, the stream continues with `fib_pair(b, a + b)`." The `(b, a + b)` *is* the recurrence — every step the pair `(a, b)` rolls forward to `(b, a + b)`. The stream-thunk evaluates the recurrence on demand.

A "find the first Fibonacci greater than 1000" query becomes:

```
#language axioma/beginner

fib_pair: func(a, b) [ (a, func() [fib_pair(b, a + b)]) ]
stream_head: func(s) [ s[1] ]
stream_tail: func(s) [ s[2]() ]

first_gt: func(s, lim) [
  h: stream_head(s);
  if h > lim then h
  else first_gt(stream_tail(s), lim)
]

fibs: fib_pair(0, 1)
println(first_gt(fibs, 1000))   # 1597
```

We never materialize the *infinite* stream of Fibonacci numbers; we only walk the prefix needed to answer the question. That's the *whole point* of lazy evaluation — **pay only for what you ask for**.

17.5 Generator-as-function (the closure variant)

There's a second formulation that doesn't use pairs at all — just closures over mutable state, à la JavaScript or Python generators:

```
#language axioma/beginner

make_counter_gen: func(start) [
  n: start;
  func() [
    v: n;
    rebind n = n + 1;
    v
```

```
  ]
]
gen: make_counter_gen(10)
v1: gen()
v2: gen()
v3: gen()
println(v1)   # 10
println(v2)   # 11
println(v3)   # 12
```

The "generator" is a closure that, when called, yields the next value and updates its internal counter. No pair, no thunk — just mutation hidden inside a closure.

The two styles trade off:

Property	Pair-based stream	Generator-as-closure
Pure (no mutation)	yes	no — updates captured state with `rebind`
Composable (`map`, `filter`)	yes — lazy chain	hard — each call advances
Re-iterable from start	yes — re-call `take_stream(n, s)`	no — generator is consumed
Familiar to imperative programmers	less	more

Pair-based wins on *composition*; generator wins on *ergonomics inside a loop*. SICP / HtDP teach the pair-based version because it's the one that fits into the design recipe. Real-world Axioma you'd reach for whichever matches the ambient style of the surrounding code.

17.6 The big picture — streams vs arrays vs generators

Three idioms for "a sequence of values":

Form	Materialized?	Lazy?	Size limit	Composes with map/filter?
Array (`[1, 2, 3]`)	yes — all in memory	no	bounded by RAM	yes (eager)
Stream (pair + thunk)	only head + thunk	yes	unbounded	yes (lazy)
Generator (closure)	only current state	yes	unbounded	awkward

Each has a place. Arrays are great when you have a *finite* known-size collection and need random access. Streams are great when the sequence is *conceptually infinite* but you only need a prefix — primes, naturals,

Fibonacci, time-stamped events. Generators are great for *adapting external sources* (file readers, network streams) into the iteration style.

17.7 The built-in infinite sequence — the open-ended range

When this chapter was first drafted, Axioma had no native lazy-sequence type and this section flagged the gap (G-10 in the inventory): Haskell writes `[1..]` for the infinite list of naturals; Axioma's closest thing was a bounded `[1..n]`.

The gap has since closed, with almost exactly the wished-for spelling. The loop examples in this section use the full language; remove the beginner pragma in a combined file. `1..` is a first-class **open-ended range** — lazy and infinite — and a *lazy* (parenthesized) comprehension streams it on demand:

```
nats: 1..                          # the naturals, as a value
squares: (n * n | n <- nats)       # lazy comprehension — nothing runs yet
first(squares, 5)                  # → [1, 4, 9, 16, 25]

first_big: 0                       # search past a threshold: loop + break
for n in nats [
  if n * n > 1000 then [
    first_big = n * n
    break
  ]
]
first_big                          # → 1024
```

Membership stays O(1) (`10^9 in nats` → `true` without iterating), `zip(1.., xs)` numbers a list, and every *materializer* (`sum`, `len`, `array`, an eager comprehension) refuses an open range with a catchable error rather than hanging. See the manual's §4 *Ranges* section for the full surface (direction, `..<`, `by` steps).

So when do you still hand-roll the `(head, thunk)` stream of this chapter? When the sequence isn't *arithmetic* — primes, Fibonacci, sensor readings. An open range gives you the integers for free; the pair-and-thunk construction (or `infinite_set("primes")`) remains the tool for everything whose next element needs *computing from the last*. That is exactly the lesson of this chapter, and it survives the feature: laziness is a *technique*, not a type.

One large slice of "computing from the last" has since become a builtin. When each element is a *function of exactly the previous one*, `iterate(f, x)` is the whole construction — the unbounded sequence `x, f(x), f(f(x)), …` as a lazy stream, no pair and no thunk:

```
powers: iterate(func(v) [v * 2], 1)   # 1, 2, 4, 8, 16, …
first(powers, 5)                      # → [1, 2, 4, 8, 16]

for i in iterate(func(v) [v * 2], 1) [    # a loop counter whose
  if i >= 20 then [ break ]               # step is any function,
  println(i)                              # and which is still
```

```
]                                          # scoped to the loop
```

A lazy stream — an open range, a lazy comprehension, or an `iterate` — is a **loop source**, walked one element at a time and never materialized. `take_while` bounds one without a `break`:

```
take_while(func(x) [x < 20], iterate(func(v) [v * 2], 1))   # → [1, 2, 4, 8, 16]
```

A stream is walked *once*: stop partway and the next loop resumes where you left off, but a stream already drained raises a catchable error rather than quietly running zero iterations.

What this does **not** cover directly is the case where the next element needs more than the previous one — Fibonacci needs the previous *two*, primes need every prime so far. The standard trick is to `iterate` over a richer **state** and project the answer out of it:

```
fib_states: iterate(func(p) [(p[2], p[1] + p[2])], (0, 1))
first(fib_states, 8)     # → [(0,1), (1,1), (1,2), (2,3), (3,5), (5,8), (8,13),
 ↪  (13,21)]
map(func(p) [p[1]], first(fib_states, 8))    # → [0, 1, 1, 2, 3, 5, 8, 13]
```

Note what that costs: the stream's elements are no longer the sequence you wanted, and you pay a `map` to recover it. When the state can't be squeezed into a fixed-size tuple — primes, which need every prime found so far — the pair-and-thunk construction of this chapter is still the tool. The lesson holds: the builtins cover the common shapes, and the technique covers the rest.

17.8 The built-in thunk — `lazy` and `force`

§17.7 was about the language growing a built-in for the *sequence*. It has since grown one for the other half of the construction too — the **thunk**, the piece §17.1 hand-rolls.

A thunk in this chapter is a zero-argument function you call to get the rest: `func() [nats_from(k + 1)]`, invoked with `()`. `lazy e` is that same idea as a *value*, and `force` runs it:

```
t: lazy (6 * 7)     # nothing has run yet
force(t)            # → 42   computed now…
force(t)            # → 42   …and cached, not recomputed
```

So the stream constructor loses the closure ceremony, and `stream_tail` becomes a `force` instead of a call:

```
nats_from: func(k) [ (k, lazy nats_from(k + 1)) ]

stream_head: func(s) [ s[1] ]
stream_tail: func(s) [ force(s[2]) ]

s: nats_from(1)
```

```
stream_head(s)                              # → 1
stream_head(stream_tail(s))                 # → 2
stream_head(stream_tail(stream_tail(s)))    # → 3
```

The difference that matters: memoization

The two thunks are not interchangeable, and the gap is the classic one. A hand-rolled `func()` thunk re-runs its body on *every* call. A `lazy` thunk runs it **once** and caches the result:

```
hand_runs: 0
hand_work: func() [
  rebind hand_runs = hand_runs + 1
  "computed"
]
hand: func() [hand_work()]    # hand-rolled thunk
hand()
hand()
println(hand_runs)            # → 2   recomputed every call

lazy_runs: 0
lazy_work: func() [
  rebind lazy_runs = lazy_runs + 1
  "computed"
]
t: lazy lazy_work()           # built-in thunk
force(t)
force(t)
println(lazy_runs)            # → 1   computed once, then cached
```

This is exactly the distinction SICP draws in §3.5.1, where `delay` starts as a bare `(lambda () exp)` and is then wrapped in `memo-proc` to stop streams recomputing their own tails. `lazy` is the memoized version, built in.

It is not a theoretical concern — §17.4's own example walks `fibs` twice:

```
tails: 0

fib_pair: func(a, b) [
  (a, lazy [ rebind tails = tails + 1
             fib_pair(b, a + b) ])
]
stream_head: func(s) [ s[1] ]
stream_tail: func(s) [ force(s[2]) ]
take_stream: func(n, s) [
  if n == 0 then []
  else [
    h: stream_head(s);
    t: stream_tail(s);
```

```
    push([], h) + take_stream(n - 1, t)
  ]
]

fibs: fib_pair(0, 1)
take_stream(10, fibs)
take_stream(20, fibs)
tails      # → 20 with `lazy`
           #   30 with the hand-rolled `func()` thunk —
           #   the first ten tails computed a second time
```

Twenty tails exist; the hand-rolled version computes thirty. The ten extra are the prefix the second walk re-derives from scratch. On this stream that is a constant-factor waste; on a stream whose tail is *expensive*, or one walked many times, it is the difference between a usable structure and an unusable one.

So which do you write?

Hand-rolling stays the right way to *learn* the mechanism — that is why this chapter does it, and why `stream_tail`'s `s[2]()` is worth having typed at least once. For code you intend to keep, `lazy` says what you mean, and it memoizes for free. And the deeper point of §17.7 survives here unchanged: the built-in did not replace the technique, it named it.

17.9 Reflection — the end of Part V

Part V's arc:

- **Chapter 15** introduced **accumulators** — carrying the growing answer forward through recursion.
- **Chapter 16** introduced **mutation** — letting a name re-bind to a new value over time.
- **Chapter 17** introduced **streams** — sequences that produce themselves on demand.

The thread: each chapter loosens one constraint of the pure functional core. Accumulators loosen "the answer comes out on the way back up" — it can come out the way down. Mutation loosens "names are fixed" — they can change. Streams loosen "all values are computed eagerly" — some values *aren't computed* until you ask.

Loosened constraints give you new *tools*, not new *rules*. A program that uses mutation everywhere is harder to reason about than one that uses mutation only where it's appropriate. A program that lazy-evaluates everything is harder to debug than one that materializes intermediate results. *Good code* picks the right tool for each spot and uses pure idioms by default.

You now have the **entire imperative/functional toolbox**. Part VI takes a different direction: Axioma-specific features that don't have HtDP equivalents — Concepts as sum-type alternatives, enums + sub-ranges, multi-valued logic, logic programming, the meta-circular evaluator. Each chapter explores one corner of Axioma that no Lisp / Python / Java textbook would cover.

17.10 Optional: built-in Sequence and Cursor

The hand-built pairs above expose the mechanism. The built-in `Sequence` packages replayable, memoized demand as a first-class value. `Stream` is its compatibility type name; `stream` and `stream_cons` still work. `Iterable` remains the common interface, and a `Generator` remains one-shot.

```
let odds :: Sequence = sequence(1..9..2)
println(collect(odds))                        # [1, 3, 5, 7, 9]
println(collect(odds))                        # [1, 3, 5, 7, 9]
let squares = sequence_map(func(x) [x * x], 1..)
let even_squares = sequence_filter(func(x) [x % 2 == 0], squares)
println(collect(sequence_take(even_squares, 4)))  # [4, 16, 36, 64]
println(first(squares, 3))                    # [1, 4, 9]
println(Stream == Sequence)                   # true
```

`sequence(source)` adapts collections, Ranges, Generators and supported Iterables. Collection conversion snapshots outer storage; nested mutable objects remain shared. Range conversion computes each cell without an intermediate Array. A Generator conversion pulls and caches only as demanded. Pulling the same Generator elsewhere changes the uncached suffix; the cached prefix remains replayable. Holding the root retains that prefix in memory.

`sequence_map` and `sequence_filter` are lazy. `sequence_take(source, n)` is a lazy prefix; `sequence_drop(source, n)` demands the skipped prefix only when its result is read. `first(source, n)` returns an Array and never demands the tail after the requested prefix. Taking zero demands nothing. Ordinary `map`/`filter`, `collect` and `force` materialize a Sequence, so bound infinite input first. Materialization stops with an error at its 1,000,000-element cap; filter refuses after 1,000,000 consecutive nonmatches, and drop accepts at most 1,000,000. Demand errors are cached, like successful cells.

A `Cursor` separates mutable position from replayable data. It accepts an ordered collection or Sequence; use `cursor(sequence(g))` explicitly for a Generator. Its read-only `.position`, `.value` and `.done` describe the current one-based position. Navigation changes only the cursor. Copying saves an independent position; binding another name to the same cursor aliases it.

```
let c = cursor([10, none, 30])
let bookmark = cursor_copy(c)
cursor_next(c)
println(c.value)                              # none
println(c.done)                               # false: none is an element
println(bookmark.value)                       # 10
cursor_seek(c, 99)
println(c.position)                           # 4: just past the end
println(c.done)                               # true
cursor_back(c)
println(c.value)                              # 30
```

Positions clamp at the beginning or one past the end. Navigation and reading `.value`/`.done` can demand cells; reading `.position` and copying do not. Backward navigation walks the cached prefix from the root; a call is limited to 1,000,000 positions. A failed navigation leaves the position unchanged, although

source effects already demanded cannot be undone. Source editing and bounded `SequenceView` slices are outside this API.

These examples use the evaluator. The VM explicitly refuses lazy callback transformations and all Cursor operations; it supports basic Sequence construction and prefix/suffix operations for otherwise supported inputs. `lazy` and custom Iterable implementations retain their VM refusal. Plain `:: Sequence`/`:: Stream` checks the container; `Sequence of T` element contracts are not introduced.

17.11 Optional: parallel demand and List suffixes

`zip` eagerly combines two or more ordered sources into an Array of tuples, stopping at the shortest. Its inputs may be Arrays, Tuples, built-in Lists, Strings, Bytes or Ranges. Strings contribute one String per Unicode rune; Bytes contribute Byte values. One finite operand bounds any open Ranges. `sequence_zip` also accepts Generators and Sequences and returns a lazy, memoized Sequence. Entirely infinite input is safe when consumption is bounded.

```
println(zip([1,2], "ab", `[true,false]))
let rows = sequence_zip(1.., 10.., "abc")
println(first(rows, 2))
println(first(rows, 2))
println(map(array, force(suffixes(`[1,2,3]))))
```

The outputs are `[(1, "a", true), (2, "b", false)]`, twice `[(1, 10, "a"), (2, 11, "b")]`, and `[[1, 2, 3], [2, 3], [3]]`. The last line uses the built-in immutable List: its nonempty suffixes share existing tails. The empty suffix is omitted. Convert an Array with `list(xs)`. A suffix is different from a fixed-width neighbour window such as `each_cons(2, xs)`.

Lazy zip demand proceeds left to right. At exhaustion it can consume one unpaired element from an earlier Generator; a requested prefix does not fetch the next row. Construction and taking zero pull nothing. Replaying cached rows or failures repeats no source effects. Sharing a Generator with another consumer changes its uncached future. Set order and Dictionary projections must be chosen explicitly before either zip. `sequence_zip` is evaluator-only; finite `zip` and List `suffixes` with basic Sequence operations also work in the VM.

Exercises

Exercise 17.1 — `integers_from`

Write `integers_from(start, step)` that yields an infinite arithmetic progression starting at `start`, advancing by `step` each time.

- `take_stream(5, integers_from(0, 1))` → `[0, 1, 2, 3, 4]`
- `take_stream(5, integers_from(10, 5))` → `[10, 15, 20, 25, 30]`
- `take_stream(5, integers_from(0, -1))` → `[0, -1, -2, -3, -4]`

Exercise 17.2 — `take_while`

Implement `take_while(pred, s)` that returns an *array* of prefix elements of `s` for which `pred` is true, stopping at the first failing element.

- `take_while(func(x) [x < 5], nats_from(1))` → `[1, 2, 3, 4]`
- `take_while(func(x) [x > 0], integers_from(10, -1))` → `[10, 9, 8, …, 1]`

(Hint: this is bounded by the first false; for an infinite stream that has no false element, this would loop. The exercise assumes a finite false-prefix.)

Exercise 17.3 — `zip_streams`

Combine two streams element-wise into a stream of tuples.

- `take_stream(5, zip_streams(nats_from(1), nats_from(100)))` → `[(1, 100), (2, 101), (3, 102), (4, 103), (5, 104)]`

Exercise 17.4 — `stream_of_squares`

Build the stream of squares `1, 4, 9, 16, …` by composing `map_stream` with `nats_from(1)`. Then verify the first five match `[1, 4, 9, 16, 25]`.

Exercise 17.5 — `every_kth`

Given a stream `s` and an integer `k`, return a new stream that yields every `k`-th element of `s` (the first, the `(k+1)`-th, the `(2k+1)`-th, …).

- `take_stream(5, every_kth(nats_from(1), 2))` → `[1, 3, 5, 7, 9]`
- `take_stream(5, every_kth(nats_from(1), 5))` → `[1, 6, 11, 16, 21]`

Exercise 17.6 (open) — the sieve of Eratosthenes

A classic stream computation: build the stream of primes by *sieving* the naturals. Algorithm:

1. Start with the stream of integers from 2 onward.
2. The head of the stream (2) is the first prime.
3. Filter out every multiple of 2 from the rest.
4. The head of *that* (now 3) is the next prime.
5. Filter out every multiple of 3.
6. … and so on, forever.

In code, this is one recursive function on streams. Each step yields one prime and constructs the "sieved" remainder as a new stream.

```
sieve: func(s) [
  p: stream_head(s);
  (p, func() [
    sieve(filter_stream(func(x) [x % p != 0], stream_tail(s)))
  ])
```

```
]
primes: sieve(integers_from(2, 1))
println(take_stream(10, primes))   # [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Verify the above. Then think:

1. Why does each prime get sieved-out of the *rest* of the stream as it's produced?
2. What's the asymptotic cost of producing the first `n` primes this way? (Hint: each prime sets up a new filter-thunk; primes near the top of the list filter through many earlier filters.)
3. What would the real sieve of Eratosthenes look like (the *efficient* algorithm)? Why is the stream version prettier even if slower?

Exercise 17.7 — replay and saved positions

Create a Sequence from a Generator yielding 10, 20 and 30. Read its first two values twice, then collect it; explain which operations consume the original Generator. Create a cursor on `[10, none, 30]`, copy its position, and move the original to exhaustion. Show both positions and explain why `.done` is needed when a value can be `none`. Would binding a second name to the original cursor save an independent position?

End of Part V. You now have:

- **Pure functional** core (Parts I–IV) — values, functions, higher-order operations, recursion on data shapes.
- **Algorithmic recursion** (Ch.14) — generative recursion with termination arguments.
- **Accumulator style** (Ch.15) — single-forward-pass recursion.
- **Mutation and loops** (Ch.16) — imperative core for state machines and counters.
- **Streams** (Ch.17) — lazy infinite sequences.

That's the full programming-with-data toolbox. Part VI takes a left turn into Axioma's distinctive features: how its Concepts replace HtDP's structures *and* sum types; how its type system handles enums and subranges; how its multi-valued logic captures *uncertainty* directly; how its logic programming engine answers queries about facts; and finally, how all of it is itself defined *in* Axioma — the meta-circular evaluator that ties the whole language back to itself.

Solutions to selected exercises: Chapter 17 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch17_solutions.md.)*

Exercise 17.8 — a bounded parallel walk

Use `sequence_zip` to pair two open ranges starting at 1 and 10 with the String `"abc"`. Read two rows twice, then read all three. Explain why replay has the same values and why the result terminates. Contrast this with three comma-separated generators in a `for` header.

Chapter 18 · Types and Data Modeling

> **What this chapter is.** A data definition describes what a value means, how to construct it, and which cases a function must handle. Axioma offers several type families for those different jobs. The shared header `type Name = …` makes declarations recognizable; it does not make schemas, records, variants and domain Concepts interchangeable.
>
> Part VI uses the full language rather than the beginner subset. Unless a block explicitly continues the preceding example, it is a complete program for a fresh file.

We will model a small drawing application. A point is a value; a cursor has changing state; a shape has alternatives; an imported row has a structural contract; and an object in a knowledge model may have a Concept classification. Choosing among these representations is part of the design recipe.

18.1 From a data definition to a representation

HtDP separates **composition** (several fields together) from **variation** (one of several cases). Its structures supply constructors, selectors and predicates:

```
(define-struct posn (x y))
(posn-x (make-posn 3 4)) ; 3
```

Axioma has a direct nominal record for that purpose:

```
type Point = struct {x:: Float, y:: Float}
let p = Point(3.0, 4.0)
println(p.x)             # 3.0
println(p is Point)      # true
```

Each field here carries a `::` annotation, so construction checks it. The annotation is optional: a bare lowercase name, as in `type Pnt = struct {x:: Float, y:: Float, z}`, declares an open field that accepts any value, the same gradual reading an unannotated parameter has.

HtDP's "a Shape is one of Circle, Rectangle or Triangle" becomes a tagged variant and a `match`. Concepts remain useful when the task needs domain entities, inheritance, facts or classification. They are not a prerequisite for making an ordinary point or a closed set of alternatives.

Use this map before choosing a declaration:

Modeling need	Choose
Name an existing type	Alias: the same values and membership
Accept several existing types	Union: an arm type, without a new tag

Modeling need	Choose
Combine values by position	Product: a Tuple with element types
Separate a role from its underlying type	Opaque: an explicit nominal wrapper
Validate named fields on imported data	Schema: a structural Dictionary contract
Store a closed immutable record	Struct: a nominal constructor and checked fields
Change fields on a closed record	Mutable struct: checked writes and reference identity
Represent alternatives with payloads	Variant: a declared constructor set
Select among ordered names	Enum: ordinal members
Classify domain entities	Concept: slots, inheritance and classification
Restrict permitted numeric or enum values	Subrange: membership within bounds

The rest of this chapter develops the choices. Chapter 19 continues with ordered enums, numeric subranges and restricted enum ranges. There is no new `type … = subrange` spelling implied by this table.

18.2 Names, unions, products and opaque wrappers

An **alias** documents intent but does not separate values. A **union** accepts an existing arm type; it adds no constructor tag. A **product** describes a Tuple, useful when positions already have an agreed meaning:

```
type Distance = Float
type Coordinate = Integer | Float
type Position = (Float, Float)
let distance :: Distance = 2.5
let coordinate :: Coordinate = 3
let position :: Position = (3.0, 4.0)
println(distance is Float)      # true
println(coordinate is Coordinate) # true
println(position[1])           # 3.0
println((3.0, "north") is Position) # false
```

Choose an alias for vocabulary, a union when the existing values themselves identify the case, and a product for a small positional bundle. For example, `Integer | Float` does not distinguish a width from a height. A struct gives those roles field names; an opaque wrapper can give a single role its own type.

An **intersection** is the dual of a union. A value satisfies `A | B` when it satisfies either arm; it satisfies `A & B` only when it satisfies *both*. A union widens a type, and an intersection narrows one:

```
type Grade = Integer & 0..100
let score :: Grade = 87
println(score)          # 87
println(87 is Grade)    # true
println(101 is Grade)   # false
println(87.0 is Grade)  # false
```

The range alone would admit the whole Float `87.0`; `Integer &` rules it out. `&` binds tighter than `|`, so `Integer & 0..100 | String` means `(Integer & 0..100) | String`. An intersection that no value can satisfy, such as `Integer & String`, is refused where it is written, because it is almost always a mistyped `|`.

```
type DrawingId = opaque Integer
let id = DrawingId(42)
println(id is DrawingId)     # true
println(42 is DrawingId)     # false
println(id is Integer)       # false
println(match id with | DrawingId(n) => n) # 42
```

The wrapper is nominal: `DrawingId(42)` is distinct from a bare Integer. Construction checks the underlying type; extraction is explicit through a constructor pattern. Use this when accidentally mixing two kinds of identifier would be a bug. An alias such as `type DrawingId = Integer` would not prevent it. These compact declarations do not require `end`.

18.3 Schemas: the shape of a Dictionary

A schema is useful at an input boundary, such as a parsed drawing row. Membership depends on the Dictionary's fields rather than a constructor name. An annotation attaches the existing persistent schema contract to supported writes; a membership test merely answers a question.

```
type PointRow = {x:: Float, y:: Float, label?:: String}
let row :: PointRow = {x: 3.0, y: 4.0}
println(row is PointRow)                  # true
println({x: 3.0, y: 4.0} is PointRow)     # true
println({x: "east", y: 4.0} is PointRow) # false
row.label = "start"
println(row.label)                        # start
```

`label?` permits absence; it does not mean that every value is accepted when present. Supported property, index and reference writes must preserve the contract. Required fields cannot be removed. The equivalent explicit body is:

```
type PointRow = schema
    x :: Float
    y :: Float
    label? :: String
end
let row :: PointRow = {x: 3.0, y: 4.0}
println(row.x) # 3.0
```

Choose a schema when multiple independently created Dictionaries should satisfy the same shape. Choose a struct when the record's nominal kind matters.

Schemas combine with `&`. The members **merge** before a literal is checked, so the literal may use the

fields of every member:

```
type Located = {x:: Float, y:: Float}
type Labelled = {label:: String}
type Marker = Located &
              Labelled
let start :: Marker = {x: 0.0, y: 0.0, label: "start"}
println(start.label)                  # start
println(start.x)                      # 0.0
println({x: 1.0, y: 2.0} is Marker)   # false
```

A field declared by both members must satisfy both of their types, and it is required if either member requires it. A long intersection continues onto the next line when the line **ends** with &; a line that *starts* with & never continues the type above, and that layout is refused. A key that no member declares is still a typo:

```
type Located = {x:: Float, y:: Float}
type Labelled = {label:: String}
let typo :: Located & Labelled = {x: 0.0, y: 0.0, lable: "start"}
```

The error names the whole type and the member the literal fails: `lable` is not `label`, so the value does not satisfy `Labelled`.

18.4 Immutable structs: named values

A struct combines a nominal constructor with closed, annotated fields:

```
type Point = struct
    x :: Float
    y :: Float
end
let p = Point(3.0, 4.0)
let q = {p with x: 6.0}
println(p.x)                        # 3.0
println(q.x)                        # 6.0
println(p == Point(3.0, 4.0))       # true
println({x: 3.0, y: 4.0} is Point) # false
println(error?(try([p.x = 9.0])))  # true
```

This declaration means the same as the brace form in §18.1. Construction uses `Point(x, y)`, field access uses `.x`, and a functional update creates another validated Point. Fields cannot be replaced or added. Equality compares immutable record values structurally. A struct neither registers domain entities in a Concept extent nor supports Concept inheritance or `has` slot extension.

Freezing is **shallow**: if a field contains an Array, a Dictionary or a mutable struct, that nested value retains its own mutation rules. Likewise, a functional update shares untouched nested values. It is not a deep clone. See §16.13 for mutation examples. The optional `record` spelling has exactly this immutable-struct

contract; §18.5's spelling note compares the forms.

18.5 Mutable structs: state with a closed contract

A cursor is a different modeling choice from a saved point. It is an object whose position changes, and aliases should see the same movement:

```
type PointCursor = struct mutable {x:: Float, y:: Float}
let cursor = PointCursor(3.0, 4.0)
let other_cursor = cursor
cursor.x = 6.0
println(other_cursor.x)                       # 6.0
println(cursor is/same other_cursor)          # true
println(cursor == PointCursor(6.0, 4.0))    # false
let copy = {cursor with y: 8.0}
println(copy is/same cursor)           # false
println(cursor.y)                     # 4.0
println(copy.y)                       # 8.0
println(error?(try([cursor.x = "east"]))) # true
println(cursor.x)                     # 6.0
```

`let` prevents rebinding the name; it does not freeze a mutable object's fields. Conversely, putting an immutable Point in a `var` does not make `point.x` writable. Binding mutability and field mutability answer different questions.

Mutable structs use reference identity for both == and `is/same`. Two fresh constructions are distinct even with identical contents. Declared field writes are checked before changing the record; incompatible writes leave it unchanged. The field set is closed. Functional updates create a distinct record, with the same shallow sharing rule as immutable structs. The body may instead be written `type PointCursor = struct mutable`, followed by field declarations and `end`. Chapter 16 explores aliases and mutation in detail.

Optional: spelling aliases, not new record kinds

This book uses `struct` as the primary spelling. In the kind position after `type Name =`, `record` is an exact alias for `struct`; it does not introduce structural membership or a different equality rule. Likewise, `mut` is a short alias for `mutable` after either marker. Thus `record` alone is immutable, while `struct mutable`, `struct mut`, `record mutable`, and `record mut` all select the same mutable-struct behavior.

```
type SavedPoint = record {x:: Float, y:: Float}
type ShortCursor = record mut
    x :: Float
    y :: Float
end
type CompactCursor = struct mut {x:: Float, y:: Float}
type LongCursor = record mutable {x:: Float, y:: Float}
let saved = SavedPoint(3.0, 4.0)
```

```
println(saved == SavedPoint(3.0, 4.0))
println({x: 3.0, y: 4.0} is SavedPoint)
println(error?(try([saved.x = 6.0])))
let cursor = ShortCursor(3.0, 4.0)
let other_cursor = cursor
cursor.x = 6.0
println(other_cursor.x)
println(cursor == other_cursor)
println(cursor == ShortCursor(6.0, 4.0))
let compact = CompactCursor(3.0, 4.0)
compact.x = 6.0
println(compact.x)
let long_cursor = LongCursor(3.0, 4.0)
long_cursor.x = 6.0
println(long_cursor.x)
```

The output is `true`, `false`, `true`, `6.0`, `true`, `false`, `6.0`, `6.0`. The immutable record keeps value equality and refuses a field write; the mutable records allow checked writes and keep reference identity. These are separately named types, not interchangeable constructors: spelling aliases do not merge nominal types. Changing only a marker or modifier in a given program changes none of these results. Each spelling accepts both brace and `end` bodies, with the same existing generic type-parameter support and limits. The VM refuses these records, just as it refuses the canonical struct forms. `record` remains available as an ordinary variable or function name. Immediately after `type Name =`, however, it always selects the declaration kind, including empty `end` bodies. To reference a type value held in a variable named `record`, write `type Alias = (record)` instead of the bare kind word; `$record` alone is not a bypass, while `($record)` is an ordinary parenthesized type reference.

18.6 Variants: closed alternatives and exhaustive functions

A drawing's geometry has alternatives with different payloads:

```
type Shape = variant Circle(radius:: Float)
    | Rect(width:: Float, height:: Float)
    | Dot
area: func(s :: Shape) [
    match s with
    | Circle(r) => 3.14159 * r * r
    | Rect(w, h) => w * h
    | Dot => 0.0
]
println(area(Circle(5.0)))    # 78.53975
println(area(Rect(4.0, 6.0))) # 24.0
println(area(Dot))            # 0.0
println(Circle(2.0) is Shape) # true
println(tag(Circle(2.0)))     # Circle
```

`Circle(2.0)` constructs a tagged value. `Dot` is a payload-free singleton, without parentheses. Constructor

patterns bind the payload positions. A variant is different from a union such as `Integer | Float`: it introduces its own tags and payload structure. The alternatives may follow on lines beginning with `|`; the declaration does not need `end`.

The existing native declaration `data Shape = Circle(Float) | Rect(Float, Float) | Dot` expresses the same alternative structure. `data` is preserved, not deprecated. It can also use explicit field names. The unified `variant` header makes its place in the family visible.

The `:: Shape` annotation on the function parameter gives `--typecheck` enough information to check constructor coverage. In a fresh file, deliberately omit Dot:

```
type Shape = variant Circle(Float) | Rect(Float, Float) | Dot
area: func(s :: Shape) [
    match s with
    | Circle(r) => r * r
    | Rect(w, h) => w * h
]
```

Run this intentional diagnostic example with `axioma --typecheck`: it reports a non-exhaustive match on Shape, naming Dot. Add `| Dot => 0.0` or a catch-all to cover it. An untyped parameter may be outside this analysis; a clean check in that case is not proof of exhaustiveness. A repeated constructor arm after an unguarded arm is diagnosed as unreachable:

```
type Shape = variant Circle(Float) | Dot
match Circle(1.0) with
| Circle(r) => r
| Circle(_) => 0.0
| _ => 0.0
```

These are static diagnostics, not runtime pattern failures. The CLI may also emit them as warnings before normal evaluation; explicit `--typecheck` reports them without running the program. An unmatched `match` returns `none`, which is distinct from `om`; that fallback is not proof that a program handles every case correctly.

Guards inspect a constructor's payload

Chapter 5 introduces ordered selection and false guards. Here the pattern first extracts fields from a constructor, and then a guard asks a question about those fields. The guard does not replace the pattern:

```
type Shape = variant Circle(Float) | Rect(Float, Float) | Dot
shape_label: func(s :: Shape) [
    case s
    | Circle(r) when r > 0 => "circle"
    | Circle(_) => "non-positive radius"
    | Rect(w, h) when w == h => "square"
    | Rect(_, _) => "rectangle"
    | Dot => "point"
```

```
]
println(shape_label(Circle(2.0))) # circle
println(shape_label(Circle(0.0))) # non-positive radius
println(shape_label(Rect(3.0, 3.0))) # square
println(shape_label(Rect(3.0, 4.0))) # rectangle
println(shape_label(Dot)) # point
```

For Circle(0.0), the Circle pattern succeeds but its first guard fails. The next Circle arm handles it. For Rect(3.0, 3.0), the square arm wins before the general rectangle arm. Reversing those arms would hide the square case. `switch s` or `match s with` can replace `case s` without changing the patterns or results. Use whichever header is clearest; there is no switch fallthrough between arms.

A guarded constructor arm alone does **not** cover every payload of that constructor. This intentionally incomplete function needs a Reading fallback:

```
type Measurement = variant Reading(Float) | Missing
reading_value: func(m :: Measurement) [
    match m with
    | Reading(x) when x > 0 => x
    | Missing => 0.0
]
```

`--typecheck` reports Reading as missing from the unguarded coverage. Listing its tag under a guard is not enough: Reading(0.0) and Reading(-1.0) can reach no arm. Add `| Reading(_) => 0.0` after the guarded arm if that is the intended policy. The checker does not prove arbitrary predicates cover the numeric domain; an unguarded constructor arm or catch-all expresses coverage directly.

Nested patterns follow nested data

A scene can wrap a Shape. Match the wrapper and its inner constructor in one pattern when that is the distinction the function needs:

```
type Shape = variant Circle(Float) | Rect(Float, Float) | Dot
type Scene = variant OnCanvas(Shape) | Hidden
scene_label: func(scene :: Scene) [
    switch scene
    | OnCanvas(Circle(r)) when r > 0 => "visible circle"
    | OnCanvas(_) => "other visible shape"
    | Hidden => "hidden"
]
println(scene_label(OnCanvas(Circle(2.0)))) # visible circle
println(scene_label(OnCanvas(Rect(2.0, 3.0)))) # other visible shape
println(scene_label(Hidden)) # hidden
```

The inner constructor must match before r is available to the guard. An unguarded OnCanvas(_) handles the remaining wrapped values. Avoid extracting several fields by hand and then testing the tag separately when a pattern can state the whole shape. Guard errors propagate; they do not silently try a later arm.

An unguarded coverage check is still only the checker's supported model, not a proof of every nested predicate or algorithm.

Optional matcher spellings and failure policy

The ordinary matcher returns `none` when no arm is selected. The `/strict` refinement makes that miss an error; add a fallback when absence is expected. Postfix matching puts the value first. The arrows `=>` and `–>` are alternatives:

```
println((1, 2) match | (x, y) -> x + y | _ -> 0) # 3
println(case "later" | "now" => 1) # none
println(error?(try(case/strict "later" | "now" => 1))) # true
```

`match/strict`, `case/strict` and `switch/strict` share that failure policy. The first bar is required with case/switch; the arms are not a brace-delimited block. Appendix A summarizes the headers. Additional pattern forms, including record rest, exact field sets, or-patterns, and as-patterns, are documented in the Manual; this lesson develops constructors, tuples, nesting and guards rather than claiming every pattern form is taught.

`Option` and `Result` are seeded variants: `Absent | Some(value)` and `Ok(value) | Err(error)`. They model optional data and success/failure without a new declaration. Railway composition carries this structure through a pipeline:

```
half: func(n) [Some(n / 2.0)]
double: func(n) [n * 2]
println(Some(10) |?> half |?> double) # 10.0
println(Absent |?> half |?> double)   # Absent
parse: func(s) [if s == "5" then Ok(5) else Err("bad input")]
println(match parse("5") with | Ok(v) => v | Err(e) => 0) # 5
```

18.7 Enums and restricted types: permitted values

For direction, four ordinal members are simpler than four payload-free entity classes:

```
type Direction = enum North, East, South, West
let facing :: Direction = East
println(facing.ord)         # 1
println(North is Direction) # true
println("East" is Direction) # false
Digit ranges 0..9
let digit :: Digit = 7
println(digit is Digit)     # true
println(12 is Digit)        # false
```

The legacy `enum Direction = North, East, South, West` and `Direction enumerates North, East, South, West` remain available. Enums have ordinals and iteration; a general variant may instead

carry different payloads. Subranges restrict an existing ordered domain. Use them when a value outside a permitted range is invalid, rather than inventing tags for each number. Chapter 19 · Enums and Subranges as Invariants teaches enum ordering, numeric bounds, restricted enum ranges and annotated function boundaries in depth.

18.8 Concepts: entities, slots and classification

A drawing program may also maintain knowledge about people and their roles. A Concept describes that domain, with entities and inherited slots:

```
type Person = concept with
    name :: String: ""            # a typed slot: every later write is checked
end
concept Artist with
    medium: "ink"
end
Person has born :: Integer: 0, city: ""   # several slots in one has, comma-separated
Artist extends Person
let ada = an Artist {name: "Ada", city: "London", medium: "pencil"}
println(ada.name)      # Ada
println(ada.city)      # London
println(ada is Artist) # true
println(ada is Person) # true
```

`type Person = concept {name: ""}` and `concept Person {name: ""}` use braces for the same slot/default body. The short form `concept Person` and unified `type Person = concept` are complete bare declarations. Slots can be added with `Person has name: ""`, and one statement can add several, separated by commas: `Person has born: 0, city: ""`. An end body explicitly opens with `with`, in either header spelling; a bare declaration does not consume the enclosing block's end.

The article `a`/`an` creates a domain entity. The brace block is the canonical slot list; `an Artist with name: "Ada", city: "London"` is a provisional second spelling of the same block, reusing the `with` that opens concept bodies. `Artist extends Person` declares inheritance; `ada is Person` queries membership. Inherited defaults can supply omitted slots. Inheritance is live: `Person has born: 0, city: ""` may come before or after `Artist extends Person`, because `Artist` reads `Person`'s slots when they are used, through the same chain `is` walks; an individual copies the slots that exist when it is created. A slot may carry a type: `name :: String: ""` is checked at creation, on assignment, in a record update and against any later default, in the concept and its instances, and an Integer converts for a Float slot; `born :: Integer` alone would start as `none`. The type does not close the slot set, though: `has` may still add slots later, so a typed concept is not a struct. Do not infer closed-record enforcement from the visual similarity of the bodies.

Choose Concepts when classification, facts, relations, inheritance or a live extent of domain entities is useful. Choose structs for ordinary closed records. One drawing system can use both: immutable geometry values and classified domain entities represent different aspects of the same application.

18.9 Partitions: checking a domain classification

An inheritance hierarchy is open to further subcategories. A partition records a proposed disjoint and exhaustive classification of a parent's live extent. It does not turn Concept inheritance into a closed variant declaration.

```
concept Animal
Animal has name: ""
Mammal extends Animal
Mammal has fur: true
Bird extends Animal
Bird has wings: 2
Fish extends Animal
Fish has gills: true
Animal partition Mammal, Bird, Fish
let dog = a Mammal {name: "Rex"}
let robin = a Bird {name: "Robin"}
println(dog is Animal)       # true
println(robin is Mammal)     # false
println(is_partitioned(Animal)) # true
let mystery = an Animal {name: "Unclassified"}
println(is_partitioned(Animal)) # false
println(mystery in partition_gap(Animal)) # true
```

The counterexample matters: creating an unclassified parent instance succeeds, and the query detects the gap. `partition_overlap(Animal)` finds instances in multiple parts; `partition_member(x, Animal)` identifies a matching part. The declaration makes the intended law inspectable, not automatically true for all future entities. `partition` plus an `if … is …` cascade does not receive the constructor-exhaustiveness check supplied by `variant`/`data` plus `match`.

18.10 The same shape problem, two models

The variant area function in §18.6 follows HtDP's recipe: the alternatives supply the cases, and each case supplies the fields. Here is the Concept alternative, in a fresh file. Its reason to exist is domain classification and inheritance, not a lack of native variants:

```
concept Shape

Circle extends Shape
Circle has radius: 0.0

Rectangle extends Shape
Rectangle has width:  0.0
Rectangle has height: 0.0

Triangle extends Shape
Triangle has base:   0.0
```

```
Triangle has height: 0.0

Shape partition Circle, Rectangle, Triangle

area: func(s) [
  if s is Circle    then 3.14159 * s.radius * s.radius
  else if s is Rectangle then s.width * s.height
  else if s is Triangle  then 0.5 * s.base * s.height
  else 0.0
]

c: a Circle    {radius: 5.0}
r: a Rectangle {width: 4.0, height: 6.0}
t: a Triangle  {base: 3.0, height: 4.0}

println(area(c))   # 78.53975
println(area(r))   # 24.0
println(area(t))   # 6.0
```

Design-recipe step	Immutable records / variants	Domain Concepts
Package fields	`struct` with annotated fields	Slots and defaults, optionally inherited
Construct	`Point(3.0, 4.0)`, `Circle(5.0)`	`a Circle {radius: 5.0}`
Select a field	`.x` or a constructor pattern	`.radius`
Test a case	`match` over variant constructors	`is` classification
Check completeness	Typed constructor coverage in `--typecheck`	Query partition laws over the live extent

18.11 Choosing a model and extending it

A closed geometry calculation benefits from variants: adding a constructor prompts a review of typed matches, and values carry only their listed payloads. A domain catalogue benefits from Concepts: shared slots and classification relationships describe entities that participate in facts and relations. Inheritance does not make every `if` cascade exhaustive, and a partition query is a check of the current extent rather than a proof about every future value.

The other families solve smaller questions within either model. Schemas validate imported Dictionaries; structs give saved points nominal identity; mutable structs hold cursor state; opaque wrappers separate drawing IDs from unrelated Integers; enums represent direction; aliases and products clarify existing data. Use a union when the existing arm types are sufficient, and a variant when new case labels or different payloads are part of the meaning.

Working with deep hierarchies

Concept inheritance can span several levels. This complete example preserves the taxonomy model, while the same hierarchy would be unnecessary for a Point:

```
concept LivingThing
LivingThing has name: ""

Animal extends LivingThing
Animal has legs: 0

Vertebrate extends Animal
Vertebrate has has_spine: true

Mammal extends Vertebrate
Mammal has fur_color: ""

Dog extends Mammal
Dog has breed: ""

rex: a Dog {name: "Rex", legs: 4, has_spine: true, fur_color: "brown", breed:
 ↪  "Labrador"}

# Every level of the hierarchy responds true
println(rex is Dog)         # true
println(rex is Mammal)      # true
println(rex is Vertebrate)  # true
println(rex is Animal)      # true
println(rex is LivingThing) # true

# And only the relevant fields exist
println(rex.breed)     # Labrador
println(rex.fur_color) # brown
println(rex.legs)      # 4
println(rex.name)      # Rex
```

Each `extends` declaration adds an inheritance relationship; `has` declares a slot or its default. Membership follows the parent chain transitively. Instances can supply inherited fields or use applicable defaults. A hierarchy describes a model's categories; it is not a substitute for choosing the fields and cases that a particular computation needs.

18.12 Recursive schemas and missing measurements

A schema can refer to itself, so a tree can be represented as Dictionaries without declaring a Concept or nominal constructor for each node:

```
type TreeRecord = {value :: Integer, children :: Array of TreeRecord}
let tree :: TreeRecord = {
  value: 1,
  children: [{value: 2, children: []}]
}
println(tree.children[1].value)    # 2
println(tree is TreeRecord)       # true
println({value: "bad", children: []} is TreeRecord)  # false
```

The recursive check validates reachable children against the schema. Supported Dictionary writes preserve their attached row contracts. This does not introduce a new deep Array mutation contract: mutating an Array stored inside a record has that Array's existing rules. Self-recursive schemas are supported; do not infer forward or mutually recursive schema declarations from this example.

Missing measurements and typed tables

An absent field and a missing measurement are different. `note?` permits the field to be absent; `Float | Na` permits a present cell holding `na`. Neither is the same as `none` or an IEEE floating-point NaN.

```
type SampleRow = {reading :: Float | Na, quality :: Integer, note? :: String}
let sample :: SampleRow = {reading: na, quality: 7}
let another_name = sample
another_name.reading = 12.5
sample.note = "checked"
println(sample.reading)                 # 12.5
println(delete sample["note"])          # checked
println(sample.note)                    # none
let revised = {sample with quality: 9}
println(sample.quality)                 # 7
println(revised.quality)                 # 9

measurements: dataframe(
  {reading: [12.5, na], quality: [7, 8]},
  {schema: SampleRow}
)
println(measurements is DataFrame)       # true
```

The first update is visible through both names. The record-update form creates a new outer Dictionary; untouched nested values remain shared. Writing a String to `sample.reading` or removing required `quality` is refused; field references and indexed writes must preserve the same contract.

A typed DataFrame checks columns against its row schema and copies input column arrays. Filtering, sorting, and supported column transformations preserve or project the schema. `df_mutate` returns a new validated table. CSV reading can apply the same schema, but missing-marker choices and column names must match the file. The current typed table surface covers scalar cells and unions; nested mutable cells are explicitly refused.

18.13 What the checks establish

The design recipe still begins with examples and the meaning of the data. A shared declaration header does not erase the distinction between a structural contract, a nominal value, a stateful record and a knowledge-domain category.

For these examples, use the normal evaluator. The browser interpreter also supports the new family. `--typecheck` checks supported annotations and typed matches; runtime constructor and mutation checks remain necessary. HM `--infer` covers a smaller fragment and does not infer property-access functions. A lack of inferred results or warnings is not an all-program type-safety guarantee.

The VM has separate limits: supported top-level transparent schemas/products retain their existing behavior, but struct/variant and enum declarations refuse; Concept, opaque and local/recursive type forms have their documented boundaries. Parser acceptance is not evidence of VM execution support. These limitations do not change the modeling choice; they determine which execution mode can run it.

Exercises

Exercise 18.1 — `perimeter`

Add a `perimeter(s)` function that computes the perimeter of a Shape. For Triangle, assume an *equilateral* triangle so the perimeter is `3 * base`.

- `perimeter(a Circle {radius: 5.0})` → ~31.42 ($2\pi r$)
- `perimeter(a Rectangle {width: 3.0, height: 4.0})` → `14.0`
- `perimeter(a Triangle {base: 5.0, height: 4.33})` → `15.0`

Exercise 18.2 — `LightState`

Model HtDP's `LightState` as a Concept hierarchy:

```
concept LightState
RedLight extends LightState
GreenLight extends LightState
YellowLight extends LightState
LightState partition RedLight, GreenLight, YellowLight
```

Then write `next_light(s)` that returns the next state (`RedLight` → `GreenLight` → `YellowLight` → `RedLight`). Build it from *is* dispatches; return *new* objects.

Exercise 18.3 — biological taxonomy

Model a small taxonomy: Animal, Vertebrate / Invertebrate (partition Animal), Mammal / Bird / Fish / Reptile / Amphibian (partition Vertebrate). Make at least one instance per leaf type. Then write `is_warm_blooded(a)` that returns `true` for Mammals and Birds, `false` for everything else.

Exercise 18.4 — `expr_eval` revisited

Convert the Chapter 9 `Lit`/`Op` expression evaluator to use explicit `Expression partition Lit, Op`. Verify the same test cases still work.

Exercise 18.5 — `count_shapes_by_type`

Given an array of Shapes, return a tuple `(num_circles, num_rectangles, num_triangles)`. Use accumulator-style recursion (Ch.15) walking the array, with three accumulators.

- `count_shapes_by_type([c, r, t, c, c])` → `(3, 1, 1)`
- `count_shapes_by_type([])` → `(0, 0, 0)`

Exercise 18.6 (open) — design your own variant data

Pick a domain you know (cards in a deck, vehicles in traffic, musical notes in a scale, file system entries, GUI widgets, events in a log) and design a *Concept hierarchy* for it. Include:

1. A root Concept with at least one shared field.
2. At least three subtypes via `extends`.
3. A `partition` and a query that checks its current extent.
4. One function that dispatches on the subtype, returning a different value per variant.

Show that `is`-dispatch and inheritance compose to make your design concise. Compare against how you'd write it in your favorite other language; note what's easier and what's harder.

Exercise 18.7 — `data LightState`

Redo Exercise 18.2 with `data` instead of Concepts + `partition`:

```
data LightState = RedLight | GreenLight | YellowLight
```

Write `next_light(s)` with `match` instead of an `is`-cascade (`RedLight` → `GreenLight` → `YellowLigh t` → `RedLight`). **Give the parameter an explicit type annotation** — `next_light: func(s :: Li ghtState) [ ... ]` — this matters: `--typecheck`'s exhaustiveness check reads a static shadow-type model, not full inference, so an *unannotated* parameter's type is unknown to it and the check is silently skipped (no warning either way — not even a false pass). With the annotation in place, deliberately delete one arm and run `axioma --typecheck` on the file — confirm you get a `non-exhaustive mat ch` warning naming the missing constructor. Put the arm back, run `--typecheck` again, confirm it's clean. Then remove the `:: LightState` annotation with the arm still missing and re-run `--typecheck` once more — confirm the warning disappears, demonstrating the annotation requirement for yourself. Compare against your Exercise 18.2 solution: which version would you rather maintain, and why?

Exercise 18.8 — an imported point and a saved point

Declare a PointRow schema and an immutable Point struct, each with Float x/y. Construct one of each. Show that a matching Dictionary satisfies PointRow but not Point. Write `save_point(row :: PointRo w)` that constructs a Point. Return a moved copy without changing the saved point. Explain why an alias for Dictionary would not provide the same row contract.

Exercise 18.9 — cursor identity and shallow sharing

Create a mutable PointCursor with Float x/y and an Array trail. Bind an alias with `let`, mutate x, then make a functional copy. Predict which x and trail changes are shared. Check a wrong-type field write leaves x unchanged. Explain why `let` and `struct mutable` are compatible.

Exercise 18.10 — identifiers, coordinates and direction

Use an opaque DrawingId, a Distance alias, an Integer-or-Float Coordinate union, a two-Float Position product, and a Direction enum. Construct and use each. Demonstrate that a bare Integer is not a DrawingId. Which definitions introduce new nominal distinctions, and which only describe existing values? Continue in Chapter 19 by restricting a numeric range using its existing `ranges` form.

Exercise 18.11 — choose a model

A drawing application imports rows, stores geometry, moves a cursor and records which artists created drawings. Choose a representation for each. Give one reason for each choice and one alternative that would lose a useful property. Where would you use a variant instead of an open Concept hierarchy? Explain why `partition` does not make a missing branch in an `is` cascade a static error.

Exercise 18.12 — guarded geometry

Use a Shape variant with Circle, Rect and Dot. Return “unit circle” for radius1, “circle” for other positive radii, “invalid circle” for non-positive radii, “square” for equal positive rectangle sides, “rectangle” for other positive sides, and “invalid rectangle” otherwise; Dot returns “point”. Use match with patterns and guards. Test a boundary and an overlapping case for each constructor. Remove Circle’s unguarded arm and inspect –typecheck’s coverage diagnostic.

Exercise 18.13 — nested scene and failure policy

Wrap Shape in an OnCanvas constructor, with Hidden as another Scene alternative. Use a nested pattern to extract a Circle radius and an outer fallback for other visible shapes. Write case and switch versions with identical results. In a separate literal-matching example, show ordinary none fallback and /strict failure. Explain why a false guard continues but a guard error does not.

Exercise 18.14 (optional) — different spellings, the same contract

Start with these declarations, keeping the canonical forms as a baseline:

```
type Point = struct {x:: Float, y:: Float}
type RecordPoint = record {x:: Float, y:: Float}
type PointCursor = struct mutable {x:: Float, y:: Float}
type ShortCursor = struct mut {x:: Float, y:: Float}
type RecordCursor = record mut
    x :: Float
    y :: Float
```

```
end
type LongCursor = record mutable {x:: Float, y:: Float}
```

Write `point_trace(make_point)` that constructs `(3.0, 4.0)`, makes a functional copy with x = 6.0, and returns the original x, copy x, equality with another `(3.0, 4.0)` construction, and whether an attempted field write is an error. Both point constructors should produce `(3.0, 6.0, true, true)`.

Write `cursor_trace(make_cursor)` that constructs `(3.0, 4.0)`, binds an alias, changes x to 6.0, and returns the alias's x, equality with the alias, equality with a fresh `(6.0, 4.0)` construction, whether writing a String to x fails, and the final x. All four cursor constructors should produce `(6.0, true, false, true, 6.0)`.

Explain why these identical traces do not make Point and RecordPoint the same nominal type, or make a Dictionary a RecordPoint. Check both membership claims. Finally rewrite the brace bodies with `end` and the end body with braces; the traces should stay the same. Use the evaluator, not the VM.

Exercise 18.15 — combining schemas

Declare `type Named = {name:: String}` and `type Priced = {price:: Float}`, then `type Product = Named & Priced`. Write `label(p :: Product) :: String`, returning `"<name> (<price>)"`, so that a pen priced 2.5 is labelled `pen (2.5)`. Show that a literal missing `price` is refused, and that a misspelled key is refused too. Then declare `type Discounted = Product & {discount?:: Float}` and compute a discounted price from a literal with all three keys. What does `discount` read on a Discounted literal without one? Finally predict, then check with `try`, what `type Bad = Named & Integer` does, and explain why the refusal comes before any value is checked.

Optional: ordering immutable variants and wrappers

For values of the same ADT, constructor declaration order decides first; fields break ties from left to right. Fields use the collection sorting rule: strings by contents, sequences component by component, nested ADTs recursively. Other field kinds keep the comparator's existing rules and display fallback. No `Ord` declaration is required, and this is not Haskell's type-class system.

```
data OrderedText = OrderedText(String)
data OrderedItems = OrderedItems(Array)
println(OrderedText("a") < OrderedText("a!"))      # true
println(OrderedItems([2]) < OrderedItems([10]))    # true
println(min(OrderedText("a!"), OrderedText("a")))  # OrderedText("a")
```

Opaque wrappers and immutable structs use the same constructor-value ordering. Changing a constructor's declaration position changes its order. Mutable records, including those inside an immutable payload, refuse ordering; choose a field key with `sort_by` instead. Operators also refuse comparisons between different ADT types. ADT declarations remain evaluator-only.

End of Chapter 18. Chapter 19 · Enums and Subranges as Invariants develops ordered enums and

restrictions on permitted values. Chapter 16.13 provides more practice with record mutation; Chapters 20–22 use classification, logic and variant data in larger programs.

Solutions to selected exercises: Chapter 18 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch18_solutions.md.)*

Chapter 19 · Enums and Subranges as Invariants

> **What this chapter is.** Two type-system tools that *narrow* the set of values a name can hold. An **enum** is a finite, ordered, named set of values — the days of the week, the suits of a deck, the states of a traffic light. A **subrange** is a *restricted* numeric or enum type — "an integer between 0 and 9," "any weekday but not the weekend." Both let you express *what's allowed* as part of the *type itself*, so violations can be caught by static checking (via `--typecheck`) where supported and by runtime annotation checks. The design goal is to make constraints explicit and reusable.

Chapter 18 · Types and Data Modeling places these forms in the full family. Here we develop ordering and restricted domains rather than repeat the schema/record/variant comparison.

19.1 The motivation — runtime checks are repetitive

Without enums, a "day of the week" gets modeled as a string:

```
# No enum — just strings
today: "Wednesday"
if today == "Saturday" or today == "Sunday" then [
  println("Weekend!")
] else [
  println("Workday.")
]
```

Three problems:

1. **Typos go undetected.** `"Wedensday"` (sic) compiles fine; only failing tests would catch it.
2. **The set of valid values is implicit.** What's a "day"? Whatever string you write. The system has no way to know `"Funday"` isn't legal.
3. **Equality checks proliferate.** Every place that dispatches on day-of-week has to repeat the seven possible string values.

The Pascal / Ada / Modula tradition fixed this fifty years ago with **enumerated types**. Axioma picks up the same tool.

19.2 Enums — the `enumerates` declaration

```
Day enumerates Mon, Tue, Wed, Thu, Fri, Sat, Sun
```

The unified spelling `type Day = enum Mon, Tue, Wed, Thu, Fri, Sat, Sun` and the legacy `enum Day = Mon, Tue, Wed, Thu, Fri, Sat, Sun` have the same meaning as `Day enumerates ...`. These compact declarations have no closing keyword.

That single line creates:

- A new type `Day` with exactly seven legal values.
- Each value (`Mon`, `Tue`, ..., `Sun`) bound as a top-level name.
- Each value can be tested by name (`today == Mon`).
- Each value has an ordering: `Mon < Tue < Wed < ...`.

An explicit parameter annotation makes Day membership part of the function contract:

```
Day enumerates Mon, Tue, Wed, Thu, Fri, Sat, Sun

is_weekend: func(d :: Day) [
  d == Sat or d == Sun
]
println(is_weekend(Mon))   # false
println(is_weekend(Sat))   # true
```

With `d :: Day`, `is_weekend("Sat")` is rejected by the runtime parameter check, and `--typecheck` diagnoses this direct literal call. Without the annotation, the same function body accepts the String and returns `false`: String `"Sat"` is not enum member `Sat`. The unannotated function does not acquire a Day-only contract just because its comparisons use Day members. Static checking is limited to supported cases; it does not replace the runtime guard.

19.3 Subranges — the `ranges` declaration

A **subrange** is a restricted integer type:

```
Digit ranges 0..9
d :: Digit: 5
println(d)       # 5

bad :: Digit: 99   # ERROR at runtime
```

Run that last line and Axioma errors:

```
type error: variable 'bad' expects type Digit, got INTEGER (value: 99)
```

The `99` doesn't fit in `Digit`'s 0..9 range, so the binding is rejected. (Compare: bare `bad: 99` would accept anything; the `:: Digit` annotation is the gate.)

Subranges are useful whenever an integer's *legal range* is narrower than `Integer`: a die roll (1..6), an hour of the day (0..23), an array index of a known-size array (1..N), an ASCII code (0..127), a percentage (0..100), an HTTP status code (100..599).

Encoding the range as a *type* makes the constraint explicit and reusable. The evaluator checks annotated bindings; the static checker can diagnose some violations before execution. A type annotation is not a promise that every invalid value will be caught statically.

19.4 Subtype-by-restriction — `extends X in a..b`

The pattern that makes the system *really* shine: an enum subtype whose values are a *contiguous interval* of the parent.

```
Day enumerates Mon, Tue, Wed, Thu, Fri, Sat, Sun
Workday extends Day in Mon..Fri
```

`Workday` is now a *new type* whose values are exactly `Mon, Tue, Wed, Thu, Fri` — *not* `Sat` or `Sun`.

```
w :: Workday: Tue   # OK
println(w)              # Tue

bad :: Workday: Sat # ERROR: type error: variable 'bad' …
```

A function whose parameter is annotated `Workday` *cannot* receive `Sat`. The compiler / runtime *guarantees* the parameter is a weekday. The function body doesn't need a `if d == Sat or d == Sun then error(...)` guard — the type *is* the guard.

This is the pattern Pascal called *subrange of an enum*; it's the same pattern modern languages with refinement types (Liquid Haskell, F*) implement at the type-theory level. Axioma's version is operationally simple — at runtime, every write to a `Workday`-typed variable checks against the interval — but the* expressive payoff* is the same: encode the invariant in the type.

19.5 The `--typecheck` pre-pass — invariants verified before running

The chapter wouldn't be complete without showing the static checker. Axioma has a `--typecheck` (alias `--check`) flag that walks a script *before running it* and reports type-error violations:

```
# day_logic.ax
Day enumerates Mon, Tue, Wed, Thu, Fri, Sat, Sun

today :: Day: "Wednesday"   # bug — string, not Day
next_day :: Day: Mon + 1    # bug — arithmetic on Day
```

Run:

```
$ axioma --typecheck day_logic.ax
day_logic.ax: line 3: type error: variable 'today' declared as Day,
  assigned value of type String
  Hint: change the annotation, or the value's literal form, so they agree
day_logic.ax: 1 type error(s) found.
```

The script never runs. Both errors are caught *before* a single line executes. For a real program with hundreds of annotated bindings and function signatures, `--typecheck` is the difference between "find the bug by running tests" and "find the bug at compile time."

19.6 Worked example — modeling a small calendar app

A calendar has events with a *day*, a *start hour*, and a *duration*. Hours range over 0..23; durations over 1..24. Days are enum members; weekday-only events use `Workday`.

```
Day enumerates Mon, Tue, Wed, Thu, Fri, Sat, Sun
Workday extends Day in Mon..Fri
Hour ranges 0..23
Duration ranges 1..24

concept Event

Event has day:      Mon         # default value
Event has start:    9
Event has duration: 1
Event has title:    ""

# A function whose parameter is annotated Workday gets
# parser-level help (and runtime enforcement).
schedule_meeting: func(d, h, dur, title) [
  an Event {day: d, start: h, duration: dur, title: title}
]

m1: schedule_meeting(Mon,  9, 1, "Stand-up")
m2: schedule_meeting(Wed, 14, 2, "Planning")

println(m1.title)
println(m2.start)
```

Then a query:

```
events: [m1, m2]
get_workday_events: func(arr) [
  filter(func(e) [e.day != Sat and e.day != Sun], arr)
]
println(len(get_workday_events(events)))   # 2
```

The `Workday`-typed parameter is the documentation *and* the runtime guarantee. The `Hour ranges 0..23` makes `schedule_meeting(Mon, 25, 1, "Bad")` a runtime error (you'll get `type error: variable expected type Hour, got Integer value 25`).

19.7 What enums + subranges replace

Compare to alternatives:

Approach	Pros	Cons
Plain strings (`"Mon"`, `"Tue"`)	familiar	typos, no exhaustiveness, no ordering
Integers (1 = Mon, 2 = Tue)	compact, ordered	meaning isn't in the value
Constants (`MON: 1; TUE: 2; …`)	named	no type protection — `MON` is just an Int
Enum class hierarchy (Java style)	OO-clean	heavy, lots of boilerplate
Axioma's `enumerates`	named, ordered, exhaustive, typed	one new declaration form to learn
Axioma's subrange	bounded numerics by type	one more
Axioma's `extends X in …`	refinement type without proofs	one more

The Axioma triple costs three lines of declaration to learn, buys you machine-checked invariants for *huge* swaths of real-world code. The Pascal tradition (Wirth, 1970) showed that this style works for teaching programming; Ada took it further; Modula-3 refined it. Axioma's port revives the tradition for a 2026 audience.

19.8 Reflection — design types so bugs become compile errors

The deep idea from this chapter: **the best way to handle a class of bugs is to make them *impossible to write*.** Not "catch them with tests," not "detect them with assertions" — make the type system refuse to compile the broken program.

- Strings instead of enums → typos are silent bugs. *Enums* turn typos into compile errors.
- Integers instead of subranges → out-of-range values silently get used. *Subranges* turn them into errors at binding time.
- Plain enums instead of subtype-by-restriction → "this can be any Day" silently allows weekends in code that needs Workdays. *Subtypes* refuse the weekend at binding time.

This is the *invariant-by-construction* principle. Every field declared with a type annotation, every function whose parameters are annotated, every `--typecheck` run, removes a class of possible bugs from your code. Real-world software that takes this seriously (Ada in aerospace, Rust in operating systems, OCaml in finance) achieves fewer production bugs *because the types do the work*.

You've now seen the four big tools in this tradition:

1. **Concepts** (Ch.18) — composite types with named slots.

2. **Inheritance** (Ch.18) — `is` makes type subset relations explicit.
3. **Enums** (this chapter) — named-finite-set types.
4. **Subranges and refinement subtypes** (this chapter) — bounded-or-restricted variants of a base type.

The next three chapters move out of the structural-type territory entirely. Chapter 20 introduces **multi-valued logic** — what if truth itself isn't binary? Chapter 21 introduces **logic programming** — what if the function calls were *queries* instead of computations? Chapter 22 closes with the **meta-circular evaluator** — an Axioma interpreter written in Axioma, tying every previous chapter back together.

A later addendum — **Chapter 54** — returns to types from the other direction. Chapter 19 is "you write the type, the checker verifies." `#language axioma/hm` is "the checker reads the body and *refuses the file* if it has no principal type." Host `--infer` on `axioma/all` is a lens that skips; the island is a language. Read 54 after this chapter if that is the next question you have. It is numbered 54 so Chapters 20–53 stay put.

Extension: related parameter types and bounds

A repeated type parameter connects fields within one constructor call. An upper bound restricts which runtime type that parameter may choose; several bounds join with `&` (`[T of Showable & Numeric]`), and each must hold.

```
data NumericPoint[T of Number] = MakePoint(x:: T, y:: T)
println(MakePoint(1, 2) is NumericPoint)       # true
println(MakePoint(1.0, 2.0) is NumericPoint)   # true
println(try(MakePoint(1, 2.0)) is Error)      # true
println(try(MakePoint("a", "b")) is Error)   # true
```

There is no automatic promotion: both direct occurrences of `T` must agree. Each constructor call chooses afresh. `Number` includes Complex; `Real` is not a seeded type. The old data-head `requires Number` spelling has been replaced by `T of Number`; function `requires:` contracts have their own role and remain supported.

This is not a fully specialized generic type system. Runtime values erase the chosen parameter type, and nested `Array of T`, dependent `T of U`, and `NumericPoint[Float]` annotations are not supported. Use explicit schemas for nested data and test construction as well as later mutation. These data declarations are interpreter-only.

Exercises

Exercise 19.1 — `Suit` enum

Declare `Suit enumerates Hearts, Diamonds, Clubs, Spades`. Write `is_red(s)` returning `true` for Hearts/Diamonds, `false` otherwise. Then `is_minor(s)` returning `true` for Diamonds and Clubs (the "minor" suits in some games).

- `is_red(Hearts)` → `true`

- `is_red(Spades)` → `false`
- `is_minor(Diamonds)` → `true`
- `is_minor(Hearts)` → `false`

Exercise 19.2 — `Digit` subrange and `safe_div`

Use `Digit ranges 0..9`. Write `safe_div(a :: Digit, b :: Digit)` that returns `a / b` if `b != 0`, else `0`. Verify that the annotation rejects out-of-range arguments:

- `safe_div(8, 2)` → `4`
- `safe_div(7, 0)` → `0`
- `safe_div(15, 3)` → runtime *type error* (15 not in 0..9)

Exercise 19.3 — `Workday` and `schedule_check`

With `Day enumerates Mon, …, Sun` and `Workday is Day in Mon..Fri`, write `schedule_check(d :: Workday, title)` that returns the string `"Scheduled: <title> on <d>"`.

- `schedule_check(Tue, "Meeting")` → `"Scheduled: Meeting on Tue"`
- `schedule_check(Sat, "Meeting")` → runtime type error

Exercise 19.4 — `Hour` and `is_business_hours`

Declare `Hour ranges 0..23`. Write `is_business_hours(h :: Hour)` returning `true` for $9 \leq h < 17$, false otherwise. Verify the type guard:

- `is_business_hours(10)` → `true`
- `is_business_hours(20)` → `false`
- `is_business_hours(-1)` → runtime type error

Exercise 19.5 — `Lock` state with enum + counter

Combine a Lock with `LockState enumerates Locked, Unlocked, Jammed` and a numeric `attempts :: Integer` counter. Each `try_open` call increments `attempts`; if attempts > 3, the lock transitions to `Jammed`. Return the new state.

```
lock: make_lock()
lock("wrong")  # attempts=1, still Locked
lock("wrong")  # attempts=2, still Locked
lock("wrong")  # attempts=3, still Locked
lock("wrong")  # attempts=4, becomes Jammed
```

(Combines Ch.16's mutation pattern with this chapter's enums.)

Exercise 19.6 (open) — `Grade` and `Honors`

Design a small academic grading system.

- `Grade enumerates Fail, Poor, Avg, Good, Top`
- `PassingGrade extends Grade in Poor..Top`

- `HonorsGrade extends Grade in Good..Top`

Write `letter_to_gpa(g :: Grade)` returning `0.0..4.0`. Write `is_passing(g :: Grade)` returning Boolean. Write `qualifies_for_honors(g :: Grade)` returning Boolean.

Then consider: *can* `qualifies_for_honors` be declared with parameter type `HonorsGrade` instead of `Grade`? What *should* it be? The right design is one where you don't have to ask "is this grade passing?" because the type *proves* it.

End of Chapter 19. Chapter 20 introduces **multi-valued logic** — the next stage of type-as-invariant pedagogy. Boolean logic is just one of many; Axioma supports Kleene three-valued, Łukasiewicz fuzzy, and Belnap four-valued logics as first-class. We'll see what they're for.

Solutions to selected exercises: Chapter 19 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch19_solutions.md.)*

Chapter 20 · Multi-Valued Logic

What this chapter is. Classical Boolean logic — true and false, nothing in between — is a clean and useful system, but it's not the only one. The world contains *missing data*, *partial measurements*, *probabilities*, and *outright contradictions*, and a logic that pretends those don't exist has to lie about them. This chapter introduces three of the logics Axioma ships as first-class citizens: **Kleene K3** (true / false / unknown), **Łukasiewicz L3 + Lₙ** (real-valued "shades of true" in [0, 1]), and **Belnap B4** (true / false / both / neither — the *paraconsistent* logic that survives contradictions instead of exploding on them). Their truth values are literally first-class: the forms Axioma *prints* (`?ᵏ`, `½ł`, `⊤⊥ᵇ`) are valid *source*, usable anywhere a value goes — including `match` patterns. (Two more logics, intuitionistic **Gödel G3** and Priest's **LP**, appear in the manual §9 and the non-classical-logic book passes.) The pedagogical bet: **"truth" is a design choice, not a physical law.**

20.1 The motivation — Boolean isn't always right

A standard Boolean test:

```
lab_result: true
symptom: false
if lab_result and symptom then [
  println("Diagnose disease X")
] else [
  println("Rule out disease X")
]
```

Now suppose the lab hasn't run yet. What's `lab_result`? *Pretending* it's `false` is a lie — we don't know. *Pretending* it's `true` is also a lie. The correct answer is "we don't know," and Boolean logic has nowhere to put that.

A second case. Two witnesses give contradictory testimony: witness A says the suspect was at the scene, witness B says the suspect was elsewhere. A Boolean variable can hold one or the other, not both. But the *evidence base* genuinely contains both claims, and downstream reasoning needs to know that.

A third case. Asking "is this medical test positive?" when the test returns a probability between 0.0 and 1.0. Boolean has to *threshold* somewhere — usually 0.5 — and discards the nuance.

Three different kinds of "not-quite-Boolean":

Kind	What it captures	Logic
Missing value	"Don't know yet"	**Kleene K3**
Partial truth	"Probably / mostly"	**Łukasiewicz L3**
Contradiction	"Both true and false simultaneously"	**Belnap B4**

Axioma supports all three. The Boolean operators (`and`, `or`, `not`, `implies`) auto-dispatch on the operand type, so you write the same expression and Axioma picks the right truth table.

20.2 Kleene K3 — three-valued logic for missing data

Kleene's three-valued logic adds one new value to Boolean: **unknown**. In Axioma, `om` (or the Greek letter Ω) is the unknown value:

```
lab: om          # haven't run the test yet
symptom: true

println(lab)              # Ω
println(lab and symptom)  # Ω  (unknown and true remains unknown)
println(lab or symptom)   # true  (true or anything is true)
println(not lab)          # Ω  (negation of unknown is still unknown)
```

K3 also has *typed* literals — `T^k` `⊥^k` `?^k` (constructor form `kleene("true")` etc.) — which run the same tables but keep the Kleene tag on results (`?^k and T^k → ?^k`). Both typed unknown and bare `om` are falsy: `if ?^k …` and `if om …` take the *else*-branch. `?^k == om` is still `true`. A typed value can therefore have the same truthiness as a bare value while preserving its logic type in an operator's result.

The K3 truth tables:

a	b	a and b	a or b	not a
true	true	true	true	false
true	false	false	true	false
true	Ω	Ω	true	false
false	false	false	false	true
false	Ω	false	Ω	true
Ω	Ω	Ω	Ω	Ω

Read the table by recognizing the pattern: an output is `true` or `false` *only when the unknown values don't matter*. If `a or b` has at least one `true`, the result is `true` regardless of what `b` actually is. If `a and b` has at least one `false`, the result is `false` regardless. Otherwise: Ω propagates.

A worked example. Diagnose-or-rule-out from §20.1, but with K3:

```
lab: om               # haven't run
symptom: true
```

```
diagnose: lab and symptom

if diagnose == true then [
  println("Diagnose disease X")
] else if diagnose == false then [
  println("Rule out disease X")
] else [
  println("Order more tests — diagnosis still unknown")
]
```

The third branch is the *new option* K3 gives you. Without it you'd be forced to misclassify the case.

20.3 Łukasiewicz L3 — shades of true

Sometimes the question isn't "is the lab test back?" but "how confident are we?" Łukasiewicz logic admits values in the *continuous* interval [0, 1]:

- `lukasiewicz(0.0)` — certainly false (the literal ⊥Ł)
- `lukasiewicz(1.0)` — certainly true (the literal ⊤Ł)
- `lukasiewicz(0.5)` — exactly half-true (the literal ½Ł)
- `lukasiewicz(0.85)` — strongly true (non-canonical values need the constructor)

```
high_confidence: lukasiewicz(0.85)
medium_confidence: lukasiewicz(0.5)
low_confidence: lukasiewicz(0.15)

println(high_confidence)             # ½Ł-style display
println(high_confidence and medium_confidence)
println(high_confidence or low_confidence)
println(not medium_confidence)
```

The L3 rules:

Operation	Formula
`a and b`	`min(a, b)`
`a or b`	`max(a, b)`
`not a`	`1 - a`
`a implies b`	`min(1, 1 - a + b)`

So `0.6 and 0.8` = `0.6`, `0.6 or 0.8` = `0.8`, `not 0.6` = `0.4`, `0.6 implies 0.8` = `min(1, 0.4 + 0.8)` = `1.0`.

When does L3 reduce to ordinary Boolean? When you restrict to `0.0` and `1.0` only. Boolean is the "two-point" subset of L3. Same operators, same conclusions.

Pedagogical caveat. Łukasiewicz is *not* the same as probability. `lukasiewicz(0.5)` does not mean "50% chance of being true"; it means "the truth value itself is exactly in the middle." Probability theory

has its own operators (Bayes' rule, marginalization, etc.) that don't match the min/max algebra above. The two systems answer different questions — don't mix them up.

20.4 Belnap B4 — surviving contradictions

The fourth logic adds two new values to the classical pair:

- `true` — known true
- `false` — known false
- `both` — known *both* true and false (contradiction)
- `neither` — known *neither* (no evidence at all)

```
witness_a: belnap("true")     # A says it's true       (the literal ⊤ᵇ)
witness_b: belnap("false")    # B says it's false      (the literal ⊥ᵇ)
contested: ⊤⊥ᵇ                # we record the contradiction — the GLUT
no_evidence: ?ᵇ               # we record the absence      — the GAP
```

(Priest's names for the two new values — the *glut* and the *gap* — are first-class too: `Belnap.glut` / `Belnap.gap`, and the digraphs `` `glut `` / `` `gap `` type them in plain ASCII. `Belnap.values` is the whole domain, `[⊤ᵇ, ⊥ᵇ, ⊤⊥ᵇ, ?ᵇ]`.)

Why does `both` get its own value? Classical Boolean *can't* hold contradictory testimony as data. If you assign one variable both `true` and `false`, you've committed to one or thrown the other away. Belnap gives you a *named value* for "we have evidence both ways" — a fact downstream code can act on.

The B4 value space:

```
both       T-evidence AND F-evidence
true       T-evidence only
false      F-evidence only
neither    no evidence either way
```

Axioma's operators are **truth-order** — they treat `true` as the "largest" value and `false` as the "smallest," with `both` and `neither` sitting in between:

- `a or b` — `true` is absorbing (T or x = T), `false` is identity (F or x = x), `both or both = both`, `both or neither = both`, `neither or neither = neither`.
- `a and b` — `false` is absorbing (F and x = F), `true` is identity (T and x = x), `true and both = both`, `true and neither = neither`, `both and neither = false`.
- `not a` — swap `true` ↔ `false`; `both` and `neither` are fixed points (negating a contradiction is still a contradiction; negating "no evidence" still leaves no evidence).

Probe these yourself — Exercise 20.3 includes a truth-table generator that you can adapt to print all sixteen B4 cells.

How does `both` actually arise in your code? Four ways:

1. **Explicit construction** — the literal `⊤⊥ᵇ` (or `belnap("both")`). You have external evidence that both T and F are claimed.
2. **Evidence combination** — `b4_join(witness_a, witness_b)` (infix `⊕`): the *knowledge-order* join builds the glut *for you* when sources conflict. This is the honest way to merge testimony — see

§20.7.

3. **Storing a B4 truth value on a stored fact** — `set_truth("parent", "alice", "bob", "both")`. The rule system propagates the `both` value through any defeasible derivations (`set_truth_combine` ⊕-merges instead of overwriting).
4. **Compound expression with a `both` operand** — once `both` is in your data, the operators preserve it correctly: `T⊥ᵇ and T` is `T⊥ᵇ`, `T⊥ᵇ or ⊥ᵇ` is `T⊥ᵇ`.

Note that **`belnap("true") or belnap("false")` is `true`**, not `both`. The *truth-order* disjunction "wins" with truth — it structurally *cannot* build a glut from classical inputs. Recording a contradiction is a *knowledge-order* act: construct `T⊥ᵇ` explicitly, or let `b4_join` (⊕) infer it from the differing testimony.

Branching on B4 values. A typed truth value behaves like one in control flow: `if v then … else …` takes the then-branch exactly when `v` is *designated* — `Tᵇ` or the glut `T⊥ᵇ` (a glut is "true-enough to act on"; `designated(v)` is the same test as a function). The gap `?ᵇ` and `⊥ᵇ` take the else-branch. For the four-way case split, the literals are `match` patterns:

```
describe: func(v) [
  match v with
  | Tᵇ  => "known true"
  | ⊥ᵇ  => "known false"
  | T⊥ᵇ => "contested"
  | ?ᵇ  => "no evidence"
]
```

The paraconsistent payoff. In Boolean logic, `false and true` = `false`, but a *single* contradictory assertion (`x = true; x = false`) makes the whole system explode — "from a contradiction, anything follows" (`ex falso quodlibet`). Belnap is **paraconsistent**: contradictions stay *local*. You can keep the contradictory fact, reason around it, and the rest of your conclusions remain valid.

This matters for *real* data: legal cases, debate transcripts, medical-evidence aggregation, web-scraped knowledge bases — sources contradict each other constantly, and a logic that crashes on the first inconsistency is unusable.

20.5 The Design Recipe — when to pick which logic

Stage 1 — data definition. Decide *what your data is*. This shapes the choice of logic:

Your data has…	Use logic
Clean true/false only	Boolean
Missing values	Kleene K3
Confidence levels	Łukasiewicz L3
Contradictions to preserve	Belnap B4

Stage 2 — signature. Write the type:

```
# Boolean version
is_diagnosable: func(lab_result, symptom) [...]

# K3 version (with comment noting it)
# is_diagnosable: lab_result :: Kleene, symptom :: Kleene -> Kleene
```

Axioma doesn't yet enforce these annotations as types, but writing them documents the intent.

Stage 3 — examples. Write at least one example per "interesting" combination. For K3 with two inputs, that's nine cases (3×3). For B4, sixteen (4×4). The exercises below show how to do this without writing all sixteen rows each time.

Stage 4 — template. For predicates over MVL inputs, the template is the same as Boolean: combine the inputs with operators. *The operators do the dispatch for you.* You rarely need a separate function per logic.

Stage 5 — body. Same as Boolean, but mind the new output values. A condition that's "true" in Boolean might be Ω or `both` in MVL.

Stage 6 — tests. Test every truth value combination that *matters* for your function. For K3, test at least one case with `om`. For B4, test at least one case each with `both` and `neither`.

20.6 A worked example — clinical-trial database

Suppose a small trial measures four things per patient: *lab_positive*, *symptom_present*, *family_history*, *age_above_50*. Each can be `true`, `false`, or `om` (not yet measured).

A patient *enters* treatment if:

> The lab is positive AND (a symptom is present OR family history is positive) — *but only if* the patient is over 50.

```
should_treat: func(lab, symptom, history, over_50) [
  lab and (symptom or history) and over_50
]

# Full data
println(should_treat(true, true, false, true))    # true
println(should_treat(false, true, true, true))    # false

# Missing lab
println(should_treat(om, true, false, true))      # Ω

# Missing history is fine if symptom is true
println(should_treat(true, true, om, true))       # true

# Missing history fatal if symptom is false
println(should_treat(true, false, om, true))      # Ω

# Under-50 always rules out regardless of unknowns
println(should_treat(om, om, om, false))          # false
```

The function is a one-liner. The K3 operators handle the "unknown" propagation. Reading the test output is the *interesting* part: any case where the answer's actually decidable returns `true`/`false`, otherwise it correctly says Ω.

That's the pedagogical payoff. **The same function works for classical Boolean and for K3 — the operators dispatch on type.** You don't write two functions; you write one and feed it different inputs.

20.7 A worked example — debate-outcome scorer with B4

Two judges score a debate on a single proposition. We want a single B4 value that records *what they said collectively* — either consensus, contradiction, or no ruling at all.

The natural `or` doesn't help here: `belnap("true") or belnap("false")` is `belnap("true")` in Axioma (truth-disjunction absorbs to `true`). What we want is a *consensus* combiner that produces `both` exactly when the two inputs disagree — and B4's **knowledge order** ships it as `b4_join` (infix ⊕, "accept everything every source says"):

```
judge_a: belnap("true")
judge_b: belnap("false")
combined: b4_join(judge_a, judge_b)     # or: judge_a ⊕ judge_b
println(combined)                       # T⊥ᵇ — conflict surfaces as the glut

verdict_describe: func(v) [
  match v with
  | Tᵇ  => "Consensus yes"
  | ⊥ᵇ  => "Consensus no"
  | T⊥ᵇ => "Judges disagree"
  | ?ᵇ  => "No judge ruled"
]

println(verdict_describe(combined))     # "Judges disagree"
println(verdict_describe(b4_join(?ᵇ, Tᵇ)))   # "Consensus yes" — no-opinion is
 ↪  ⊕-identity
```

Under the hood, `b4_join` is a five-case table you could write yourself (Exercise 20.4 has you do exactly that, then check your answer against the builtin): equal inputs pass through, `?ᵇ` is the identity, and any genuine T-vs-F conflict — or an existing glut — yields `T⊥ᵇ`. Its dual `b4_meet` (⊗, "consensus": keep only what all sources agree on) maps the same conflict to `?ᵇ` instead.

The point of B4 is *not* that Axioma's `or` magically detects contradictions — it's that B4 gives you a **value space** with a named cell for "evidence both ways" *and the pair of combination operators that live on it*, which downstream code can dispatch on (the `match` above). Compare to a Boolean approach: you'd have to add a side-channel boolean called `disagreement_flag` and remember to check it at every site that handles the verdict. Belnap *folds the flag into the value*.

20.8 Exercises

20.1 — `definitely_pass`

A test has three Kleene-valued inputs: `submitted`, `passed_grade`, `attendance_ok`. Each is `true`, `false`, or `om`. Write `definitely_pass(submitted, passed_grade, attendance_ok)` returning a Kleene value that is:

- `true` if **all three** are `true`,
- `false` if **any** of them is `false`,
- `om` otherwise.

Test with these six cases (compute the expected output before running):

```
println(definitely_pass(true, true, true))      # ?
println(definitely_pass(true, false, true))     # ?
println(definitely_pass(true, om, true))        # ?
println(definitely_pass(om, om, om))            # ?
println(definitely_pass(om, false, om))         # ?
println(definitely_pass(false, om, true))       # ?
```

Hint: this is exactly `submitted and passed_grade and attendance_ok` — the K3 operator does the dispatch.

20.2 — `confidence_score`

Three sub-scores in Łukasiewicz [0, 1] capture how confident we are in a candidate's *technical*, *communication*, and *teamwork* skills:

- `tech: lukasiewicz(...)`
- `comm: lukasiewicz(...)`
- `team: lukasiewicz(...)`

Hire if **all three are at least 0.7**. Reject if **any one is at most 0.3**. Otherwise "interview again."

Write `hiring_decision(tech, comm, team)` returning the string `"hire"`, `"reject"`, or `"interview again"`.

Hint: Łukasiewicz values don't support </>= directly — those operators belong to the *numeric* algebra, not the L3 truth algebra. But the *algebra itself* gives you a clean test. Compute `tech and comm and team` (the L3 `and` is `min`). If the meet equals `lukasiewicz(0.7)` *or higher*, you've cleared the bar. To express "or higher" in L3, use the fact that `a or b` is `max(a, b)`: if `(meet) or lukasiewicz(0.7) == meet`, then `meet >= 0.7`.

Equivalently, accept the Łukasiewicz inputs but also accept raw floats as a fallback API: write a helper `at_least(v, threshold)` that returns `(v or lukasiewicz(threshold)) == v`.

20.3 — K3 truth-table generator

Write `k3_truth_table(name, op)` where `op` is a two-argument function and `name` is its display name. The function should print a 3×3 truth table for `op`:

```
a       b       a name b
true    true    true
```

```
true    false   false
true    om      om
false   true    false
false   false   false
false   om      false
om      true    om
om      false   false
om      om      om
```

(That's the K3 `and` table — your function should produce it when called as `k3_truth_table("and", func(a, b) [a and b])`.)

20.4 — `belnap_aggregate` (consensus combiner)

Write a *consensus combiner* over three Belnap witnesses:

```
belnap_aggregate: func(w1, w2, w3) [...]
```

Each `wi` is a Belnap value (`belnap("true")`, `belnap("false")`, `belnap("both")`, or `belnap("neither")`). The result should satisfy:

- If all three are equal, return that value.
- If two say `true` and one says `false` (or vice-versa), return `both` — the witnesses contradict each other.
- If some are `neither` (no opinion) and the rest agree on one truth value, return that truth value (no-opinion witnesses don't change the consensus).
- If at least one is `both`, return `both` (existing contradiction propagates).

Then write `verdict_describe(v)` that prints:

- `"Consensus yes"` if the result is `belnap("true")`
- `"Consensus no"` if the result is `belnap("false")`
- `"Disagreement"` if the result is `belnap("both")`
- `"No evidence"` if the result is `belnap("neither")`

Hint: build it from a two-argument helper `combine_two(a, b)` that handles the four cases above; then `belnap_aggregate(w1, w2, w3) = combine_two(combine_two(w1, w2), w3)`.

Check your answer against the builtin: your `combine_two` should agree with `b4_join` on all sixteen input pairs, and your three-witness aggregate with the variadic `b4_join(w1, w2, w3)`. Write the comparison loop over the domain rather than sixteen hand-written cases (bind it first — `dom: Belnap.values` — since `foreach` wants a simple iterable expression):

```
dom: Belnap.values
foreach a in dom [
  foreach b in dom [
    expect("agrees with ⊕", combine_two(a, b), b4_join(a, b))
  ]
]
```

20.5 — Łukasiewicz `meet_all` and `join_all`

Write two functions:

- `meet_all(values)` — given a non-empty array of Łukasiewicz values, return the *meet* (the L3 `and` reduced across the array — that is, the minimum truth value).
- `join_all(values)` — same but using *join* (the L3 `or` — the maximum).

Examples:

```
v: [lukasiewicz(0.5), lukasiewicz(0.8), lukasiewicz(0.2)]
println(meet_all(v))   # 0.2ł
println(join_all(v))   # 0.8ł
```

Hint: this is `reduce` from Chapter 10, with the L3 operator as the combiner. The L3 `and` *is* the meet; the L3 `or` *is* the join. You don't need to extract any floats — the operators do everything for you.

20.6 (open) — Multi-source medical aggregator

You're aggregating diagnostic input from four sources for a suspected disease:

- `lab_result`: Kleene (true / false / unknown)
- `symptom_severity`: Łukasiewicz [0, 1]
- `expert_a_opinion`: Belnap
- `expert_b_opinion`: Belnap

Write `combined_diagnosis(lab, severity, opinion_a, opinion_b)` that returns a *string* describing the overall picture:

- `"Treat"` — lab true, severity ≥ 0.7, experts agree on true
- `"Don't treat"` — lab false, severity ≤ 0.3, experts agree on false
- `"Expert disagreement"` — expert join is `both`
- `"Need more data"` — lab is `om` or severity unclear
- `"Order tests"` — neither expert ruled

You'll have to *mix* the three logics. Decide for yourself which combinations the function should resolve and which it should pass through. Document your design choice as a comment above the function.

20.9 Reflection — why MVL belongs in CS1

Three reasons, in increasing order of importance.

One. Real-world data is missing all the time. Survey responses skipped, sensors offline, lab tests not yet run. A language that pretends those don't exist forces students into two equally-bad habits — fake values (sentinel constants like −1) or scattered null-checks. K3 gives them the right tool: a logical value that means *exactly* "I don't know."

Two. Real-world data is sometimes *uncertain*. Confidence scores, model outputs, expert opinions on a sliding scale. Łukasiewicz gives an algebra for combining them without collapsing back to threshold-Boolean prematurely.

Three. Real-world data is sometimes *contradictory*. Witnesses disagree; sources conflict; legacy systems and new systems hold incompatible facts. Belnap is the *only* classical-extension logic that doesn't melt down when this happens. Students who learn it *first* — before being trained to assume cleanly-Boolean data — write more robust code their whole career.

The pedagogical bet stands: **truth is a design choice.** Choose your logic, and the operators come along for free.

The next chapter steps up one level, from *the value* of a fact to *the inference* of facts from rules — Prolog-style logic programming.

> *"The truth is rarely pure and never simple."* — Oscar Wilde. He was right, and now Axioma agrees.

Solutions to selected exercises: Chapter 20 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch20_solutions.md.)*

20.10 Optional: probability is not missing information

A probability is a measure over events. It is distinct from a fuzzy membership grade, a model's confidence, or Belnap's `unknown`. Axioma's named probability table requires a supplied marginal or a hard observation. An absent entry is an error; it is not an implicit 0.5.

```
chance: prob_logic_kb()
add_prob_formula(chance, "rain", 0.3)
add_prob_formula(chance, "coin", 0.5)
println(prob_and(chance, "rain", "rain"))
println(prob_and(chance, "rain", "coin"))
println(is_error(try(query_prob(chance, "snow"))))
set_prob_evidence(chance, "rain", true)
println(query_prob(chance, "rain"))
```

The output is `0.3`, `0.15`, `true`, `1.0`. Repeating the same event does not make independent evidence. Distinct events in the default `independent` mode are assumed independent unless an explicit conditional is supplied. The `bayesian` mode requires dependence information when the marginals alone do not determine a joint. Both modes check supplied pairs for consistency; neither constructs a full Bayesian network or guarantees a globally consistent joint over three or more events. `markov_logic` is currently refused.

A hard observation changes that named event's local value. It does not update other marginals. An explicit `bayesian_update` needs a likelihood and the stored pre-observation evidence marginal. These probability builtins can run in the VM, but the example's `try` error capture requires the evaluator.

Exercise 20.7 — Repetition is not independence. Explain why rain AND rain has probability 0.3, while rain AND an independent fair coin has probability 0.15. What additional information would you need if the coin event were replaced by wet grass? Why should a missing probability not become `unknown` in B4 or 0.5? See the chapter solutions.

20.11 Optional: truth tables and evaluation order

The truth tables in this chapter answer: **given two values, what is the result?** Evaluation order answers a different question: **which expressions run to obtain those values?** A known result does not by itself promise that the interpreter will skip an expression.

For ordinary `and` / `or` expressions, Axioma skips the right operand only for Boolean `false` / Boolean `true`, respectively. A typed Kleene, Belnap, Łukasiewicz or Intuit3 value on the left still evaluates the right operand, as does bare `om`. This is the current implementation policy. For example, Kleene false determines the value of a conjunction, but it is a `Kleene`, not a `Boolean`, so it does not activate the Boolean-only gate.

The `probe` function below prints its label and then returns its value. Each comment lists the output in order:

```
probe(label, value) = [println(label); value]
println(false and probe("Boolean", true))        # false; no label
println(kleene("false") and probe("K3", true))    # K3, then ⊥ᵏ
println(belnap("true") or probe("B4", false))     # B4, then ⊤ᵇ
println(kleene("false") andalso probe("skip", true)) # false
println(belnap("true") orelse probe("skip", false))  # true
```

`andalso` / `orelse` apply the same truthiness conversion used by a condition and return a plain Boolean. They can skip a call with a non-Boolean left value, but they also discard the multivalued result. Use them for a Boolean guard; retain `and` / `or` when unknown or conflicting information must survive.

The other ordinary binary logical connectives are **eager**: `xor`, `iff`, `implies` (also `impl`), NAND ⊼ and NOR ⊽ evaluate the right operand after the left succeeds. This remains true when Boolean algebra determines the answer from the left alone. Negation evaluates its single operand. Ordinary function calls evaluate their arguments before calling, including `nand(...)` and `nor(...)`. An error in the left operand propagates before the right runs. See Appendix A for the complete alias table and a trace of all five eager connectives.

Exercise 20.8 — Predict the calls. Using `probe` above, predict every printed label and result before running these expressions. Explain why the second and third lines differ, and why the fourth line still calls `probe`.

```
println(false implies probe("A", false))
println(kleene("false") and probe("B", true))
println(kleene("false") andalso probe("C", true))
println(nand(false, probe("D", true)))
```

Chapter 21 · Logic Programming

What this chapter is. A *logic program* is a collection of **facts** and **rules**, with answers extracted via **queries**. Instead of writing *how to compute*, you write *what is true*, and the system figures out the rest. The tradition runs from Prolog (1972) through Datalog and modern logic engines like ErgoAI; Axioma's logic subsystem inherits from all three. The pedagogical bet: **declarative knowledge deserves a different programming style than computation.** You won't replace functions with rules — you'll add rules to your toolkit for the cases where they're a better fit.

21.1 The motivation — relationships are not functions

A genealogy program. You want to know "who are John's grandchildren?" In functions, you write:

```
children_of: func(p) [...]              # returns array
grandchildren_of: func(p) [
  kids: children_of(p)
  nested: map(children_of, kids)
  flatten(nested)
]
```

You write `children_of` once. Then `grandchildren_of`. Then `great_grandchildren_of`. Then `siblings_of`. Then `cousins_of`. Each one is a *separate function*, each one has to *re-walk the family tree*, each one needs careful handling of duplicates, missing data, and infinite loops.

The relational view is different:

```
relation parent(child, parent)
# or the short reserved alias:  rel parent(child, parent)

parent("Mary", "John")
parent("Tom", "John")
parent("Alice", "Mary")
parent("Bob", "Mary")
```

Five facts. Nothing computational. Then you *query*:

```
# Who are John's children?
{C | parent(C, "John")}        # {"Mary", "Tom"}
```

```
# Who are John's grandchildren?
{G | parent(G, P) and parent(P, "John")}    # {"Alice", "Bob"}

# Who are siblings?
{(A, B) | parent(A, X), parent(B, X), A != B}
```

The structure of the data drove the queries. You never wrote `grandchildren_of` — the query expression said what you wanted, and the engine figured out how. *That's the relational style.*

21.2 Facts and the `relation` / `rel` declaration

Before you can store facts, declare the relation (`relation` and `rel` are the same reserved word — two spellings):

```
relation parent(child, parent)
rel age(person, years)          # short form — same as `relation age(...)`
relation friend(a, b)
```

That's like declaring a database schema. After the declaration, the relation's *name* (like `parent`) becomes a predicate you can populate with facts:

```
parent("Mary", "John")
parent("Tom", "John")
parent("Alice", "Mary")
parent("Bob", "Mary")
```

Each line is *not* a function call — it's an **assertion**. The fact is stored in the knowledge base, indexed for query.

Each fact has a *grounding*: by default, raw `parent("Mary", "John")` is a **datum** (no derivation, no justification). You can promote facts to **axioms** (most-trusted) or write them as **postulates** (tentative claims) using refinement syntax — but for CS1 the default works fine.

21.3 Queries via set comprehensions

You met the comprehension surface in §10.9 — array comprehensions, set comprehensions, filters, walrus bindings, dict comprehensions, lazy generators. The same notation works for logic-programming queries; the only thing that changes is what sits on the right of `<-`. There, instead of an array or a set, you write a **relational call** like `parent(X, Y)`, and the comprehension becomes a query against the fact store.

The basic shape:

```
{output_pattern | V <- relation(args), V <- relation(args), ...}
```

The body is a comma-separated list of **generators**, each of the form `V <- relation(args)`, where `V` is a fresh **variable** (capitalized identifier) and `relation(args)` is a query pattern with literals, underscores, or other variables in its argument positions.

```
# All parents of Mary
{P | P <- parent("Mary", P)}              # {"John"}

# All known parents (any direction)
{P | P <- parent(_, P)}                    # underscore = "don't care"

# Pair-flat output (no <- generator needed for full-tuple)
{(C, P) | parent(C, P)}                    # all pairs

# Multi-generator (variable shared across generators)
{G | P <- parent("Alice", P), G <- parent(P, G)}    # Alice's grandparents
```

The underscore _ in an argument position means "any value matches, don't bind." It's the difference between "I want to *use* the parent's name later" (`P`) and "I just want to know *that* a parent exists" (`_`).

For more complex predicates that combine relations or apply arithmetic filters, the recommended idiom is **define a rule** and then query the derived relation. We'll see that pattern in §21.5.

21.4 Variables (capitalized identifiers)

In the query body, **any identifier starting with a capital letter** acts as a logic variable — a name that the engine fills in to make facts match. Lower-case identifiers (and quoted strings) are *constants*.

A single capital letter (`X`, `Y`, `Z`, `P`) is the Prolog-traditional way. Multi-letter capitalized names (`Child`, `Parent`) work too:

```
{(Child, Parent) | parent(Child, Parent)}    # same as {(C, P) | ...}
```

Note: the single-letter quantifier shorthands `A` (∀) and `E` (∃) were *retired* in June 2026, so `A` and `E` are now ordinary logic variables — use any capital you like (`A`, `E`, `X`, `Y`, `Z`, ...). Quantify with the words `forall` / `exists` or the glyphs `∀` / `∃`.

When a variable appears in **multiple** positions of a query, the engine **unifies** the bindings — that is, both positions must end up with the same value:

```
# A grandparent: someone who is parent of a parent
{G | parent(P, G) and parent(C, P)}
# Here P appears twice — both occurrences must match the same person.
```

That's the heart of logic programming: **multi-position unification** is what lets relational queries do work that imperative code would need explicit nested loops for.

21.5 Rules — deriving new facts from old

Sometimes the same query pattern shows up over and over. You can name it as a **rule**:

```
relation grandparent(child, gp)

# Whenever there's a parent-of-parent chain, infer grandparent
grandparent(C, G) whenever parent(C, P) and parent(P, G)
```

The rule connective is the word `whenever` — the code now says exactly what the comment says, and that word choice is doing logical work: English "whenever" means "in *every* case where," which is precisely what a rule head claims. This is the primary spelling. Also first-class: `head if body` (same clause; gated on an uppercase logic-var argument because `if` doubles as the postfix conditional), and the operator twins `<==` (legacy `<=`) and Prolog's `:-` in arrow or Prolog dress. Once the rule is in scope, queries can use `grandparent` like any other relation:

```
# Who are Alice's grandparents?
{G | grandparent("Alice", G)}    # {"John"}

# Whose grandparent is John?
{C | grandparent(C, "John")}     # {"Alice", "Bob"}
```

The engine **evaluates the rule body lazily** — at query time, it walks the parent facts, finds matches, and produces the answers. You wrote *no* computation. You wrote a *truth condition*, and the engine inferred the answers.

Rules can also chain through **themselves**. The classical Prolog/Datalog idiom of *two* clauses for one relation — a base case plus a recursive case — works exactly as in Datalog: clauses for the same head accumulate, and the engine iterates them to a fixpoint. Transitive closure is the canonical example (§21.9 runs one on real data):

```
relation reachable(a, b)
reachable(X, Y) whenever edge(X, Y)
reachable(X, Y) whenever edge(X, Z) and reachable(Z, Y)
```

For the *single-step* case, one rule is enough:

```
relation grandparent(child, gp)
grandparent(X, G) whenever parent(X, P) and parent(P, G)
```

A common gotcha — `X != Y` in rule bodies. Equality and inequality *between rule variables* aren't enforced by the rule engine the way they would be in classical Prolog. Writing `sibling(X, Y) <= parent(X, P) and parent(Y, P) and X != Y` produces all pairs *including* the self-pairs `(X, X)`. The clean idiom is **rule then post-filter**:

```
relation sibling_raw(x, y)
sibling_raw(X, Y) whenever parent(X, P) and parent(Y, P)

# Post-filter: drop the self-pairs
all_sibs: {p | p <- {(X, Y) | sibling_raw(X, Y)}, p[1] != p[2]}
```

That's the convention this chapter will follow for any "inequality-on-variables" filter.

21.6 Strict vs. defeasible rules

Some inference is **strict** — "always follows": plain `head whenever body` (also `head if body`; operator twin <= / <==). Other inference is **typical** — "usually follows, but with exceptions" — and you write it with exactly that word: `typically head whenever body` (synonym `normally`; marker spelling `rule~ head if body`, read the tilde as "roughly"; operator twin <~~).

```
relation bird(x)
relation flies(x)
relation penguin(x)

bird("robin")
bird("tweety")
bird("opus")
penguin("opus")

# Defeasible rule: birds typically fly — written as said
typically flies(X) whenever bird(X)

# Query — who flies?
{X | flies(X)}    # {"opus", "robin", "tweety"} — all three derived defeasibly
```

The rule `typically flies(X) whenever bird(X)` (≡ `rule~ flies(X) if bird(X)` ≡ `flies(X) <~~ bird(X)`) produces **conjectures** (weaker than theorems). To handle the exception — penguins don't fly — we explicitly **cancel** the conclusion:

```
cancel("flies", "opus")

{X | flies(X)}                          # without filter — still includes opus
{X @conjecture | flies(X)}              # only un-canceled conjectures
```

The `@conjecture` tag is a **filter** — only facts at that grounding level pass through. The default `{X | flies(X)}` includes canceled facts (with their cancellation noted), so you can audit *which* defeasible rules were overridden.

Compare this to two alternatives:

- **Boolean fact `is_flyer`** — every species needs a separate Boolean stored fact. The "default flying" pattern is implicit in your data.

- **Functional `can_fly(species)` with hardcoded cases** — the code structure encodes the inference, not the data.

The defeasible rule says **the rule is the default, the cancellations are the exceptions**, and that's *exactly* what the cognitive model looks like for most natural-world knowledge.

21.7 Forward and backward chaining

The rules above all use <= (backward chaining) — Axioma *starts from the query* and works backward through rule bodies to find supporting facts.

There's a dual operator, ==> (forward chaining), which says "whenever this body matches, derive the head":

```
# Forward: every parent-of-parent chain forces a grandparent fact
parent(X, Y) and parent(Y, Z) ==> grandparent(X, Z)
```

Same semantics, *opposite direction*. Forward chaining is what production rule systems (CLIPS, Drools) do — every time the rule body matches a fact pattern, the head fact gets added to the knowledge base.

The defeasible forward variant ~~> works analogously:

```
bird(X) ~~> flies(X)
```

For Axioma's purposes, the four operators are interchangeable for simple queries — the engine picks whichever evaluation strategy is cheaper. The difference matters for *side effects* (forward chaining can trigger automatic notifications) and for **performance** (forward is better for many-fact, few-query workloads; backward for the opposite).

For CS1, use `head whenever body` for "rule that says when something is true" and `typically head whenever body` for "rule with exceptions."

21.8 The Design Recipe for logic programs

The recipe takes a different shape here.

Stage 1 — data declaration. Decide *what relations exist*. That's the schema. Each relation has a name and an arity (number of arguments). Capitalize positions you'll often query *into* (it makes the query patterns readable).

Stage 2 — fact base. Write the ground facts. These are the *data* of the logic program — the leaves of the inference tree. Comments explaining the data are encouraged ("These are the actual measured parent-child relationships in the 1850 census…").

Stage 3 — rules. Add rules for derived relationships. For each rule, ask: is it **always** true (plain `whenever`) or **usually** true (`typically … whenever`)? Write the strict version first; only defeasible if you know there are real exceptions to handle.

Stage 4 — queries. Express the question you want answered as a set comprehension. Variables for what

you want, ground values for what you know, underscores for what doesn't matter.

Stage 5 — verification. Print the result; verify by hand-walking the facts. If the answer is wrong, the bug is either in the facts (data error), the rule (logic error), or the query (asking the wrong question). Walking through each in turn isolates the bug.

21.9 A worked example — course prerequisites

A small university catalog:

```
relation course(code, title)
relation prereq(course, requires)

course("CS101", "Intro to Programming")
course("CS201", "Data Structures")
course("CS301", "Algorithms")
course("CS401", "Compilers")
course("MATH101", "Calculus I")
course("MATH201", "Discrete Math")

prereq("CS201", "CS101")
prereq("CS201", "MATH101")
prereq("CS301", "CS201")
prereq("CS301", "MATH201")
prereq("CS401", "CS301")
prereq("MATH201", "MATH101")
```

Direct queries:

```
# What are the direct prereqs of CS401?
{R | prereq("CS401", R)}                  # {"CS301"}

# What courses require CS101 directly?
{C | prereq(C, "CS101")}                  # {"CS201"}

# All prereq pairs
{(C, R) | prereq(C, R)}
```

Transitive query — what does CS401 *eventually* require?

The classic Prolog/Datalog idiom — a base clause plus a recursive clause — works verbatim: clauses for the same head accumulate, and the engine iterates them to a fixpoint:

```
relation eventually_requires(course, requires)
eventually_requires(C, R) whenever prereq(C, R)
eventually_requires(C, R) whenever prereq(C, P) and eventually_requires(P, R)

{R | R <- eventually_requires("CS401", R)}
```

```
# {"CS101", "CS201", "CS301", "MATH101", "MATH201"}
```

When you'd rather see each depth on its own — while debugging, or when the maximum depth is known and small — explicit **multi-hop comprehension chains** remain a fine alternative:

```
hop1: {R | R <- prereq("CS401", R)}
hop2: {R | P <- prereq("CS401", P), R <- prereq(P, R)}
hop3: {R | P <- prereq("CS401", P), Q <- prereq(P, Q), R <- prereq(Q, R)}

all_eventual: hop1 ∪ hop2 ∪ hop3
println(all_eventual)
# {"CS301", "CS201", "CS101", "MATH101", "MATH201"}
```

Each hop is a self-contained query you can inspect on its own; the union (`∪`) combines the depth-1, depth-2, and depth-3 results.

Defeasible query — what's the *typical* term to take CS401?

```
relation nominal(course, term)      # catalog data: each course's nominal term
nominal("CS101", 1)
nominal("CS201", 2)
nominal("CS301", 3)
nominal("CS401", 4)

relation typical_term(course, term)

# Courses are typically taken in their nominal term
typically typical_term(C, T) whenever nominal(C, T)

# But students with credit for CS101 from high school skip ahead
cancel("typical_term", "CS101", 1)
typical_term("CS101", 0)    # they got credit before college
```

The defeasible rule gives the *baseline*. Cancellation + explicit fact handles the exception.

21.10 Why this style matters

Three pedagogical claims.

One — relational thinking transfers. Once students grasp relations + queries + rules, they have the conceptual model for SQL, for graph databases (Neo4j), for knowledge graphs (RDF / SPARQL), for inference engines (CLIPS, Drools), and for modern AI systems' fact stores. The vocabulary is common; only the syntax varies.

Two — declarative style is debuggable in a different way. A buggy function fails *during computation*. A buggy logic program produces *wrong query results*. The debugging tools differ: instead of `print` statements through a call stack, you ask "which facts contradict?", "which rule fired unexpectedly?", "what proof leads to this conclusion?" Axioma's `why` and `proof` operators are the debugging analog:

```
why grandparent("Alice", "John")
# Explanation for: grandparent(Alice, John)
# This is a theorem derived by strict inference from:
#   - parent("Alice", "Mary")  [datum]
#   - parent("Mary", "John")  [datum]
```

The reasoning is **legible**. You can read why a fact was derived, fact by fact.

Three — defeasibility is the right model for real knowledge. Most things we know about the world are typical, not absolute. Birds fly (mostly). Mammals give birth (mostly). Coffee shops are open in the morning (mostly). A programming language that forces every assertion to be *either* absolute *or* not-asserted-at-all loses an enormous chunk of how humans actually reason. Defeasible rules + cancellation capture exactly the right shape.

21.11 The `#language axioma/knowledge-core` fence

Chapter 21 so far is **host** logic programming: `relation`, bare facts, `whenever`, defeasible `typically`, `retract`, comprehensions, and ordinary `func` sitting next to the rules. That is `axioma/all` (this book's default once you leave beginner).

`#language axioma/knowledge-core` is a **monotonic proof-core**. It is a language, not a lens. The file may contain concepts, entities, frame slots, `assert/axiom`, strict Horn rules, and `derive/prove/query`. It may **not** contain general computation. The point is the same honesty move as Chapter 54: if a skip would be the wrong answer, shrink the language until the bad program cannot run.

A family-tree kernel that *is* in the core — notice there is no `relation` line. The schema is the asserted atoms plus the rule:

```
#language axioma/knowledge-core
assert/axiom parent("Mary", "John")
assert/axiom parent("Alice", "Mary")
grandparent(G, A) whenever parent(G, P) and parent(P, A)
thesis: derive/result grandparent("Alice", "John")
println(thesis.status)       # derived
println(thesis.grounding)    # theorem
```

`whenever` is still the chapter's primary rule spelling. `derive/result` returns a structured answer (`status`, `grounding`, `justification`) instead of a silent "maybe it fired."

What the fence refuses, with the diagnostic you will actually see:

You write	The core says
`func(x) [x]`	`function literal is not available` — facts and rules, not computation
`1 + 2`	`operator + is not available` — use relation calls, `is`, frames, `and`

You write	The core says
`retract parent("Mary", "John")`	`retract is not available` — retraction is non-monotonic
`typically flies(X) whenever bird(X)`	`defeasible rule is not available` — completeness is targeted at the strict Horn fragment; use `<==` / `whenever`
a `while` loop	`loop is not available`

`relation parent(...)` is a host schema form. The core does not take it: write `assert/axiom parent(...)` and let the atom be the schema. (The current diagnostic names an internal node type — treat it as "this declaration is not in the core," and use `assert/axiom`.)

Two nearby names, three jobs:

Name	What it actually does
`axioma/all`	this chapter: `relation`, defeasible rules, `retract`, `func`
`axioma/knowledge`	**mild** — some helper names are refused (`insert` → write `assert/axiom`); `retract` still runs
`axioma/knowledge-core`	**AST allowlist** — the proof-core in this section

`axioma/knowledge` is not a silent alias of the core. If you need the fence, write `knowledge-core`.

Stay on `axioma/all` for the rest of this chapter's exercises (they use `relation` and, in 21.4, defeasible rules). Step onto the core when the *point* of the file is that it is a Horn theory.

A failed search is not proof that nothing exists

A recursive relation query can exceed an implementation limit before it finishes. Returning an empty set in that situation would falsely report that no answer exists. Axioma instead returns `IncompleteReasoningError`, which is also an `Error`.

```
relation reached(n)
assert reached(0)
reached(N) whenever reached(M) and N == M + 1 and N <= 100
query_result: try({N | N <- reached(N)})
println(query_result is IncompleteReasoningError)  # true on this revision
println(query_result is Error)                     # true
```

This deliberately exceeds the current eager derivation-pass budget on its first query. It demonstrates a resource boundary, not an infinite mathematical set. Some positive facts may remain cached; a later query can make further progress. Negation must not interpret an incomplete positive search as proof of absence. The exact budgets are implementation details: handle the error or redesign the query rather than relying on a specific limit as part of your algorithm.

The same distinction matters in applications: **no answer found after a completed search**, **search incomplete**, and **claim disproved** are three different outcomes. A checked theorem has a further trust boundary; Appendix E explains why its certified conclusion cannot be changed by mutating an ordinary input record.

21.12 Exercises

21.1 — Build a family-tree fact base

Declare the relations `parent(child, parent)` and `married(a, b)`. Populate with at least six parent facts and three married facts that describe a family of your choosing (real or fictional). Then write four queries:

1. Get the set of all parents.
2. Get the set of all children of a specific person (using the generator form `{C | C <- parent(C, "name")}`).
3. Get the set of all married couples as `(X, Y)` tuples using `{(X, Y) | married(X, Y)}`.
4. Define a `sibling_raw(x, y)` *rule* that holds when `x` and `y` share a parent. Add the rule with `<=`, then query `{p | p <- {(X, Y) | sibling_raw(X, Y)}, p[1] != p[2]}` to get just the proper-sibling pairs.

21.2 — Grandparent rule + query

Using your family-tree base from 21.1, define the `grandparent(grandchild, gp)` relation as a strict rule. Then query:

1. All grandparents (any direction).
2. The grandparents of one specific grandchild.
3. The grandchildren of one specific grandparent.

21.3 — Course prerequisite chain

Use the catalog from §21.9. Then write queries for:

1. The direct prereqs of "CS401" (one-hop).
2. *All* eventual prereqs of "CS401" — write `hop1`, `hop2`, `hop3` as in §21.9 and union them with `u`.
3. The courses that have NO prereqs — start with `{C | C <- course(C, _)}`, then filter out any `C` that appears in the left position of `prereq`. Use the comprehension: `{c | c <- all_courses, not (c in has_prereq)}`.

21.4 — Defeasible bird-flies, with exception

Declare:

```
relation bird(x)
relation penguin(x)
relation ostrich(x)
```

```
relation flies(x)
```

Add facts for at least five birds, one penguin, and one ostrich. Write the defeasible rule "birds fly." Then *cancel* the conclusions for the penguin and the ostrich. Query the remaining flyers using the `@conjecture` tag and verify the right ones survive.

21.5 — Two-hop friend recommendation

Declare `friend(a, b)` and add at least eight friendship facts (use names like "alice", "bob", "carol", etc.). Friendships are symmetric — if you write `friend("alice", "bob")`, *also* write `friend("bob", "alice")` (Axioma doesn't infer symmetry automatically).

Then write a query for **friend-of-friend recommendations**: all `X` such that some person `P` is friends with both "alice" and `X`, where `X` is not already alice's friend and `X != "alice"`. Print the list of recommended new friends for "alice."

21.6 (open) — Genealogy reasoner

Build a small genealogical knowledge base for a fictional family of your choosing — *at least* twelve people, *at least* three generations. Include relations for `parent`, `married`, `gender`, `birth_year`.

Then implement (as rules + queries) at least *four* derived relationships of your choice. Suggestions:

- `sibling(X, Y)` — same parent
- `aunt_or_uncle(A, N)` — sibling of a parent
- `cousin(X, Y)` — children of siblings
- `descendant_of(X, A)` — recursive parent chain
- `ancestor_of(X, A)` — the inverse
- `step_parent(X, Sp)` — married to one's parent without being a biological parent

Write at least one query for each derived relation and verify by hand against the fact base.

21.7 — A Horn file on the proof-core

Rewrite the grandparent kernel from §21.5 as `#language axioma/knowledge-core`: no `relation` line, `assert/axiom` for the two parent facts, a `whenever` rule, `derive/result` for `grandparent("Alice", "John")`. Print `status` and `grounding`. Then, in a *second* file (or a commented block you do not run), try `retract` or `typically` and record the diagnostic.

21.13 Reflection — when to reach for logic programming

Three signs the relational style is the right fit:

Sign one — your data is "things in relationships." Genealogies, social networks, organization charts, knowledge graphs, dependency chains — anything where the *structure between* the items matters more than the items themselves.

Sign two — you find yourself writing many similar functions. Five different "find all X" functions that all walk the same graph differently are a hint that you've implemented a relational query as imperative code. A logic program lets the structure *be* the program.

Sign three — exceptions matter. If you're constantly catching edge cases ("but not the penguins…"), defeasible rules give you a clean separation: the rule is the default, the cancellations are the data exceptions, and both are explicit.

Three signs to stay with functions:

- The data is a *single value* per input, not a network.
- You need fine control over *evaluation order*.
- The structure of the algorithm matters more than the structure of the data.

A real program uses both. Functions for computations, relations for relationships. The two compose well — a function can call a query, a query can use a function as a filter condition. Axioma supports the full mix.

The next chapter steps up one final level — implementing a small Axioma interpreter *in Axioma itself*, the meta-circular evaluator.

> *"In logic programming, the program is the specification and the proof is the execution."* — Robert Kowalski, 1985.

Solutions to selected exercises: Chapter 21 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch21_solutions.md.)*

21.14 Optional: choose complete relations or ordered search

The ordinary relations taught above keep their existing KB behavior. For an explicit contract, use `relation/datalog` for safe finite relational analysis and `relation/prolog` for ordered search over possibly open terms. The query block names the contract too. A comma or semicolon never switches engines.

A Datalog query returns a complete set. Heads, filters and negative goals must be range restricted by positive relational goals in every alternative. Negative dependencies must be stratified; terms cannot grow through recursive heads. The evaluator refuses unsafe rules and reports resource limits as errors.

```
relation/datalog edge(x, y)
relation/datalog path(x, y)
assert edge(1, 2)
assert edge(2, 3)
path(X, Y) whenever edge(X, Y) or (edge(X, Z) and path(Z, Y))
reach: [logic/datalog | {Y | Y <- path(1, Y)}]
println(reach.result)
println(reach.complete)
```

This prints `{2, 3}` and `true`. Choose this mode when you need the entire finite answer relation, such as all prerequisite pairs.

Prolog mode searches clauses and goals in order. Fresh logical variables and structural unification let the same append relation find list splits:

```
relation/prolog append(a, b, c)
append([], Ys, Ys) :- true
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs)
splits: [logic/prolog | (A, B) <- append(A, B, [1, 2])]
println(take_solutions(9, splits.result))
println(splits.complete)
```

This prints `[(`[], `[1, 2]), (`[1], `[2]), (`[1, 2], `[])]` and `true`. Asking for nine answers exhausts this three-answer search. Asking for only one would leave `.complete` false. Choose this mode when order, the first answer, duplicate proofs, or open structures are part of the task. Use `next_solution` to pull one answer and `close_solutions` to release a search you no longer need. A resource limit is `IncompleteReasoningError`, never proof that no answer exists.

Within explicit logical goals, `,` aliases `and` and `;` aliases `or`. Conjunction binds more tightly. Argument separators and host statements retain their existing grammar. `cut`, `prune`, and standalone `!` succeed and discard alternatives created since entering the current predicate; caller choices and goals to the right remain. `once(goal)` keeps one enclosed solution. `fail` rejects the current branch. `not goal` tests finite failure, discarding the enclosed bindings. Cut-fail can suppress a fallback clause, but with unbound variables it does not enumerate the complement of a relation.

The optional `[logic/prolog/tabled | ...]` strategy completes variant tables for positive recursion before yielding answers. It can handle finite cyclic and left-recursive searches, but refuses cut, once and procedural negation. Infinite terms or answers can still reach a resource limit. Datalog cannot call Prolog; Prolog can read a completed Datalog snapshot. None of these query results is automatically asserted into the KB. These modes run in the evaluator; the VM refuses them.

Exercise 21.8 — An observable first choice. Declare two ordered clauses of a Prolog relation. Pull one answer and inspect completeness. Then compare exhausting the remaining stream, closing it early, and placing `prune` in the first clause. Explain why cancellation is not exhaustion and why a Datalog relation cannot contain this control. See the chapter solutions.

The full contract, including tagged tuple terms and reusable residual variables, is in Explicit Datalog and Prolog modes.

21.15 Optional: explanations are candidates

Deduction establishes consequences from premises. Abduction asks which premises could explain an observation. Axioma's `abduce` examines matching rule heads and returns one-step candidate bodies; it does not prove those bodies or recursively find a minimal explanation.

```
relation reported(person)
relation saw(person)
reported("alice") whenever saw("alice")
println(abduce("reported", "bob"))
```

```
println(abduce("reported", "alice"))
```

This prints `[]`, then `[("saw(\"alice\")", "strict")]`. A matching predicate name is insufficient: the head's literal must match too. Repeated logical variables must unify, and supported nested list/array and tagged-tuple terms are matched structurally. The result retains the complete body, including guards and other premises. Calls in head arguments are not executed to manufacture a match. Unsupported terms report an error.

A candidate such as `saw("alice")` remains a hypothesis until independently checked. Similarly, `understand` reports possible consequences; one matching premise does not establish every other premise. Both cognitive operations require the evaluator. See the reasoning candidate contract for supported terms and limits.

Exercise 21.9 — A plausible cause is not proof. If a report could follow from seeing an event or hearing a rumour, what should the two abduced candidates mean? Explain why accepting either candidate as an asserted fact would be an invalid learning rule. What additional observations could discriminate them? See the chapter solutions.

21.16 Optional: keep statement data typed

A symbolic database must distinguish a number from text that spells the number. Bracket facts now retain native typed elements instead of using printed text as identity. Bare fact words are Words; a quoted phrase is one String.

```
assert [mary has age 25]
assert [mary has age "25"]
assert ["new york" is city]
ages: query/result [?who has age ?age]
println(len(ages.matches))                  # 2
places: query/result [?place is city]
println(places.matches[1].bindings.place)  # new york
retract [mary has age "25"]
remaining: query/result [?who has age ?age]
println(remaining.matches[1].bindings.age) # 25
```

Plain `query` prints matches; `query/result` returns a Dictionary for programs. Each typed row carries native `bindings`, `elements`, identity, display text, grounding and an Epistem. Its deep form is a fresh snapshot. Changing a returned Array does not change the fact. Comma-separated Arrays can hold nested data:

```
assert ['entry, "a", [1, 2]]
entry: query/result ['entry, ?label, [?first, 2]]
println(entry.matches[1].bindings.first)   # 1
assert [alice likes bob]
assert [bob likes bob]
reflexive: query/result [?x likes ?x]
```

```
println(len(reflexive.matches))          # 1
```

Unlike ordinary pattern captures in chapter 5, repeated query `?x` must match the same typed value.

Each `?_` ignores its position independently. Word/String and numeric types remain distinct for statement identity. Ordinary `[mary has age]` expressions still produce Arrays of Strings; knowledge operations supply the typed fact context.

Inspect `complete` and `status` before relying on an absence of matches. If any visible record still has only legacy text, `complete` is false and status is `"legacy_ambiguous"`; old text matches appear separately in `legacy_matches`. Their original types and phrase boundaries cannot be recovered from text alone.

The evaluator supports session save/reload of typed facts and retractions. Scripts are transient by default. `/persist` means external KB storage and currently refuses for typed bracket facts until a compatible adapter exists. The VM refuses these statement operations. The contract reference lists supported values, structural limits and provenance serialization limits.

Optional administration boundary. The evaluator's explicit `--canonical-kb-session` mode persists supported complete concept and relation *definitions* into a prepared sandbox or explicitly enabled shared KB. It requires an identity/version manifest and reloads exact UUIDs. A shared session attached to the same ordinary KB retains its existing fact operations; this does not add typed bracket fact persistence or make these operations available in the VM. Each definition declaration commits separately. See the Manual's "Ordinary persistence in an explicit canonical session" for setup, receipts and restart recovery. The examples here continue to run without that external administration mode.

Exercise 21.10 — Preserve meaning across a restart. Explain why the two age facts above must have separate identities. Retract the String-valued fact and query the remaining value. Explain why a legacy text-only result cannot certify that no numeric fact exists, and why a memory system should inspect completeness before turning a query miss into a learned conclusion.

21.17 Optional: production rules choose actions

A logical clause describes a consequence of its premises. A production rule selects work to perform when its condition matches the current facts. Its definition retains code; its action runs only when `deduce` selects an activation. An activation is one rule with one set of captured values and supporting facts.

```
assert [ada waits]
assert [bob waits]
assert [ada is urgent]
served_order: []
serve: func(person) [
    push(served_order, person)
    retract [person, 'waits]
    assert [person, 'served]
]
rule regular salience 1: [?person waits] implies serve(person)
rule urgent salience 10: [?person waits] and [?person is urgent] implies serve(person)
println(len(served_order))              # 0: definitions are inert
```

```
deduce
println(served_order)                    # ['ada, 'bob]
println(_last_deduce_trace[1]["rule"])   # urgent
println(_last_deduce_complete)           # true
remaining: query/result [?person waits]
println(len(remaining.matches))          # 0
```

Higher salience gives urgent work priority. Ada's action removes her waiting fact, so the regular activation for Ada disappears. Bob still receives service. The engine rechecks the agenda after each action; it does not permanently mark an entire rule as finished after its first binding.

`and` joins patterns left to right; guards using captured values follow their generators. Guards must return Boolean and should have no side effects because they can run repeatedly. `or` unions alternatives. The typed matching rules from §21.16 still apply. Captures stay local, while assertions and retractions performed by an action persist in the run's environment.

Each unchanged activation fires once per run. `no_loop` additionally suppresses activations newly enabled by that same rule's action, while allowing its other initial matches. Calling `deduce` again begins a fresh run; removing completed work is what keeps the example from serving someone twice across runs.

Check completion and errors before treating a run as learning evidence. The production pass allows 100 successful firings and has separate matching limits. An incomplete run is not evidence that no work exists. An action error retains earlier effects; the engine does not roll them back. Its trace describes what executed, not a logical proof that the selected policy was correct.

Canonical Horn derivation follows the production pass; the two engines do not jointly iterate to a fixpoint. Production source must be reloaded after restart, and the VM refuses these rules. See the production reference.

Exercise 21.11 — Priority and consumed work. Add an urgent waiting fact for Carol. Confirm the urgent rule can serve two different people. Then call `deduce` again and explain why nobody is served twice. What would change if the action kept the waiting facts? Would `no_loop` make the work permanently complete across separate runs? See the chapter solutions.

Chapter 22 · The Meta-Circular Evaluator

> **What this chapter is.** A *meta-circular evaluator* is an interpreter for a language, **written in that same language**. We're going to build a tiny Axioma interpreter *in Axioma itself*. The tradition runs from Lisp's `apply` (1960) through SICP's evaluator (1985) through every modern language that exposes its parser as a library. Axioma exposes `parse`, `ast_eval`, `headof`, `argsof`, `fullform`, and `quote` — everything needed to read code, inspect it as data, and evaluate it. The pedagogical bet: **once you've written an interpreter for a language, you understand that language at a level no other exercise can teach.**

This is the capstone chapter. It ties together Chapter 8 (trees as Concepts), Chapter 11 (lambdas as values), Chapter 12 (local environments), and the design recipe writ large.

For a compact map of the Go interpreter that runs this chapter, see Appendix G — How the Axioma Interpreter Works. The evaluator we build here handles a teaching subset and uses Axioma's existing parsing and reflection tools; it is not a complete self-hosted Axioma.

22.1 The big idea — code is data

When you write `2 + 3` in Axioma, two things are happening:

1. The **lexer + parser** convert that text into an *AST* (abstract syntax tree) — a tree of nodes that records *what kind* of expression this is, *what operator* it uses, *what arguments* it takes.
2. The **evaluator** walks the AST and produces a value.

For most of this book you've only seen step (2) — you write code, it runs. This chapter pulls back the curtain on step (1). Axioma exposes the AST as a first-class value:

```
ast1: parse("2 + 3")
println(ast1)                  # <AST: (2 + 3)>
println(ast_type(ast1))        # InfixExpression
println(headof(ast1))          # +
println(argsof(ast1))          # ["2", "3"]
```

`parse` converts a string of code into an AST. `ast_type` tells you *what kind* of node. `headof` gives the operator/function. `argsof` gives the operands.

You can also *capture* an AST without parsing — using `quote`:

```
ast2: quote(2 + 3)        # same thing, no string
println(fullform(ast2))        # +(2, 3)
```

`fullform` renders the AST in Mathematica-style prefix form, which makes the *structure* obvious: an `InfixExpression` is really a function call `+(2, 3)`.

And you can *run* an AST:

```
println(ast_eval(parse("2 + 3")))   # 5
```

That round-trip — *parse a string into an AST, then evaluate the AST* — is the heart of every interpreter ever written. You've just done it in two lines of Axioma.

22.2 What our small evaluator handles

We'll build an evaluator for a *subset* of Axioma:

- **Numeric literals**: `42`, `3.14`
- **Variables**: `x`, `y`
- **Arithmetic**: `+`, `-`, `*`, `/`
- **Conditionals**: `if cond then a else b`
- **Lambdas + applications**: `lambda x => x * 2`, `f(3)`

That's enough to express factorials, list-walks (in a recursive language with pairs), and most of what we've done in the first half of this book.

The full Axioma evaluator handles concepts, comprehensions, relations, mutation, MVL, and a hundred other things. Our subset is the **lambda-calculus + arithmetic** core. Real Axioma is built on the same skeleton, with more cases in the dispatch.

22.3 Representing the AST as Concepts

We'll use Chapter 8's idiom — model AST nodes as Concepts:

```
concept NumNode
NumNode has value: 0

concept VarNode

VarNode has name: ""

concept OpNode

OpNode has op: ""
OpNode has left: 0
OpNode has right: 0

concept IfNode

IfNode has condition: 0
```

```
IfNode has then_branch: 0
IfNode has else_branch: 0

concept LambdaNode

LambdaNode has param: ""
LambdaNode has body: 0

concept AppNode

AppNode has callee: 0
AppNode has arg: 0
```

Six concepts. Each one captures *one kind* of AST node. An arithmetic expression `(2 + 3) * 4` is a tree:

```
        OpNode("*")
        /         \
   OpNode("+")     NumNode(4)
   /         \
NumNode(2)  NumNode(3)
```

In Axioma:

```
n2: a NumNode { value: 2 }
n3: a NumNode { value: 3 }
n4: a NumNode { value: 4 }
add: an OpNode { op: "+", left: n2, right: n3 }
mul: an OpNode { op: "*", left: add, right: n4 }
```

That AST corresponds to `(2 + 3) * 4`. Reading the tree from the root tells you the structure: a multiplication of an addition with a number.

22.4 The evaluator — first cut, arithmetic only

The simplest evaluator: just numeric expressions, no variables, no functions.

```
eval_expr: func(node) [
  if node is NumNode then node.value
  else if node is OpNode then [
    l: eval_expr(node.left)
    r: eval_expr(node.right)
    if node.op == "+" then l + r
    else if node.op == "-" then l - r
    else if node.op == "*" then l * r
    else if node.op == "/" then l / r
    else 0
  ]
  else 0
```

```
]

println(eval_expr(mul))   # (2 + 3) * 4 = 20
```

Read the dispatch carefully:

- If the node is a `NumNode`, return its value.
- If it's an `OpNode`, **recursively** evaluate both children, then apply the operator.
- Otherwise, return 0 (we'll improve the fallback below).

That's the **structural recursion** pattern from Chapter 7 applied to a tree. Each Concept type gets one case; the recursion mirrors the data shape.

22.5 Environments — making variables work

Now add variables. A variable looks up its value in an **environment** — a list of name/value pairs.

```
# Environment: an array of (name, value) tuples
empty_env: []

# Look up: walk the list, return value of first match
env_lookup: func(ev, nm) [
  if len(ev) == 0 then 0
  else if ev[1][1] == nm then ev[1][2]
  else env_lookup(ev[2..len(ev)], nm)
]

# Extend: cons a new binding onto the front
env_extend: func(ev, nm, item_value) [
  [(nm, item_value)] + ev
]

# Build an env: x = 10, y = 20
e0: empty_env
e1: env_extend(e0, "x", 10)
e2: env_extend(e1, "y", 20)

println(env_lookup(e2, "x"))   # 10
println(env_lookup(e2, "y"))   # 20
println(env_lookup(e2, "z"))   # 0 (default — not found)
```

The evaluator threads the environment through every recursive call:

```
eval_expr: func(node, ev) [
  if node is NumNode then node.value
  else if node is VarNode then env_lookup(ev, node.name)
  else if node is OpNode then [
    l: eval_expr(node.left, ev)
```

```
    r: eval_expr(node.right, ev)
    if node.op == "+" then l + r
    else if node.op == "-" then l - r
    else if node.op == "*" then l * r
    else if node.op == "/" then l / r
    else 0
  ]
  else 0
]

# x + 5, with env { x → 10 }
nx: a VarNode { name: "x" }
n5: a NumNode { value: 5 }
expr: an OpNode { op: "+", left: nx, right: n5 }
println(eval_expr(expr, e2))   # 10 + 5 = 15
```

The variable `x` gets *looked up* at evaluation time, in the environment passed by the caller. That's how scoping works under the hood — every function in a real evaluator threads an environment through every recursive call.

22.6 Conditionals — first non-trivial dispatch

`if cond then a else b` evaluates `cond` first, then *only one of* `a` or `b` based on the result:

```
eval_expr: func(node, ev) [
  if node is NumNode then node.value
  else if node is VarNode then env_lookup(ev, node.name)
  else if node is OpNode then [
    l: eval_expr(node.left, ev)
    r: eval_expr(node.right, ev)
    if node.op == "+" then l + r
    else if node.op == "-" then l - r
    else if node.op == "*" then l * r
    else if node.op == "/" then l / r
    else if node.op == "<" then if l < r then 1 else 0
    else if node.op == "==" then if l == r then 1 else 0
    else 0
  ]
  else if node is IfNode then [
    c: eval_expr(node.condition, ev)
    if c != 0 then eval_expr(node.then_branch, ev)
    else eval_expr(node.else_branch, ev)
  ]
  else 0
]
```

Notice we **don't** evaluate both branches up front — the order matters. If we did `a: eval_expr(node.then_branch, ev); b: eval_expr(node.else_branch, ev)` and then chose between them, we'd

evaluate dead code. In a pure subset that doesn't matter; in a real language with side effects, it absolutely does.

We also added < and == as ops, using `0` and `1` as stand-ins for booleans (this is how C represents booleans; classical Lisp evaluators do the same).

Test:

```
# if (x < 10) then 100 else 200, with x = 5
cmp: an OpNode { op: "<", left: a VarNode { name: "x" }, right: a NumNode { value: 10 }
 ↪  }
then_b: a NumNode { value: 100 }
else_b: a NumNode { value: 200 }
ifn: an IfNode { condition: cmp, then_branch: then_b, else_branch: else_b }
println(eval_expr(ifn, e1))   # x=10, so x<10 is false → 200
```

22.7 Lambdas and applications — the hard case

A **lambda** is a function value. In an interpreter, it's represented as a *closure* — the lambda's body AST plus the environment that was active when the lambda was created:

```
concept ClosureValue
ClosureValue has param: ""
ClosureValue has body: 0
ClosureValue has captured_env: 0

eval_expr: func(node, ev) [
  if node is NumNode then node.value
  else if node is VarNode then env_lookup(ev, node.name)
  else if node is OpNode then [
    l: eval_expr(node.left, ev)
    r: eval_expr(node.right, ev)
    if node.op == "+" then l + r
    else if node.op == "-" then l - r
    else if node.op == "*" then l * r
    else if node.op == "/" then l / r
    else 0
  ]
  else if node is IfNode then [
    c: eval_expr(node.condition, ev)
    if c != 0 then eval_expr(node.then_branch, ev)
    else eval_expr(node.else_branch, ev)
  ]
  else if node is LambdaNode then
    a ClosureValue { param: node.param, body: node.body, captured_env: ev }
  else if node is AppNode then [
    fval: eval_expr(node.callee, ev)
    aval: eval_expr(node.arg, ev)
```

```
    if fval is ClosureValue then [
      new_env: env_extend(fval.captured_env, fval.param, aval)
      eval_expr(fval.body, new_env)
    ] else 0
  ]
  else 0
]
```

Two new cases:

- **`LambdaNode`** — when we evaluate a lambda, we don't *run* its body. We **package up** the body, the param name, and the *current environment* into a `ClosureValue`.
- **`AppNode`** — when we evaluate a function call `(fn arg)`, we evaluate `fn` first (yielding a closure), then the arg, then *extend the closure's captured env* with the param bound to the arg value, and run the body in that extended env.

The line *"extend the closure's captured env"* is where **lexical scoping** comes from. A lambda remembers the env it was *defined* in, not the env in which it's *called*. That's the secret behind closures.

Test — the classic "make adder" pattern:

```
# (lambda x. (lambda y. x + y)) 3   →  closure that adds 3 to its arg
# Apply that to 4 →  3 + 4 = 7

# Build inner: x + y (where x and y are variables looked up at runtime)
body_inner: an OpNode {
  op: "+",
  left: a VarNode { name: "x" },
  right: a VarNode { name: "y" }
}
# Outer lambda: lambda x. (lambda y. x + y)
lam_inner: a LambdaNode { param: "y", body: body_inner }
lam_outer: a LambdaNode { param: "x", body: lam_inner }

# Apply outer to 3: yields closure capturing x = 3
app1: an AppNode { callee: lam_outer, arg: a NumNode { value: 3 } }
# Apply that closure to 4: yields 7
app2: an AppNode { callee: app1, arg: a NumNode { value: 4 } }

println(eval_expr(app2, empty_env))   # 7
```

That's a curried `make_adder` in our tiny interpreter. The inner lambda captures `x = 3` at the moment the outer lambda is applied; when we then apply the inner lambda to 4, it sees both `x` (from its captured env) and `y` (just bound to 4) — and computes 7.

22.8 The recipe — applied to interpreters

The Design Recipe scales:

Stage 1 — data definition. The AST: one Concept per node type. This is the *grammar* of the language you're interpreting.

Stage 2 — signature. `eval_expr: func(node, ev) -> Value` — takes an AST node + environment, returns a value.

Stage 3 — examples. For each AST node type, an example: a small expression whose value you know, with the environment if needed.

Stage 4 — template. Match the data: one `else if node is NodeType` per Concept. Same shape as Chapter 8's tree walks, but with more branches.

Stage 5 — body. For each case, do the work:

- Literal node → return the literal value.
- Compound node → recursively evaluate children, then combine.
- Special case (`If`, `Lambda`, `App`) → custom dispatch.

Stage 6 — tests. A test per node type plus a few compound tests that exercise multiple node types together.

The interpreter is just **a function over a tree-shaped data type**. Same recipe. Bigger data definition.

22.9 Macros — transforming code before evaluation

A macro receives argument syntax and returns an AST. Its expansion happens during **evaluator preparation**, before ordinary statements run; it is not ordinary function application and not a parser rewrite.

```
macro unless(test, body) quasiquote(
  if not unquote(test) then unquote(body) else none
)
unless(false, println("runs"))          # runs
unless(true, println("does not run"))   # no output
println(macroexpand(unless(false, 7)) is AST)  # true
```

`quasiquote` builds a syntax template; `unquote` inserts argument syntax. The macro body must return an AST. Wrapping a macro body in an ordinary array literal changes its result type and is not an interchangeable function-body spelling. By contrast, a block *inside* `quasiquote` is part of the code being generated.

Hygiene: temporary names belong to the expansion

A macro should not overwrite an unrelated variable in the caller. Axioma renames template-introduced bindings for each expansion.

```
macro twice(expression) quasiquote([temporary: unquote(expression); temporary +
↪  temporary])
temporary: 100
println(twice(5))               # 10
```

```
println(temporary)              # 100
println(twice(twice(3)))        # 12
```

Here the input expression is evaluated once, then its value is reused. Spliced argument names retain the caller's meaning. Free names written in the template resolve in the macro's definition context. This makes three different origins visible: caller argument, definition helper, and expansion-local temporary.

`macroexpand` recursively exposes the prepared code, including fresh names. A freshly parsed AST has not yet undergone that preparation. Do not expect a macro body to read an earlier ordinary runtime binding from the same file: expansion precedes execution of those statements. Macros are currently an interpreter feature; the VM explicitly refuses unsupported macro compilation. Start with functions unless syntax itself must be transformed.

22.10 Exercises

22.1 — Extend the evaluator with subtraction

Take the evaluator from §22.6 (the version without lambdas) and add support for `node.op == "-"` if it isn't already there. Then build the AST for `(10 - 3) * 2` and verify it evaluates to 14.

22.2 — Add a local-binding form

Add a new Concept `LetNode` with slots `name`, `value`, and `body`. Semantically, the object-language form `bind x = e1 in e2` should evaluate `e1` in the current env, then evaluate `e2` in the env extended with `x` bound to that value.

Extend `eval_expr` to handle `LetNode`. Then build the AST for nested local bindings equivalent to `bind x = 3 in bind y = 4 in x + y` and verify it evaluates to 7.

22.3 — Pretty-print an AST

Write `pretty(node)` that converts an AST back into a human-readable string. Cases:

- `NumNode { value: n }` → `str(n)`
- `VarNode { name: nm }` → `nm`
- `OpNode { op: o, left: l, right: r }` → `"(" + pretty(l) + " " + o + " " + pretty(r) + ")"`
- `IfNode { cond, then_branch, else_branch }` → `"if " + pretty(cond) + " then " + pretty(then_branch) + " else " + pretty(else_branch)`
- (and so on for any other node types you've added)

Then for the AST `(2 + 3) * 4` you built in §22.3, `pretty(mul)` should produce `"((2 + 3) * 4)"`.

22.4 — Use Axioma's `parse` + `ast_eval` to round-trip

Write a function `run(s)` that takes a string of code, parses it with Axioma's built-in `parse`, evaluates it with `ast_eval`, and returns the value.

```
println(run("2 + 3"))            # 5
println(run("(7 - 3) * 5"))      # 20
println(run("if 1 then 100 else 200"))  # 100
```

This *isn't* using our hand-written evaluator — it's using the *real* Axioma evaluator, via the AST-as-data interface. Compare and contrast with the hand-written one.

22.5 — Inspect an AST with `headof` and `argsof`

Write `describe(s)` that takes a string of code, parses it, and prints the head and args of the parsed AST. Examples:

```
describe("2 + 3")
# head: +
# args: ["2", "3"]

describe("sqrt(16)")
# head: sqrt
# args: ["16"]

describe("if true then 1 else 2")
# head: If
# args: ["true", "1", "2"]
```

22.6 (open) — Build a tiny language

Pick a *small* domain — Boolean expressions, a tiny stack machine, a four-function calculator with memory, anything you want — and design:

1. The AST (one Concept per node type).
2. The evaluator (one case in `eval_expr` per Concept).
3. Three test programs in your language.
4. A pretty-printer (from Ex 22.3) for it.

Document your design choices. There's no single right answer — the *point* is that you can design your own language now, because you understand what an interpreter does.

22.11 Reflection — why this chapter is the capstone

Three reasons.

One — you understand evaluation. Once you've written `eval_expr` for a subset of a language, you *know* what "evaluation" means. Not in the textbook sense, in the operational sense. You know what an environment is. You know what a closure is. You know why lexical scoping isn't some arbitrary rule — it's the direct consequence of how the evaluator threads environments. The mystery is gone.

Two — you understand the design recipe at the largest scale. The recipe says: *data shape determines program shape*. An interpreter is the case where the *data is itself a program*. The recipe still applies. The

dispatch over the AST is structural recursion, just bigger.

Three — you can extend any language. Most "languages" you'll use professionally — DSLs, query languages, config formats — turn out to be ASTs walked by an evaluator. JSON config files. SQL. CSS. They're all this same pattern. After this chapter, you don't see them as opaque tools; you see their *structure*. That's the gift this course was leading toward.

The textbook is over. The language is yours.

> *"The art of programming is the art of organizing complexity, of mastering the multitude and avoiding its bastard chaos as effectively as possible."* — Edsger Dijkstra, 1972. The meta-circular evaluator is one place where the multitude really does get mastered — in 50 lines of code that *is* the language it describes.

Solutions to selected exercises: Chapter 22 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch22_solutions.md.)*

Volume II — Data Structures and Beyond

Volume II picks up where Volume I leaves off. Volume I gave you the design-recipe vocabulary and Axioma's signature features. Volume II shows how to use them to build the standard CS2 data-structure curriculum — lists, stacks, queues, hash tables, trees, graphs — and (in chapters to come) extends past that curriculum into algorithm analysis, stateful systems, knowledge representation, and the neuro-symbolic frontier.

Chapter numbering continues from Volume I (Volume II starts at Chapter 23).

Chapter 23 · Linked Lists from Concepts

What this chapter is. Your first "self-built" data structure. Axioma's arrays are wonderful — fast, indexable, familiar — but they're not the only way to model a sequence. A **linked list** is a sequence built out of *links* — each element holds its value and a reference to the rest. This chapter shows you how to build one from Concepts, how the operations look in code, and *why* you'd ever do this when arrays already exist. The answer turns out to be more interesting than it sounds.

23.1 The motivation — what's wrong with arrays?

Nothing! Axioma arrays are excellent for most jobs. They give you O(1) random access, they're built in, and Chapter 6 already taught you how to use them.

But arrays have hidden tradeoffs that show up at the edges:

Operation	Array cost	Linked list cost
`arr[i]` (random access)	**O(1)**	O(n)
Append to end	O(1) amortized	O(n) (or O(1) with tail pointer)
Prepend to front	O(n) (shift everything)	**O(1)**
Insert in middle	O(n)	O(1) *if you have the link*
Length	O(1) (stored)	O(n) (walk to count)

Two operations stand out: **prepend** and **insert-in-middle** are O(n) for arrays and O(1) for linked lists. If your program spends most of its time *adding things to the front of a sequence* (a stack), or *splicing items into a sequence at known positions* (an editor's text buffer), a linked list is genuinely faster.

The deeper reason to learn linked lists: they're your first **recursively-defined data structure built by hand**. Every balanced tree, every graph, every parse tree in Chapter 22 is built the same way — Concepts pointing to other Concepts. Master linked lists, and the rest of the data-structures curriculum unfolds from the same template.

23.2 The shape

A linked list is *either*:

- the **empty list**, or
- a **cons cell** — a value plus a reference to the rest of the list.

That's the entire data definition. Two cases, recursive. Encoded as Concepts:

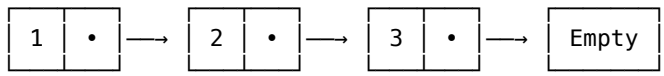
```
concept Empty                # the empty-list marker

concept Cons                 # a cell with a value + rest
Cons has head: 0                 # the value at this position
Cons has tail: 0                 # the rest of the list
```

The names `head` and `tail` are classical (Lisp 1960 → ML → Haskell → many others). Some traditions use `car` and `cdr` (from the original IBM-704 register names); we'll stick with the modern names.

A three-element list `[1, 2, 3]` represented this way:

```
Cons(1, Cons(2, Cons(3, Empty)))
```

Or drawn as boxes-and-arrows:



Each box is a Concept instance. The arrows are *references* — `tail` slot in one cell pointing at the next cell.

23.3 Building a list

```
concept Empty
concept Cons
Cons has head: 0
Cons has tail: 0

nil: an Empty {}

l3: a Cons { head: 3, tail: nil }
l2: a Cons { head: 2, tail: l3 }
l1: a Cons { head: 1, tail: l2 }

# l1 now represents the list [1, 2, 3]
println(l1.head)              # 1
println(l1.tail.head)         # 2
println(l1.tail.tail.head)    # 3
```

Building from the right (the tail) is the natural order — each new cell's `tail` slot needs the *already-built* rest of the list. A helper makes this less painful:

```
list_of: func(arr) [
  result: nil
  i: len(arr)
  while (i >= 1) [
    result = a Cons { head: arr[i], tail: result }
```

```
    i = i - 1
  ]
  result
]

my_list: list_of([1, 2, 3, 4, 5])
println(my_list.head)         # 1
```

We walk the array *right-to-left* so each `cons` happens after its tail is already built. That gives us the same left- to-right order in the resulting list as in the input array.

23.4 The recipe — structural recursion over lists

Every list operation follows the same template:

```
walk_template: func(lst) [
  if lst is Empty then BASE_CASE
  else COMBINE(lst.head, walk_template(lst.tail))
]
```

Two cases (Empty vs. Cons), one recursive call. Identical to Chapter 7's recursion-on-arrays template, except the recursive call is `fn(lst.tail)` instead of `fn(arr[2..len(arr)])`.

Length

```
length: func(lst) [
  if lst is Empty then 0
  else 1 + length(lst.tail)
]

println(length(my_list))        # 5
println(length(nil))            # 0
```

Each cell contributes 1; the empty list contributes 0. The total is the count of cells.

Sum

```
sum: func(lst) [
  if lst is Empty then 0
  else lst.head + sum(lst.tail)
]

println(sum(my_list))           # 1 + 2 + 3 + 4 + 5 = 15
```

Identity element 0 (for +); combine via addition.

Find-largest

```
largest: func(lst) [
  if lst is Empty then 0     # caller guarantees non-empty list, but 0 is safe
  else if lst.tail is Empty then lst.head
  else [
    r: largest(lst.tail)
    if lst.head > r then lst.head else r
  ]
]

println(largest(my_list))       # 5
```

Two structural sub-cases for non-empty: "exactly one element" and "more than one." The recursive call gives us the largest of the rest; we compare with the current head.

To-array

Converting a linked list back to an Axioma array is itself a walk over the list. The iterative form is the cleanest:

```
list_to_array: func(lst) [
  result: []
  cur: lst
  while (not (cur is Empty)) [
    result = result + [cur.head,]
    cur = cur.tail
  ]
  result
]

println(list_to_array(my_list))   # [1, 2, 3, 4, 5]
```

(The trailing comma is §6.6's singleton rule at work: in block positions a bare `[x]` reads as a one-statement block, so `[x,]` is the spelling that means "one-element array" everywhere. You may also meet the older `[] + [x]` wrapper in existing code — same effect.)

23.5 Three classical operations

Append

Concatenate two linked lists. The recursion is on the *first* list — when we reach Empty, we hand off the second list as-is.

```
append_lists: func(a, b) [
  if a is Empty then b
```

```
  else a Cons { head: a.head, tail: append_lists(a.tail, b) }
]

l_a: list_of([1, 2, 3])
l_b: list_of([4, 5])
l_c: append_lists(l_a, l_b)
println(list_to_array(l_c))     # [1, 2, 3, 4, 5]
```

Notice the cost. `append_lists` makes a new cons cell for *every* element of the first list (a recursive call per cell). That's O(length(a)). The second list is *shared* — its cells aren't copied. That's structural sharing in action.

Reverse

Reverse a list. The naive structural recursion (`reverse(tail) + [head]`) is $O(n^2)$ because each `+` re-walks the partial result. The accumulator version (Chapter 15 territory) is O(n):

```
reverse: func(lst) [
  reverse_helper(lst, nil)
]

reverse_helper: func(lst, acc) [
  if lst is Empty then acc
  else reverse_helper(lst.tail, a Cons { head: lst.head, tail: acc })
]

lr: reverse(my_list)
println(list_to_array(lr))      # [5, 4, 3, 2, 1]
```

The accumulator `acc` collects the reversed prefix; the recursive call moves one cell from the input list to the front of `acc`. After we've walked the entire input, `acc` holds the full reversal.

Map

Apply a function to every element. The structural recursion walks the list and rebuilds it with each element transformed:

```
map_list: func(f, lst) [
  if lst is Empty then nil
  else a Cons { head: f(lst.head), tail: map_list(f, lst.tail) }
]

doubled: map_list(func(x) [x * 2], my_list)
println(list_to_array(doubled)) # [2, 4, 6, 8, 10]
```

The function is passed as a value (Chapter 10's higher-order toolkit) and applied at each cell. Compare with array `map` — same conceptual shape, just operating on linked structure instead of indexed storage.

23.6 Structural sharing

Here's a subtle but important property of linked lists:

```
l: list_of([1, 2, 3, 4, 5])
r: reverse(l)
```

After these two lines, `l` *still points to its original cells*. `reverse` built brand-new cells for the reversed result; it didn't mutate the original. That's because *every function in this chapter builds new cells, never modifies existing ones*.

Arrays in some languages (Python, JavaScript) get mutated in place when you do `arr.reverse()`. That can cause subtle bugs — calling `.reverse()` on a list passed to your function *also* reverses the caller's list. Linked lists in this chapter never have that problem. They're **immutable** by construction.

This idea — *functions return new values rather than modifying old ones* — is the seed of **functional** data-structure design (Okasaki 1998, the source for the two-stack queue pattern you'll see in Chapter 24). Immutable linked lists are the simplest case.

23.7 When linked lists are the right tool

For 90% of CS2 work, **Axioma arrays are better than linked lists**. They're built in, indexable in O(1), and require no custom Concepts.

The cases where linked lists win:

1. **Heavy front-end mutation** — if your program prepends thousands of items and never indexes by position, a linked list's O(1) prepend beats an array's O(n) shift.
2. **Need for immutability + sharing** — when many "versions" of a list need to share most of their structure (think: undo history, persistent data structures), linked lists give you that almost for free.
3. **Pedagogical visibility** — to *understand* recursive data, you need to *build* a recursive data structure. Linked lists are the smallest possible such structure.
4. **A stepping stone to trees and graphs** — every later chapter builds Concept-based recursive structures. Mastering this template here makes those chapters easy.

Most production Axioma code uses arrays for sequences. Most **books about** Axioma still teach linked lists.

23.8 Exercises

23.1 — `nth`

Write `nth(lst, n)` — return the element at position `n` (1-indexed) in the linked list `lst`. Examples:

```
l: list_of([10, 20, 30, 40, 50])
println(nth(l, 1))   # 10
println(nth(l, 3))   # 30
```

```
println(nth(l, 5))   # 50
println(nth(l, 6))   # 0   (default for out-of-bounds; caller's responsibility)
```

Hint: recurse on `n` and `lst` *simultaneously* — at each step, either we've reached `n == 1` and return the head, or we recurse with `n - 1` on the tail.

What's the cost of `nth`? Compare with `arr[i]` on an Axioma array.

23.2 — `contains`

Write `list_contains(lst, x)` — return `true` if `x` appears anywhere in `lst`, `false` otherwise.

```
l: list_of([1, 2, 3, 4, 5])
println(list_contains(l, 3))   # true
println(list_contains(l, 99))  # false
println(list_contains(nil, 7)) # false
```

23.3 — `filter_list`

Write the linked-list analog of `filter` (Chapter 10). Given a predicate `pred` and a list `lst`, return a new list containing only the elements for which `pred(x)` returns true.

```
l: list_of([1, 2, 3, 4, 5, 6, 7, 8])
evens: filter_list(func(x) [x % 2 == 0], l)
println(list_to_array(evens))   # [2, 4, 6, 8]
```

Hint: structural recursion. For each cell, either include it (`cons head onto recursive result`) or skip it (`return recursive result`).

23.4 — `fold_list`

Write the linked-list analog of `reduce`. Given a binary function `op`, a seed value `acc`, and a list `lst`, return the result of folding `op` over the list:

```
l: list_of([1, 2, 3, 4, 5])
println(fold_list(func(a, b) [a + b], 0, l))    # 15
println(fold_list(func(a, b) [a * b], 1, l))    # 120
```

Hint: this is the most general list-walk function. `length`, `sum`, `largest`, and many others can be written as specializations of `fold_list`.

23.5 — `zip_lists`

Given two linked lists `a` and `b`, return a single linked list whose elements are the `(a[i], b[i])` tuples. When the shorter list runs out, stop.

```
l_a: list_of([1, 2, 3])
l_b: list_of(["a", "b", "c", "d"])
z: zip_lists(l_a, l_b)
println(list_to_array(z))    # [(1, "a"), (2, "b"), (3, "c")]
```

Hint: recurse on *both* lists. The base case is "either list is Empty."

23.6 (open) — Persistent insert

Write `insert_at(lst, n, x)` — return a *new* linked list that is `lst` with `x` inserted at position `n` (the new element ends up at position `n`; original elements at `n` and after shift to `n+1`, etc.). The original `lst` is unchanged.

```
l: list_of([1, 2, 3, 4])
l2: insert_at(l, 2, 99)
println(list_to_array(l2))   # [1, 99, 2, 3, 4]
println(list_to_array(l))    # [1, 2, 3, 4]   — original unchanged
```

The interesting part: *how much of the original list does `l2` share with `l`?* Trace it and explain in a comment. This is the seed of persistent data structures.

23.9 Reflection — the template is more important than the structure

You'll rarely use linked lists in production Axioma code. You'll *constantly* use the **structural-recursion template** you just internalized:

```
walk_template: func(structure) [
  if structure is BaseCaseConcept then BASE_VALUE
  else COMBINE(structure.slot, walk_template(structure.other_slot))
]
```

That template is the engine for every Concept-based data structure in the rest of this book. Trees: same template, two recursive children. Graphs: same template, plus a "visited" guard. Parsed expressions in Chapter 22's evaluator: same template. Conceptual graphs in Chapter 35: same template.

You haven't just learned a data structure. You've learned how to build *every* data structure.

The next chapter ports stacks and queues — the simplest *abstract data types* — and shows how the same operations can have multiple implementations with different costs.

> *"Lists are the original objects."* — Many places. The insight is that a list of items is the most natural recursive data: *something at the front, and the rest of the list.* Once you see that shape, you'll see it everywhere.

Solutions to selected exercises: Chapter 23 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch23_solutions.md.)*

23.10 Optional — built-in Lists and sequence operators

The `Empty` and `Cons` Concepts above teach how to implement a linked structure. Axioma also supplies a built-in `List`, with the literal `` `[1, 2] `` and empty `` `[] ``. Constructors `list()`, `list(array)` and `cons(head, tail)` remain available. The `+:` and `:+` operators work on these built-in Lists and on Arrays. They do not operate on our handmade Concepts.

```
println(1 +: 2 +: `[])       # `[1, 2]
println([] :+ 1 :+ 2)           # [1, 2]
println([1, 2] :+ [3, 4])       # [1, 2, [3, 4]]
println((1 +: [2]) :+ 3)        # [1, 2, 3]
println((1 + 2) +: [4])         # [3, 4]
```

Each operation adds **one element**, even when that element is a collection. The result has the same kind as the sequence operand. `+:` groups from the right; `:+` groups from the left. Both bind as tightly as word `cons`, above addition, so parenthesize an arithmetic head. A mixed chain requires explicit parentheses.

Type annotations still use `::`.

List prepend shares the tail in O(1); List append copies the spine in O(n). Array prepend and append both copy the outer storage in O(n). This copying append differs from the mutable Array append discussed earlier in the chapter. Nested values keep their identity, and explicit Array element constraints are preserved. Changing an element slot in the result does not change the original:

```
let original = [1, 2]
let extended = original :+ 3
extended[1] = 99
println(original)              # [1, 2]
println(extended)              # [99, 2, 3]
```

`cons` remains List-only. Strings and other collection kinds are outside the new operators' scope. To accumulate efficiently with a List, prepend and then reverse once instead of repeatedly appending.

Exercise 23.7 — Build without mutation

Using built-in sequences, start with an empty List and use `+:` to build `` `[1, 2, 3] ``. Then start with an empty Array and use `:+` to build `[1, 2, 3]`. Explain the total cost when extending each construction to n elements. Finally, append `[4, 5]` as one element and predict the resulting shape.

23.11 Optional: writing built-in List literals

The backtick prefix distinguishes a built-in List from an Array. It touches `[` and has no closing backtick. A singleton needs no comma; a trailing comma is also accepted. These are ordinary evaluated expressions, not quoted code.

```
let empty = `[]
let linked = `[1, 2, 3]
println(empty is List)               # true
println(linked)                      # `[1, 2, 3]
println(linked == [1, 2, 3])          # false
println(0 +: linked)                 # `[0, 1, 2, 3]
println(linked :+ 4)                 # `[1, 2, 3, 4]
println(`[1 + 2, `[4], [5]])          # `[3, `[4], [5]]
```

Elements evaluate once in written order. A nested collection is one element; even a Range stays one value, so `len(`[1..3])` is 1. The List spine is immutable, but an Array stored inside can still be mutated. Printing keeps the literal form, such as `` `[1, 2] ``. The new literal constructs the built-in List directly, independently of any binding named `list`.

Existing `[h | t]` and `[]` patterns still work for both sequence types. This literal does not introduce another comprehension or pattern syntax. Evaluator and VM construct it; HM inference currently skips this form.

Exercise 23.8 — Values inside a List

Build a List containing the result of `1 + 2`, a singleton List containing 4, and an Array containing 5. Predict its printed form and its length. Explain why changing that Array changes the value observed through the List without changing the List's spine.

23.12 Optional: spreading between Arrays and Lists

Use `...` inside either literal to insert an Array or List's elements. The outer brackets choose the result type. The functions `list(collection)` and `array(collection)` remain useful conversions, and `list()` remains supported. Literals are the preferred form for writing and printing explicit List values.

```
let source = [1, 2]
let linked = `[...source]
let copied = [...linked]
println(linked)                      # `[1, 2]
println(copied)                      # [1, 2]
println(`[0, ...source, ...linked, 3]) # `[0, 1, 2, 1, 2, 3]
println(`[source])                   # `[[1, 2]]
push(source, 9)
println(linked)                      # `[1, 2]
```

Each operand evaluates once, in written order. Its outer elements are copied at that point; later changes to the source's outer storage do not change the result. Nested mutable values are shared. Spreading preserves values and does not transfer an Array's element contract to the new collection; annotate the result when it needs a contract.

Only Arrays and Lists are spreadable. A Range, Set, Tuple, or other collection needs an explicit conversion first. `` `[xs] `` stores the whole value as one element; `` `[...xs] `` inserts its elements. Rest parameters and

sequence patterns retain their current syntax. Evaluator, VM and gradual type checking support spreading; HM inference skips it.

Exercise 23.9 — A snapshot during construction

Create an Array containing 1. Construct a List that spreads it, calls a function that appends 2 to the Array and returns 0, then spreads the Array again. Predict the result and explain which part of the new List observes the mutation.

Chapter 24 · Stacks, Queues, and Deques

What this chapter is. Three of the smallest, oldest, most useful data structures — and three good ways to learn the most important idea in CS2: an **abstract data type** (ADT) is an *interface*, not a *representation*. The same stack operations (`push`, `pop`, `top`, `empty?`) can be backed by an Axioma array, a linked list of Concepts, or the global interpreter stack. The user of the stack doesn't care which — they get the same behavior, possibly at different costs. The same goes for queues and deques.

24.1 The abstract data type idea

Most data structures have **two layers**:

1. **The interface** (the ADT). A small set of operations that describe *what* the structure does. For a stack: `push`, `pop`, `top`, `empty?`.
2. **The representation**. The actual data layout (linked-list of Concepts? Axioma array? Forth-style global stack?) and code that implements the interface.

The key insight: **clients work against the interface, not the representation**. Once you commit to "this is a stack," the consumer code doesn't care how it's implemented — switch representations later and nothing breaks.

This chapter shows the same three ADTs (Stack, Queue, Deque) implemented in multiple ways, with measured complexity contrasts.

24.2 Stack — the simplest ADT

A **stack** is LIFO — *last in, first out*. Operations:

Operation	Effect
`push(s, x)`	Add x on top
`pop(s)`	Remove and return the top
`top(s)`	Return the top without removing
`empty?(s)`	Is the stack empty?

Representation 1 — linked list of Concepts

The Chapter 23 linked list IS a stack:

```
concept Empty
concept Cons
Cons has head: 0
Cons has tail: 0

stack_empty: an Empty {}

stack_push: func(s, x) [
  a Cons { head: x, tail: s }
]

stack_top: func(s) [
  if s is Empty then 0
  else s.head
]

stack_pop: func(s) [
  if s is Empty then s
  else s.tail
]

stack_is_empty: func(s) [
  s is Empty
]
```

All four operations are **O(1)**. The "current top" lives at the front of the linked list, where `push` and `pop` both take constant time.

```
s1: stack_push(stack_empty, 10)
s2: stack_push(s1, 20)
s3: stack_push(s2, 30)
println(stack_top(s3))    # 30
s4: stack_pop(s3)
println(stack_top(s4))    # 20
```

Note: `pop` returns a *new* stack with the top removed; the old stack is unchanged. Persistent, just like the linked lists from Chapter 23.

Representation 2 — Axioma array

A stack-by-array uses *the end* of the array as the top:

```
arr_push: func(s, x) [
  s + ([] + [x])
]

arr_top: func(s) [
  if len(s) == 0 then 0
```

```
  else s[len(s)]
]

arr_pop: func(s) [
  if len(s) == 0 then s
  else s[1..(len(s) - 1)]
]

arr_is_empty: func(s) [
  len(s) == 0
]

a1: arr_push([], 10)
a2: arr_push(a1, 20)
a3: arr_push(a2, 30)
println(arr_top(a3))    # 30
println(arr_pop(a3))    # [10, 20]
```

Cost. Each `arr_push` builds a *new* array; the underlying + concatenation is O(n). Same for `pop` — slicing copies. **So the array stack is O(n) per operation**, not O(1). For most practical uses, the Concept-based stack is faster *and* simpler.

(Production Axioma code has two better tools: mutable arrays push/pop in place at true O(1) — Chapter 16 territory — and the native **List** is exactly the persistent structure this section hand-builds: `cons(x, l)` is O(1) prepend with the tail *shared*, `first`/`rest` are the O(1) reads, and nothing ever copies. What `arr_push` teaches by construction, `List` provides as a type.)

Representation 3 — the global interpreter stack (Forth-style)

Axioma has a built-in *global* stack used by Pop-11/Forth operations. We won't teach it deeply (it's Chapter 39 material), but it exists, and stack-based programming has a long, productive history.

The pedagogical choice for the rest of this chapter: **Concept-based stacks** (Representation 1). Constant-time operations, immutable, and the recursive structure is the heart of every later data structure.

24.3 Queue — FIFO with a twist

A **queue** is FIFO — *first in, first out*. Operations:

Operation	Effect
`enqueue(q, x)`	Add `x` at the back
`dequeue(q)`	Remove and return the front
`front(q)`	Return the front without removing
`empty?(q)`	Is the queue empty?

The challenge: both operations need to be O(1), but a queue has *two* active ends. A naive linked list gives O(1) at the front and O(n) at the back, or vice versa. We need both ends to be fast.

Representation 1 — two stacks (the Okasaki trick)

Use **two** stacks, `in` and `out`. New items get pushed onto `in`. To dequeue, pop from `out`; if `out` is empty, flip `in` onto `out` (reversing it).

```
concept Queue
Queue has in_stack: 0     # incoming items, newest on top
Queue has out_stack: 0    # outgoing items, oldest on top

queue_empty: a Queue { in_stack: stack_empty, out_stack: stack_empty }

queue_enqueue: func(q, x) [
  a Queue {
    in_stack: stack_push(q.in_stack, x),
    out_stack: q.out_stack
  }
]

queue_dequeue: func(q) [
  if not stack_is_empty(q.out_stack) then [
    # Fast path: out has items
    new_out: stack_pop(q.out_stack)
    a Queue { in_stack: q.in_stack, out_stack: new_out }
  ] else if stack_is_empty(q.in_stack) then [
    # Both empty: no-op
    q
  ] else [
    # Out is empty, in has stuff: flip in onto out, then pop
    flipped: stack_reverse(q.in_stack)
    new_out: stack_pop(flipped)
    a Queue { in_stack: stack_empty, out_stack: new_out }
  ]
]

queue_front: func(q) [
  if not stack_is_empty(q.out_stack) then stack_top(q.out_stack)
  else if stack_is_empty(q.in_stack) then 0
  else stack_top(stack_reverse(q.in_stack))
]

queue_is_empty: func(q) [
  stack_is_empty(q.in_stack) and stack_is_empty(q.out_stack)
]
```

We need `stack_reverse`:

```
stack_reverse: func(s) [
  stack_reverse_helper(s, stack_empty)
]
```

```
stack_reverse_helper: func(s, acc) [
  if s is Empty then acc
  else stack_reverse_helper(s.tail, stack_push(acc, s.head))
]
```

Cost. enqueue is always O(1). dequeue is O(1) *amortized* — most calls are O(1), but the occasional "flip" is O(n). Averaged over many calls, each operation is O(1).

Why this is beautiful. Two stacks, each O(1) on their own end, working together to give us a queue with O(1) amortized both ways. No mutation needed, no doubly-linked structure, no auxiliary pointers. The whole thing fits in 20 lines.

This trick is due to Chris Okasaki (1998), who popularized functional data-structure design. The version above is the simplest case.

```
q1: queue_enqueue(queue_empty, 1)
q2: queue_enqueue(q1, 2)
q3: queue_enqueue(q2, 3)
println(queue_front(q3))      # 1
q4: queue_dequeue(q3)
println(queue_front(q4))      # 2
```

24.4 Deque — both ends fast

A **deque** (double-ended queue) supports push and pop at *both* ends:

Operation	Effect
`push_front(d, x)`	Add x to the front
`push_back(d, x)`	Add x to the back
`pop_front(d)`	Remove and return from the front
`pop_back(d)`	Remove and return from the back

The two-stack trick generalizes:

```
concept Deque
Deque has front_stack: 0
Deque has back_stack: 0

deque_empty: a Deque { front_stack: stack_empty, back_stack: stack_empty }

deque_push_front: func(d, x) [
  a Deque { front_stack: stack_push(d.front_stack, x), back_stack: d.back_stack }
]

deque_push_back: func(d, x) [
```

```
  a Deque { front_stack: d.front_stack, back_stack: stack_push(d.back_stack, x) }
]
```

`pop_front` and `pop_back` are dual to the queue's `dequeue` — pop from your own stack, or flip the other if yours is empty:

```
deque_pop_front: func(d) [
  if not stack_is_empty(d.front_stack) then [
    a Deque { front_stack: stack_pop(d.front_stack), back_stack: d.back_stack }
  ] else if stack_is_empty(d.back_stack) then d
  else [
    flipped: stack_reverse(d.back_stack)
    a Deque { front_stack: stack_pop(flipped), back_stack: stack_empty }
  ]
]

deque_pop_back: func(d) [
  if not stack_is_empty(d.back_stack) then [
    a Deque { front_stack: d.front_stack, back_stack: stack_pop(d.back_stack) }
  ] else if stack_is_empty(d.front_stack) then d
  else [
    flipped: stack_reverse(d.front_stack)
    a Deque { front_stack: stack_empty, back_stack: stack_pop(flipped) }
  ]
]
```

Same O(1) amortized cost on both ends, same immutability, same elegant lack of mutation.

In practice, mature implementations balance the two stacks periodically (every time one gets much larger than the other) so that single-side patterns don't degrade. The balancing version is the *real* Okasaki deque; ours is the pedagogical simplification.

24.5 The Design Recipe — ADTs

When designing an ADT:

Stage 1 — interface. List the operations clients will call. Stack: push/pop/top/empty?. Queue: enqueue/dequeue/front/empty?. Five-or-fewer operations is usually right.

Stage 2 — invariants. What must be true of every state? "The stack always has at least the elements pushed minus those popped, in LIFO order." Spelling out invariants makes the implementation correctness obvious.

Stage 3 — representation. Pick the underlying data: linked list of Concepts? Array? Two stacks? Each choice has cost implications.

Stage 4 — implement and time. Write the operations, then *measure*. Empirical timing on inputs of size 100, 1000, 10000 reveals whether your "O(1)" is really O(1) in practice.

Stage 5 — test against the interface. A good test exercises only the interface — never reaches into the

representation. That way the same tests run unchanged when you swap implementations.

24.6 Exercises

24.1 — Stack-balanced parens

Write `balanced(s)` — given a string `s` of parentheses (`(`, `)`, `[`, `]`, `{`, `}`), return `true` if they're properly balanced.

```
println(balanced("(())"))         # true
println(balanced("([{}])"))       # true
println(balanced("(()"))          # false (unclosed)
println(balanced(")"))            # false (unopened)
println(balanced("([)]"))         # false (interleaved)
```

Hint: walk the string. For each open paren, push onto a stack. For each close paren, pop and check the match. At the end, the stack must be empty.

24.2 — Reverse a list using a stack

Write `reverse_via_stack(arr)` that uses an explicit Stack ADT to reverse an array. Push every element of the input onto a stack, then pop them all back into a new array.

```
println(reverse_via_stack([1, 2, 3, 4, 5]))
# [5, 4, 3, 2, 1]
```

Compare with the recursive `reverse` from Chapter 23. Same asymptotic cost (O(n)), but the *imperative* style with an explicit stack is sometimes clearer for iterative algorithms.

24.3 — Queue length

Write `queue_length(q)` — return the number of elements in the queue.

```
q: queue_enqueue(queue_enqueue(queue_enqueue(queue_empty, 1), 2), 3)
println(queue_length(q))    # 3
q4: queue_dequeue(q)
println(queue_length(q4))   # 2
```

Hint: `length` of `in_stack` plus `length` of `out_stack`.

24.4 — Deque palindrome check

Write `is_palindrome(arr)` — given an array, return `true` if its elements read the same forwards and backwards. Use a deque: push every element onto the deque (back), then *pop from both ends* and compare until you reach the middle.

```
println(is_palindrome([1, 2, 3, 2, 1]))   # true
println(is_palindrome([1, 2, 3, 4, 5]))   # false
println(is_palindrome(["a", "b", "a"]))    # true
println(is_palindrome([]))                 # true (vacuously)
```

24.5 — Stack-based RPN calculator

Write `rpn(tokens)` — evaluate a *reverse-Polish-notation* expression given as an array of tokens. Numbers go onto a stack; operators pop two numbers, compute, and push the result. Support `+`, `-`, `*`, `/`.

```
println(rpn([1, 2, "+"]))               # 3 (= 1 + 2)
println(rpn([1, 2, "+", 3, "*"]))       # 9 (= (1 + 2) * 3)
println(rpn([5, 1, 2, "+", 4, "*", "+", 3, "-"]))   # 14
```

Hint: walk the tokens. If a token is a number, push it. If it's an operator, pop two, apply, push the result. After all tokens, the stack should contain exactly one value — the answer.

24.6 (open) — Browser history with deque

Model a browser's tab history as a deque. Operations:

- `visit(url)` — go to a new URL (push onto back).
- `back()` — go to the previous URL (pop back, return new current).
- `forward()` — go forward after a back (we'll need a *second* deque for the forward stack — this is the *two-deque pattern*).
- `current()` — return the current URL.

Document your choice of representation. Real browsers do this with stacks rather than deques; explain why a deque (your choice) is also valid.

24.7 Reflection — interface > representation

The biggest lesson of this chapter is *not* the data structures themselves. It's the **discipline of separating interface from representation**.

When you design a piece of code that uses a queue, you should be able to specify *just*:

> "I need a queue: a thing where I can `enqueue` items at the back and `dequeue` items from the front."

And then **plug in any implementation that satisfies that contract**. Today maybe you use the two-stack queue. Next month you discover a measurement that shows the doubly-linked list is faster for your workload. You swap representations *without touching the calling code*.

This is the seed of **encapsulation** — the idea that underpins object-oriented programming, modules, libraries, microservices, and APIs. Every "good API" is just an ADT with a stable interface and swappable implementation underneath.

The next chapter brings hash tables — a *single* representation (array of buckets) that supports a much richer interface than stacks or queues.

> *"Programs must be written for people to read, and only incidentally for machines to execute."* — Hal Abelson, in the preface to *Structure and Interpretation of Computer Programs*. ADTs are the discipline that makes that possible: clients read the interface, not the implementation.

Solutions to selected exercises: Chapter 24 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch24_solutions.md.)*

Chapter 25 · Hash Tables and Dictionaries

> **What this chapter is.** A hash table answers a question that lists and stacks can't answer well: *given a key, find the value, fast.* The trick is **hashing** — turning the key into an array index in O(1). The chapter shows the built-in `dict()` / hash-literal syntax, opens the lid on how hashing actually works, builds a hand-rolled hash table from Concepts so you understand the trade-offs, and ends with a tour of *collision resolution* — the part of hash-table design that determines how robust your implementation really is.

25.1 The motivation — lookup by key

So far we can:

- Walk a list looking for a match: O(n).
- Use an array index when we already know the position: O(1).

Neither gives us the third option we want most often:

> **Given a *key* (a string, a number, anything), find the associated *value*, in constant time, regardless of how many entries are stored.**

That's a hash table — also called a dictionary, map, or associative array depending on the language. It's the data structure behind every database index, every spell- checker, every compiler's symbol table, every cached web request, and every modern programming language's "object" or "dictionary" type.

25.2 The built-in — `dict()` and hash literals

Axioma gives you hash tables two ways:

```
# The function form
d: dict()
d["alice"] = 30
d["bob"] = 25
println(d["alice"])    # 30

# The literal form
ages: {"alice": 30, "bob": 25, "carol": 22}
println(ages["bob"])   # 25
println(ages.alice)    # 30   — dot access also works for string-keyed hashes
```

Both produce a `Dictionary`. Keys can be values, including numbers; string keys are a convenient special

case. The dot-access shortcut (`ages.alice`) works for a string key spelled as a valid identifier.

Cost. Insertion and lookup are O(1) on average. That's the entire selling point.

> **Why build our own?** The built-in hash already does everything this chapter needs — literals read fine and post-creation mutation (`h["a"] = 1`) works. But the point of the chapter is to see what's *under the hood*, so from §25.4 onward we hand-roll a **Concept-based hash table**: buckets, hashing, collision handling, the works.

Pairs, tuples, and keys by value

A Pair represents two associated values. It is distinct from a Tuple, although the dictionary constructor can consume either representation.

```
labels: dict([1 -> "one", 2 -> "two"])
println(labels[2])                      # two
println(labels is Dictionary)           # true
println((1 -> "one") is Pair)           # true
println((1, "one") is Tuple)            # true
println(dictionary([(3, "three")])[3]) # three
```

The arrow can also introduce a function. A **bare identifier** followed by `->` is a lambda parameter, as in `x -> 2*x`. To make a Pair whose left member is the value of a variable, use the explicit constructor: `pair(key, value)`. Parenthesizing a name still permits the lambda reading. Do not infer meaning from the arrow alone. Numeric and other supported keys compare by value; dot notation is only the convenient string-key spelling. Mutable compound keys require special care: changing the value used as a key makes the intended lookup contract hard to reason about. Prefer stable, simple keys in introductory programs.

25.3 What's happening under the hood

The trick: take the key, compute a number, use that number as an array index.

```
"alice"  ──hash──→  4827361  ──% N──→  bucket #3
```

Three pieces:

1. **A hash function** that turns any key into an integer. Good hash functions distribute keys uniformly: similar keys get *different* hashes, so two real keys rarely collide.
2. **A bucket array** of fixed size N. The hash gets reduced modulo N to give a bucket number.
3. **A collision strategy** for what to do when two different keys hash to the same bucket.

In Axioma's built-in `dict`, all three are hidden — you get the interface, the implementation is internal. For **learning**, we'll build our own.

25.4 A hand-rolled hash table

Start with the data definition:

```
concept HashBucket
HashBucket has key: ""
HashBucket has value: 0
HashBucket has nxt: 0     # for chaining — see §25.6

concept HashTable

HashTable has buckets: 0   # array of `HashBucket`s or `nil`
HashTable has size: 0      # number of buckets
HashTable has count: 0     # number of stored entries
```

Each slot in the `buckets` array is either `nil` (empty) or a `HashBucket` Concept holding a key, a value, and a `nxt` pointer for chained collisions.

A trivial hash function for strings (the *djb2* hash, much- used and easy to remember):

```
djb2_hash: func(s) [
  h: 5381
  i: 1
  while (i <= len(s)) [
    code: ord(s[i..i])
    h = h * 33 + code
    i = i + 1
  ]
  if h < 0 then -h else h    # ensure positive
]

println(djb2_hash("alice"))   # some large positive number
println(djb2_hash("bob"))     # different large positive number
```

Properties of a good hash function:

- *Deterministic.* Same key in → same hash out, every time.
- *Uniform.* Different keys spread evenly across the integer range, reducing clustering. Collisions remain possible and must be handled.
- *Fast.* O(length-of-key), not O(n^2).
- *Good mixing.* Changing one bit of the key ideally changes many bits of the hash (the **avalanche property**).

djb2 is not the strongest hash function in the world, but it's fast, simple, and good enough for teaching. Production code uses SipHash, MurmurHash, or xxHash.

25.5 The simplest hash table — direct addressing

```
nil_bucket: 0    # we'll use 0 as the "empty slot" sentinel

make_table: func(size) [
  buckets: []
  i: 1
  while (i <= size) [
    buckets = buckets + ([] + [nil_bucket])
    i = i + 1
  ]
  a HashTable { buckets: buckets, size: size, count: 0 }
]

put_simple: func(table, key, item_value) [
  idx: djb2_hash(key) % table.size + 1
  new_bucket: a HashBucket { key: key, value: item_value, nxt: nil_bucket }
  table.buckets[idx] = new_bucket
  a HashTable {
    buckets: table.buckets,
    size: table.size,
    count: table.count + 1
  }
]

get_simple: func(table, key) [
  idx: djb2_hash(key) % table.size + 1
  bucket: table.buckets[idx]
  if bucket == nil_bucket then 0
  else if bucket.key == key then bucket.value
  else 0
]
```

This works **as long as no two keys collide**. In practice, collisions are inevitable — the *birthday paradox* says that in a table of 100 buckets, you'll see your first collision around the 13th distinct key. We need a real collision strategy.

25.6 Collision resolution — separate chaining

The simplest collision resolution: **chain colliding entries together** in a linked list per bucket.

```
put_chained: func(table, key, item_value) [
  idx: djb2_hash(key) % table.size + 1
  existing: table.buckets[idx]
  new_bucket: a HashBucket {
    key: key,
    value: item_value,
```

```
    nxt: existing
  }
  table.buckets[idx] = new_bucket
  a HashTable {
    buckets: table.buckets,
    size: table.size,
    count: table.count + 1
  }
]

get_chained: func(table, key) [
  idx: djb2_hash(key) % table.size + 1
  bucket: table.buckets[idx]
  while_chain_lookup(bucket, key)
]

while_chain_lookup: func(bucket, key) [
  if bucket == nil_bucket then 0
  else if bucket.key == key then bucket.value
  else while_chain_lookup(bucket.nxt, key)
]
```

When inserting, push the new `HashBucket` onto the head of the chain (O(1)). When looking up, walk the chain (O(chain length)). If the table is sized so that the **load factor** (count / size) stays around 0.75, the *average* chain length is roughly 1, and lookups are O(1) on average.

When the load factor gets too high, the table needs to **rehash** into a larger bucket array. We won't cover rehashing here — the basic algorithm is "make a new double-sized table, walk every chain, re-insert each entry."

25.7 The other strategy — open addressing

An alternative: store everything *in the bucket array itself*, with no chains. If a collision happens, **probe** to the nxt available slot.

```
hash(key) % N = 7
bucket 7 occupied → probe bucket 8
bucket 8 occupied → probe bucket 9
bucket 9 free → store here
```

Linear probing is the simplest version (just increment). Better strategies include quadratic probing (probe by 1, 4, 9, 16…) and double hashing (use a second hash to determine the probe step). Open addressing is faster *when* the load factor is low (≤ 0.7) — better cache locality, no pointer chasing — but degrades sharply as the table fills.

For most teaching purposes, **separate chaining** is the simpler and more forgiving choice. Production hash tables often use open addressing because of the cache benefit.

25.8 Custom Concept keys

The hash function in §25.4 only handles strings. For a key that's a Concept (e.g., a `Point`), you'd need:

```
concept Point
Point has x: 0
Point has y: 0

point_hash: func(p) [
  djb2_hash(str(p.x) + "," + str(p.y))
]
```

The trick is to produce a *canonical string representation* of the Concept and hash that. Same Concept → same string → same hash. The string-then-hash pattern is general — every language that supports "user-defined hashable objects" uses some variant of it.

25.9 Two-level dispatch via hash

A classic use case: a function that dispatches on a string key, with O(1) lookup. Without a hash, you'd write:

```
# Without hash — O(n) dispatch via if/else cascade
dispatch_slow: func(cmd) [
  if cmd == "open" then "Opening..."
  else if cmd == "close" then "Closing..."
  else if cmd == "list" then "Listing..."
  else if cmd == "kill" then "Killing..."
  else "Unknown command"
]
```

With a hash, the dispatch becomes:

```
cmd_table: {
  "open":  "Opening...",
  "close": "Closing...",
  "list":  "Listing...",
  "kill":  "Killing..."
}

dispatch_fast: func(cmd) [
  if cmd in cmd_table then cmd_table[cmd]
  else "Unknown command"
]

println(dispatch_fast("close"))   # Closing...
println(dispatch_fast("xyz"))     # Unknown command
```

Same behavior. For 4 commands, the cascade is fine. For many commands, a hash table avoids a linear chain of string comparisons. The actual speedup depends on hashing costs and the workload; it is not equal to the number of commands.

This pattern — *replace cascades with table lookup* — is one of the most common refactorings in performance work.

25.10 Exercises

25.1 — Word frequency

Write `word_freq(words)` — given an array of words, return a hash table mapping each unique word to its count.

```
words: ["the", "cat", "sat", "on", "the", "mat", "and", "the", "rat"]
freq: word_freq(words)
println(freq["the"])    # 3
println(freq["cat"])    # 1
println(freq["dog"])    # 0 (or some default)
```

Hint: walk the array. For each word, set `freq[word] = (freq[word] or 0) + 1`. Or: check `word in freq` and dispatch.

25.2 — Hash a small Concept

Write a hash function for a `Point` Concept (slots `x` and `y`, both integers). Verify that:

- Equal Points produce equal hashes.
- Distinct Points typically produce distinct hashes (try 10 random points and count unique hashes).

```
hash_point: func(p) [ ?? ]

p1: a Point { x: 3, y: 7 }
p2: a Point { x: 3, y: 7 }   # equal coords
p3: a Point { x: 7, y: 3 }   # swapped — should be different
println(hash_point(p1) == hash_point(p2))   # true
println(hash_point(p1) == hash_point(p3))   # false (almost certainly)
```

25.3 — Implement djb2

Write `djb2_hash(s)` from scratch. Then hash these 10 words and check whether the resulting hash values modulo 100 are reasonably uniformly distributed (no bucket should hold more than 3 of them):

```
words: ["the", "of", "and", "a", "to", "in", "is", "you", "that", "it"]
buckets: make_table(100)
# ... walk words, hash, count per bucket ...
```

25.4 — Chain-based hash table

Build a small chain-based hash table from scratch (using the `HashBucket` / `HashTable` Concepts from §25.4). Implement `put`, `get`, and `contains`. Insert 5 key-value pairs and verify they all round-trip.

```
t: make_table(8)
t1: put_chained(t, "a", 1)
t2: put_chained(t1, "b", 2)
println(get_chained(t2, "a"))   # 1
println(get_chained(t2, "b"))   # 2
println(get_chained(t2, "c"))   # 0 (default for not-found)
```

25.5 — Replace a cascade

Take this if/else cascade and rewrite it as a hash-table dispatch:

```
day_to_number: func(d) [
  if d == "Mon" then 1
  else if d == "Tue" then 2
  else if d == "Wed" then 3
  else if d == "Thu" then 4
  else if d == "Fri" then 5
  else if d == "Sat" then 6
  else if d == "Sun" then 7
  else 0
]
```

Then time both versions on 10,000 random calls. Which is faster? By how much?

25.6 (open) — Most-frequent words

Write `top_n_words(words, n)` — given an array of words and an integer `n`, return the `n` most frequent words sorted by count (descending).

```
speech: [... ~1000 words ...]
top10: top_n_words(speech, 10)
println(top10)
```

This combines Ex 25.1 (frequency count) with a sort. There are several reasonable implementations — comment on which you chose and why.

25.11 Reflection — the constant-time miracle

Hash tables are *unreasonably effective*. They turn a class of problems (key-value lookup) from O(n) to O(1) with a small constant overhead. That single transformation has shaped every major piece of software written in the last 50 years:

- **Databases**: every index is a hash or B-tree.
- **Compilers**: symbol tables, keyword recognition.
- **Web servers**: routing, caching, session lookup.
- **Operating systems**: file-handle tables, process tables.
- **Networking**: routing tables, ARP caches.
- **Spell-checkers**: dictionary lookup.

For Axioma specifically, hash tables underpin the environment system (Chapter 12's local bindings), the KB fact store, and the `dict()` builtin you use directly.

The nxt chapter takes the *other* fundamental dictionary strategy — balanced trees — and shows how they give you ordered keys at the price of O(log n) per operation. When you need *sorted* iteration over your keys, hash tables fail and balanced trees win.

> *"Algorithms + Data Structures = Programs."* — Niklaus Wirth, 1976. The hash table is one of the half-dozen data structures that turn the equation from impossible to tractable.

Solutions to selected exercises: Chapter 25 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch25_solutions.md.)*

Chapter 26 · Balanced Trees

> **What this chapter is.** A hash table gives O(1) lookup but no ordering. A sorted array gives O(log n) lookup *if you never insert*, but O(n) per insertion. A **binary search tree** combines them — O(log n) lookup *and* insertion *if it stays balanced*. This chapter builds a plain BST, demonstrates why insertion order can wreck it, and sketches the **red-black tree** fix.

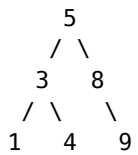
26.1 The motivation — sorted lookup that survives insertion

You sometimes need:

- Look up by key, fast.
- Insert new keys, fast.
- Walk the keys *in sorted order*, fast.

Hash tables fail the third (unordered iteration). Sorted arrays fail the second (insertion shifts the tail).

A **binary search tree** (BST) gives you all three at *O(log n)*. Each node holds a key, a left child (smaller keys), and a right child (larger keys). Look-up walks one side or the other; insertion walks to a leaf and adds.



Search for 4: $4 < 5 \rightarrow$ go left. $4 > 3 \rightarrow$ go right. Found. Search for 7: $7 > 5 \rightarrow$ right. $7 < 8 \rightarrow$ left. Not found (empty).

26.2 BST as Concepts

```
concept TreeEmpty
concept TreeNode
TreeNode has key: 0
TreeNode has value: 0
TreeNode has left: 0
TreeNode has right: 0

tree_empty: a TreeEmpty {}
```

```
tree_insert: func(t, k, v) [
  if t is TreeEmpty then a TreeNode { key: k, value: v, left: tree_empty, right:
  ↪  tree_empty }
  else if k < t.key then a TreeNode { key: t.key, value: t.value, left:
  ↪  tree_insert(t.left, k, v), right: t.right }
  else if k > t.key then a TreeNode { key: t.key, value: t.value, left: t.left, right:
  ↪  tree_insert(t.right, k, v) }
  else a TreeNode { key: t.key, value: v, left: t.left, right: t.right }   # update
]

tree_get: func(t, k) [
  if t is TreeEmpty then 0
  else if k < t.key then tree_get(t.left, k)
  else if k > t.key then tree_get(t.right, k)
  else t.value
]

tree_contains: func(t, k) [
  if t is TreeEmpty then false
  else if k < t.key then tree_contains(t.left, k)
  else if k > t.key then tree_contains(t.right, k)
  else true
]
```

Three-case structural recursion. Smaller-than → left. Larger-than → right. Equal → found.

```
t1: tree_insert(tree_empty, 5, "five")
t2: tree_insert(t1, 3, "three")
t3: tree_insert(t2, 8, "eight")
t4: tree_insert(t3, 1, "one")
t5: tree_insert(t4, 4, "four")
println(tree_get(t5, 4))         # "four"
println(tree_contains(t5, 7))    # false
```

26.3 In-order traversal — a free sort

Walking a BST in *in-order* (left, then root, then right) visits keys in ascending sorted order:

```
tree_inorder: func(t) [
  if t is TreeEmpty then []
  else tree_inorder(t.left) + ([] + [t.key]) + tree_inorder(t.right)
]

println(tree_inorder(t5))   # [1, 3, 4, 5, 8]
```

This is **tree sort**: insert every element into a BST, then in-order-walk to get them sorted. O(n log n) when the tree is balanced. Less practical than mergesort (Chapter 29) but a beautiful application of the

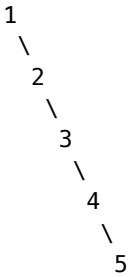
structure.

26.4 The problem — skewed trees

What happens if you insert keys *in sorted order*?

```
bad: tree_insert(tree_insert(tree_insert(tree_insert(tree_insert(tree_empty, 1, "a"),
 ↪  2, "b"), 3, "c"), 4, "d"), 5, "e")
println(tree_inorder(bad))   # [1, 2, 3, 4, 5]
```

The keys come out sorted, but the tree itself is a **right-leaning chain**:



Searching for 5 walks all 5 levels — O(n), not O(log n). Insertion order *fully sorted* gives **worst-case BST behavior**. Reverse-sorted is symmetric and equally bad.

For a BST to deliver its promised O(log n), the tree must be **balanced** — the depth of the deepest leaf must be $\leq 1.5 \times$ depth of the shallowest, give or take a constant. Achieving balance for arbitrary insertion orders requires *self-balancing* trees.

26.5 Red-black trees — the standard fix

A **red-black tree** is a BST with extra invariants attached to each node (each is "red" or "black") that collectively bound the tree's worst-case height to O(log n). On every insert and delete, the tree is locally rotated and recolored to keep the invariants.

The five red-black invariants:

1. Every node is red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, both its children are black.
5. Every simple path from a node to its descendant leaves contains the same number of black nodes.

Insert and delete each take O(log n), do at most O(1) rotations and O(log n) recolorings.

Implementing a full red-black tree is *substantial* — usually ~100 lines in any language, including Axioma. It's a Chapter 26 *exercise* in CS2 textbooks, not Chapter 26 *prose*. The exercises at the end of this chapter walk through the pieces.

The takeaway: **the tree's invariants do the work**. Once the invariants are maintained, lookup, insert,

and delete are all O(log n) without any explicit rebalancing logic in the calling code.

26.6 Exercises

26.1 — Tree height

Write `tree_height(t)` — the number of edges on the longest path from root to leaf. A single-node tree has height 0; the empty tree has height -1 (or you can use 0; document your choice).

```
t: tree_insert(tree_insert(tree_insert(tree_empty, 5, "a"), 3, "b"), 8, "c")
println(tree_height(t))   # 1
```

26.2 — Tree min and max

Write `tree_min(t)` and `tree_max(t)` — return the smallest and largest keys. In a BST, the smallest is the leftmost node; the largest is the rightmost.

```
t: tree_insert(tree_insert(tree_insert(tree_insert(tree_empty, 5, "a"), 3, "b"), 8,
 ↪  "c"), 1, "d")
println(tree_min(t))   # 1
println(tree_max(t))   # 8
```

26.3 — Count nodes

Write `tree_size(t)` — the number of key-value pairs stored.

26.4 — Range query

Write `tree_range(t, lo, hi)` — return an array of all keys in `[lo, hi]` in sorted order. Use the BST property to **prune** the recursion: if the current node's key is less than `lo`, only recurse right; if greater than `hi`, only recurse left.

```
t: tree_insert(tree_insert(tree_insert(tree_insert(tree_insert(tree_empty, 5, "a"), 3,
 ↪  "b"), 8, "c"), 1, "d"), 7, "e")
println(tree_range(t, 3, 7))   # [3, 5, 7]
```

26.5 — Detect imbalance

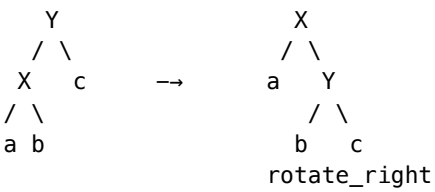
Write `is_balanced(t)` — return `true` if the absolute difference of left and right subtree heights is ≤ 1 at every node, `false` otherwise.

```
# Build a skewed tree by sorted insertion
skewed: tree_insert(tree_insert(tree_insert(tree_empty, 1, "a"), 2, "b"), 3, "c")
println(is_balanced(skewed))   # false (degenerate chain)
```

```
# Insert in a balance-friendly order
balanced_t: tree_insert(tree_insert(tree_insert(tree_empty, 2, "a"), 1, "b"), 3, "c")
println(is_balanced(balanced_t))   # true
```

26.6 (open) — Tree rotation

A **rotation** is the local rearrangement that keeps a tree balanced. Implement `rotate_right(t)` and `rotate_left(t)`:




```
rotate_right: func(t) [ ?? ]
rotate_left:  func(t) [ ?? ]
```

Both should return a new tree (no mutation). Verify that in-order traversal of the rotated tree equals the in-order traversal of the original. (Rotation preserves order.)

This is the building block of red-black, AVL, and splay trees. With rotation in hand, you can implement any self-balancing variant.

26.7 Reflection — invariants do the work

Three takeaways:

- **BST is a one-shot data structure.** Worst-case insertion order produces a chain that defeats the O(log n) promise. Balanced variants exist precisely because unbalanced trees are *one bad pattern away from disaster*.
- **Rotations preserve order.** That's the *deep* property that makes self-balancing possible at all. Rearranging the tree structurally doesn't change which key is "smaller than" another — only how the tree finds them.
- **Invariants encode correctness.** Once you have a clear invariant ("every red node has black children"), the code that maintains it almost writes itself. Most data- structure design is invariant design.

The next chapter shifts from *trees in isolation* to *graphs* — where edges no longer just go "down" from parent to child, but anywhere.

> *"Many small simple bug-free programs can be combined to form a large bug-free program."* — Dijkstra, 1972. Trees are the canonical "small simple bug-free program" — a recursive structure where each node only sees its immediate children, never the global shape.

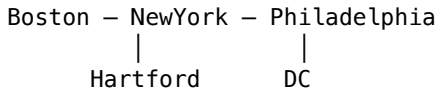
Solutions to selected exercises: Chapter 26 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch26_solutions.md.)*

Chapter 27 · Graphs as Reified Relationship Concepts

What this chapter is. A graph is *nodes* + *edges* — the most general data structure in the standard CS2 curriculum. Trees are graphs without cycles; lists are graphs where each node has one successor. Graphs underpin social networks, road maps, the web, every dependency system, and every knowledge graph. The chapter shows two representations — the classical *adjacency-list* form and Axioma's strength, the *reified-edge* form — and walks through breadth-first and depth-first traversals.

27.1 The two representations

Take a graph of five cities with roads between them:



Representation 1 — adjacency list. A hash table from each node to the list of its neighbors:

```
road_adj: {
  "Boston":       ["NewYork"],
  "NewYork":      ["Boston", "Philadelphia", "Hartford"],
  "Philadelphia": ["NewYork", "DC"],
  "Hartford":     ["NewYork"],
  "DC":           ["Philadelphia"]
}
```

Compact, fast neighbor lookup. The standard choice in most CS2 textbooks.

Representation 2 — reified Edge Concepts. Each edge is its own first-class Concept:

```
concept Road
Road has source: ""
Road has target: ""
Road has miles: 0

r1: a Road { source: "Boston", target: "NewYork", miles: 215 }
r2: a Road { source: "NewYork", target: "Philadelphia", miles: 95 }
r3: a Road { source: "NewYork", target: "Hartford", miles: 117 }
```

```
r4: a Road { source: "Philadelphia", target: "DC", miles: 140 }

roads: [r1, r2, r3, r4]
```

Each `Road` is a Concept instance you can pass to functions, store in lists, query against. **Edges become first-class citizens.**

Why bother? Three reasons:

1. **Edge attributes.** Roads have lengths, traffic, tolls, surface types. With reified edges, those are just slots on the Road Concept. An adjacency list can also store edge records; Concept instances make that choice explicit and available to the knowledge model.
2. **Edge identity.** You can query *"which road connects Boston to New York?"* and get a *thing back* — not just a boolean. That thing has its own slots and history.
3. **Logic-programming integration.** Reified edges plug directly into Axioma's relation system (Chapter 21). `parent`, `friend`, `prereq` from Book 1 were already reified relationships — the Concept-and-slots view is the same idea, factored differently.

The textbook uses **reified edges** as the primary representation. Adjacency lists appear when the algorithm specifically benefits from O(1) neighbor lookup.

27.2 Neighbors and the basic queries

Given an array of Road Concepts, finding a node's neighbors is a filter-then-extract:

```
neighbors_of: func(roads, node) [
  result: []
  i: 1
  while (i <= len(roads)) [
    if roads[i].source == node then [
      result = result + ([] + [roads[i].target])
    ]
    if roads[i].target == node then [
      result = result + ([] + [roads[i].source])
    ]
    i = i + 1
  ]
  result
]

println(neighbors_of(roads, "NewYork"))
# ["Boston", "Philadelphia", "Hartford"]
```

We check both ends because the example treats roads as *undirected*. For directed graphs, drop the second `if`.

Cost. `neighbors_of` is O(E) — it walks every edge. For small graphs this is fine; for large ones, build an adjacency-list index from the edge array once.

27.3 BFS — breadth-first search

The classical shortest-path algorithm for unweighted graphs. From a start node, visit nodes level by level — all distance-1 neighbors, then distance-2, then distance-3, and so on.

The shape: a **queue** of "to-visit" nodes, and a **set** of "already-visited" markers.

```
bfs_distances: func(roads, start) [
  # Initialize: distances dict starts with {start: 0}, queue with [start]
  distances: {}
  distances = put_t(make_table(64), start, 0)
  queue: [start]

  while (len(queue) > 0) [
    node: queue[1]
    queue = queue[2..len(queue)]
    nbrs: neighbors_of(roads, node)
    i: 1
    while (i <= len(nbrs)) [
      n: nbrs[i]
      if not contains_t(distances, n) then [
        distances = put_t(distances, n, get_t(distances, node) + 1)
        queue = queue + ([] + [n])
      ]
      i = i + 1
    ]
  ]
  distances
]
```

(Load the complete hash-table prelude at the start of Chapter 25's solutions, which defines `make_table`, `put_t`, `get_t`, and `contains_t`. The simplified table in the chapter itself uses different names.)

```
d: bfs_distances(roads, "Boston")
println(get_t(d, "Boston"))         # 0
println(get_t(d, "NewYork"))        # 1
println(get_t(d, "Philadelphia"))   # 2
println(get_t(d, "DC"))             # 3
```

Cost. With adjacency lists and a queue supporting constant-time operations, BFS is `O(V + E)`. Our readable edge-list version scans all roads to find each node's neighbors and rebuilds its queue Array, so that bound does not describe this implementation. Hash-table load also matters.

BFS is the algorithm behind: shortest path in unweighted graphs, "six degrees of separation," friend-of-friend suggestions (Chapter 21 Ex 21.5 was essentially BFS at depth 2), web-crawler frontier expansion.

27.4 DFS — depth-first search

Same shape, but use a *stack* instead of a queue. DFS goes deep first, backtracks when it hits a dead end:

```
dfs_visit: func(roads, start) [
  visited: make_table(64)
  visited = put_t(visited, start, 1)
  stk: [start]
  order: []
  while (len(stk) > 0) [
    node: stk[len(stk)]
    stk = stk[1..(len(stk) - 1)]
    order = order + ([] + [node])
    nbrs: neighbors_of(roads, node)
    i: 1
    while (i <= len(nbrs)) [
      n: nbrs[i]
      if not contains_t(visited, n) then [
        visited = put_t(visited, n, 1)
        stk = stk + ([] + [n])
      ]
      i = i + 1
    ]
  ]
  order
]
```

DFS visits in a *different order* than BFS — it goes as deep as it can before backing up.

Cost. The usual `O(V + E)` bound assumes adjacency lists and efficient stack and visited-set operations. This teaching version instead scans the edge list and rebuilds Arrays; account for those costs separately.

DFS is the algorithm behind: cycle detection, topological sort, strongly-connected-component analysis, maze solving.

27.5 Cycle detection

A graph has a *cycle* if you can leave a node and return to it without retracing an edge. The simplest cycle check uses DFS plus a "currently on the path" marker:

```
has_cycle: func(roads) [
  nodes: all_nodes(roads)
  visited: make_table(64)
  result: false
  i: 1
  while (i <= len(nodes) and not result) [
    if not contains_t(visited, nodes[i]) then [
      if cycle_dfs(roads, nodes[i], "", visited) then [result = true]
```

```
    ]
    i = i + 1
  ]
  result
]
```

(`all_nodes` walks every Road and collects the union of sources and targets; `cycle_dfs` is the per-component DFS with parent tracking. Full implementation in the solutions.)

For the road graph above, `has_cycle` returns `false` — it's a tree (well, almost; the chain Boston-NY-Phil-DC plus the spur NY-Hartford has no cycles). Adding a road from `DC` back to `Boston` would create one.

27.6 A worked example — course prerequisites

Recall Chapter 21's course-prerequisite graph. The same data modeled with reified edges:

```
concept Prereq
Prereq has course: ""
Prereq has requires: ""

prereqs: [
  a Prereq { course: "CS201", requires: "CS101" },
  a Prereq { course: "CS201", requires: "MATH101" },
  a Prereq { course: "CS301", requires: "CS201" },
  a Prereq { course: "CS301", requires: "MATH201" },
  a Prereq { course: "CS401", requires: "CS301" },
  a Prereq { course: "MATH201", requires: "MATH101" }
]
```

Topological sort — order the courses so prerequisites come before dependents. The standard algorithm: repeatedly pick a course with no unfulfilled prerequisites, output it, remove its outgoing edges.

The implementation lives in the exercises. The key insight: *the same graph, two views* — Chapter 21 used facts and rules, this chapter uses reified edges and explicit traversal. **Both are valid; pick the one that matches your debugging style.**

27.7 Exercises

27.1 — all_nodes

Write `all_nodes(roads)` — return an array of all unique node names referenced as either source or target of any edge.

```
println(all_nodes(roads))
# ["Boston", "NewYork", "Philadelphia", "Hartford", "DC"]
```

```
# (order may vary)
```

27.2 — Path existence (BFS-based)

Write `has_path(roads, src, dst)` — return `true` if there's any sequence of edges from `src` to `dst`. Use BFS.

```
println(has_path(roads, "Boston", "DC"))      # true
println(has_path(roads, "DC", "Boston"))      # true (undirected!)
println(has_path(roads, "Boston", "Mars"))    # false
```

27.3 — Edge weight sum

Each Road has a `miles` slot. Write `total_miles(roads)` — sum of every road's mileage.

```
println(total_miles(roads))   # 567 = 215 + 95 + 117 + 140
```

27.4 — Topological sort (course prereqs)

Implement `topo_sort(prereqs)` — given an array of `Prereq` edges, return an array of course names in dependency order (every course comes after all its prerequisites).

```
println(topo_sort(prereqs))
# ["CS101", "MATH101", "CS201", "MATH201", "CS301", "CS401"]
# (one of several valid orderings — your output may differ)
```

Hint: repeatedly pick a course whose prerequisites are all already in the output, append it, remove its outgoing edges from consideration. Stop when no courses remain or when no course is pickable (in which case there's a cycle).

27.5 — Cycle detection

Implement `has_cycle(roads)` — return `true` if the (undirected) graph has any cycle. Use DFS with parent tracking.

```
println(has_cycle(roads))   # false — it's a tree
with_cycle: roads + ([] + [a Road { source: "DC", target: "Boston", miles: 600 }])
println(has_cycle(with_cycle))   # true
```

27.6 (open) — Dijkstra's shortest path

Implement `shortest_path(roads, src, dst)` using Dijkstra's algorithm — given weighted Road edges, find the route with minimum total `miles` from `src` to `dst`.

```
println(shortest_path(roads, "Boston", "DC"))
# ["Boston", "NewYork", "Philadelphia", "DC"]
```

Document your choice of priority-queue representation (simple linear scan? Concept-based heap?) and analyze the time complexity.

27.8 Reflection — graphs are universal

The whole chapter rests on one insight: **almost every interesting data structure is a graph in disguise**. Trees are acyclic graphs. Linked lists are graphs where each node has one outgoing edge. Family trees, web pages, file systems, dependency graphs, social networks, knowledge bases — all the same shape.

Once you can express data as nodes + edges and traverse it with BFS or DFS, you have a *generic vocabulary* for half of all algorithm problems. The other half is sorting and arithmetic.

For Axioma specifically, the **reified-edge** representation is the bridge between data-structure code (this chapter) and knowledge-representation code (Chapters 34–38). Once an edge is a first-class Concept, it can carry truth values, provenance, defeasibility — everything Book 1 Chapter 21 taught about logic programs.

The next chapter steps back from data structures and introduces the language of *cost* — Big-O notation and empirical complexity measurement, the tools that let you compare data-structure choices objectively.

> *"In graph problems, the structure is usually more important than the algorithm."* — Tarjan, paraphrased from a talk. The choice of representation (adjacency list vs. reified edges) often matters more than which BFS or DFS variant you use to walk it.

Solutions to selected exercises: Chapter 27 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch27_solutions.md.)*

Chapter 28 · The Graphs Landscape — Four Kinds, One Language

What this chapter is. Chapter 27 taught you *plain mathematical graphs* — vertices, edges, BFS, DFS, the classic data-structures repertoire. That's *one* graph kind. Axioma also supports three other graph formalisms, each engineered to answer a question that a plain graph can't. Sowa's **Conceptual Graphs** add typed concepts and n-ary relations. Peirce's **Existential Graphs** add *diagrammatic logic* — propositions as inscriptions on a sheet, negation as enclosure. McCarthy-style **Context Graphs** add multi-context truth — the same proposition can be true in one context and false in another, with Belnap B4 truth values per context. By the end of this chapter you will know all four formalisms, the question each is best at, and how to call them from Axioma.

28.1 The four-kind taxonomy

Kind	Inventor	Best for
Plain graphs	Euler 1736; graph theory	Networks, paths, connectivity, centrality — Chapter 27
Conceptual Graphs	John Sowa, 1976	Semantic networks with *typed* concepts, *n-ary* relations, and the FOL bridge
Existential Graphs	C. S. Peirce, 1896	Diagrammatic logic — propositions on a sheet, negation as enclosure, the five rules of inference as visual moves
Context Graphs	McCarthy `ist(c,p)`; MCS; CKR	Multi-context reasoning with truth-per-context, including Belnap B4 — beliefs, sources, hypotheses, temporal slices

These four formalisms live in Axioma *side-by-side* because they answer **different** questions. Don't pick by the shape of the diagram; pick by the question you need to answer.

The remainder of this chapter takes each formalism in turn. We cover the theory in just enough depth to use it, then move directly into Axioma code that you can run.

28.2 Plain graphs (recap)

Chapter 27 already drove this home, but for completeness of the taxonomy: a **plain graph** is a pair `(V, E)` where `V` is a set of vertices and `E` is a set of edges between them. Edges can be directed or undirected, weighted or unweighted, and a vertex can be any value your language allows.

Axioma's `graph()` and `digraph()` builtins give you a plain graph; reified `Edge` concepts let you store metadata per edge. Plain graphs are the right answer when your question is **structural** — *shortest path, centrality, connected components, cycles, topological sort.* None of those questions need types on the vertices or truth values on the edges.

You'd reach for a plain graph for:

- A road network ("shortest route from London to Tokyo")
- A social network ("who knows whom, two hops out")
- A dependency graph ("compile order for these modules")
- A computer network ("does packet X reach host Y")

You'd reach for one of the *other three* formalisms when your question is **semantic** or **epistemic** — what the nodes *mean*, what's *believed* about them, or whether their meaning depends on a context.

28.3 Sowa Conceptual Graphs

John Sowa's **conceptual graphs** (CGs) are the modern form of the *semantic network* tradition that goes back to Quillian's 1968 thesis. The signature commitment is that nodes come in **two kinds** — *concept nodes* and *relation nodes* — and the graph is bipartite: concepts only connect to relations, not to other concepts.

28.3.1 Concepts and relations

A **concept node** is a typed entity. It has a *type* (a concept name like `Person`, `Animal`, `Country`) and an optional *referent* — the specific individual being referred to.

A **relation node** is also typed. It has a *type* (the relation name like `Loves`, `MotherOf`, `Owns`) and an **arity** — the number of concept-positions it connects. Crucially, relations are **n-ary**, not just binary. The relation `Between(x, y, z)` ("x is between y and z") has arity 3 — that's not expressible as chained binary relations without losing information.

Concept nodes connect to relation nodes through *argument positions*. If `Loves` has arity 2, then it has position-0 (the lover) and position-1 (the loved one). The arc from concept to relation carries the position — that's how `Loves(John, Mary)` is distinguished from `Loves(Mary, John)` in the graph.

28.3.2 Bracket notation

Axioma supports Sowa's **bracket notation** directly. Concept nodes are written `[Type: "referent"]`, relation nodes are written `(RelationType)`, and arcs between them are written `→`. The whole graph is wrapped in `<<<` and `>>>`.

```
g: <<<[Person: "John"] → (Loves) → [Person: "Mary"]>>>
println("Concepts:", g show concepts)
println("Relations:", g show relations)
println("Arcs:", g show arcs)
```

Running this produces:

```
Concepts: [{id: "...", referent: "John", referent_type: "individual", type: "Person"},
           {id: "...", referent: "Mary", referent_type: "individual", type: "Person"}]
Relations: [{arity: 2, id: "...", type: "Loves"}]
Arcs: [{from: "...john-id...", position: 0, to: "...loves-id..."},
       {from: "...mary-id...", position: 1, to: "...loves-id..."}]
```

The `g show concepts` / `g show relations` / `g show arcs` queries are Axioma's natural-language introspection form for CGs. They return arrays of records — easy to iterate, easy to dump as JSON, easy to feed into the next processing stage.

28.3.3 Generic vs. individual concepts

Bracket notation distinguishes:

- **Individual** concepts — `[Person: "John"]` — refer to a specific named individual.
- **Generic** concepts — `[Animal]` (no colon) — refer to *some* member of the type, unspecified.

This matters because the meaning of a CG is *different* in the two cases. `[Animal] → (IsA) → [Mammal]` is a *universal* statement ("animals are mammals" — or more strictly, "there exists an animal-that-is-a-mammal"). `[Animal: "Fido"] → (IsA) → [Mammal]` is a *particular* statement ("Fido is a mammal").

```
universal:   <<<[Animal] → (IsA) → [Mammal]>>>
particular:  <<<[Animal: "Fido"] → (IsA) → [Mammal]>>>
```

28.3.4 Chained relations

CGs naturally express *chains* of related concepts:

```
tax_chain: <<<[Cat: "Yojo"] → (IsA) → [Mammal] → (IsA) → [Animal]>>>
println("Concepts:", tax_chain show concepts)
```

Three concepts (`Cat: "Yojo"`, `Mammal`, `Animal`), two relation instances (both `IsA`, arity 2). Yojo is a Cat, Cat is a Mammal, Mammal is an Animal — and the graph makes the chain *visible as structure*, not as inference.

(Note: `taxonomy` and `chain` are both reserved words in Axioma — they live in the concept-formation vocabulary — so this chapter uses neutral names like `tax_chain` when binding a CG variable.)

28.3.5 Worked example — kinship

A small kinship graph:

```
family: <<<[Person: "Alice"] → (MotherOf) → [Person: "Bob"]>>>
println("Concepts:", family show concepts)
println("Relations:", family show relations)
println("Arcs:", family show arcs)
```

This says, in one line: Alice is the mother of Bob.

To express more relationships, build more graphs and combine them. The CG formalism doesn't *require* you to cram everything into one graph — it's natural to keep each fact as a small graph and let the relation-store do the indexing.

28.3.6 The FOL bridge

Every CG has a *corresponding first-order logic formula*. Sowa's original 1976 paper proves the formalisms are expressively equivalent — they're two notations for the same logical content. The translation is mechanical:

CG	FOL
`[Person: "John"]`	`Person(John)`
`[Animal]`	`∃x. Animal(x)`
`[Person: "J"] → (Loves) → [Person: "M"]`	`Person(J) ∧ Person(M) ∧ Loves(J, M)`
`[Animal] → (IsA) → [Mammal]`	`∃x. ∃y. Animal(x) ∧ Mammal(y) ∧ IsA(x, y)`

Because of this bridge, the rules you wrote in Chapter 21 (<=, <~~, ==>, ~~>) can reason over CG-derived facts just as readily as over relation-form facts. The CG is a *prettier* surface for the same underlying logical content.

28.3.7 When to use Sowa CGs

Reach for a CG when:

- The relation is **n-ary** (arity > 2 carries information that pairwise edges lose).
- The nodes need **typing** (Person vs. Animal vs. Country).
- You want a **graphical surface** that maps cleanly to FOL.
- You're moving between **semantic-network and logical** representations — CGs are the bridge.

If your question is structural ("shortest path"), use a plain graph. If your question is "is this proposition true in *some* context but not *another*?", use a context graph (§28.5).

28.4 Peirce Existential Graphs

In 1896, Charles Sanders Peirce invented a **diagrammatic notation for logic** that he called *existential graphs*. Peirce's claim — vindicated by 20th-century logicians — was that EGs are not just an alternative

to algebraic logic but a *better* one for revealing what implication *means*. His student Frederik Stjernfelt called EGs "the most elegant logical system ever devised."

28.4.1 The Sheet of Assertion

In Peirce's system, the work surface is a literal **sheet of paper** — the *Sheet of Assertion*. Anything written on the sheet is asserted to be true. Two statements written side-by-side are *conjoined* — the sheet expresses logical AND by adjacency.

```
sheet: eg_alpha("Two facts on one sheet")
p: eg_predicate("Sunny", 0, 100.0, 100.0)
q: eg_predicate("Warm", 0, 300.0, 100.0)
eg_add_predicate(sheet, p)
eg_add_predicate(sheet, q)
println(eg_format(sheet))
```

This asserts *Sunny AND Warm* — two predicates living on the same sheet at the same nesting depth.

28.4.2 Cuts — negation as enclosure

Peirce's most striking move: **negation is an enclosed oval**. To assert *not P*, draw a closed curve (a *cut*) around the predicate `P`. Anything inside a cut is *denied* by the sheet.

```
sheet2: eg_alpha("Not raining")
rain: eg_predicate("Raining", 0, 100.0, 100.0)
eg_add_predicate(sheet2, rain)
cut: eg_cut(80.0, 80.0, 150.0, 50.0)
eg_add_cut(sheet2, cut)
println(eg_format(sheet2))
```

A predicate `Raining` inside a cut asserts *not Raining*. The cut is the negation operator made visible.

Double negation cancels. A predicate enclosed in two nested cuts is back to being asserted. Peirce called this the *double-cut elimination rule* — one of the five basic inference rules for EGs.

28.4.3 Lines of identity (Beta graphs)

The propositional fragment (just predicates and cuts) is called **Alpha**. To get full first-order logic, Peirce added **lines of identity** — heavy lines connecting two predicate-arguments that share an individual.

In Axioma, beta graphs are constructed with `eg_beta`:

```
beta: eg_beta("All humans are mortal")
h: eg_predicate("Human", 1, 100.0, 200.0)
m: eg_predicate("Mortal", 1, 300.0, 200.0)
eg_add_predicate(beta, h)
eg_add_predicate(beta, m)
```

```
println(eg_format(beta))
```

A line connecting the argument of `Human(x)` to the argument of `Mortal(x)` says: *the x in Human is the same x as in Mortal.* The line is Peirce's way of expressing variable-sharing without writing a variable letter.

28.4.4 Modal operators (Gamma graphs)

Peirce went on to add **dashed cuts** for modal operators — necessity, possibility, knowledge, belief. This level is called **Gamma**.

```
gamma: eg_gamma("Modal logic")
p: eg_predicate("Lives_underwater", 1, 200.0, 200.0)
eg_add_predicate(gamma, p)
modal_cut: eg_cut(180.0, 180.0, 180.0, 50.0, "dashed")
eg_add_cut(gamma, modal_cut)
println(eg_format(gamma))
```

A dashed cut wraps a modal claim: this graph reads "it is *possible* that something lives underwater" (or under a different convention, "necessarily…"). The exact modal reading depends on the cut style and the surrounding context.

28.4.5 Worked example — encoding modus ponens

Modus ponens — *from P and P→Q, infer Q* — looks dramatic in EG notation because **implication is not a primitive**. In EG, `P → Q` is expressed as `¬(P ∧ ¬Q)`, which is *a cut containing P and an inscribed cut around Q*.

```
mp: eg_alpha("Modus ponens: if P then Q")
p_pred: eg_predicate("P", 0, 100.0, 100.0)
q_pred: eg_predicate("Q", 0, 300.0, 100.0)

# The structure is: outer cut around (P + inner cut around Q)
outer_cut: eg_cut(50.0, 50.0, 400.0, 100.0)
inner_cut: eg_cut(250.0, 80.0, 120.0, 50.0)

eg_add_predicate(mp, p_pred)
eg_add_predicate(mp, q_pred)
eg_add_cut(mp, outer_cut)
eg_add_cut(mp, inner_cut)

println(eg_format(mp))
```

Reading: "not (P and not-Q)" — equivalently, "if P then Q". Every implication in EG is a double-cut structure.

28.4.6 The five inference rules

Peirce's *complete* inference system for EGs has only five rules. Each is a *visual transformation*:

Rule	What it does
Insertion	At an odd depth, insert anything. (Logic on negative areas allows weakening.)
Erasure	At an even depth, erase anything. (Logic on positive areas allows weakening from premise to conclusion.)
Iteration	Copy a graph element into a nested area. (Reuse a premise inside a deeper context.)
Deiteration	Erase an iterated copy. (The inverse of iteration.)
Double-cut	Two concentric cuts with nothing between them can be added or removed. (Double negation cancels.)

Axioma's `eg_insert_cut`, `eg_erase_cut`, and `eg_iterate` implement three of these directly. Together they make EGs not just a notation but an *executable logic* — you can perform inference by visual transformation, which is what makes Peirce's system so striking.

28.4.7 When to use EGs

Reach for an EG when:

- You want to **teach** logic. The visual nature of cuts makes implication and negation *intuitive* in a way the algebraic notation isn't.
- You want a **diagrammatic** representation that survives printing — EGs work on whiteboards and slide decks where algebraic formulas blur.
- You want a logically **complete** but visually **minimal** system. Five rules, three primitives (predicate, cut, line of identity).

If your reasoning needs to plug into Axioma's strict- and-defeasible rule engine, use the relation/rule form from Chapter 21 — EGs are a *parallel* logic surface, not a feeder for the main rule pipeline.

28.5 Context Graphs

A **context graph** is John McCarthy's `ist(c, p)` formalism made first-class: every assertion lives inside a *named context*, and the same proposition can carry different truth values in different contexts. Context graphs are the workhorse of multi-perspective reasoning in Axioma and the foundation of Cascade's source- provenance layer.

28.5.1 McCarthy's `ist(c, p)`

McCarthy proposed `ist(c, p)` as a primitive that reads "in context `c`, proposition `p` is true." This single primitive turns out to be enough to express:

- **Beliefs** — `ist(alice_believes, P)` says Alice believes P, without committing the system itself to P.

- **Source provenance** — `ist(cnn_reports, P)` vs. `ist(reuters_reports, P)` lets two news sources disagree without contradicting the system.
- **Modal scope** — `ist(under_hypothesis_H, P)` lets you reason hypothetically.
- **Temporal slices** — `ist(year_1990, P)` vs. `ist(year_2025, P)` lets the same proposition change truth value over time.
- **Layered ontologies** — Multi-Context Systems (MCS) and Contextualized Knowledge Repositories (CKR) are built on `ist(c, p)`.

28.5.2 Creating a context graph

```
g: context_graph("worldmodel", "World Model")
fiction:  context_create(g, "fiction",  "Fiction",  "domain",      "b4")
reality:  context_create(g, "reality",  "Reality",  "domain",      "b4")
```

The four positional arguments to `context_create` are:

1. The host graph
2. A short ID
3. A human-readable label
4. A *context type* — one of `temporal`, `epistemic`, `source`, `hypothetical`, `domain`
5. A *logic kind* — `b4` (Belnap), `kleene`, `luka` (Łukasiewicz), or `bool`

Each context can therefore carry its own *logic system*. Use B4 when truth values can be paraconsistent. Use Kleene K3 when "unknown" is meaningful. Use Łukasiewicz L3 when you need continuous truth.

28.5.3 Asserting facts in a context

```
context_assert(g, "fiction", "sherlock", "lives_at_baker_street", "true",  1.0)
context_assert(g, "reality", "sherlock", "lives_at_baker_street", "false", 0.99)
```

The signature of `context_assert` is:

```
context_assert(graph, ctx_id, element_id, fact, truth, confidence)
```

Where `truth` is one of `"true"`, `"false"`, `"both"` (B4 paraconsistent), or `"neither"` (B4 gappy), and `confidence` is a float in `[0, 1]`.

So we've recorded that Sherlock lives at Baker Street is **true in fiction** and **false in reality**. No contradiction — different contexts.

28.5.4 The DAG hierarchy

Contexts form a directed acyclic graph. A child context inherits its parent's assertions; this is what makes multi-context reasoning scale beyond hand-curated tables.

```
g: context_graph("hier", "Intelligence Hierarchy")
global_ctx: context_create(g, "global",  "Global", "domain",    "b4")
```

```
western:    context_create(g, "western", "Western Sources",     "epistemic", "b4")
eastern:    context_create(g, "eastern", "Eastern Sources",     "epistemic", "b4")
cia:        context_create(g, "cia",     "CIA",                 "epistemic", "b4")
mossad:     context_create(g, "mossad",  "Mossad",              "epistemic", "b4")

context_set_parent(g, "western", "global")
context_set_parent(g, "eastern", "global")
context_set_parent(g, "cia",     "western")
context_set_parent(g, "mossad",  "eastern")
```

The graph now has a tree:

```
global
├── western
│    └── cia
└── eastern
     └── mossad
```

28.5.5 Inheritance via projection

When you `context_project` a context, you get its own assertions *plus* all inherited assertions from its ancestors:

```
context_assert(g, "global", "iran", "nuclear program active",  "true", 0.95)
context_assert(g, "western", "iran", "breakout < 6 months",     "true", 0.70)
context_assert(g, "cia",    "iran", "enrichment at 60%",        "true", 0.90)
context_assert(g, "mossad", "iran", "enrichment at 60%",        "true", 0.95)
context_assert(g, "mossad", "iran", "breakout < 6 months",      "false", 0.85)

println(context_project(g, "cia"))
```

This prints the CIA's *full picture* — its own assertions (`enrichment at 60%`), Western's (`breakout < 6 months`), and Global's (`nuclear program active`). No need to merge by hand.

28.5.6 Lift and lower

Two complementary operations move facts between contexts:

- **`context_lift(g, from, to, element, fact)`** — *promotes* a fact upward. Use when one analyst's assessment becomes authoritative for everyone.
- **`context_lower(g, from, to)`** — *propagates* parent defaults to a child. Use when you want a child context to inherit explicitly (so subsequent edits to the parent don't affect the child).

```
context_lift(g, "mossad", "global", "iran", "enrichment at 60%")
println(context_project(g, "global"))
# Now Mossad's enrichment assessment is global.

lowered: context_lower(g, "global", "eastern")
```

```
println("Lowered " + str(lowered) + " facts to eastern context")
```

28.5.7 Resolution under disagreement

What is the *true* truth of a proposition that's claimed in multiple contexts with different values? Axioma's `context_resolve` collapses across contexts using Belnap's *knowledge join*:

```
println(context_resolve(g, "iran", "breakout < 6 months"))
# Western says TRUE @0.70, Mossad says FALSE @0.85
# → B both (85% confidence)
```

A `true` and a `false` from two contexts resolve to `both` (the Belnap paraconsistent value). Resolution *preserves* the disagreement rather than averaging it away — that's the right behavior for intelligence fusion, where "two sources disagree" is itself a fact you want to act on.

The other resolution strategies — *credulous* (highest-confidence wins), *skeptical* (any disagreement → unknown), *prioritized* (defer to a specific context) — are configurable via the `context_resolve` extended-form signature.

28.5.8 Diff and merge

Two further operations support multi-context workflows:

- **`context_diff(g, c1, c2)`** — returns three lists: facts only in `c1`, only in `c2`, and *conflicting* (same fact, different truth values).
- **`context_merge(g, c1, c2, new_id, strategy)`** — creates a new context combining `c1` and `c2`. The strategy is one of `"union"`, `"intersection"`, `"priority"`, or `"b4_join"`.

Use `context_diff` for *audit trails* — what changed between analyst reports? Use `context_merge` for *synthesis* — what's the union of CIA and Mossad's assessments?

28.5.9 When to use context graphs

Reach for a context graph when:

- The same proposition can have **different truth values in different settings**.
- You need to model **beliefs**, **sources**, **hypotheses**, or **temporal slices** without losing the per-context information.
- You want **Belnap B4 truth** preserved through reasoning — paraconsistent contradictions live naturally in contexts.
- You're building a **knowledge graph with provenance** — Cascade is the canonical example.

If you don't need multi-context reasoning, plain graphs and CGs are simpler. Context graphs are *the* answer when the question is "according to whom?"

28.6 Choosing the right graph

The taxonomy resolves to a four-row decision table:

Need	Use
Shortest path / centrality / clustering / cycles	**Plain graphs** (Chapter 27)
Typed concepts + n-ary relations + FOL bridge	**Sowa Conceptual Graphs**
Diagrammatic logic + negation as enclosure	**Peirce Existential Graphs**
Multi-context truth + beliefs / sources / hypotheses	**Context Graphs**

These are *complementary*, not competing. A real knowledge-representation system uses several of them side-by-side. Cascade — Volume III — uses **plain graphs** for traversal and centrality, **CGs** for typed entity-relation extraction from natural language, and **context graphs** for source provenance.

28.7 Compositions — when you need more than one

A few patterns appear repeatedly in practice:

- **Plain + Context.** Run BFS over a plain graph; record the truth values of the edges in a context graph. Each context represents one analyst's view; the plain graph carries the structure.
- **CG + Context.** A CG captures the *structure* of a claim ("Acme acquired Beta on date D for amount A"); the context graph records *whether the claim is true* per source. CGs supply the schema; contexts supply the truth.
- **EG + Cascade reasoning.** Use EGs for *teaching* the formal logic that underlies Cascade's reasoning rules; use Cascade's actual rule engine to do the inference at scale.

The four graph kinds are *layers* — pick the layer that matches the question, and let other layers contribute what they're best at.

28.8 Exercises

28.1 — Build a CG by hand

Build a Sowa CG `family` representing *Mary is the mother of Alice*, using concepts of type `Person` and a relation `MotherOf`. Print its concepts, relations, and arcs.

Solution:

```
family: <<<[Person: "Mary"] → (MotherOf) → [Person: "Alice"]>>>
println("Concepts:", family show concepts)
println("Relations:", family show relations)
println("Arcs:", family show arcs)
```

Expected (IDs vary):

```
Concepts: [{id: "...", referent: "Mary", referent_type: "individual", type: "Person"},
           {id: "...", referent: "Alice", referent_type: "individual", type: "Person"}]
Relations: [{arity: 2, id: "...", type: "MotherOf"}]
Arcs: [{from: "<mary>", position: 0, to: "<motherof>"},
       {from: "<alice>", position: 1, to: "<motherof>"}]
```

28.2 — A taxonomy chain

Build a CG showing the chain *Yojo* → *IsA* → *Cat* → *IsA* → *Mammal* → *IsA* → *Animal*. Count how many concepts and relations it has.

Solution:

```
tax_chain: <<<[Cat: "Yojo"] → (IsA) → [Mammal] → (IsA) → [Animal]>>>
chain_concepts: tax_chain show concepts
chain_relations: tax_chain show relations
println("Concepts:", len(chain_concepts))
println("Relations:", len(chain_relations))
```

Expected: 3 concepts (`Yojo`, `Mammal`, `Animal` — note that `Cat` is the *type* of the Yojo concept, not a concept itself), 2 relation instances (both `IsA`).

28.3 — Express *not P* in an EG

Build an alpha EG asserting *not P*. Use one predicate and one cut. Print the graph.

Solution:

```
neg: eg_alpha("Not P")
p: eg_predicate("P", 0, 100.0, 100.0)
eg_add_predicate(neg, p)
cut: eg_cut(80.0, 80.0, 100.0, 50.0)
eg_add_cut(neg, cut)
println(eg_format(neg))
```

Expected: a graph with 1 predicate `P/0` and 1 cut. The cut around the predicate is the negation operator. The formatted output shows the predicate at one depth and the cut at the same depth, which together encode "*not P*."

28.4 — Encode modus ponens in EG

Build an alpha EG that encodes `P → Q` using only predicates and cuts (no implication primitive). Use two predicates and two cuts.

Solution:

```
mp: eg_alpha("If P then Q")
p_pred: eg_predicate("P", 0, 100.0, 100.0)
q_pred: eg_predicate("Q", 0, 300.0, 100.0)

outer_cut: eg_cut(50.0, 50.0, 400.0, 100.0)
inner_cut: eg_cut(250.0, 80.0, 120.0, 50.0)

eg_add_predicate(mp, p_pred)
eg_add_predicate(mp, q_pred)
eg_add_cut(mp, outer_cut)
eg_add_cut(mp, inner_cut)

println(eg_format(mp))
```

The structure encodes `¬(P ∧ ¬Q)`, which is `P → Q`. The outer cut means "not (the conjunction inside)"; the inner cut around `Q` means "not Q"; their combination means "not (P and not-Q)" — which is exactly material implication.

28.5 — Multi-context Sherlock

Build a context graph with two contexts (`fiction` and `reality`). Assert that *Sherlock lives at Baker Street* is true in `fiction` and false in `reality`. Project each context and print what it sees.

Solution:

```
g: context_graph("sherlock_graph", "Sherlock World Model")
fic: context_create(g, "fiction", "Fiction",  "domain", "b4")
rea: context_create(g, "reality", "Reality",  "domain", "b4")

context_assert(g, "fiction", "sherlock", "lives_at_baker_street", "true",  1.0)
context_assert(g, "reality", "sherlock", "lives_at_baker_street", "false", 0.99)

println("Fiction view:")
println(context_project(g, "fiction"))
println("Reality view:")
println(context_project(g, "reality"))
```

Expected: fiction shows the proposition as `true`, reality shows it as `false`. Both are simultaneously recorded; neither overrides the other.

28.6 — Resolving a disagreement

Extend Exercise 28.5: add a third context (`literary_critic`) that asserts the proposition as `both` (paraconsistent — the critic sees Sherlock as fictionally-living and historically-not-living *at the same time*). Then call `context_resolve` over all three and report the result.

Solution:

```
g: context_graph("sherlock_three", "Three Views")
fic:  context_create(g, "fiction",         "Fiction",    "domain", "b4")
rea:  context_create(g, "reality",         "Reality",    "domain", "b4")
crit: context_create(g, "literary_critic", "Critic",     "domain", "b4")

context_assert(g, "fiction",         "sherlock", "lives_at_baker_street", "true",
 ↪  1.0)
context_assert(g, "reality",         "sherlock", "lives_at_baker_street", "false",
 ↪  0.99)
context_assert(g, "literary_critic", "sherlock", "lives_at_baker_street", "both",
 ↪  0.85)

println("Resolution:")
println(context_resolve(g, "sherlock", "lives_at_baker_street"))
```

Expected: the resolution returns a Belnap **B both** value, reflecting that the proposition is *simultaneously* true (in fiction) and false (in reality). The B4 knowledge join is the principled way to combine contradictory truth values without collapsing them to "unknown" or picking one arbitrarily.

28.9 Reflection — graphs as a typed alphabet

The four-kind taxonomy is, in the end, a recognition that "graph" is a *family* of formalisms, each tuned to a different question. Internalize three things.

One — plain graphs do structure, not semantics. A plain graph tells you *how things are connected*, not *what they mean*. The moment your question becomes "what *kind* of entity is this?" or "is this *true*?", you've left plain-graph territory.

Two — Sowa CGs are the FOL bridge. When you need typed concepts and n-ary relations, CGs give you a clean surface that maps mechanically to FOL. Anything you can write as a CG, you can prove about with the rule engine from Chapter 21.

Three — context graphs are the source of *truth-per-perspective*. The single hardest engineering problem in knowledge representation is *whose truth?* Context graphs answer it explicitly. Every fact has a context; resolution across contexts is principled (B4 knowledge join) and *preserves* disagreement.

Peirce's existential graphs are the outlier of the four — they're not for *storing* knowledge but for *reasoning visually about logic itself*. They're the right tool when you're teaching, when you need to think through an inference move on a whiteboard, or when you want to show non-mathematicians what "implication" really *means*.

The next chapter takes a parallel turn: when knowledge is *geometric* rather than relational, you reach for **conceptual spaces** (Chapter 29) — Gärdenfors's formalism for prototypes, similarity, and convex regions in quality dimensions. Different geometry, same idea: the right structure for the right question.

> *"The graph is not the territory."* — Borrowed from Korzybski. Different graph formalisms map different parts of the territory. Axioma lets you keep the territory; the choice of formalism is

the choice of map.

Solutions to selected exercises: Chapter 28 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch28_solutions.md.)*

Chapter 29 · Conceptual Spaces (Gärdenfors)

What this chapter is. Peter Gärdenfors's *conceptual spaces* are a **geometric** alternative to the graph-based KR you saw in Chapter 28. Where Sowa's CGs ask *which entity relates to which*, Gärdenfors asks *where each entity sits in a continuous space of qualities*. The payoff is that **similarity becomes distance** and **categories become regions** — both intuitions that graphs handle awkwardly. Axioma exposes ~15 builtins for creating quality dimensions, anchoring prototypes, defining convex regions, and asking distance and membership questions. This chapter walks the API, the theory, and the cases where graph thinking *can't* answer what spatial thinking answers in one query.

29.1 The premise — geometry over taxonomy

Gärdenfors's 2000 book *Conceptual Spaces: The Geometry of Thought* argued that human concepts have *gradient* structure that propositional and graph-based representations can't capture. Three observations drove the framework:

1. **Some category members are *more typical* than others.** A robin is a more prototypical bird than a penguin. A chair is more typical *furniture* than a bean-bag. Classical set theory treats both as full members; humans clearly don't.
2. **Similarity has a gradient.** Red is more similar to orange than to green. *Hot* and *warm* are closer than *hot* and *cold*. Graphs don't naturally express degree-of-difference.
3. **Categories have *fuzzy* boundaries.** Where exactly does *red* end and *orange* begin? Where does a *cup* become a *mug* become a *bowl*? Sharp set membership gives the wrong answer.

Gärdenfors's solution: model meaning as **geometry**. A concept lives in a **quality-dimension space**. Colors live in HSV (hue, saturation, value). Sounds live in (pitch, loudness, timbre). Tastes live in (sweet, sour, bitter, salty, umami). A *category* is a **convex region** of that space, centered on a **prototype**. Similarity is **distance**. Membership is **degree of containment**.

This is not a metaphor: the framework is operational, and Axioma exposes it as a working API with normalized metrics, circular dimensions, fuzzy membership, and interpolation along the manifold.

29.2 The functional API

Axioma's conceptual-spaces vocabulary is a *functional* surface, not a natural-language one. (The `ColorSpace extends ConceptualSpace` form from the pilot draft does not actually work — see §29.10 for the gap.) You build a space step-by-step with builtin calls:

```
color: conceptual_space("Color", "perceptual HSV space")
add_dimension(color, "Hue", 0, 360, "circular", true)
add_dimension(color, "Saturation", 0, 100)
add_dimension(color, "Value", 0, 100)
add_prototype(color, "Red",    [0,   100, 50], "salience", 1.0)
add_prototype(color, "Green",  [120, 100, 50], "salience", 1.0)
add_prototype(color, "Blue",   [240, 100, 50], "salience", 1.0)
add_prototype(color, "Yellow", [60,  100, 50], "salience", 0.9)
```

Three subtleties hidden in those eight lines:

- **Keyword args.** `add_dimension` and `add_prototype` take *keyword* options after the required positional args. `"circular", true` and `"salience", 0.9` are key/value pairs. Passing a bare 5th positional (e.g. `add_dimension(..., true)`) is **silently ignored** — no error, just the default. This is a UX trap; see the notes after each table below.
- **`conceptual_space(name, desc)` constructs a value, not a Concept.** You can't use dot notation (`color.dimensions` errors). Use the `*_of` builtins instead: `dimensions_of(color)`, `prototypes_of(color)`, etc.
- **Coordinates are arrays of floats** matched positionally to dimensions in order of declaration. Mismatched length is a hard error.

29.3 The full API surface

The 15 working builtins:

Builtin	Signature	Returns
`conceptual_space(name, [desc])`	Construct a new space	`*ConceptualSpace`
`add_dimension(s, name, min, max, [kw...])`	Add a quality dimension	Null (mutates)
`add_prototype(s, name, coords, [kw...])`	Anchor a prototype point	Null (mutates)
`add_region(s, name, type, definition)`	Add a named region	Null (mutates)
`dimensions_of(s)`	Array of dimensions	`Array<Dimension>`
`prototypes_of(s)`	Array of prototypes	`Array<Prototype>`
`regions_of(s)`	Array of regions	`Array<Region>`
`space_to_concepts(s)`	All prototypes as records	`Array<Record>`
`distance(s, p1, p2)`	Normalized Euclidean (or circular)	Float
`nearest_prototype(s, point)`	Name of closest prototype	String
`in_prototype(s, point, name)`	Boolean membership (radius-checked)	Boolean

Builtin	Signature	Returns
`prototype_membership(s, point, name)`	Fuzzy membership in [0, 1]	Float
`interpolate(s, p1, p2, t)`	Linear interp at parameter t∈[0,1]	Array
`interpolate_path(s, [points], steps)`	Multi-point smooth path	Array of arrays
`between(s, point, p1, p2)`	Is point between p1 and p2?	Boolean
`visualize_space(s)`	ASCII summary string	String
`space_stats(s)`	Record with counts + metric	Record

Keyword args on `add_dimension`: `"circular", true` (wrap-around metric), `"unit", "kg"` (display only), `"weight", 0.5` (importance multiplier), `"normalized", true` (rescale to [0,1] internally).

Keyword args on `add_prototype`: `"salience", 0.9` (importance multiplier in distance), `"radius", 20` (extent — turns point into a ball).

29.4 Quality dimensions

A **quality dimension** is one axis of variation along which two stimuli can differ. Wavelength is a dimension. Pitch is. Sweetness is. Spatial position is.

The empirical content of the framework lies in the claim that *real* dimensions come in small natural bundles:

- **Color** integrates hue, saturation, value — and hue is *circular* (red wraps back to red through magenta).
- **Sound** integrates pitch, loudness, timbre.
- **Taste** integrates sweet, sour, salty, bitter, umami.

The dimensions in each bundle are *not independent* in any useful psychological sense — moving along one is psychologically distinct from moving along another, and distance has to reflect that.

Axioma represents this with a per-dimension `weight` (salience multiplier) and `circular` flag:

```
mood: conceptual_space("Mood", "Russell circumplex")
add_dimension(mood, "Valence", -100, 100, "weight", 1.0)
add_dimension(mood, "Arousal", -100, 100, "weight", 1.0)
add_prototype(mood, "Excited", [60, 80])    # high val, high aro
add_prototype(mood, "Calm",    [60, -40])   # high val, low aro
add_prototype(mood, "Angry",   [-60, 80])   # low val, high aro
add_prototype(mood, "Sad",     [-50, -50])  # low val, low aro

println(nearest_prototype(mood, [70, 70]))   # → "Excited"
println(nearest_prototype(mood, [-70, -20])) # → "Sad"
```

A *circular* dimension makes distance wrap modularly:

```
println(distance(color, [0, 100, 50], [350, 100, 50]))
# → 0.0278 — very near (hue wraps around through 360)

println(distance(color, [0, 100, 50], [180, 100, 50]))
# → 0.5 — antipode on the hue circle, maximum hue distance
```

Without `"circular", true` the distance would have been `350/360 ≈ 0.97` for the first pair — *wrong*, because 350° and 0° are *both red*. The flag is load-bearing.

29.5 Prototypes and salience

A **prototype** is a named point anchoring a category. It records:

- A **name** (the category label),
- An array of **coordinates** matching the dimensions,
- A **salience** in `[0, 1]` (how much weight this prototype carries in nearest-prototype voting),
- A **radius** in dimension-space units (0 means *point prototype*; positive means *ball-shaped extent*).

Prototype theory comes from Eleanor Rosch's 1970s work on *basic-level categories*. The empirical observation:

- Children acquire *basic-level* categories (dog, chair, bird) before *superordinate* (animal, furniture) or *subordinate* (dachshund, rocking chair).
- People list features faster for basic-level than for other levels.
- Judgments of typicality (*"how good an example of bird is X?"*) are consistent across speakers.

Gärdenfors's bet: those judgments *are* distance from prototype. Axioma's `prototype_membership` computes exactly that, returning a fuzzy value in `[0, 1]`:

```
salmon: [15, 70, 70]              # pinkish color
println(prototype_membership(color, salmon, "Red"))    # approximately 0.81
println(prototype_membership(color, salmon, "Yellow")) # approximately 0.80
println(prototype_membership(color, salmon, "Green"))  # 0.7650993858356716
```

Membership decays smoothly with distance, so two prototypes can both have *partial* claim on a point — the mathematical model of *fuzzy categorization*.

29.6 Convex regions

Some categories aren't a single point. *Warm colors* isn't a prototype — it's a *region* covering reds, oranges, yellows. *Edible* isn't centered at one taste — it covers a wide convex slice of the taste manifold.

`add_region` records a named region with a definition:

```
cs: conceptual_space("Color2D", "")
add_dimension(cs, "Hue", 0, 360, "circular", true)
add_dimension(cs, "Saturation", 0, 100)
```

```
# A convex region: hue in 30°–90°, saturation > 50
add_region(cs, "Yellowish", "convex", [[30, 90], [50, 100]])

# A radial region: ball of radius 50 around a point
add_region(cs, "NearRed", "ball", [0, 100])

println(regions_of(cs))
# → [Region(Yellowish: convex), Region(NearRed: ball)]
```

The `type` argument is a string tag (`"convex"`, `"ball"`). The `definition` is a list whose interpretation depends on the type. **Caveat:** region semantics are not yet fully load-bearing in queries — `regions_of` returns them, but there's no `in_region` builtin yet (the closest analogue is `in_prototype` with `radius` set, which uses a ball not a polytope).

29.7 Distance and similarity

The metric on a conceptual space is **normalized Euclidean**:

$$d(p, q) = \frac{1}{\sqrt{n}} \sqrt{\sum_i w_i (p_i - q_i)^2}$$

where `n` is the number of dimensions, `w_i` is the salience weight on dimension *i*, and the sum is taken over all dimensions — with the special rule that *circular* dimensions use `min(|p − q|, range − |p − q|)` in place of the bare difference.

Why normalized. Spaces with different numbers of dimensions become comparable. A distance of `0.5` in a 2D space and a distance of `0.5` in a 5D space mean roughly the same thing — both are "half-range".

Why domains matter. The original *Conceptual Spaces* book distinguishes integral dimensions from separable ones. Axioma now exposes that distinction with `add_domain`: dimensions grouped into a quality domain are measured inside the domain first, then domain distances are combined by weighted sum. A space with no declared domains keeps the old flat Euclidean default.

Why `space_similarity()` has an explicit name. Axioma keeps concept-level `similarity(concept1, concept2)` for prototype/ontology comparisons, and uses `space_similarity(space, p1, p2)` for coordinate geometry. The same split applies to `typicality(concept)` versus `space_typicality(space, point)`.

29.8 Interpolation and betweenness

Two builtins that have no graph-based analogue:

`interpolate(s, p1, p2, t)` — gives you the point at parameter $t \in [0, 1]$ on the straight line from p1 to p2. $t = 0$ returns p1, $t = 1$ returns p2, $t = 0.5$ the midpoint.

```
mid: interpolate(color, [0, 100, 50], [120, 100, 50], 0.5)
println(mid)   # → [60, 100, 50] — green-yellow midpoint
println(nearest_prototype(color, mid))   # → "Yellow"
```

The point between Red and Green is Yellow. **That's a content claim about the perceptual manifold.** It's true *because* hue is circular and we set the prototypes at the canonical RGB primaries — geometry expresses the empirical fact.

`between(s, point, p1, p2)` — Boolean: is `point` on or near the line from `p1` to `p2`?

```
println(between(color, [60, 100, 50], [0, 100, 50], [120, 100, 50]))
# → true (Yellow IS between Red and Green)

println(between(color, [240, 100, 50], [0, 100, 50], [120, 100, 50]))
# → false (Blue is not between Red and Green)
```

`interpolate_path` generalizes to multi-point sequences, useful for animation or smooth scrubbing through a gradient.

29.9 Worked example — categorizing tastes

Build a 3D taste space, anchor four prototypes, then classify novel points:

```
taste: conceptual_space("Taste", "primary tastes")
add_dimension(taste, "Sweet",  0, 100)
add_dimension(taste, "Sour",   0, 100)
add_dimension(taste, "Bitter", 0, 100)

add_prototype(taste, "Lemon",     [15, 95,  5], "salience", 1.0)
add_prototype(taste, "Coffee",    [10,  5, 90], "salience", 1.0)
add_prototype(taste, "Honey",     [95,  5,  5], "salience", 1.0)
add_prototype(taste, "DarkChoc",  [50, 10, 70], "salience", 0.8)

# A mystery item with moderate sweet, low sour, high bitter
mystery: [40, 15, 75]
println(nearest_prototype(taste, mystery))          # → "DarkChoc"
println(prototype_membership(taste, mystery, "DarkChoc"))  # 0.9317314234233945
println(prototype_membership(taste, mystery, "Coffee"))    # 0.8170356137292041
println(prototype_membership(taste, mystery, "Lemon"))     # 0.5323383419690717

# A blend point on the line from Lemon to Honey at 30%
lh_blend: interpolate(taste, [15, 95, 5], [95, 5, 5], 0.3)
println(lh_blend)                                    # [39.0, 68.0, 5.0]
println(nearest_prototype(taste, lh_blend))          # → "Lemon" still
```

Notice how *one query* answers what graph KR would need a custom rule for: "which named category is this point *most like*?" The geometric formalism gives the answer directly.

29.10 Honest limits and gaps

Four gaps worth knowing about:

1. **The natural-language surface is broken.** The pilot draft suggested `ColorSpace create` / `extends ConceptualSpace` / `has dimension: Hue from 0 to 360, circular`. None of those actually work today — the ConceptualSpace-as-Concept path errors out. Use the functional API throughout.
2. **Region semantics are partial.** `add_region` and `regions_of` work, but there's no `in_region` builtin. Use `in_prototype(s, p, name)` with `radius` set for ball regions; convex polytopes have no query yet.
3. **5th positional arg silently ignored.** Both `add_dimension` and `add_prototype` switch to keyword args after the required positionals. Passing `add_prototype(s, name, coords, 0.9)` silently uses salience 1.0, not 0.9. You must write `"salience", 0.9`.
4. **`in_prototype` returns over-generously.** The radius is normalized by `sqrt(n_dimensions)`, which means a radius of 20 in a 2D space with [0,100] axes effectively covers the whole space. If you want crisp ball regions, use `prototype_membership > threshold` instead.

These are *documentation gaps* and *UX gaps*, not theory gaps — the underlying math is sound, the API just hasn't been polished. Treat the functional core (`distance`, `space_similarity`, `nearest_prototype`, `prototype_membership`, `interpolate`, `between`) as production-ready and the surface around it as a work in progress.

29.11 When to reach for spaces over graphs

Question	Spaces win	Graphs win
"X *is-a* Y?"	—	✓ subclass chains
"X *relates to* Y how?"	—	✓ labeled edges
"How similar is X to Y?"	✓ distance	hard
"Which prototype is X closest to?"	✓ nearest_prototype	needs custom code
"Is X a *typical* member of Y?"	✓ prototype_membership	hard
"Generate a smooth transition X → Y"	✓ interpolate	impossible
"Is Z *between* X and Y in some quality?"	✓ between	impossible
"Which N items *cluster*?"	✓ k-means on coords	needs external metric

Use both. Chapter 28 showed Sowa CGs for *structural* KR. This chapter shows conceptual spaces for *gradient* KR. Production systems like Cascade often run both representations side by side: a graph for "Tesla is a *kind of* Car" and a conceptual space for "this customer's preference vector is closest to Cluster #4".

29.12 Exercises

29.1 — Build the mood space

Construct a 2D conceptual space `Mood` with dimensions `Valence ∈ [−100, 100]` and `Arousal ∈ [−100, 100]`. Add four prototypes: Excited (60, 80), Calm (60, -40), Angry (-60, 80), Sad (-50, -50). Classify three novel points: (70, 70), (0, 0), (-30, 60). Print the nearest prototype and the membership scores against all four.

29.2 — Circular hue matters

Build a 1D `Hue` space with one dimension `Hue ∈ [0, 360]` *circular*. Add prototype `Red` at `[0]`. Compute the distance from Red to each of `[10]`, `[180]`, `[350]`, `[359]`. Then build a *second* 1D space with the same prototypes but `Hue` *not* circular and compute the same four distances. Print both sets, side by side. Explain the difference in one sentence.

29.3 — Interpolation paths

In the taste space from §29.9, generate the path from Lemon to Coffee via `interpolate_path([Lemon, DarkChoc, Coffee], 5)`. Print each waypoint and its `nearest_prototype`. Does the sequence move smoothly through DarkChoc as the midpoint?

29.4 — Fuzzy classification

Build a 3D color space (HSV) with prototypes Red, Green, Blue at the canonical positions and `"radius", 30` each. For ten points spaced evenly along the hue circle (0°, 36°, 72°, …, 324°), print which prototypes claim it (membership > 0.7). Some points should have *no claim*; some should have *multiple*.

29.5 — Salience matters

In a 2D space with dimensions `X` and `Y`, both `[0, 100]`, add two prototypes: `Heavy` at `[10, 10]` with `"salience", 1.0` and `Light` at `[10, 90]` with `"salience", 0.3`. Classify the point `[10, 50]`. Then swap the saliences and reclassify the same point. The nearest prototype should change. Explain why — what exactly does salience do to the metric?

29.6 (open) — Design a conceptual space for your domain

Pick a domain you know well — programming languages, music genres, food cuisines, political stances, customer segments. Sketch:

- 3–5 quality dimensions (with ranges and units)
- 5+ named prototypes with coordinates
- 2 named regions

Then ask the framework three classification questions of real items in your domain. Reflect: where do the prototypes feel right? Where do they feel arbitrary? Where would a *latent* (data-driven) dimension serve better than a human-named one?

Solutions to selected exercises: Chapter 29 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch29_solutions.md.)*

29.13 What you learned

- The Gärdenfors framework: meaning is geometry, concepts are convex regions, similarity is distance.
- The functional API: `conceptual_space`, `add_dimension`, `add_prototype`, `add_region`, plus query builtins `distance`, `nearest_prototype`, `prototype_membership`, `interpolate`, `between`.
- Quality dimensions can be *circular* (hue) or *salience- weighted* (one dimension matters more than another).
- Prototypes are *named points*; convex regions generalize to *named extents*.
- Fuzzy membership (`prototype_membership`) is the cognitively realistic alternative to crisp set inclusion.
- Interpolation and betweenness are first-class — *no graph KR offers them*.
- Where graphs and spaces complement each other in production KR systems.

The next pilot to promote — Ch.31 Lojban-in-Axioma — returns to relational/predicate KR with named places (selbri) and a bridge to FOL. After that, Volume III continues with Cascade-specific chapters that *combine* graphs and spaces with neural signals.

> *"To be is to be the value of a bound variable."* — Quine. Gärdenfors's reply: to be is also to *sit somewhere in a quality space*. Both ontologies pay rent.

Chapter 30 · Gödelization

What this chapter is. Gödelization is the trick Kurt Gödel used in his 1931 incompleteness proofs: encode a piece of syntax — a formula, a proof, an entire program — as a *single large integer*, in a way that the encoding and decoding are themselves computable arithmetic. The trick is the bridge between *symbolic* reasoning ("is this proof valid?") and *arithmetic* reasoning ("does this number satisfy this arithmetic predicate?"). It underlies the incompleteness theorems, the halting problem, the recursion theorem, quine programs, and modern zero-knowledge proofs. This chapter covers the trick, the mathematical move it enables, and the working Axioma surface (`encode` and `decode`) you can use today.

30.1 The encoding trick

Every program is, ultimately, a sequence of symbols. Every symbol can be assigned a unique integer code. Once you have a way to *combine* a sequence of integers into a *single* integer — reversibly — you have a way to encode any program as one big number.

Gödel's original 1931 method used **prime-factor encoding**: assign each symbol an integer code, then encode the sequence $[c_1, c_2, c_3, \ldots, c_n]$ as

$$2^{c_1} \cdot 3^{c_2} \cdot 5^{c_3} \cdot \ldots \cdot p_n^{c_n}$$

where p_n is the n-th prime. By the fundamental theorem of arithmetic, this product is unique, so the sequence can be recovered by repeated division — no information is lost. The cost is staggering integer growth (even a 10-symbol program produces a number with hundreds of digits), but the *principle* is what matters.

Modern variants use base-k representation or pairing functions (Cantor's $\langle x, y\rangle = \frac{1}{2}(x + y)(x + y + 1) + y$) that grow more slowly. Axioma's `encode` builtin uses an internal prime-power variant backed by Go's `big.Int` arbitrary-precision integers — so encodings stay reversible no matter how large.

What's so special about *one* integer?

Two reasons.

First — arithmetic predicates can talk about syntax. Once a program is a number, you can ask arithmetic questions about that number — *"is N a valid encoding?"*, *"is N the encoding of a program that halts on input M?"* — and these questions are about *integers*. They live in the same mathematical universe as `2 + 2 = 4`. That's the bridge.

Second — the encoding-and-decoding operation is itself computable. If you can write a primitive-recursive function `decode(N)` that recovers the original symbols from `N`, then *any* arithmetic system

rich enough to express primitive recursion can, in principle, talk about programs *internally*. That's what makes the self-reference moves work.

30.2 What Axioma exposes

The Gödelization layer in Axioma is **partially shipped**. The minimal API that works in this build:

Builtin	Works?	Purpose
`encode(s)`	[done]	Encode a string-of-Axioma-source as a Gödel number
`decode(g)`	[done]	Decode a Gödel number back to source
`demonstrateIncompleteness()`	[done]	Pedagogical text walkthrough of Gödel's proof

The following are *registered* but currently unreachable or unstable in the user surface:

Builtin	Works?	Why not
`isGödel(g)` / `gödelValue(g)`	[no]	The ö character is rejected by the user-script lexer
`selfEncode()`	[no]	Hangs (encoding pipeline loops on the canned self-reference string)
`diagonalize(s)`	[no]	Integer-parse error on the resulting big number
`createFormalSystem(name)`	[no]	Hangs
`provable(stmt)`	[no]	Hangs
`createGodelSentence([sys])`	[no]	Hangs (depends on `createFormalSystem`)
`proveIncompleteness([sys])`	[no]	Hangs
`searchForProof(stmt, depth)`	[no]	Hangs

The chapter treats the working surface concretely and the broken surface conceptually — when the build is fixed, the chapter's exercises will gain runnable solutions for the formal-system half of the material.

30.3 Working with encode and decode

Two primitives, one inverse pair.

```
gn: encode("x + 1")
println(decode(gn))          # "x + 1" — round-trips
```

A few invariants you can rely on.

30.3.1 Round-trip identity

```
println(decode(encode("hello")) == "hello")    # true
println(decode(encode("42"))    == "42")       # true
println(decode(encode("x")))                   # "x"
```

For any string `s` that `encode` accepts, the round-trip `decode(encode(s))` recovers `s`.

30.3.2 What `encode` accepts

`encode` *parses* its input as Axioma syntax before encoding. Valid Axioma expressions encode cleanly:

```
encode("x")          # ok
encode("42")         # ok
encode("x + 1")      # ok
encode("hello")      # ok — bare identifier counts
encode("true")       # ok
```

Strings that aren't well-formed Axioma may not encode:

```
encode("encode(\"x\")")
# → ERROR: parse errors: expected '=>' or '->' or '→' …
```

This is a deliberate design choice — Gödelization is, at heart, encoding *well-formed syntax*, not arbitrary byte sequences. The encoder doubles as a syntax check.

30.3.3 Distinct expressions, distinct numbers

```
gn_x: encode("x")
gn_y: encode("y")
println(decode(gn_x) == decode(gn_y))       # false
```

Two *different* strings produce *different* Gödel numbers (modulo any AST-normalization the encoder does — see below). The inverse property — that two Gödel numbers are "equal" iff they encode the same expression — is what makes Gödelization useful as a *hash with provenance*.

30.3.4 Validation + byte-level preservation

`encode` parses its input as a syntax check, but *encodes the literal bytes* of the source — whitespace, comments, formatting all preserved. So two equivalent-but-differently- formatted strings produce *different*

Gödel numbers:

```
g1: encode("x + 1")
g2: encode("x   +   1")
println(decode(g1))              # "x + 1"
println(decode(g2))              # "x   +   1" — extra spaces preserved
println(decode(g1) == decode(g2))    # false
```

The parse step is purely a validity guard ("is this parseable as Axioma?"); the encoding itself is over the byte stream. If you wanted *AST-level* fingerprinting (spaces ignored), you'd parse, render the canonical form, *then* encode — a two-step recipe the chapter doesn't build but the principles support.

30.4 The historical move — Cantor → Gödel

Cantor's *diagonal argument* (1891) proved that there are more real numbers than natural numbers by *listing* a hypothetical enumeration of reals, then constructing a new real that differs from the *n*-th listed real in the *n*-th decimal place. The new real can't be on the list — contradicting the hypothesis that the list was complete.

Gödel's 1931 paper *Über formal unentscheidbare Sätze* (*On formally undecidable propositions*) ported Cantor's diagonal to *formal systems*. The translation:

Cantor	Gödel
Enumerate real numbers	Enumerate *formulas* via Gödel numbering
Diagonal disagrees with the *n*-th	*G*: "the formula with Gödel number *n* is not provable"
New real isn't on the list	*G* is true but not in the provable set

The translation requires that the formal system be *rich enough to talk about its own Gödel numbers* — i.e., it must contain enough arithmetic to define primitive recursive functions over the integers. Peano Arithmetic qualifies. Robinson Arithmetic qualifies. Anything weaker might not.

Once you can talk about your own encoding, the diagonal move produces a sentence that *says of itself* that it's unprovable. By case analysis:

- If *G* is provable, then *G* is true, so *G* is not provable — contradiction. So the system is *inconsistent*.
- If *G* is not provable, then *G* is true (it says so), so the system is *incomplete*.

Either way, the system can't be both consistent and complete. **Mathematics cannot be totally mechanized.**

You can see Axioma's pedagogical narration of this move:

```
println(demonstrateIncompleteness())
```

The output walks through the three steps — *create a formal system, construct the Gödel sentence, prove*

incompleteness — and concludes with the canonical implication: *"Any consistent formal system is incomplete."* Use it as a teaching aid.

30.5 The halting problem — Gödelization meets diagonalization

Turing's 1936 *On Computable Numbers* used the same diagonalization-over-Gödel-numbers move to prove the *halting problem* undecidable. The argument:

1. Suppose there were a program `halts(P, I)` that correctly decides whether program `P` halts on input `I`.
2. Encode every program as a Gödel number. Now `halts` is really `halts(g_P, g_I)` — a predicate over integers.
3. Construct `diag(g_P) = if halts(g_P, g_P) then loop else stop`. Diagonalize: ask `diag` about itself, on its own Gödel number.
4. *If* `diag(g_diag)` halts, then `halts(g_diag, g_diag)` says it loops. Contradiction.
5. *If* `diag(g_diag)` loops, then `halts(g_diag, g_diag)` says it halts. Contradiction.

The diagonalization works *because Gödel numbering lets a program take another program as an integer argument*. The trick is the same as Cantor's, applied to programs-as-data instead of reals-as-digits.

This is why the halting problem can't be solved by any algorithm. The undecidability isn't a *limit of human cleverness* — it's a structural consequence of the same move that powers all of programming-language theory.

30.6 Kleene's recursion theorem — self-reference made routine

Stephen Kleene's 1938 *recursion theorem* says that *every computable transformation `f` of program-encodings has a fixed point* — a program `P` such that `f(g_P)` encodes a program that behaves identically to `P` itself. The proof uses Gödel numbering plus a double-substitution trick.

Practical consequence: **self-referential programs are mechanical, not magic.** Once you have Gödel numbering, you can build:

- **Quines** — programs that print their own source.
- **Self-modifying programs** — programs that take their own Gödel number, transform it, and execute the result.
- **Reflective interpreters** — Chapter 22's meta-circular evaluator gets its self-reference for free from Gödelization.

The recursion theorem is *why* the meta-circular evaluator can interpret itself. The fixed-point argument shows that a meaningful self-interpreter must exist; Gödel numbering shows you how to construct it.

30.7 The quine pattern

A **quine** is a program that prints its own source code. The classical recipe, paraphrased:

```
s: "s: ?; println(s with ? replaced by quoted-s)"
println(s with ? replaced by quoted-s)
```

The trick is the *self-substitution*. With Gödel numbering, quines become straightforward: a program that asks for its own Gödel number, decodes it, and prints the result.

Conceptual sketch (the Axioma `selfEncode` builtin that would power this is not yet reliable):

```
# Conceptual — selfEncode is not yet reliable in the current build
me: selfEncode()
my_source: decode(me)
println(my_source)
```

Once `selfEncode` is fixed, this is a 3-line quine. The *mechanism* is what's important: a quine is just a fixed point of the function "given a program-as-number, return the source-as-string." Gödelization makes that fixed point computable.

In the meantime, you can build quines the *classical* way by hand, using string manipulation:

```
# A working quine via string manipulation (no Gödelization)
q: "q: ; println(q)"
# Print q with the literal value of q substituted into the placeholder
# (Exercise — left as Exercise 30.4)
```

Either route works. Gödelization is the *theoretical* route; classical quoting is the *practical* route. They illustrate the same fixed-point principle.

30.8 Where Gödelization shows up in Axioma and Cascade

Three places it's used internally — *each behind the scenes*, none yet exposed to user code in this build:

1. **The meta-circular evaluator** (Chapter 22) — when the evaluator needs to *talk about* an AST it's evaluating, the Gödel number is the canonical handle. Inside the Go implementation, this is plumbed via the `GödelNumberObject` type and `godelEncode`/`godelDecode` helpers in `evaluator/builtins.go`.
2. **Provenance tracking** — Chapter 21's `proof` builtin walks the derivation chain that produced a conclusion. In principle that chain can be Gödel-encoded into a single compact integer for cryptographic commitment; the Cascade roadmap calls for this in the knowledge-graph-export path.
3. **The MCP server's code-analysis tools** (Chapter 34) — `parse_axioma` returns an AST, which can be Gödel-encoded for cross-process commitment. The MCP protocol passes AST shapes (not Gödel numbers) today; the encoder is reserved as a future optimization.

You won't usually call `encode`/`decode` directly in CS1 / CS2 code. The presence of the primitives makes

meta-arithmetic reasoning possible when you do need it, and the *theoretical content* of Gödelization is foundational to understanding what Cascade does at the provenance and reflection layers.

30.9 What to teach with

Several of the formal-system builtins (`selfEncode`, `isGödel`, `diagonalize`, the `createFormalSystem` / `provable` chain) are not reliable in the current build — §30.3's table is the honest inventory.

The right way to *teach* Gödelization is the theoretical treatment in §§30.4–30.7. The right way to *demonstrate* it concretely is the `encode`/`decode` working pair in §30.3 plus the `demonstrateIncompleteness` pedagogical text in §30.4. Anything more — running an actual incompleteness proof end-to-end, building a quine via `selfEncode`, diagonalizing programmatically — is on the roadmap once the formal-system pipeline is fixed.

30.10 Exercises

30.1 — Round-trip identity

Show that `decode(encode(s)) == s` for at least four different valid Axioma expressions. Use `println` to display each round-trip.

Solution:

```
println(decode(encode("42"))    == "42")        # true
println(decode(encode("x"))     == "x")          # true
println(decode(encode("x + 1")) == "x + 1")      # true
println(decode(encode("hello")) == "hello")      # true
```

All four print `true`. The encoder normalizes whitespace via the AST, but for these examples the canonical and input forms agree.

30.2 — Distinct expressions, distinct numbers

Encode two different expressions. Show via `decode` comparison that they produce different Gödel numbers (comparing decodings is the user-script-friendly way to check, since the Gödel number itself is a `big.Int` that doesn't expose easily to user code).

Solution:

```
gn_x: encode("x")
gn_y: encode("y")
println(decode(gn_x) == decode(gn_y))       # false
println(decode(gn_x))                       # "x"
println(decode(gn_y))                       # "y"
```

Different expressions decode to different strings, so their encodings must be distinct.

30.3 — Whitespace is preserved

Show that `encode` preserves whitespace by encoding two syntactically equivalent expressions with different spacing. Compare the decoded outputs.

Solution:

```
g1: encode("x + 1")
g2: encode("x   +   1")   # extra whitespace
println(decode(g1))            # "x + 1"
println(decode(g2))            # "x   +   1" — extra spaces preserved
println(decode(g1) == decode(g2))   # false
```

`encode` parses its input for *validity* — `"x  +  1"` must be syntactically meaningful — but the encoding itself is byte-level. The two expressions produce *different* Gödel numbers because their byte representations differ.

If you wanted whitespace-insensitive fingerprinting, you'd canonicalize before encoding (parse → re-render in a canonical style → encode). That's a two-step recipe; `encode` alone does only step 3.

30.4 — The quine substitution pattern

Demonstrate the *substitution* move at the heart of every quine, using `replace`. We won't build a perfect quine here (getting the escaping byte-exact across all levels of indirection is famously fiddly — see below), but we'll make the *pattern* concrete: a string with a placeholder, where the placeholder is replaced by the string's own quoted form.

Solution:

```
q: "q: X; println(X)"
quoted_q: "\"q: X; println(X)\""
# Replace the placeholder X with the quoted form of the template
prog: replace(q, "X", quoted_q)
println(prog)
```

This prints:

```
q: "q: X; println(X)"; println("q: X; println(X)")
```

The *pattern* is right — `q` plus a `println` of `q`'s quoted form — but notice that the *output* of running this program would not be the input: running the printed program prints `"q: X; println(X)"`, not the full program we just emitted. A genuine quine requires *two-level substitution* (the template substitutes itself into both its own definition *and* the print statement).

The Lisp form makes the recipe transparent because of how it represents code-as-data:

```
((lambda (q) (list q (list 'quote q)))
 '(lambda (q) (list q (list 'quote q))))
```

Building a byte-exact Axioma quine is a worthwhile follow- up exercise; the path is to think of the program

as a pair *(template, data)* where `data` is the template itself, and the run prints `(template, quote(data))`.

(`replace(s, from, to)` is Axioma's standard string-replace builtin; you can confirm with `replace("aXa", "X", "b")` producing `"aba"`.)

30.5 — Hand-encode "P" with small primes

Without using Axioma at all — on paper — encode the single-character string "P" using the classical prime-power method. Assume the symbol "P" has code `16` (the position of `P` in the Latin alphabet). What is the Gödel number?

Solution:

A single symbol with code `c` encodes as `2^c` (one symbol, the first prime is 2). So `P → 2^16 = 65536`.

The point of doing this by hand is to feel *how fast the numbers grow*. A two-symbol string `"AB"` with codes `(1, 2)` encodes as `2^1 · 3^2 = 18`. A six-symbol string encodes to a multi-billion-digit number under the classical scheme. Axioma's variant grows more slowly but the *phenomenon* is the same.

30.6 (open) — Why does Gödelization still matter in 2025?

In your own words, give three modern applications of Gödelization. The applications can be in computer science, mathematics, philosophy, or AI safety.

Sample answers:

- **Compilers and interpreters** — every "code as data" story (LISP, macros, eval, JIT compilation) is Gödelization under a different name. The compiler treats source code as a structured object it can manipulate; that's exactly what Gödel did with formulas.
- **Cryptographic commitment** — zero-knowledge proofs (zk-SNARKs, zk-STARKs) encode entire computations as polynomials over a finite field. That polynomial *is* the Gödel number of the computation, in a more compact arithmetic encoding. The commitment lets a prover convince a verifier that a computation ran correctly without showing the computation.
- **AI alignment / formal verification** — to prove a property of an AI system, you need to *encode the AI system* as something a proof assistant can talk about. Gödelization is the bridge. The HOL theorem prover, Coq, Lean — all of them rely on the syntax-as-data move Gödel pioneered.
- **Self-modifying systems** — Kleene's recursion theorem guarantees that a system can refer to its own code. This is the mathematical bedrock of every AI agent that reasons about its own behavior.
- **Halting / undecidability** — every "this is undecidable" theorem in computer science traces back to the Gödel-Turing diagonalization. Rice's theorem (no non-trivial property of a program is decidable from the source) is the most general form.

Gödel's trick from 1931 is, in retrospect, one of the half-dozen *moves* that underwrite all of computer science.

30.11 Reflection — three things to internalize

One — syntax is data. That's the move. Anything you can write down as a string of symbols can be a number, and arithmetic operations on that number correspond to syntactic operations on the original string. The encoding is *just bookkeeping*; the *idea* is that the universe of formal systems and the universe of arithmetic *are the same universe*, viewed differently.

Two — diagonalization is universal. Cantor used it on reals. Gödel used it on formulas. Turing used it on programs. Russell used it on sets. Every limitative result in mathematics and computer science is, at its core, *this same trick*: enumerate a class of objects, construct one that disagrees with each listed object, conclude that the class wasn't fully listable. Gödel numbering is what makes the trick applicable to syntactic objects.

Three — the move is older than computer science but *designed* computer science. Gödel didn't have computers. Turing didn't have computers when he proved the halting problem. The encoding-syntax-as-numbers move is *prior* to computation; it's what *enabled* the formal definition of computation that came after. Modern computer science is, in a deep sense, the *industrial application* of a 1931 mathematical trick.

The next chapter (Ch.31, Lojban-in-Axioma) takes a completely different turn: instead of encoding syntax as numbers, it encodes natural-language relations as syntactically unambiguous predicates with named places. Both directions — making syntax computable, making language unambiguous — are part of the same broader project: bridging the formal and the natural.

> *"This statement cannot be proved."* — Gödel's Liar-paraphrase. The astonishing part isn't that there are unprovable truths; it's that *every sufficiently rich system contains its own undecidability*. Gödel numbers are how you make that fact mechanical.

Solutions to selected exercises: Chapter 30 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch30_solutions.md.)*

Chapter 31 · Lojban-in-Axioma

What this chapter is. **Lojban** is the most ambitious constructed language in modern linguistics: a ~1300-word vocabulary engineered so that every sentence has *exactly one* syntactic parse and *exactly one* first-order-logic translation. Axioma adopts Lojban's signature mechanism — the **selbri with named places** — as the `define relation` form. This single syntactic idea pays off three ways: it documents argument ordering as part of the schema, it produces machine- checkable relational predicates, and it composes with Volume I's Horn-rule logic-programming machinery from Chapter 21. This chapter walks the theory, the API, and the spots where the framework strains in production KR.

31.1 Why Lojban interests language designers

In 1955 James Cooke Brown proposed **Loglan** as a test bed for the Sapir-Whorf hypothesis: a language whose grammar was *first-order logic*, so that thinking in it would prove or disprove that language shapes cognition. Lojban (1987) is the descendant — a community-maintained language with:

- **~1300 gismu** (root predicate words),
- **A YACC-style grammar** that produces unambiguous parse trees for every well-formed sentence,
- **Direct translation to FOL** for every assertion,
- **An open-source dictionary** mapping each gismu to its argument structure.

The Sapir-Whorf experiment never produced clean results, but the engineering goal is independently valuable: a language that *cannot be misparsed* is a language *you can run inference over without a natural-language model in the loop*. That's a kingdom in modern KR work.

The signature feature is the **selbri** — a predicate word that carries a *fixed* list of argument slots called **places**, each with a known *semantic role*.

Lojban's gismu `dunda` ("gives") has three places:

Place	Role	Example filler
x1	giver	"John"
x2	gift	"book"
x3	recipient	"Mary"

Every assertion using `dunda` fills those three slots in the same order. Compare to English:

- *"John gave Mary the book"*
- *"Mary was given the book by John"*
- *"The book was given to Mary by John"*

Three surface forms, three different orderings of the same three roles. In Lojban there's *one* surface form:

`John dunda le cukta Mary` (John gives the book to Mary)

And one FOL formula: `dunda(John, book, Mary)`.

That predictability is what Axioma adopts.

31.2 The Axioma surface — `define relation`

Axioma exposes Lojban's named-places idea through the **`define relation`** statement. The syntax has four variants of increasing detail:

```
# Bare — argument names only
relation parent(p, c)

# With type annotations
relation age(p :: String, y :: Integer)

# With semantic roles (Lojban places)
relation gives(g -> "giver", x -> "gift", r -> "recipient")

# With both types and roles
relation cite(paper :: String -> "paper",
              author :: String -> "author")
```

The `-> "role"` form is the Lojban analog: it labels each argument position with its semantic role, lifting *place structure* into the language. Once defined, the relation is **first-class**:

```
gives("John", "book", "Mary")
gives("Alice", "flower", "Bob")
gives("Carol", "book", "Dave")

# Query: who gave a book?
println({X | X <- gives(X, "book", _)})
# → {"Carol", "John"}

# Query: what did Alice give?
println({Y | Y <- gives("Alice", Y, _)})
# → {"flower"}

# Query: who received a book?
println({Z | Z <- gives(_, "book", Z)})
# → {"Dave", "Mary"}

# All three at once
println({(g, gift, r) | gives(g, gift, r)})
# → {("Alice", "flower", "Bob"),
#    ("Carol", "book", "Dave"),
```

```
#    ("John", "book", "Mary")}
```

The wildcard _ matches any value. The pattern-variable X binds the value at that position and propagates through the comprehension.

31.3 Arity enforcement and type-as-documentation

Arity is **strictly checked**:

```
relation gives(g -> "giver", x -> "gift", r -> "recipient")

gives("John", "book", "Mary")     # ok — 3 args
gives("John", "book")              # ERROR — expects 3, got 2
```

Types, by contrast, are **documentation only** at runtime:

```
relation age(p :: String -> "person", y :: Integer -> "age")
age("Alice", 30)
age("Bob", "thirty")    # passes — no runtime type check
println({(P, Y) | age(P, Y)})
# → {("Alice", 30), ("Bob", "thirty")}
```

This is a *deliberate* choice: the relational subsystem lets you mix value shapes during exploratory KR work, then use `--typecheck` (or a downstream linter) to enforce conformance when you ship. For the Lojban analogy: the places *contract for what makes sense*; the runtime tells the user, not the typechecker, when something looks wrong.

31.4 Composition with Horn rules

The big payoff of choosing Lojban-style places is that relations defined this way are **immediate premises** for the Horn-rule machinery you learned in Ch.21. Strict and defeasible rules compose unchanged:

```
relation parent(p -> "parent", c -> "child")
parent("Alice", "Bob")
parent("Bob", "Carol")
parent("Carol", "Dave")

# Strict derivation — grandparent
grandparent(X, Z) <= parent(X, Y) and parent(Y, Z)

# Recursive derivation — ancestor
ancestor(X, Y) <= parent(X, Y)
ancestor(X, Z) <= parent(X, Y) and ancestor(Y, Z)

println({Y | Y <- ancestor("Alice", Y)})
```

```
# → {"Bob", "Carol", "Dave"} — all transitive descendants

println({X | X <- ancestor(X, "Dave")})
# → {"Alice", "Bob", "Carol"} — all ancestors
```

The rule body `parent(X, Y) and ancestor(Y, Z)` is a *Lojban-style logical conjunction*: each predicate is a selbri call with its named places, and the shared variable `Y` enforces unification across the two predicates. **This is FOL.** No translation step. The relational subsystem is the language's FOL bridge.

31.5 The Lojban grammar layer (advanced)

Axioma includes a deeper Lojban surface in `grammar/ lojban.go` that lets you read actual Lojban-flavored prose:

```
s: <<John loves Mary>>      # constrained-language literal
println(words_of(s))            # → ["John", "loves", "Mary"]

lj: cl_translate(s, "lojban")
println(lj)                     # → <<John loves Mary>> [lojban]
```

The constrained-language literal `<<...>>` is a typed container that knows what *natural-language* mode it's in. `cl_translate` switches modes. The deeper grammar parser (`parse_grammar`) extracts **bridi** — Lojban sentences with a selbri and zero or more **sumti** (arguments).

Caveat: the grammar parser depends on a populated **vocabulary registry** that maps each English/Lojban word to its part-of-speech and (for selbri) its place structure. In a fresh Axioma session the registry is empty:

```
parse_grammar(<<John loves Mary>>)
# ERROR: no selbri (predicate) found in bridi

vocab_lookup("loves")
# ERROR: word 'loves' not found in vocabulary
```

Production systems populate the registry from a corpus of gismu definitions before calling `parse_grammar`. The in-language *capability* exists; the *out-of-the-box data* doesn't. See §31.9.

31.6 What Lojban-style places buy you in production

Three concrete wins, ordered by impact:

1. **Self-documenting facts.** A new team member sees `gives("John", "book", "Mary")` and the *function signature* of `gives` immediately tells them which argument is the giver, which the gift, which the recipient. Compare to a graph-edge KR where every edge type needs an out-of-band convention document.

2. **Mechanical FOL compilation.** Because every relation has a fixed arity and place structure, the compiler can mechanically translate every `define relation` into a FOL predicate symbol with a known signature. No NLP step.
3. **Cross-team consistency.** Axioma's `define relation affects doc: "..."` update form lets you *amend* documentation without breaking the schema. The compiler can warn when two teams independently define `gives` with different place structures, catching the most common form of KR drift.

Cascade's entity-extraction pipeline lives partly in this register: the symbolization layer in `.axioma/symbolization/lojban.md` (a design doc) defines how to normalize natural-language claims into Lojban-style predicate-with-places form before they enter the knowledge graph. The savings compound as the graph grows.

31.7 Worked example — a citation graph

```
relation cite(paper :: String -> "paper",
                     author :: String -> "author") doc: "P cites A"

relation wrote(author :: String -> "author",
                      paper  :: String -> "paper")

# Some facts
cite("Pearl1988", "Russell")
cite("Pearl1988", "Norvig")
cite("Russell2010", "Pearl")
wrote("Pearl", "Pearl1988")
wrote("Russell", "Russell2010")
wrote("Norvig", "Russell2010")

# Who has authored a paper that cites someone?
println({W | wrote(W, P) and cite(P, A)})
# → {"Pearl", "Russell", "Norvig"}

# Lifted self-citation rule
self_cite(A) <= wrote(A, P) and cite(P, A)
println({A | A <- self_cite(A)})
# → {} — no self-cites in our data

# Mutual citation
mutual(A, B) <= wrote(A, P1) and cite(P1, B)
                and wrote(B, P2) and cite(P2, A)
println({(A, B) | mutual(A, B)})
# → {("Russell", "Pearl"), ("Pearl", "Russell")}
```

Notice how the *named places* on `cite` and `wrote` prevent the typical bug of confusing argument order: `cite(paper, author)` is unambiguously *paper cites author*, not *author cites paper*. A graph KR with labeled edges would document this with edge-type names (`AUTHORED`, `CITED_BY`), but here the documentation is

inline in the call signature — the Lojban win.

31.8 Doc strings and refinement

Two further sugar forms:

```
# Trailing doc string
relation supports(party :: String -> "party",
                  policy :: String -> "policy")
    doc: "The party publicly endorses the policy"

# Refinement — transient definition (no KB persist)
relation/transient scratch(x -> "x", y -> "y")

# Refinement — explicit persist (default in REPL)
relation/persist core_rel(p -> "p", c -> "c")
```

An opt-in canonical session is an exception to the ordinary REPL persistence default: it requires explicit `/persist` and a manifest binding for each stored definition. Its initial bounded profile accepts ordered argument names, types and roles, but refuses relation `doc:` and annotations. These examples describe the ordinary language/session behavior; use the Manual's canonical session section when testing external storage and exact-UUID reload.

The `doc:` annotation surfaces in the relation's metadata (introspectable from external tools that read the session). The refinements `/persist` and `/transient` match the same vocabulary you saw for bindings and `axiom` in Volume I — relation definitions can be temporary scaffolding or permanent schema.

Optional: reuse a vocabulary and reopen its KB

A self-contained declarative vocabulary can be imported with a plain unaliased `import "ontology.ax"`. In the supported subset, concepts, relation schemas, ground facts and logical rules become visible in the caller; adding a fact through an imported relation is visible to caller queries. All declared predicates must be exported. Repeating the same import preserves existing state and avoids repeating rules. Changed source or conflicting definitions refuses. Aliased or selective imports and modules with arbitrary helper functions retain their existing isolated module behavior.

Shared-KB reload preserves typed numeric concept defaults and supported closed expression actions. A closed action uses only its untyped parameters, scalar literals, arithmetic/comparison/Boolean operators, conditionals and returns; captures, other function calls, property access and advanced contracts refuse. Loading parses the approved function form without executing stored code. Old unsupported display-only values require re-saving from their source. Relation argument schemas, arity and saved facts reload, but generic KB rows do not encode rule ASTs or every relation modifier; reimport the authoritative module for those. See the Manual's "Typed concept values across a KB restart" and "Declarative knowledge modules" for exact boundaries. This optional material does not change the examples or exercises below.

31.9 Honest limits

Four gaps to be aware of:

1. **Block-form `define relation X { places: [...] }` is not implemented.** The pilot draft showed this form; only the parenthesis form with `:: Type` and `-> "role"` actually works.
2. **`vocab_lookup` and `parse_grammar` need a populated registry.** Out of the box neither has any Lojban or English gismu loaded. `parse_grammar(<<John loves Mary>>)` produces the error "no selbri found". Building a Lojban-prose-reading pipeline requires populating `vocab.GlobalRegistry` from a vocab file first — outside what this chapter teaches.
3. **Types are documentation, not contracts.** `age("Bob", "thirty")` passes runtime checks even with `y :: Integer`. Use the `--typecheck` pre-pass or downstream linters to enforce type conformance.
4. **No introspection builtin for relation metadata.** There's no `relation_places("gives")` or `relation_arity("gives")` builtin. The arity is enforced by the runtime but not queryable from inside Axioma.

These are *coverage gaps* rather than design flaws — the core idea (Lojban-style places as predicate signatures) is sound and load-bearing. The richer grammar/vocab features are present but require setup work outside Axioma proper.

31.10 Exercises

31.1 — Place a relation

Define a relation `taught` with three places: instructor, subject, student. Add facts that Alice taught calculus to Bob, Carol, and Dave; that Eve taught logic to Bob. Query: who taught Bob? What did Alice teach? How many students learned calculus?

31.2 — A small citation graph

Define relations `cite(paper -> "paper", author -> "author")` and `wrote(author, paper)`. Build the six-fact example from §31.7. Query: find authors that have been cited by someone they wrote a paper with.

31.3 — Type annotations as documentation

Define a relation `birthday(person :: String -> "person", year :: Integer -> "year")`. Add three facts, including one with an Integer year and one with a String year (e.g. `"unknown"`). Verify both pass. Then sketch what a runtime type-check would look like.

31.4 — Recursive ancestor

Define `parent(p -> "parent", c -> "child")` and add a four-generation chain Alice → Bob → Carol → Dave. Write strict rules for `grandparent` and `ancestor`. Query: print all grandparents, all of Alice's descendants, and all of Dave's ancestors.

31.5 — Arity bug

Define a relation `loves(lover, beloved)` with 2 places. Try calling it with 1 argument and with 3 arguments. Print the error messages. Explain why arity enforcement catches a class of bugs that named-place semantics do *not* (and conversely).

31.6 (open) — Map a domain

Pick a domain you know well — recipes, scientific papers, customer-support tickets, sports events, legal contracts. Define five relations with Lojban-style places. Pay attention to *place order*: which role goes in `x1`? Justify your ordering in 1-2 sentences per relation.

Solutions to selected exercises: Chapter 31 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch31_solutions.md.)*

31.11 What you learned

- Lojban's premise: a language with *one* parse and *one* FOL reading per sentence, anchored on the **selbri with named places**.
- Axioma's `define relation` adopts that signature vocabulary with four variants (bare / typed / role- labeled / both).
- Arity is enforced strictly; types are documentation with no runtime check; the `--typecheck` pre-pass and downstream linters carry the contract.
- Relations compose immediately with Horn rules — strict, defeasible, forward, backward — without translation.
- Deeper Lojban features (grammar parsing, vocabulary lookup, English↔Lojban translation) exist but need a populated vocabulary registry.
- Three production wins: self-documenting facts, mechanical FOL compilation, cross-team consistency.

This closes Volume II's *Part X — Symbolic Knowledge Frontiers* (Ch.28-33). Volume III continues with the Cascade architecture (Ch.35), where every primitive you just learned — graphs, spaces, NSM primes, Lojban-style places, Schank ACTs — lives side-by-side in a production knowledge graph.

> *"The number of places of a selbri is its valency."* — Lojban Reference Grammar. Axioma's `define relation` form makes valency *part of the schema* rather than *part of the convention document* — the discipline that compounds.

Chapter 32 · Natural Semantic Metalanguage (Wierzbicka)

What this chapter is. Anna Wierzbicka and Cliff Goddard have spent four decades arguing for a startling claim: every human language contains a small set of ~65 **semantic primes** — atoms of meaning that are *irreducible* (you cannot define them in simpler terms) and *universal* (they appear in every language ever studied). Every other concept can be paraphrased using only these primes. Axioma exposes the primes as a first-class registry with three validators, a seven-language translation table, and seven grammar rules. This chapter walks the surface, then drives it with the workflow Wierzbicka herself uses: **write an explication; validate; refine; iterate.**

32.1 The premise — irreducibility plus universality

The NSM bet rests on two claims that have to hold *simultaneously* for the theory to be more than a curiosity:

1. **Irreducibility.** A prime cannot be defined without circularity. Try to define "want" without using "want", "desire", "wish", or any of their cognates — you fail every time. Try the same for "know", "think", "good", "this", "I". They are *load-bearing primitives* of thought.
2. **Universality.** Every natural language has *some word or grammatical device* expressing each prime. Japanese has 私 (watashi) for I. Russian has хороший (khoroshy) for GOOD. Chinese has 不 (bu) for NOT. No language has been documented that lacks them.

If both claims are true, then a definition written *entirely in primes* is automatically translatable to every language by substitution. Cultural baggage shouldn't intrude because the primes are too thin to carry baggage.

This is a strong claim. NSM has critics — see §32.10 — but the framework has produced ~3,000 published definitions ("explications") across emotions, social roles, cultural keywords, ethics terms, and grammar particles. Axioma gives you the machinery to write your own.

32.2 The 66 primes, by category

Axioma's registry has **66 primes** (the canonical Wierzbicka set is 65; the registry splits some `BE` forms more finely) organized into **16 categories**. Here is the full inventory:

Category	Primes
substantives	I, YOU, SOMEONE, SOMETHING, PEOPLE, BODY
relational substantives	KIND, PART
determiners	THIS, THE SAME, OTHER
quantifiers	ONE, TWO, SOME, ALL, MUCH, LITTLE
evaluators	GOOD, BAD
descriptors	BIG, SMALL
mental predicates	THINK, KNOW, WANT, DON'T WANT, FEEL, SEE, HEAR
speech	SAY, WORDS, TRUE
actions, events	DO, HAPPEN, MOVE, TOUCH
location, existence	BE (SOMEWHERE), THERE IS, BE (SOMEONE/SOMETHING), HAVE, BE (SOMEONE'S)
life, death	LIVE, DIE
time	WHEN, NOW, BEFORE, AFTER, A LONG TIME, A SHORT TIME, FOR SOME TIME, MOMENT
space	WHERE, HERE, ABOVE, BELOW, FAR, NEAR, SIDE, INSIDE
logical	NOT, MAYBE, CAN, BECAUSE, IF
intensifier	VERY, MORE
similarity	LIKE

Note the *flavor* of the list:

- No content nouns. There is no DOG, no TREE, no MOTHER.
- No abstractions. No JUSTICE, no LOVE, no MIND.
- No middling abstractions. No NUMBER, no COLOR.
- Only meta-vocabulary you need to *describe* concepts.

Every other word in any human language is, in NSM's bet, *reducible* to these. That bet is what the registry operationalizes.

32.3 The registry API

Five entry points:

```
nsm_primes()                              # all 66 primes (array of records)
nsm_prime("THINK")                        # registry entry for one prime
is_nsm_prime("think")                     # case-insensitive boolean check
nsm_primes_by_category("mental_predicates")
nsm_categories()                          # 16 category names
```

Each registry entry is a record:

```
p: nsm_prime("THINK")
println(p.canonical_number)   # 22
println(p.category)           # "mental_predicates"
println(p.valency)            # 2 (binary: experiencer + thought)
println(p.arguments)          # roles + descriptions
println(p.examples)           # ["I think something", "you think like this"]
println(p.grammar_rules)      # ["complementation"]
println(p.allolexes)          # related surface forms
println(p.translations)       # 7-language table
```

Valency matters: it tells you how many positions the prime takes in a clause. `THINK` is valency 2 (someone thinks something); `I` is valency 0 (just refers); `SAY` is valency 3 (speaker says content to addressee).

Allolexes are *non-canonical surface forms that count* as the prime. The registry records, for example, that "me" is an allolex of `I`, that "thing" is an allolex of `SOMETHING`, that "person" is an allolex of `SOMEONE`, and that "many" is an allolex of `MUCH`. When the validators read text, they accept allolexes.

```
primes: nsm_primes()
i: 1
while (i <= len(primes)) [
    if len(primes[i].allolexes) > 0 then [
        println(primes[i].prime, "→", primes[i].allolexes)
    ]
    i = i + 1
]
# I       → ["me"]
# SOMEONE → ["someone", "person"]
# SOMETHING → ["something", "thing"]
# MUCH    → ["many"]
# WHEN    → ["time"]
# WHERE   → ["place"]
# ... etc
```

32.4 The seven grammar rules

Primes don't combine arbitrarily. NSM grammar is severely constrained — a deliberate choice, because the goal is to make sure the grammar itself is also universally expressible. Axioma's registry records seven rules:

Rule	Pattern	Example
quantification	`{quantifier} {substantive}`	"many people", "two things"
modification	`{evaluator/descriptor} {substantive}`	"something good", "big thing"
complementation	`X {mental_predicate} [that] Y`	"I think something good"
temporal_specification	`{event} {temporal_prime}`	"this happened before"

Rule	Pattern	Example
spatial_specification	`{substantive} {spatial_prime}`	“something here”
causation	`X happened because Y`	“I feel bad because of this”
conditional	`if X then Y`	“if this happens, something bad happens”

You can query them programmatically:

```
rules: nsm_grammar_rules()
i: 1
while (i <= len(rules)) [
    r: rules[i]
    println(r.name, ":", r["pattern"])
    println("  applicable to:", r.applicability)
    i = i + 1
]
```

Each rule lists its `applicability` (which primes can participate), `examples` (good uses), and `violations` (common bad uses). The violations are pedagogically valuable — they show what *looks* like NSM but isn’t.

32.5 The four validators

Four validators sit on top of the registry. They have different strictness levels and return different shapes. Pick the one whose strictness matches your task.

`nsm_validate_text(text)` — strict boolean

The simplest. Returns `true` iff every word in the input is either a prime or an allolex of a prime. Punctuation and the connective “to” are tolerated.

```
nsm_validate_text("I think something good")     # true
nsm_validate_text("I love you")                  # false ("love" not a prime)
nsm_validate_text("someone did something bad")   # false ("did" isn't lemmatized)
```

Surprise: “did something bad” fails because the validator doesn’t lemmatize verb tenses. `nsm_validate_explication` is more forgiving.

`nsm_validate_explication(text)` — rich detail

The validator you actually want for iterative refinement. Returns a record with five fields:

```
r: nsm_validate_explication("X feels something bad because of this")
println(r.valid)              # true
println(r.used_primes)        # ["FEEL", "SOMETHING", "BAD", "BECAUSE", "THIS"]
println(r.grammar_issues)     # []
println(r.suggestions)        # []
println(r.warnings)           # []
```

This input is accepted: the validator recognizes its normalized primes and permits the variable. Suggestions are diagnostic aids when supplied, not a guarantee of grammatical or semantic correctness. The workflow is:

1. Write a paraphrase using whatever words feel natural.
2. Run `nsm_validate_explication` on it.
3. Read `grammar_issues` + `suggestions`.
4. Rewrite, replacing the flagged words.
5. Repeat until `valid: true`.

`nsm_validate_grammar(text)` — currently permissive

A stub. Returns `{errors: [], valid: true}` for almost anything. Don't rely on it for grammar checking yet; the prose explanation of the 7 rules in §32.4 is what to internalize.

`nsm_check_structure(text)` — currently permissive

Same. Returns `{issues: [], valid: true}` regardless of input. Reserved for future structural analysis.

The honest summary: **use `nsm_validate_explication` for real work; the other three are convenience wrappers or placeholders.**

32.6 Near-match: `nsm_find_similar`

When you're not sure if a word maps to a prime, ask directly:

```
nsm_find_similar("person")    # ["PERSON"] — wait, that's an allolex display
nsm_find_similar("good")      # ["GOOD"]
nsm_find_similar("feel")      # ["FEEL"]
nsm_find_similar("happy")     # []
nsm_find_similar("hate")      # []
nsm_find_similar("kill")      # ["I"]
```

The matcher does substring lookup against canonical primes *and* allolexes. It is loose — short words like "kill" match "I" (because "kI ll" contains "I" as a substring), so treat its output as a *hint* not an oracle. The empty list for "happy" and "hate" is correct and informative: those are *not* primes, so an NSM explication can't use them directly. You'd have to decompose them — see §32.8.

32.7 Translation across seven languages

Each prime carries translations into seven languages: `english`, `spanish`, `french`, `german`, `russian`, `japanese`, `chinese`.

```
nsm_translate("GOOD", "french")        # "bon"
nsm_translate("THINK", "japanese")     # "思う"
nsm_translate("BECAUSE", "russian")    # "потому что"
nsm_all_languages()
# ["chinese", "english", "spanish", "french", "german",
#  "russian", "japanese"]
```

The translation table is the *operational core* of NSM's universality claim. If the primes really are universal, this table should *always work*. Where the framework strains is when a target language has *multiple* words for what English calls one prime — German has "denken" (reasoned thought) and "glauben" (belief-like thought), and the registry has to pick one as canonical for THINK. Those choice points are the empirical edges of NSM and where the most interesting linguistic argument lives.

A complete translation pipeline:

```
translate_explication: func(text, target_lang) [
    words: split(text, " ")
    out: ""
    i: 1
    while (i <= len(words)) [
        w: words[i]
        if is_nsm_prime(w) then [
            out = out + nsm_translate(uppercase(w), target_lang) + " "
        ] else [
            # Allolex or punctuation — pass through
            out = out + w + " "
        ]
        i = i + 1
    ]
    out
]
```

(That's a sketch; production code would handle multi-word primes like "DON'T WANT" specially. Real translation pipelines in Cascade do exactly this kind of prime-by-prime substitution.)

32.8 Worked example — explicating "promise"

Wierzbicka's classic example. The English word "promise" is *not* a prime. To explicate it in NSM, paraphrase using only primes. Here is the iterative workflow.

Attempt 1

```
v1: "X promises Y to do something"
println(nsm_validate_explication(v1))
```

Output:

```
{valid: false,
 used_primes: ["DO", "SOMETHING"],
 grammar_issues:
   ["'X' is not an NSM prime",
    "'promises' is not an NSM prime",
    "'Y' is not an NSM prime",
    "'to' is not an NSM prime"],
 suggestions: ["Did you mean 'I'?", ...]}
```

`X` and `Y` aren't primes (they're variables, but the validator doesn't know that). `promises` is what we're trying to *define* — we can't use it in its own definition. `to` is grammar particle.

Attempt 2 — replace variables with people

```
v2: "someone says: I will do something"
println(nsm_validate_explication(v2))
```

Still fails: "says" is not a prime (SAY is), "will" is not a prime.

Attempt 3 — lemmatize and remove tense

```
v3: "someone say this: I do something"
println(nsm_validate_explication(v3))
# valid: true, used_primes: ["SOMEONE", "SAY", "THIS", "I", "DO", "SOMETHING"]
```

Better. But this is a *bare assertion*, not a promise. A promise involves commitment about the future, awareness by the listener, and intention. NSM's full explication is typically:

```
someone (X) says this to someone (Y):
  I think this: I will do something good for you
  I want you to know this
```

In purely-prime form:

```
v4: "I say this: I think I do something good. I want you know this"
println(nsm_validate_explication(v4))
```

This still has some grammatical roughness — NSM explications are typically *line-by-line* prose, one proposition per line, with explicit time-words for the future. The full Wierzbicka explication of "promise" runs about 8 lines. The takeaway: **decomposing one common word into primes typically takes 5-10 propositions**. That's the discipline.

Storing the explication

Once you've validated, store it as a `Concept`:

```
concept Promise
Promise has explication: [
    "someone X say this:",
    "I think this: I do something good for someone Y",
    "I want someone Y know this"
]
Promise has used_primes: ["SOMEONE", "SAY", "THIS", "I", "THINK", "DO",
                          "SOMETHING", "GOOD", "WANT", "KNOW"]
Promise has language_neutral: true
```

The `language_neutral: true` flag is the payoff: a prime-only definition, by construction, translates to any of the 7 supported languages by substitution.

32.9 Where NSM lives in a KR system

Three places NSM pays off in a knowledge-representation pipeline like Cascade:

1. **Multilingual fact bases.** If an entity's definition is in prime form, the system can serve it in any supported language by table lookup — no re-extraction per language.
2. **Definition lint.** Auto-extracted definitions from LLMs often contain the term being defined (circular) or rely on domain jargon (unparseable downstream). `nsm_validate_explication` catches both: circular definitions fail because the term won't be a prime; jargon-heavy definitions fail with long `grammar_issues` lists.
3. **Cultural-keyword analysis.** Wierzbicka has shown that *culture-specific* words (Russian *toska*, Japanese *amae*, German *Gemütlichkeit*) decompose into culture-*neutral* primes plus structural patterns. This lets a KR system *capture* cultural concepts without claiming they reduce to English glosses.

32.10 Honest limits

NSM has earned a substantial critical literature. The honest summary:

- **The "irreducibility" claim is testable but not closed.** Different NSM researchers occasionally propose different prime sets — 60, 65, 70 — depending on whether they treat certain compounds (BE SOMEONE'S) as one prime or two.
- **The "universality" claim is empirical.** It has survived contact with hundreds of languages but is always one fieldwork-discovery away from falsification. The framework is *bet*, not theorem.
- **Translation isn't always crisp.** Picking `glauben` over `denken` for THINK in German is a judgment call. Different translators make different picks.
- **The grammar is constraining.** Some natural paraphrases just can't be written in NSM grammar. You sometimes need 3 sentences where 1 felt right.

These are *features*, not bugs, for KR work — the constraints are what make prime-form definitions

machine-checkable. But it is worth knowing where the framework strains. Compare Ch.33 (Schank's CD) for a *different* set of universal primitives with different tradeoffs.

32.11 Exercises

32.1 — Inventory by category

Use `nsm_categories()` and `nsm_primes_by_category` to print one line per category, of the form `category: N primes`. Total should sum to 66.

32.2 — Allolex lookup

Find every prime that has at least one allolex. Print the prime alongside its allolex list. (You can use this to teach yourself the lemmatization the validators expect.)

32.3 — Validate-and-refine

Start with the candidate definition for "regret": *"someone feel bad because they did something"*. Run `nsm_validate_explication`, read the issues, rewrite, repeat until `valid: true`. Print each iteration along with its `valid` flag.

32.4 — Translation pipeline

Implement the `translate_explication` function from §32.7 in full. Then run it on the validated explication *"I think this is good"* into French, German, and Japanese. Print all three.

32.5 — Decompose an emotion

Pick one of these emotional concepts and write an NSM explication using only primes: **surprise**, **gratitude**, **embarrassment**. Validate it. Store the validated explication as a `Concept` with `explication` and `used_primes` slots.

32.6 (open) — Where does the framework strain?

NSM's 65/66 primes have been periodically updated as fieldwork uncovers languages that challenge specific universals. Use `nsm_primes_by_category` on the `space` category (`WHERE, HERE, ABOVE, BELOW, FAR, NEAR, SIDE, INSIDE`) and ask yourself: are any of these *culturally specific*? Hint: orientation systems differ dramatically across languages — some lack absolute ABOVE/BELOW and use cardinal directions instead.

Solutions to selected exercises: Chapter 32 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch32_solutions.md.)*

32.12 What you learned

- The NSM premise: ~65 irreducible, universal semantic primes form a metalanguage powerful enough to paraphrase every other concept.
- The 16 categories and the surface API (`nsm_primes`, `nsm_prime`, `is_nsm_prime`, `nsm_primes_by_category`, `nsm_categories`).
- The seven grammar rules and the four validators — with `nsm_validate_explication` as the workhorse and the other three as placeholders or convenience wrappers.
- The validate-or-iterate workflow: write paraphrase, validate, read suggestions, rewrite. The discipline that produces translatable definitions.
- The seven-language translation table that makes prime-form definitions language-neutral by construction.
- The honest limits: irreducibility is a bet, universality is empirical, grammar is constraining.

The next chapter (33) goes deeper into the *action* half of the universal-primitives game with Schank's Conceptual Dependency. Where NSM proposes **universal nouns and verbs**, CD proposes **a restricted set of action verbs** — different cut, same ambition.

> *"What I want to know is whether what we call 'mind' is universal. The only way to test that is to define mental words in words that everyone shares."* — Anna Wierzbicka, paraphrased

Chapter 33 · Schank's Conceptual Dependency

What this chapter is. Roger Schank's **Conceptual Dependency** (CD), introduced in his 1972 paper *Conceptual Dependency: A Theory of Natural Language Understanding*, claims that every action verb in every human language reduces to one of about a dozen **primitive ACTs**. "Give," "donate," "sell," "lend," "transfer," "bequeath," "award" — all reduce to a single primitive: `ATRANS` (abstract transfer of possession). The surface vocabulary varies wildly; the conceptual spine is small and shared. Axioma exposes the registry of eleven primitive ACTs as a first-class API, three working builtins, and a discipline for grounding domain-vocabulary against a shared semantic core. By the end of this chapter you will know the eleven primitives, their role slots, the verbs that decompose into each, and where CD pays off in Cascade's entity-extraction pipeline.

33.1 The core insight

Schank's bet, made in 1972 and refined through the 1970s *Scripts, Plans, Goals, and Understanding* line of work: **human action verbs are *not* fundamental units of meaning**. They're surface forms that decompose into a small set of *cognitive primitives* — operations the mind models the world with.

Three sentences:

- *John gave Mary the book.*
- *John sold Mary the book for $10.*
- *John donated the book to Mary.*

A naive semantic system treats "give," "sell," and "donate" as three *distinct* relations. Build a knowledge graph that way and you'll have hundreds of trade-related predicates with no machinery for noticing that they're all describing the same kind of event.

Schank's move: factor the meaning. The shared core is the **primitive ACT** — here, `ATRANS(actor=John, object=book, from=John, to=Mary)`. The differences between "give," "sell," and "donate" are *modifiers* on that primitive:

- *give* — bare ATRANS, no consideration mentioned
- *sell* — ATRANS plus a counter-transfer (`for $10`)
- *donate* — ATRANS plus *gratuitous* flavor (no consideration expected)

The same primitive runs under all three. Queries like "what transferred between Alice and Bob today?" can now match every verb-instance at once — because they all share the `ATRANS` spine.

This is the same engineering insight as type erasure in programming languages, or normalized forms in

relational databases: **the surface presentation is not the data**. Find the underlying invariant, store it, then derive surface forms from the invariant on demand.

33.2 What Axioma exposes

Three builtins, all working in the current build:

Builtin	Purpose
`cd_primitives()`	Return the full registry — array of `{name, roles, description}` records
`cd_roles("ATRANS")`	Return the role-name array for a specific primitive
`is_cd_primitive("ATRANS")`	Boolean — is this string a registered primitive? (case-insensitive)

A quick taste:

```
prims: cd_primitives()
println("Total primitives:", len(prims))          # Total primitives: 11
println("First entry:", prims[1])
println("ATRANS roles:", cd_roles("ATRANS"))
println("Is ATRANS a primitive?", is_cd_primitive("ATRANS"))
println("Is XFRANS?", is_cd_primitive("XFRANS"))
```

Output:

```
Total primitives: 11
First entry: {description: "abstract transfer of possession, control, or relationship",
 ↪  name: "ATRANS", roles: ["actor", "object", "from", "to"]}
ATRANS roles: ["actor", "object", "from", "to"]
Is ATRANS a primitive? true
Is XFRANS? false
```

Three builtins is enough for the *retrieval* side of CD. For the *decomposition* side — assigning verbs to primitives and stitching them into knowledge graphs — you use ordinary Axioma machinery (Concepts, relations, the rule engine from Chapter 21) with the primitive name as the spine.

33.3 The eleven primitives

Schank's original 1972 set had eleven primitives — refined across many subsequent papers but the eleven below are the canonical roster Axioma exposes. Each primitive has a **name**, a fixed **role slot list**, and a **description** explaining what it means.

Primitive	Roles	Description
`ATRANS`	`actor, object, from, to`	abstract transfer of possession, control, or relationship
`ATTEND`	`actor, object`	directing attention or perception toward something
`EXPEL`	`actor, object`	moving something out of the body or container
`GRASP`	`actor, object`	grasping or taking hold of an object
`INGEST`	`actor, object`	taking something into the body or system
`MBUILD`	`actor, content`	building or changing a mental state
`MOVE`	`actor, body_part, direction`	moving a body part or controlled object
`MTRANS`	`actor, information, from, to`	mental transfer of information
`PROPEL`	`actor, object, force`	applying force to an object
`PTRANS`	`actor, object, from, to`	physical transfer of location
`SPEAK`	`actor, content, to`	producing communicative speech or text

A small reading guide on each.

`ATRANS` — abstract transfer of possession

The *ownership* primitive. Anything that changes who controls or owns something. Verbs: give, donate, sell, lend, return, grant, award, transfer, cede, bequeath, inherit, surrender, confiscate, gift. *Not* INGEST or GRASP — those are about *physical* possession; ATRANS is about the *abstract right* to possess.

`ATTEND` — directing perception

The *attention* primitive. Verbs that route a sense toward an object. Verbs: look, see, watch, listen, hear, smell, taste, observe, notice. Some compound verbs that imply ATTEND: "He glared at her" decomposes to ATTEND (eyes) plus an affective modifier.

`EXPEL` — moving outward from a container

Verbs about *removal* from a body or vessel. Verbs: exhale, excrete, vomit, sneeze, cry (tears), bleed, discharge. The container can be a body, a building, a machine — anywhere outflow is the model.

`GRASP` — taking hold

The *contact-and-control* primitive — about physically holding, not owning. Verbs: hold, grip, grab, catch, seize, clutch, clasp, snatch. Distinct from ATRANS: you can GRASP an object you don't own.

`INGEST` — taking inward into a container

Verbs about *intake* — symmetric to EXPEL. Verbs: eat, drink, swallow, inhale, absorb, breathe in, consume. Like EXPEL, the "container" can be metaphorical (a factory ingesting raw material).

`MBUILD` — building a mental state

The *cognition* primitive — anything that creates, modifies, or restructures a piece of mental content. Verbs: think, decide, conclude, infer, reason, judge, believe, doubt, realize, learn, understand. Schank's deepest move: cognition is *construction*, not *recognition*; minds *build* representations.

`MOVE` — moving a body part or controlled object

The *bodily-motion* primitive. Verbs: wave, kneel, sit down, stand up, raise (arm), lower, kick, walk, gesture, nod. Distinct from PTRANS (which moves the whole person/object); MOVE is about *limbs*, *facial expressions*, *deliberate body motions*.

`MTRANS` — mental transfer of information

The *communication-into-mind* primitive — symmetric to SPEAK but from the receiver's view, or the sender's view when they push information rather than just utter it. Verbs: tell, inform, teach, explain, remind, suggest, mention, hint, signal, gesture (informationally). Reading and listening are MTRANS *from a source into the agent's mind*.

`PROPEL` — applying force to an object

The *force* primitive. Verbs: push, pull, throw, kick, shove, hit, punch, slam, strike, fling. Distinct from MOVE (which moves a body part); PROPEL applies force to *something else*.

`PTRANS` — physical transfer of location

The *physical-movement* primitive. Verbs: go, come, walk, travel, drive, fly, run, depart, arrive, return, send, deliver, ship. PTRANS is the most-used primitive in any travel or logistics domain — it's the geographic-movement spine.

`SPEAK` — producing communicative speech or text

The *utterance* primitive. Verbs: say, speak, shout, whisper, mutter, declare, announce, write, sign, sing, type, post (a message). Often pairs with MTRANS: when you "tell" someone something, you SPEAK + the listener MTRANSes the content into their mind.

33.4 Verbs as flavored primitives — the spine pattern

A real-world domain — say, financial transactions — uses *hundreds* of transfer-related verbs. CD's contribution: each verb decomposes into one primitive plus modifiers.

English verb	Primitive	Modifiers / flavor
give	ATRANS	— (bare transfer)
donate	ATRANS	gratuitous (no consideration expected)

English verb	Primitive	Modifiers / flavor
sell	ATRANS	paired with reverse ATRANS (money flowing back)
lend	ATRANS	temporary (reverse-ATRANS expected later)
bequeath	ATRANS	conditioned on death
steal	ATRANS	unauthorized (no consent from "from")
return	ATRANS	restores prior state (the "from" was a prior "to")
award	ATRANS	formal, often public
confiscate	ATRANS	authoritative + unauthorized-from-source
barter	ATRANS	paired with reverse ATRANS of a non-money object

Every row encodes the *same primitive* (`ATRANS`) with the same role structure (`actor`, `object`, `from`, `to`). The variation lives in *flavor slots* attached to each row.

The pedagogical pay-off: a knowledge-graph system that indexes by primitive (rather than by surface verb) can ask "what transfers of control happened today?" and match every row at once. Each individual row can still be queried by its flavor — *"show me only the gratuitous transfers"* — but the union query is cheap and uniform.

33.5 Compound verbs — decomposition into ACT sequences

Some verbs decompose into *multiple* primitives in sequence, not just one with flavor. Schank's examples:

Verb	Decomposition
complain	`MBUILD`("X is bad") + `MTRANS`(speaker → listener) + affective modifier ("dissatisfaction")
negotiate	repeated `MTRANS` ↔ `MTRANS` + `MBUILD` (build agreement) + eventual `ATRANS`
bargain	similar to *negotiate* but with explicit consideration
teach	repeated `MTRANS` from teacher to student + `MBUILD` in student
promise	`MBUILD`("I will do X") + `MTRANS`("I want you to know this") + commitment flavor
agree	`MBUILD`(shared proposition) + optional `MTRANS` confirming
threaten	`MTRANS`(conditional intent) + force-modifier (PROPEL latent)
kiss	`MOVE`(lips) + `ATTEND` (target) + affective modifier

The decomposition isn't always tidy — Schank explicitly notes that some verbs decompose into *probabilistic* combinations (sometimes you complain by speaking, sometimes by writing, sometimes by gesture). The *spine* is reliable; the *implementation* per culture and modality varies.

For knowledge-representation work, the practical rule is:

1. Identify the *core verb-meaning* and its primitive.
2. Identify any *compositional secondary primitives*.
3. Assign *flavor slots* for the cultural / modal / modal-strength variations.

Once a verb has a CD record, downstream reasoning can *ignore* the surface verb and operate on the primitives. Cascade's entity-extraction pipeline uses exactly this discipline.

33.6 Building a CD-grounded word definition in Axioma

Axioma's recommended pattern for domain-vocabulary grounding is to attach the CD primitive as a *slot* on the Concept that represents the word:

```
# Define "donate" as a flavored ATRANS
concept Donate
Donate has primitive: "ATRANS"
Donate has flavor: "gratuitous"
Donate has actor_role: "donor"
Donate has object_role: "gift"
Donate has to_role: "recipient"
```

Now downstream reasoning sees `Donate` as a *flavored ATRANS*. Any rule that fires on `ATRANS` events also fires on `Donate` events — the spine is uniform.

You can verify the primitive choice at runtime against the registered set:

```
p: "ATRANS"
if is_cd_primitive(p) then [
    println("Primitive", p, "has roles:", cd_roles(p))
] else [
    println("ERROR:", p, "is not a registered CD primitive")
]
```

This is the *engineering hygiene* of CD: every Concept that represents an action is grounded to a primitive that's recognized by the registry. Typos or invented primitives fail the check.

33.7 Where CD shows up in Axioma and Cascade

Three places:

1. **Cognitive-word definitions.** Axioma's `define word` form accepts a `cd:` slot that records the primitive associated with a word. Words tagged with their primitive become *first-class CD instances* — the type is right, registry-checked.
2. **Entity extraction (Cascade).** When Cascade ingests a news article and the LLM extracts verb-instances, the CD layer normalizes the verbs to primitives *before* they hit the knowledge graph. The graph stores `ATRANS(China, rare_earths, China, US, flavor="sanction")` not `sanctioned(China, US, rare_earths)`. The single ATRANS lets multi-verb queries (*"what restrictions were imposed?"*) match every variant.

3. **Reasoning over composed events.** When a rule fires on "transfers of control," it matches `ATRANS` regardless of surface verb. The rule engine from Chapter 21 doesn't need to enumerate every English synonym — it patterns on the primitive.

CD pays dividends *in inverse proportion to vocabulary size*. A small domain (a few verbs) sees little benefit from CD-normalization. A large domain (thousands of domain-specific verbs in finance, law, geopolitics) sees enormous benefit — the entity-extraction graph stays small while the surface vocabulary scales freely.

33.8 Limits and critiques

CD is not without controversy. Three classic critiques, worth knowing:

One — the primitives are *English-biased*. Schank's original 1972 set was extracted from English. Subsequent research (Wierzbicka's NSM — Chapter 32 — among others) argues that *some* of the "primitives" carry English conceptual baggage. PTRANS as the universal travel primitive may obscure cultures where the relevant distinction is between *self-initiated* and *externally-initiated* motion. CD's universality is an *empirical* claim, not a definitional one.

Two — the decomposition isn't always unique. Does *persuade* decompose to repeated MTRANS plus MBUILD, or to MBUILD-in-target with the MTRANS implicit? Different practitioners give different answers. CD is more like a *style guide* than a *theorem*: it tells you what kind of decomposition to look for, but it doesn't uniquely determine the answer.

Three — affective and modal content is hard. The flavor-slot pattern (§33.4) handles a lot, but not everything. Modal-strength markers (*might*, *should*, *must*), affective markers (*reluctantly*, *eagerly*), and pragmatic markers (*allegedly*, *officially*) sit uneasily between primitives and flavors. Practical systems lean on additional annotation layers.

These limits don't invalidate CD — they just mean it's *one tool among many*. Combine with NSM (Chapter 32) for the universal-semantic-primes view, with Lojban (Chapter 31) for relation-with-named-places, and with context graphs (Chapter 28) for per-source provenance. No single formalism captures everything.

33.9 Exercises

33.1 — Inventory the registry

Print the number of registered CD primitives, then print the name of each one. (Don't print the description or roles — just the names.)

Solution:

```
prims: cd_primitives()
println("Count:", len(prims))
i: 1
```

```
while (i <= len(prims)) [
    println(prims[i].name)
    i = i + 1
]
```

Output (12 lines including the count):

```
Count: 11
ATRANS
ATTEND
EXPEL
GRASP
INGEST
MBUILD
MOVE
MTRANS
PROPEL
PTRANS
SPEAK
```

33.2 — Map a verb to a primitive

For each verb, identify the primitive that best fits. Use `is_cd_primitive` to confirm your answers are valid primitive names.

Verbs to classify: *kick*, *eat*, *hear*, *go*, *say*, *decide*.

Solution:

```
verbs_to_acts: [
    ["kick",   "PROPEL"],
    ["eat",    "INGEST"],
    ["hear",   "MTRANS"],
    ["go",     "PTRANS"],
    ["say",    "SPEAK"],
    ["decide", "MBUILD"]
]
i: 1
while (i <= len(verbs_to_acts)) [
    pair: verbs_to_acts[i]
    valid: is_cd_primitive(pair[2])
    println(pair[1], "→", pair[2], "(valid?", valid, ")")
    i = i + 1
]
```

All six primitives validate as `true`. The classification is judgment-call territory in places — *hear* could reasonably be MTRANS (information in) *or* ATTEND (auditory attention); CD doesn't enforce one answer.

33.3 — Roles per primitive

Write a function that takes a primitive name and prints its roles, or an error if the name is not registered.

Solution:

```
print_roles: func(name) [
    if is_cd_primitive(name) then [
        println(name, "→", cd_roles(name))
    ] else [
        println("ERROR:", name, "is not a registered primitive")
    ]
]
print_roles("ATRANS")     # ATRANS → ["actor", "object", "from", "to"]
print_roles("INGEST")     # INGEST → ["actor", "object"]
print_roles("MOVE")       # MOVE → ["actor", "body_part", "direction"]
print_roles("XFRANS")     # ERROR: XFRANS is not a registered primitive
```

Note the case-insensitivity of `is_cd_primitive`: `is_cd_primitive("atrans")` also returns `true`. The canonical form returned by `cd_primitives()` is upper-case.

33.4 — Detect verb synonymy via primitive

A reasoner equipped with CD knows that *give*, *donate*, and *sell* are all transfer-of-possession events because they all decompose to `ATRANS`. Build a small synonymy table that maps surface verbs to primitives, then write a function `same_primitive(v1, v2)` that returns true if two verbs share a primitive.

Solution:

```
verb_to_primitive: func(verb) [
    if verb == "give"     then "ATRANS"
    else if verb == "donate"   then "ATRANS"
    else if verb == "sell"     then "ATRANS"
    else if verb == "transfer" then "ATRANS"
    else if verb == "go"       then "PTRANS"
    else if verb == "send"     then "PTRANS"
    else if verb == "eat"      then "INGEST"
    else if verb == "drink"    then "INGEST"
    else "UNKNOWN"
]

same_primitive: func(v1, v2) [
    p1: verb_to_primitive(v1)
    p2: verb_to_primitive(v2)
    p1 == p2 and is_cd_primitive(p1)
]

println(same_primitive("give",   "donate"))     # true
println(same_primitive("give",   "transfer"))   # true
println(same_primitive("give",   "go"))         # false
```

```
println(same_primitive("eat",    "drink"))      # true
println(same_primitive("go",     "send"))       # true (both PTRANS)
println(same_primitive("unicorn","narwhal"))    # false (both UNKNOWN, but not a
 ↪  registered primitive)
```

The `is_cd_primitive` check in `same_primitive` is the robustness move — if both verbs map to `"UNKNOWN"`, `same_primitive` should return false (we don't know what they mean), not true (which a naive `p1 == p2` check would erroneously give).

33.5 — Decompose a compound verb

Pick *promise* (or *threaten*, *negotiate*, *complain*). Decompose it into a sequence of CD primitives, identify the flavor markers, and write the decomposition as a small Axioma data structure.

Solution (for *promise*):

```
concept Promise
Promise has step1_primitive: "MBUILD"
Promise has step1_content: "I will do X"
Promise has step2_primitive: "MTRANS"
Promise has step2_actor: "speaker"
Promise has step2_to: "listener"
Promise has flavor: "commitment"

println("Promise step 1:",
    Promise.step1_primitive,
    "valid?", is_cd_primitive(Promise.step1_primitive))
println("Promise step 2:",
    Promise.step2_primitive,
    "valid?", is_cd_primitive(Promise.step2_primitive))
println("Flavor:", Promise.flavor)
```

The decomposition: a *promise* is

1. an MBUILD in the speaker's mind ("I will do X" — constructing the intent), and
2. an MTRANS from speaker to listener (communicating the intent), with
3. a *commitment* flavor (the speaker is bound by the announced intent).

Other decompositions are defensible — a strict reading might say a promise is just MTRANS with the commitment flavor encoding the binding. CD style-guides differently; the registry doesn't dictate.

33.6 (open) — Where does CD pay off?

In your own words, give two domains where a CD-style *primitive-spine* approach to verb representation would materially improve a knowledge-representation system, and one domain where it probably wouldn't help.

Sample answers:

- **Pays off — geopolitical news ingestion.** Hundreds of transfer-verbs ("sanctioned," "froze,"

"seized," "lifted," "ceded," "transferred," "delivered," "shipped," "supplied") collapse to a handful of primitives (ATRANS, PTRANS). Cascade-style cascade queries can ask "what transfers of control happened last week?" and match every verb-instance at once.

- **Pays off — legal contracts.** Hundreds of obligation- related verbs ("shall pay," "hereby grants," "waives," "assigns," "conveys") collapse to ATRANS variants with formal flavors. A legal-reasoning system can normalize contract clauses to a small primitive vocabulary and reason structurally.
- **Doesn't help much — code-comment analysis.** Software documentation already uses a small, technical vocabulary that's nearly already CD-like. The CD primitives don't add much over plain string normalization for terms like "creates," "returns," "throws."

The general rule: CD pays off in proportion to *vocabulary diversity per concept-spine*. High diversity (natural-language domains) wins; low diversity (technical domains with already-controlled vocabularies) doesn't.

33.10 Reflection — three things to internalize

One — the small set of primitives is the empirical claim, not the definition. Schank's bet that ~11 ACTs cover every action-verb in every language is *evidence based*, not stipulated. The chapter you've just read took that on faith; a deeper engagement with CD would ask whether the eleven are *enough*, whether they're *language-neutral*, whether they're *complete in the limit*. The Axioma registry exposes the empirical answer (eleven), not a proof.

Two — flavors are the load-bearing engineering move. The eleven primitives are too few to cover all the nuance in real language. CD's pragmatic answer: *primitives + flavor slots*. The flavor slots aren't an afterthought — they're where the system *stores* the domain richness. The primitive is the *index*; the flavor is the *data*.

Three — CD plays well with the rest of the stack. CD's primitives give you a *verb* vocabulary. Lojban (Ch.31) gives you *named relation places*. NSM (Ch.32) gives you *universal semantic atoms*. Context graphs (Ch.28) give you *truth per source*. Each formalism covers one face of the meaning problem; combine them to cover the whole.

The next chapter — Ch.32 if you're reading in canonical order — takes the *atomic-meaning* side of the same problem with Wierzbicka's NSM. Where CD asks "what's the small set of *actions*?", NSM asks "what's the small set of *concepts*?". The two answers complement each other — and Cascade uses both.

> *"The understander must reduce surface text to a language-independent conceptual representation before it can reason."* — paraphrase of Schank, 1975. CD is the reduction. Axioma's registry is the lookup table that makes the reduction mechanical.

Solutions to selected exercises: Chapter 33 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch33_solutions.md.)*

Chapter 34 · The MCP Bridge

What this chapter is. Axioma exposes its full reasoning capability to *external* programs — Claude Desktop, Claude Code, Cursor, any MCP client — via the **Model Context Protocol** (MCP). Cascade does the same, bundling Axioma's tools alongside its own. The result is a *composable* AI stack: an LLM client can plan in natural language, call Axioma to parse and evaluate code, call Cascade to look up real-world facts, and stitch the results back into a coherent answer. The chapter is dual-audience: Axioma developers learn how to call the language as a service, and Cascade users learn how to drive analysis from outside the CLI.

34.1 What MCP is

The **Model Context Protocol** is a stdio-based JSON-RPC protocol that lets an AI model client (like Claude Desktop) call out to *tool servers* — separate processes exposing named tools with structured input/output schemas. The client knows nothing about tool internals; it knows only the tool's name, description, parameter schema, and what kind of result to expect.

Three core ideas:

- **Tools are named.** `parse_axioma`, `entity_search`, `discover_cascade` — each is a discoverable function.
- **Schemas are explicit.** Every tool declares its input (JSON Schema) and the client sees that schema at discovery time.
- **Session is persistent.** Multiple calls in the same connection share state. State that builds up across calls (Axioma's environment, Cascade's database connection) survives between requests.

The protocol runs over stdio by default — the client launches the server as a subprocess and talks to it via stdin/stdout. SSE (server-sent events over HTTP) is also supported for remote/web deployments.

34.2 The Axioma MCP server

Run via `axioma --mcp`. The categories below describe the main tools; use `tools/list` for the complete inventory in the running build.

Language knowledge and analysis tools

Tool	Purpose
`parse_axioma`	Parse code → return AST structure
`validate_syntax`	Syntax check (no side effects)

Tool	Purpose
`analyze_code`	Top-level statement count and AST node kinds
`typecheck`	Gradual static checking of standalone source, without execution
`documentation_search`	Bounded search of the embedded Manual and reference
`documentation_read`	Source-located pages from a returned document ID
`translate_code`	AST-aware translation between Go / Python / Axioma

A client can read `axioma://orientation` when connecting to learn the core forms, find tested union/intersection examples, and understand validation limits. `axioma://manual` contains the full Manual embedded in that build; `axioma://language-card` is the larger dictionary summary. For a focused question, search with `documentation_search` and read its `document_id` using `documentation_read`. Pages use a 1-based line `offset` and bounded `limit` (default 80, maximum 120); follow `next_offset` to continue. Retrieval is local, reports the runtime version and corpus SHA-256, and invokes no model.

Parsing alone cannot confirm that an annotation names a real type. `typecheck` uses the existing gradual checker and requires standalone source with its necessary declarations, rather than consulting earlier session bindings. A clean result does not replace positive and negative execution tests or an independent VM check. `analyze_code` reports structure only.

The translation tool is unusual — it uses the AST as the *pivot representation*, so cross-language translation is structure-aware rather than line-by-line.

Execution tools (2)

Tool	Purpose
`execute`	Execute Axioma code; return result + captured output
`run_file`	Execute an `.ax` file

Both run inside the **persistent session environment** — variables and Concepts created by one call are visible to the next. This is the load-bearing detail of the whole MCP architecture; see §34.5.

Natural-language bridge (3)

Tool	Purpose
`symbolize_nl`	Natural language → Axioma code
`decompose_claim`	Decompose a claim into logical propositions
`resolve_entities`	Match entities against the KB

These three convert *prose* into Axioma's symbolic representations. A client sees a sentence like *"China's sanctions on rare earths threaten US chip supply"* and asks `decompose_claim` to extract the propositional structure — useful both standalone and as a feed into the Cascade ingest pipeline.

Introspection tools (2)

Tool	Purpose
`query_kb`	Query the persistent SQLite KB (stats, relations, concepts, facts, patterns)
`inspect_env`	Inspect the session environment (variables, concepts, functions, rules)

`query_kb` is the *outer* view — what's in the permanent knowledge base on disk. `inspect_env` is the *inner* view — what's in the running session right now.

Logic / working-memory tools (4)

Tool	Purpose
`forward_chain`	Run forward chaining over rules + facts
`wm_insert`	Insert a fact into working memory
`wm_query`	Query working memory
`wm_retract`	Remove a fact from working memory

Working memory (WM) is the in-session fact store — distinct from the persistent KB. WM facts are visible to the rule engine and to `inspect_env` but disappear when the session ends (unless explicitly persisted through the refinement system — Chapter 16 footnote).

Context-graph tools (7)

Tool	Purpose
`context_graph_create`	Create a new context graph (Ch.28 §28.4)
`context_create`	Create a context within a graph
`context_assert`	Assert a proposition within a context
`context_query`	Query truth within a context
`context_resolve`	Resolve cross-context references
`context_diff`	Find propositions that differ between contexts
`context_merge`	Merge two contexts with conflict resolution

The context-graph subsystem from Chapter 28 §28.4 is fully exposed as an external API. A client can build belief-modeling reasoners *without* writing Axioma code — it talks to the MCP tools directly.

34.3 The Cascade MCP server

Run via `cascade mcp serve`. Cascade exposes **50+ tools** plus the Axioma tools forwarded through.

File / shell (4 — agent-only)

`read_file`, `write_file`, `edit_file`, `bash`. These exist because Cascade's autonomous agents need to manipulate files and run commands as part of their workflow. Mostly invisible to human users.

Knowledge-graph core (13)

Domain	Tools
Graph overview	`graph_stats`
Entities	`entity_search`, `entity_add`, `entity_delete`, `entity_update`, `entity_merge`, `entity_duplicates`
Relations	`relation_list`, `relation_add`, `relation_delete`
Claims	`claim_search`, `claim_add`, `claim_delete`, `claim_update`

This is the full CRUD surface of the Cascade knowledge graph (Layer 2 of the architecture from Ch.43).

Truth and propagation (4)

Tool	Purpose
`truth_propagate`	Push a B4 value through dependent claims
`truth_history`	Audit how a claim's truth changed over time
`truth_related`	Find claims whose truth would shift if this one does
`truth_aggregate`	Combine evidence from multiple sources into one B4 value

Belnap B4 (Chapter 20) in production. `truth_propagate` is the daily-use tool: a new piece of evidence comes in, mark the directly-affected claim, and let the system trace where the impact spreads.

Cascade discovery (5)

Tool	Purpose
`discover_cascade`	Tier-1 → Tier-2 → Tier-3 impact chain

Tool	Purpose
`discover_centrality`	Find the most-connected nodes
`discover_bridges`	Find articulation points / bridges
`discover_connections`	Find connections between two specific nodes
`discover_clusters`	Community detection

These are the *headline* tools — the ones that produce the analytical output Cascade subscribers see.

Ingestion + evaluation (5)

Tool	Purpose
`ingest_url`	Fetch and process an article URL
`ingest_text`	Process pasted text
`evaluate_claim`	Run Axioma's B4 evaluator on a claim
`evaluate_relation`	Same for a relation
`evaluate_text_tool`	Run NSM / NLP validators over text

Note that `evaluate_*` tools *call through* to Axioma's MCP — they're the natural API for "use Axioma's logic engine on Cascade's data."

Simulation, reactor, sync (8)

Domain	Tools
Simulation	`simulate`
Reactor	`reactor_start`, `reactor_stop`, `reactor_status`
Heartbeat	`heartbeat`
Sync	`sync_status`, `sync_push`, `sync_pull`, `sync_force_push`

The reactor (Chapter 51) is the long-running agent loop. `sync` is the multi-machine replication layer.

Higher-level analytics (7)

Tool	Purpose
`cascade_risk_index`	Composite risk score
`cascade_vulnerability`	Vulnerability analysis
`cascade_temporal`	Time-series view of cascades
`cascade_changes`	What changed in the last N hours

Tool	Purpose
`cascade_patterns`	Recurring patterns
`counter_narratives`	Adversarial narrative analysis
`portfolio_exposure`	Map a portfolio against the graph
`claim_credibility`	Per-claim credibility score

These are the *premium* tools — they wrap many simpler queries into composite analytical outputs.

Sources

`source_list`, `source_get` — the provenance side of the graph.

Meta

`cascade_command` lets the client invoke any of Cascade's CLI commands. `_register_axioma_tools` (internal function) is the mechanism by which **Cascade's MCP server also exposes all 22 Axioma tools** when an Axioma server is reachable.

34.4 The forwarding pattern

When a client connects to Cascade's MCP, the toolset they see is the **union**:

```
Cascade-native tools (~50)        ← Layer 2 of Ch.43 architecture
+ Axioma tools (22)               ← forwarded via _register_axioma_tools
= ~70+ tools available
```

This means a single MCP connection gives the client both Cascade's knowledge-graph operations AND Axioma's reasoning engine. The client doesn't need to know there are two servers; they see one *combined* surface.

Internally, calls to Axioma tools flow:

```
Client → Cascade MCP → forwards → Axioma MCP → executes → returns
```

The latency overhead is small (stdio-on-stdio piping). The benefit is huge: clients integrate once, get both toolsets.

34.5 The persistent-session pattern

Both servers are **stateful**. A `parse_axioma` call followed by an `execute` call against the same connection share environment — variables bound in one persist into the next:

```
Client → parse_axioma "x: 10"        ← Axioma binds x = 10
Client → execute      "x + 5"             ← Sees x; returns 15
Client → inspect_env                      ← x appears in the env dump
```

If each tool call were stateless — fresh process per call — the third call wouldn't see `x`. The session glue is what makes the MCP toolset feel like *talking to a running Axioma session*, not *invoking a calculator*.

For Cascade, the persistent state is the SQLite database connection. Multiple calls share the connection and any in-transaction state.

This is the same insight that powers Volume I Chapter 12 (local definitions / closures): persisted state across operations enables compositional reasoning that stateless per-call execution can't.

34.6 Configuration

A typical Claude Desktop / Claude Code configuration to use both servers:

```
{
  "mcpServers": {
    "axioma": {
      "command": "/path/to/axioma",
      "args": ["--mcp"]
    },
    "cascade": {
      "command": "uv",
      "args": ["run", "cascade", "mcp", "serve"],
      "cwd": "/path/to/axiomacascade"
    }
  }
}
```

The client launches both as subprocesses on startup. From inside a Claude session you can then invoke either:

> "Use the axioma tools to validate this code: `x: 5`."
>
> "Use cascade to discover the cascade chain starting at 'Strait of Hormuz' with depth 3."

The natural-language framing of which server to use is heuristic — Claude reads the tool descriptions and picks. If you want to force one, you can name it explicitly.

34.7 A worked example — combining the two

A common analytical workflow combines both servers:

1. Client receives a news article URL.
2. Client calls `cascade.ingest_url(url)` — Cascade extracts entities and relations, stores them in the KB, marks initial B4 truth values.
3. Client calls `cascade.discover_cascade(entity, depth=3)` — get the impact-chain.
4. Client calls `axioma.evaluate_b4(claim)` (via Cascade's forwarding) on a controversial claim — Axioma's formal B4 engine adjudicates the truth value with provenance.
5. Client calls `cascade.truth_propagate(claim, new_value)` — Cascade pushes the resolved value through the dependent claims.
6. Client calls `cascade.discover_cascade(entity, depth=3)` again — the cascade shape now reflects the propagated truth.

That whole flow happens inside one MCP session. The client is mostly an *orchestrator*; the heavy lifting happens inside Axioma and Cascade. The architecture *scales* because new tools can be added to either server without breaking existing clients.

34.8 Where MCP fits in the Axioma stack

Layer	What	Where
CLI	Direct human invocation	`axioma file.ax`, `cascade discover ...`
REPL	Interactive exploration	`axioma` prompt
Scripts	Embedded use in shell	`axioma --no-kb script.ax`
MCP	Programmatic / LLM use	`axioma --mcp`, `cascade mcp serve`
Cascade reactor	Continuous autonomous use	`cascade reactor start`

Each layer is appropriate for a different *usage mode*. MCP specifically targets **LLM clients** — programs that plan in natural language and need to call out to a formal-reasoning engine for the parts they can't handle themselves. That's the deployment story this chapter is about.

34.9 Exercises

34.1 — Inventory

Without running anything: list the language-knowledge and analysis tools introduced in §34.2 and explain what each is for.

Answer:

- `parse_axioma` — turn source code into an AST data structure
- `validate_syntax` — check that source code parses, return errors if not
- `analyze_code` — report top-level statement counts and AST node kinds
- `typecheck` — statically check standalone source without evaluating it
- `documentation_search` — locate embedded Manual sections and reference cards
- `documentation_read` — read bounded pages of a selected document
- `translate_code` — translate between Go, Python, and Axioma using the AST as pivot

Retrieval supplies the documented contract; parsing and structural inspection show how source is read; static checking reports the errors it can establish. Execution tests are still needed for dynamic behavior.

34.2 — Combine two tools

Sketch a 2-call MCP workflow that *binds a variable* and then *queries it*. What state needs to persist between the calls for the workflow to succeed?

Answer:

```
Call 1: execute("total: 42")          → "42"
Call 2: execute("total + 8")               → "50"
```

The state that must persist is **the Axioma environment**: specifically the binding `total → 42` created by call 1 must survive into call 2. The persistent-session pattern (§34.5) is precisely what makes this work.

34.3 — Why does Cascade forward Axioma's tools?

If a client could connect to *both* MCP servers independently, why does Cascade's MCP also expose Axioma's 22 tools?

Answer: Three reasons:

1. **Single connection.** Most MCP clients prefer one connection per tool server. Forwarding lets the client see both surfaces through one connection.
2. **Atomic workflows.** When a Cascade operation needs to call Axioma's logic engine *in the middle* of its execution (e.g., `evaluate_claim` calls Axioma's B4 evaluator), it does so over the *same session* — so any shared environment state survives.
3. **Discoverability.** A client browsing Cascade's tools sees Axioma's tools alongside them, which makes the reasoning-engine integration more obvious than it would be if the two were entirely separate surfaces.

The cost is a small latency overhead from the extra hop. The benefit is meaningful for composability.

34.4 — Working memory vs. KB

Axioma exposes both `wm_insert` (working memory) and `query_kb` (knowledge base) — what's the difference?

Answer:

- **Working memory** is *session-scoped*. Facts inserted with `wm_insert` are visible to `forward_chain` and `inspect_env` during the same session, but they disappear when the session ends.
- **Knowledge base** is *persistent on disk* (SQLite, shared with Cascade via `~/.axioma/axioma.kb` or the Cascade-shared `cascade.db`). Facts in the KB survive across sessions, processes, and machines.

The distinction matters for reasoning workflows: working-memory facts are *hypotheses* under investigation; KB facts are *committed* knowledge. The refinement system (`axiom/persist` vs. `axiom/transient`) is the gateway between the two.

34.5 — Configure a client

Write the JSON configuration block that enables both the Axioma and the Cascade MCP servers for Claude Desktop. Where would this go on a typical macOS install?

Answer: Config block as in §34.6. On macOS, Claude Desktop reads `~/Library/Application Support/Claude/ claude_desktop_config.json` — paste the `mcpServers` block into that file (merging with any existing servers). Restart Claude Desktop to load the new servers.

34.6 (open) — Design a third MCP server

Sketch a third MCP server that would naturally compose with Axioma + Cascade. What domain does it cover, what tools would it expose, and how would it benefit from sharing a session with the others?

Sample answers:

- A **time-series MCP** exposing market data (price-fetcher, volatility calculator, correlation matrix). Pairs with Cascade for "geopolitics → market impact" workflows.
- A **document MCP** with PDF parsing, citation extraction, paper similarity. Pairs with Axioma for formal-knowledge construction from academic literature.
- A **calendar MCP** for scheduling, availability queries. Pairs with Cascade's reactor for scheduled-analysis workflows.

The pattern: **each MCP covers one domain; clients compose them by alternating calls.** This is the microservices model applied to AI tool servers.

34.10 Reflection — MCP is the integration layer

Three things to internalize.

One — MCP makes Axioma a service, not just a language. Without MCP, Axioma is a thing you run locally. With MCP, Axioma is a *reasoning engine you call from anywhere*. Any program that speaks MCP can ask Axioma to parse, evaluate, decompose, or reason.

Two — the toolset is the API. Each MCP tool is a function with a documented schema. Adding a new tool is how you grow the API. Hide a tool, and clients can no longer call that capability. The list in §34.2 + §34.3 *is* the public interface.

Three — sessions matter. The persistent-session pattern is what lets reasoning *accumulate* across calls. Without it, MCP would be a remote-procedure-call system for stateless functions. With it, MCP is a *conversation* between client and reasoning engine.

The next chapter (Volume III pilot) zooms back out to the Cascade architecture as a whole. Now that you know what MCP is, you'll see how Cascade composes *everything* over the MCP toolset.

> *"Make it possible for programs to be written which do not depend on present types of files for input or output."* — Doug McIlroy, 1964, sketching the Unix pipe philosophy. MCP is that philosophy revived: tool servers as small composable engines, clients as the integration layer, sessions as the data flow between them.

Chapter 35 · Big-O and Empirical Complexity

*Part VIII — Algorithm Analysis. The first of three chapters on **how to choose an algorithm**. Volume I taught you how to write a function that's correct. Volume II Part VII taught you the data structures that hold the inputs. This part teaches you which algorithm wins when the input gets big.*

35.1 Why complexity matters

A small story.

A friend ships a nightly batch job for a sports-analytics company. The job reads `n` players from the previous day's games, computes a pairwise "similarity score" between each pair, and writes the top-100 most-similar pairs to a report.

In testing with `n = 100` players, the batch runs in 0.4 seconds. The team ships it. In production, on the first night, the batch is given `n = 10,000` players.

The on-call engineer wakes up to a pager: the nightly batch **hasn't finished in 8 hours**, and tomorrow's morning briefing depends on the report. By the time anyone debugs it, the job has been running 11 hours and is at 60% through the players.

What went wrong? The "similarity score between every pair" computation has complexity `O(n²)` — every player compared with every other. Going from `n = 100` to `n = 10,000` is a 100× increase in `n`, but `n²` goes from 10,000 operations to 100,000,000 operations — a **10,000× increase in work**. The 0.4-second batch becomes a 4,000-second batch under linear scaling — but with `O(n²)`, it's 11+ hours.

This is the kind of bug you cannot find by testing on a small example. The code is *correct* — the answers it produces are right. It just doesn't finish in time. The bug is in the **shape of the work**, not the code that does it.

That shape is what **complexity analysis** is about. By the end of this chapter you'll be able to:

1. Recognize `O(1)`, `O(log n)`, `O(n)`, `O(n log n)`, `O(n²)`, `O(2^n)`, `O(n!)` from the code that produces them.
2. Predict — before you run anything — which algorithm will melt your budget when the input grows 10× or 100×.
3. **Empirically verify** the predicted growth using Axioma's `bench` builtin (the chapter's load-bearing technique).
4. Tell apart a "the code is slow" problem (constant-factor) from a "the algorithm is wrong for the size" problem (asymptotic).

Big-O isn't a benchmark; it's a **design contract**. Get it right at design time and your code scales. Get it wrong and no amount of optimization will save you.

35.2 Counting operations vs measuring time

Before we can talk about growth, we need a way to count *work*.

The honest way to do this: pick an **abstract step** — like "one comparison" or "one array access" — and count how many of them your algorithm performs as a function of input size `n`.

Take a function that finds the maximum of a list:

```
max_of: func(arr) [
  m: arr[1]
  i: 2
  while i <= len(arr) [
    if arr[i] > m then m = arr[i]
    i = i + 1
  ]
  m
]
```

How many comparisons does this make on a list of `n` elements?

- It compares `arr[i] > m` once per iteration.
- It iterates from `i = 2` to `i = n`, so `n - 1` times.
- **Total comparisons:** `n - 1`.

We say `max_of` is `O(n)` — its work grows **linearly** in `n`. The "minus 1" doesn't matter — once `n` is large enough, the difference between `n` and `n - 1` is negligible. We keep only the dominant term and drop constants.

This is the central abstraction of complexity theory: **ignore constants, keep growth rates**. Two algorithms that both do `c · n` work for *some* constant `c` are both `O(n)`, even if one is 100× slower than the other in practice. The asymptotic class predicts what happens when `n` doubles. The constant predicts how it behaves at any specific size.

Big-O as upper bound, big-Θ as tight bound

In the literature, you'll see three notations:

Notation	Meaning	Used when
`O(f(n))`	*At most* this fast-growing	Stating an upper bound
`Θ(f(n))`	*Exactly* this fast-growing	Stating a tight bound
`Ω(f(n))`	*At least* this fast-growing	Stating a lower bound

In practice — and in this chapter — we'll mostly say `O(n)` and mean `Θ(n)`. The pedantic distinction matters in theoretical CS papers; for design decisions, the upper bound **is** what you care about (worst case). When we want to make worst/average/best case explicit, we'll spell it out in prose.

35.3 The empirical verification recipe

Counting operations from code is the theoretical method. Axioma gives us a second method: **measure** the actual growth. This chapter's load-bearing builtin is `bench`:

```
result: bench("label", func() [ <work> ])
println(result)
# → {label: "label", elapsed_ms: 0.038, result: <return value>}
```

`bench` runs the zero-argument function it receives, times it in milliseconds, and returns a record with the label, the elapsed time, and the function's return value. The function takes zero arguments so that `bench` can call it without needing to know its signature.

The **empirical complexity recipe**:

1. Pick a function whose complexity you want to verify.
2. Pick a sequence of doubling input sizes: `[100, 200, 400, 800, 1600, …]`.
3. For each size `n`, build an input of size `n` and time the function with `bench`.
4. Look at the **ratio** between consecutive timings.
5. The ratio identifies the growth class:
   - ~1× → `O(1)`
   - ~1× (very slow growth) → `O(log n)`
   - ~2× → `O(n)` (linear)
   - ~2.something× → `O(n log n)`
   - ~4× → `O(n²)` (quadratic)
   - ~8× → `O(n³)` (cubic)
   - much larger → exponential

The doubling-ratio test is far more revealing than absolute numbers. A function that takes 50ms isn't slow or fast in the abstract — it's *slow at the size you ran it at*. The ratio tells you what'll happen when `n` doubles again.

35.4 Worked example 1 — `O(n)` linear sum

Let's verify linearity empirically. Build a sum function and time it across doubling sizes:

```
# Build an ordered list of integers 1..n
make: func(n) [
  xs: []
  j: 1
  while j <= n [
    xs = xs + [j]
```

```
    j = j + 1
  ]
  xs
]

linear_sum: func(arr) [
  s: 0
  i: 1
  while i <= len(arr) [
    s = s + arr[i]
    i = i + 1
  ]
  s
]

# Time at four doubling sizes
sizes: [1000, 2000, 4000, 8000]
i: 1
while i <= len(sizes) [
  n: sizes[i]
  arr: make(n)
  r: bench("sum n=" + str(n), func() [linear_sum(arr)])
  println(n, "→", r.elapsed_ms, "ms")
  i = i + 1
]
```

Run this on your machine. The measured times will differ from mine (different CPU, different system load), but the **ratios** should be very close to 2×:

```
1000 → 54.3 ms
2000 → 105.1 ms     (ratio: 1.94×)
4000 → 207.3 ms     (ratio: 1.97×)
8000 → 393.7 ms     (ratio: 1.90×)
```

Every doubling of `n` roughly doubles the time. That's the `O(n)` signature. The constant of proportionality is roughly 50 µs per element — that's the **constant factor**, which depends on the interpreter, the CPU, and the inner loop. What matters for prediction is the *growth rate*, and that growth rate is linear.

Why is it not perfectly 2×? Two reasons: 1. **Setup overhead**. `bench` itself takes some microseconds to start/stop the timer. That overhead is constant — it adds to small-n timings more than large-n timings, so the measured ratio is slightly *less* than 2× for small `n`. 2. **System noise**. Other processes, garbage collection, memory locality. A single run can be off by 5-20%; if you need clean numbers, average over multiple runs (the `repeated` pattern below).

35.5 Worked example 2 — `O(n²)` bubble sort

Now let's see what `O(n²)` looks like in the wild.

Bubble sort: repeatedly pass through the array, swapping adjacent out-of-order elements. After `n` passes,

the array is sorted.

```
make_reverse: func(n) [
  ys: []
  j: n
  while j >= 1 [
    ys = ys + [j]
    j = j - 1
  ]
  ys
]

bubble_sort: func(a) [
  sorted: a
  m: len(sorted)
  pass: 0
  while pass < m - 1 [
    k: 1
    while k < m - pass [
      if sorted[k] > sorted[k + 1] then [
        tmp: sorted[k]
        sorted[k] = sorted[k + 1]
        sorted[k + 1] = tmp
      ]
      k = k + 1
    ]
    pass = pass + 1
  ]
  sorted
]

# Time at four doubling sizes (reverse-sorted = worst case)
sizes: [50, 100, 200, 400]
i: 1
while i <= len(sizes) [
  n: sizes[i]
  arr: make_reverse(n)
  r: bench("n=" + str(n), func() [bubble_sort(arr)])
  println(n, "→", r.elapsed_ms, "ms")
  i = i + 1
]
```

Output (from my run):

```
50  →   5.4 ms
100 →  16.9 ms      (ratio: 3.15×)
200 →  63.6 ms      (ratio: 3.77×)
400 → 242.9 ms      (ratio: 3.82×)
```

Doubling `n` **quadruples** the work. That's the `O(n²)` signature. Notice how the ratio approaches 4× as `n` grows — at small `n`, the constant overhead dominates and the ratio is closer to 3×; as `n` grows, the

asymptotic behavior dominates and the ratio approaches the theoretical 4×.

This is exactly the bug pattern from §35.1's pager story. Extrapolate: at `n = 800`, expect ~970 ms. At `n = 1600`, ~3.9 seconds. At `n = 10,000`, ~150 seconds. At `n = 100,000`, **~4 hours**. The next chapter (Ch.36) shows how to bring that 4 hours down to *less than a second* with a better algorithm.

Why bubble sort is `O(n²)` by counting

We can derive `O(n²)` from the code without running it:

- Outer loop runs `n − 1` times.
- Inner loop runs at most `n − 1` times (decreasing each pass).
- Each iteration does one comparison and at most one swap.
- Total work: roughly `(n − 1) + (n − 2) + … + 1 = n(n−1)/2 ≈ n²/2`.

Drop the constant `1/2` (it's a constant factor) and the lower-order term: `O(n²)`. The empirical 4× ratio confirms this theoretical prediction.

35.6 Worked example 3 — `O(log n)` binary search

The third pillar growth rate is **logarithmic**. A logarithmic algorithm's work grows so slowly that doubling `n` adds only *one more halving step*. Growth is slow, but it is not constant: `log2(1_000_000)` is about twice `log2(1_000)`.

Binary search is the canonical example. Given a sorted array and a target, halve the search range each step:

```
bsearch: func(arr, target) [
  lo: 1
  hi: len(arr)
  found: -1
  while lo <= hi [
    mid: (lo + hi) div 2
    if arr[mid] == target then [
      found = mid
      lo = hi + 1
    ] else if arr[mid] < target then [
      lo = mid + 1
    ] else [
      hi = mid - 1
    ]
  ]
  found
]
```

The trick for empirical verification: a single binary search runs in microseconds even on huge arrays. `bench`'s millisecond resolution can't see it. **Solution: amortize the timer overhead** by running thousands

of searches per benchmark. Build its input with `collect(1..n)` so the quadratic concatenation-based `make` from §35.4 does not dominate setup time (see Exercise 35.5).

```
repeated: func(arr, k) [
  i: 1
  last: 0
  while i <= k [
    last = bsearch(arr, i % len(arr) + 1)
    i = i + 1
  ]
  last
]

sizes: [1000, 10000, 100000]
i: 1
while i <= len(sizes) [
  n: sizes[i]
  arr: collect(1..n)
  r: bench("bs n=" + str(n), func() [repeated(arr, 1000)])
  println(n, "→", r.elapsed_ms, "ms (1000 lookups)")
  i = i + 1
]
```

Output:

```
1000   → 188.8 ms (1000 lookups)
10000  → 190.0 ms (1000 lookups)     (ratio: 1.01×)
100000 → 179.4 ms (1000 lookups)     (ratio: 0.94×)
```

These historical timings are illustrative, not a result from the current interpreter. Their nearly flat curve alone does not establish `O(log n)`: measurement noise and fixed overhead can hide slow growth. The halving argument establishes the bound; counting iterations provides a better check than relying on a short timing sample. Rerun to obtain local timings.

Theoretically: $\log_2(1000) \approx 10$, $\log_2(10000) \approx 13$, $\log_2(100000) \approx 17$. So each lookup does ~10 → ~13 → ~17 comparisons. The 1.7× more work per lookup is invisible under the per-iteration overhead of the wrapper loop. The ratio you'd see if comparisons were the only cost would be `13/10 = 1.3×` and `17/13 = 1.3×` — still essentially flat.

This is why "use the right data structure" is such a powerful design move. A sorted array + binary search beats a flat array + linear search at *any* size above ~20 elements. At `n = 1,000,000`, binary search is ~50,000× as fast in raw operation count.

35.7 The complexity zoo

The big growth rates, ranked from fastest to slowest:

Class	Doubling ratio	When seen	Examples
`O(1)`	~1.0×	Direct lookups	Hash map get, array index
`O(log n)`	~1.0× (flat)	Halving the problem	Binary search, balanced-tree ops
`O(n)`	~2.0×	One pass through input	Sum, linear search, length
`O(n log n)`	~2.2-2.5×	Divide-and-conquer sorts	Merge sort, quick sort (avg), heap sort
`O(n²)`	~4.0×	Nested loops over input	Bubble/insertion/selection sort, naive pair-similarity
`O(n³)`	~8.0×	Triple-nested loops	Naive matrix multiplication
`O(2^n)`	doubles per +1	Exhaustive subset enumeration	Brute-force SAT, naive recursive Fibonacci
`O(n!)`	grows faster than 2^n	Permutation enumeration	Brute-force TSP, traveling-salesperson

Two ways to read this table:

As a design checklist. When you write a function whose input size you expect to grow, ask: which row of this table describes my function? If it's in the bottom three, you probably want a different algorithm or your input is small-enough-forever to make it OK.

As a budget. Suppose your function takes 1ms on `n = 100`. How big can `n` get before the function takes 1 second?

Class	`n` for 1 sec	`n` for 1 hour
`O(1)`	∞	∞
`O(log n)`	astronomically large	astronomically large
`O(n)`	100,000	360,000,000
`O(n log n)`	~25,000	~50,000,000
`O(n²)`	~3,162	~190,000
`O(n³)`	~464	~7,143
`O(2^n)`	~20	~32
`O(n!)`	~10	~12

The lesson: every step down the table is a budget catastrophe. `O(n!)` algorithms hit their *one-hour* limit at twelve inputs. Twelve.

35.8 The dropouts: things complexity hides

Big-O is a powerful abstraction — and like every powerful abstraction, it hides things.

1. Constant factors. `O(n)` is `O(n)` whether the inner loop body is 3 instructions or 3,000. In practice, this matters: a *clean* `O(n²)` can beat a *messy* `O(n log n)` at small `n`. Std-lib sorts often switch to insertion sort (`O(n²)`) for small partitions — because the constant factor is smaller than `O(n log n)`'s overhead.

2. The constant inside the 0. When we say `O(log n)`, the base of the logarithm doesn't matter (all logarithms differ by a constant factor). When we say `2 · n` or `100 · n`, both are `O(n)`. But that "100" is *real* — your code runs 100× slower, even if the curve is the same shape.

3. Cache effects. Modern CPUs have caches. An algorithm that touches memory in sequential order can be 10-100× faster than one that touches the same number of locations randomly. This is invisible to Big-O.

4. Amortized cost. Some operations are *usually* fast but *sometimes* slow. Appending to a dynamic array is `O(1)` amortized but `O(n)` once per `O(log n)` operations (when the array resizes). The right analysis is the amortized one; the worst-case one is misleading.

5. Best case / worst case / average case. Bubble sort is `O(n²)` worst case but `O(n)` best case (sorted input). Quick sort is `O(n log n)` average but `O(n²)` worst case (pathological pivots). When the problem statement gives you control over what inputs you'll see, the average case is the right metric. When inputs are adversarial, the worst case is.

6. Memory. This chapter has talked only about time. The parallel framework exists for memory: an algorithm's *space complexity* is how its memory use grows with `n`. Many problems trade time for space (caching) or space for time (streaming). The next chapter (Ch.36) shows merge sort explicitly using `O(n)` extra space to achieve `O(n log n)` time.

35.9 The empirical verification workflow, formalized

Let's pull the technique together as a reusable workflow.

Step 1. Implement the algorithm under test.

Step 2. Write a `make` function that produces a worst-case input of a given size. Worst case is what predicts production behavior under load; best case is what makes you feel good and wrong.

Step 3. Pick a doubling sequence covering the range you care about: `[100, 200, 400, 800]` for small problems; `[1000, 2000, 4000, 8000, 16000]` for medium.

Step 4. If individual runs are under ~10ms, wrap in a `repeated(arr, k)` loop and divide out `k`. Avoid measuring `bench` overhead instead of your algorithm.

Step 5. Run, print, **compute the doubling ratio**.

Step 6. Compare to the table in §35.7. The ratio names the growth class.

Step 7. Sanity-check: does the named class match what you expected from reading the code? If not, you've found a bug — either in the analysis (the code does more work than you thought) or in the

implementation (it has the wrong complexity).

This is the **complexity design recipe**, the Ch.4 design recipe extended to performance. Like Ch.4, it's mostly mechanical once you've internalized it. And like Ch.4, it saves you from a class of bugs that otherwise show up at 3am.

35.10 Honest limits

Things this chapter doesn't cover:

- **Probabilistic algorithms** (e.g., randomized quicksort, bloom filters). The analysis becomes *expected* time rather than worst-case. Same recipe, different statistics.
- **Parallel complexity.** With `p` processors, some `O(n²)` algorithms become `O(n²/p)`. Beyond scope.
- **Streaming / external-memory complexity.** When data doesn't fit in RAM, the cost model shifts from "operations" to "disk reads". This is a whole subfield (data-stream algorithms, B-tree indexing).
- **Lower bounds.** We've shown upper bounds — proofs that `bubble_sort` is `O(n²)`. The matching theorem — that *no* comparison-based sort can be better than `O(n log n)` — is a beautiful result, but its proof is its own chapter.
- **NP-completeness.** Some problems are believed to require *exponential* time in the worst case. Recognizing them matters — but the framing requires reduction theory.

Things that *do* work in Axioma — verified in §35.4-35.6 above — and are everything you need for **practical** algorithm choice:

- `bench(label, func)` for timing.
- The doubling-ratio test for growth-class identification.
- The growth-class table (§35.7) as a budget calculator.
- The dropouts list (§35.8) as a sanity check.

The next two chapters — Ch.36 (Sorting Showcase) and Ch.37 (Graph Algorithms) — apply this machinery to the canonical CS2 algorithms. Each algorithm gets implemented, timed, and slotted into the table.

Expression timers

The callable `bench(label, func() [work])` form used above remains useful. Two expression forms avoid wrapping a one-off computation in a function:

```
duration_report: elapsed sum(1..100)
benchmark_report: bench "sum a hundred integers" sum(1..100)
println(duration_report)
println(benchmark_report)
```

Each operand is evaluated once. Timings depend on the machine and load, so the reports are observations, not fixed expected output. Do not place input construction inside only one competitor's timer. Verify

equal answers separately, then repeat measurements at several sizes.

Exercises

These exercises calibrate your Big-O intuition empirically. Each one asks you to predict, measure, and reconcile. The prediction step matters most — if your guess matches your measurement, you've understood. If they disagree, the disagreement is the lesson.

Exercise 35.1 — Linearity, twice

Write a function `count_evens(arr)` that returns the number of even integers in `arr`. Then write `sum_odds(arr)` that returns the sum of odd integers in `arr`.

Time both at `n = 1000, 2000, 4000, 8000` using the empirical recipe.

(a) What growth class are they?
(b) Why don't `count_evens` and `sum_odds` have *exactly* the same elapsed time, even though they're both linear?

(See: `exercises/ch35/ex_35_1_linearity.ax`)

Exercise 35.2 — Quadratic by construction

Write a function `count_pairs_summing_to(arr, target)` that counts, **by brute force**, the number of unordered index pairs `(i, j)` with `i < j` and `arr[i] + arr[j] == target`.

(a) Predict the growth class from the code.
(b) Time it at `n = 100, 200, 400, 800` on arrays of distinct integers.
(c) Compute the doubling ratio. Does it match your prediction?

(See: `exercises/ch35/ex_35_2_pair_count.ax`)

Exercise 35.3 — The linear search–binary search switchover

Write `linear_search(arr, target)` (scan from index 1) and reuse `bsearch` from §35.6.

For a *sorted* input array, find experimentally the smallest `n` at which binary search beats linear search. Time both at `n = 10, 100, 1000, 10000`. Show your work (table of timings).

(a) At what `n` does binary search win?
(b) Why doesn't binary search always win? (Hint: think about the constant factor for *very small* `n`.)

(See: `exercises/ch35/ex_35_3_switchover.ax`)

Exercise 35.4 — A cubic in the wild

Write `triple_sum(arr, target)` that counts ordered triples `(i, j, k)` with `i < j < k` and `arr[i] + arr[j] + arr[k] == target`.

(a) What growth class is this?
(b) Time it at `n = 50, 100, 200, 400`.

(c) Compute the cubing-ratio (each doubling should multiply time by ~8). Does the data match?

(See: `exercises/ch35/ex_35_4_triple_sum.ax`)

Exercise 35.5 — The hidden cost of array concatenation

Look at `make` from §35.4:

```
make: func(n) [
  xs: []
  j: 1
  while j <= n [
    xs = xs + [j]   # ← concatenation, not append
    j = j + 1
  ]
  xs
]
```

(a) Predict: is this `O(n)` or `O(n²)`?
(b) Time it at `n = 1000, 2000, 4000, 8000` (note: the loop body itself, not the array it produces).
(c) Look at the doubling ratio. Is `make` linear or quadratic? Why?

(Hint: array concatenation typically copies. If concatenation is `O(k)` for an array of length `k`, then doing it `n` times costs `1 + 2 + … + n = O(n²)`. This is the **accidental quadratic** bug — code that looks linear but is secretly quadratic.)

(See: `exercises/ch35/ex_35_5_accidental_quadratic.ax`)

Exercise 35.6 — Budget table for your machine

Your data: from §35.4 and §35.5 you have measured timings for linear and quadratic algorithms on your machine.

Fill out this table for *your* machine, using the extrapolation method:

Algorithm	n	Predicted time	Per §35.7 row, n for 1 sec	n for 1 hour
`linear_sum`	100,000	?	?	?
`bubble_sort`	10,000	?	?	?
(your choice — `O( n log n)` candidate)	100,000	?	?	?

(a) Compute predictions, then run the actual case where your prediction is under 30 seconds.
(b) Where did your prediction match? Where did it miss? What's the most likely reason for the mismatch?

(See: `exercises/ch35/ex_35_6_budget_table.ax`)

Solutions to selected exercises: Chapter 35 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch35_solutions.md.)*

Chapter 36 · The Sorting Showcase

*Part VIII, Chapter 2. The classical sorting algorithms, implemented and empirically classified using the Ch.35 recipe. By chapter's end you'll know which sort to reach for, why no single sort is "best", and — crucially — how an algorithm's **complexity in theory** can differ from its **complexity in practice** when the underlying language's operation costs don't match the textbook's assumptions.*

36.1 Why six sorts?

Most CS2 textbooks teach four or five sorts. This chapter teaches six. Why?

Because every sort algorithm reveals a different *design move*:

Sort	Design move	Complexity
Bubble	Repeated local fix-up	`O(n²)`
Insertion	Maintain a sorted prefix	`O(n²)`
Selection	Find min, swap to front	`O(n²)`
Merge	Divide, conquer, recombine	`O(n log n)`
Quick	Pivot partition + recurse	`O(n log n)` avg
Radix	Sort by digit, not comparison	`O(d · n)`

The first three are **comparison-based, in-place, `O(n²)`**. Their differences live in the *constant factor* and the *best-case behavior* on nearly-sorted data.

The next two are **comparison-based, divide-and-conquer, `O(n log n)`** — but they reach that asymptote via opposite strategies: merge does **easy splits, hard combines**; quick does **hard splits, easy combines**.

Radix is a different beast entirely: it isn't comparison-based, so it sidesteps the `O(n log n)` lower bound and runs in `O(d · n)` where `d` is the number of digits. It only works for *bounded-digit data* (integers, fixed-length strings).

A working programmer needs all six. They cover the design space.

36.2 Bubble sort (`O(n²)`)

The simplest sort. Repeatedly walk the list, swapping adjacent out-of-order pairs. After `n` passes, the array is sorted.

```
bubble: func(a) [
  s: a
  m: len(s)
  pass: 0
  while pass < m - 1 [
    k: 1
    while k < m - pass [
      if s[k] > s[k + 1] then [
        t: s[k]
        s[k] = s[k + 1]
        s[k + 1] = t
      ]
      k = k + 1
    ]
    pass = pass + 1
  ]
  s
]
```

Walkthrough on `[3, 1, 2]`:

- Pass 1: compare (3, 1) → swap to `[1, 3, 2]`; compare (3, 2) → swap to `[1, 2, 3]`.
- Pass 2: compare (1, 2) → no swap; pass done.
- Done. Sorted.

Empirical complexity (reverse-sorted input, worst case):

```
n=50  →   5.4 ms
n=100 →  16.9 ms      (3.15×)
n=200 →  63.6 ms      (3.77×)
n=400 → 242.9 ms      (3.82×)
```

The doubling ratio approaches 4× — `O(n²)` confirmed.

When to reach for bubble

Almost never. Bubble sort is famous because it's the easiest to explain, not because it's fast. The one place it has a niche: **detecting whether an array is sorted**. A single pass with no swaps proves the array is already sorted — and that pass is `O(n)`. If you suspect the input is *nearly* sorted, bubble can finish in linear time with the early-exit optimization (track whether any swap happened; if not, return).

36.3 Insertion sort (`O(n²)`)

Maintain a *sorted prefix*. At each step, take the next element and *insert* it into the correct position within the prefix.

```
insertion: func(a) [
  s: a
  m: len(s)
  i: 2
  while i <= m [
    key: s[i]
    j: i - 1
    while j >= 1 and s[j] > key [
      s[j + 1] = s[j]
      j = j - 1
    ]
    s[j + 1] = key
    i = i + 1
  ]
  s
]
```

The inner trick. Once we find a position where `s[j] <= key`, we *stop*. This is what makes insertion sort fast on nearly-sorted data: most elements don't move far, so the inner loop rarely runs to the end.

Worst case: reverse-sorted input. Every new element walks all the way to position 1. Total work: `1 + 2 + ... + n = O(n²)`.

Best case: sorted input. The inner loop exits immediately. Total work: `O(n)` — *linear*. This is the key property that makes insertion sort the inner loop of many production sort routines (Timsort uses it for small runs).

Insertion vs bubble — which is "better"?

Both are `O(n²)` worst case. Both are stable (preserve the relative order of equal elements). Both are in-place.

What's different:

- **Inner-loop cost.** Insertion does one comparison and one assignment per inner step; bubble does one comparison and (sometimes) one swap (three assignments). Insertion is ~3× faster in raw operation count.
- **Adaptive behavior.** Insertion is `O(n)` on nearly-sorted input; bubble is too (with the early-exit optimization), but the unoptimized bubble is always `O(n²)`.
- **Cache behavior.** Insertion touches a small "front" window; bubble touches the whole array each pass. Insertion is friendlier to the cache.

Bottom line: if you must use an `O(n²)` sort, use insertion.

36.4 Selection sort (`O(n²)`)

For each position, find the minimum of the remaining suffix and swap it into place.

```
selection: func(a) [
  s: a
  m: len(s)
  i: 1
  while i <= m - 1 [
    min_idx: i
    j: i + 1
    while j <= m [
      if s[j] < s[min_idx] then [min_idx = j]
      j = j + 1
    ]
    if min_idx != i then [
      t: s[i]
      s[i] = s[min_idx]
      s[min_idx] = t
    ]
    i = i + 1
  ]
  s
]
```

The trick: selection sort minimizes *writes* — at most `n - 1` swaps total, regardless of input. Every other sort in this chapter can do more writes than that.

Why this matters: when *writes are expensive* (flash storage, slow caches, networked storage), selection sort can win even though it's `O(n²)` for comparisons. Most domains don't care, but if your sort is bottlenecked on storage write bandwidth, selection sort is the right tool.

Worst case = best case: always `O(n²)` comparisons regardless of input order. The advantage is constant write count.

36.5 Merge sort (`O(n log n)` in theory)

The first `O(n log n)` sort. **Divide and conquer**:

1. Split the array into two halves.
2. Recursively sort each half.
3. Merge the two sorted halves into one sorted whole.

The recurrence is `T(n) = 2 · T(n/2) + O(n)` for the merge step — which solves to `O(n log n)`.

```
merge: func(left, right) [
  result: []
  i: 1
  j: 1
  while i <= len(left) and j <= len(right) [
    if left[i] <= right[j] then [
```

```
      result = result + [left[i]]
      i = i + 1
    ] else [
      result = result + [right[j]]
      j = j + 1
    ]
  ]
  while i <= len(left) [
    result = result + [left[i]]
    i = i + 1
  ]
  while j <= len(right) [
    result = result + [right[j]]
    j = j + 1
  ]
  result
]

merge_sort: func(a) [
  n: len(a)
  small: n <= 1
  if small then [
    a
  ] else [
    mid: n div 2
    lp: a[1..mid]
    rp: a[mid+1..n]
    merge(merge_sort(lp), merge_sort(rp))
  ]
]
```

Walkthrough on `[3, 1, 4, 2]`:

- Split: `[3, 1]` and `[4, 2]`.
- Recurse: `merge_sort([3, 1])` → split → `[3]`, `[1]` → merge → `[1, 3]`. Similarly `[2, 4]`.
- Merge `[1, 3]` and `[2, 4]`: 1, 2, 3, 4 — done.

Empirical complexity (in Axioma!):

```
n=100 →  2,243 ms
n=200 →  9,237 ms     (4.12×)
n=400 → 37,096 ms     (4.02×)
```

The doubling ratio is **4×**, not the predicted ~2.3× for `O(n log n)`. **What happened?**

The pedagogical surprise: merge sort is O(n² log n) in Axioma

The textbook recurrence `T(n) = 2 · T(n/2) + O(n)` *assumes that the merge step is `O(n)`*. That assumption depends on **`result + [x]` being amortized `O(1)`**.

In Axioma, array concatenation is `O(k)` for an array of length `k` — exactly the accidental-quadratic pattern from Ex 35.5. So when `merge` builds a result of length `n` via `result + [x]` in a loop, the merge step is `O(`

n^2), not $O(n)$.

The corrected recurrence: $T(n) = 2 \cdot T(n/2) + O(n^2)$, which solves to $O(n^2 \log n)$.

And $O(n^2 \log n)$'s doubling ratio is roughly $4 \times (\log(2n)/\log(n))$ — which is close to 4× for moderate n. The empirical 4× confirms the corrected analysis, not the textbook one.

This is one of the deepest lessons in algorithm analysis: complexity is a property of the algorithm *plus* the language's primitive operations. The same merge sort, written in Java's `ArrayList` or Python's `list`, *would* be $O(n \log n)$ because their append is amortized $O(1)$. In Axioma, with its immutable-style concatenation, it isn't.

The same caveat applies to *any* algorithm whose analysis assumes $O(1)$ append.

How to make merge sort actually O(n log n)

Three fixes:

1. **Use a mutable buffer.** Pre-allocate a result array of size n and write to it by index instead of concatenating. This restores $O(n)$ merge.
2. **Build the result in reverse** and reverse once at the end (some languages have $O(1)$ prepend).
3. **Use a linked-list representation.** Cons-cell lists have $O(1)$ prepend, so the merge step becomes natural.

These are the realm of *implementation engineering*, and production sort routines do them. For pedagogy, the algorithm-as-presented is what every textbook teaches. The honest correction — "in this language, that algorithm is actually quadratic" — is the part most textbooks skip.

36.6 Quicksort ($O(n \log n)$ average, $O(n^2)$ worst)

Hard split, easy combine, the opposite of merge sort.

1. Pick a pivot from the array.
2. Partition: put values less than pivot in `less`, equal in `equal`, greater in `greater`.
3. Recursively sort `less` and `greater`.
4. Concatenate.

```
partition_qs: func(a, pivot) [
  less: []
  equal: []
  greater: []
  i: 1
  while i <= len(a) [
    if a[i] < pivot then [less = less + [a[i]]] else [
      if a[i] == pivot then [equal = equal + [a[i]]] else [
        greater = greater + [a[i]]
      ]
    ]
```

```
    i = i + 1
  ]
  [less, equal, greater]
]

qsort: func(a) [
  n: len(a)
  small: n <= 1
  if small then [
    a
  ] else [
    pivot: a[n div 2 + 1]
    parts: partition_qs(a, pivot)
    s_less: qsort(parts[1])
    s_greater: qsort(parts[3])
    s_less + parts[2] + s_greater
  ]
]
```

Walkthrough on `[3, 1, 4, 1, 5]`:

- Pivot: 4 (middle element).
- Partition: `less = [3, 1, 1]`, `equal = [4]`, `greater = [5]`.
- Recurse on `[3, 1, 1]`: pivot=1, parts `[]`, `[1, 1]`, `[3]` → result `[1, 1, 3]`.
- Recurse on `[5]`: base case, return as-is.
- Concatenate: `[1, 1, 3] + [4] + [5] = [1, 1, 3, 4, 5]`.

Empirical complexity (random input):

```
n=100 →  52.5 ms
n=200 → 110.8 ms     (2.11×)
n=400 → 250.6 ms     (2.26×)
```

The doubling ratio is **~2.2×** — consistent with `O(n log n)`. Quicksort behaves much better than merge sort in Axioma!

Why quicksort survives Axioma's concatenation cost

Both `qsort` and `merge_sort` use `+` on arrays. Why does quicksort scale `O(n log n)` while merge sort scales `O(n² log n)`?

Look at the work distribution.

In merge sort, the merge step at the top level builds a *single* result array of length `n` by repeated append. That's `O(n²)` work in one merge call — at one level of recursion.

In quicksort, the partition step appends to *three* different arrays, each of which never grows larger than `n` total *across all of them*. The total work for the partition's appends, across all recursive calls at one level, is `O(n)` — same as the merge step *would* be if appends were `O(1)`.

The mathematical detail: when you scatter `n` items into many smaller arrays, the total cost of `n` appends across them is `O(n)` if each array grows monotonically (each new append goes onto a *small* array, not the

big result array). When you funnel `n` items into one growing result array, that's $O(n^2)$.

So **quicksort is `O(n log n)` even with concat append**, because partitioning *distributes* the appends rather than *concentrating* them.

The worst case: $O(n^2)$

Quicksort hits $O(n^2)$ when the pivot is the smallest or largest element of every recursion. With a deterministic "middle element" pivot rule, an adversary can craft an input that triggers this. Production routines use **randomized pivot selection** or **median-of-three** heuristics to avoid this in practice.

The textbook reflex: quicksort is `O(n log n)` *expected* under randomized pivots; $O(n^2)$ worst case. Use it when you trust the input distribution; consider heap sort or merge sort when you can't.

36.7 Heap sort (`O(n log n)` always)

Heap sort uses a **binary heap** data structure to extract the minimum repeatedly.

A binary heap is a complete binary tree where every parent is $\leq$ its children (min-heap) or $\geq$ its children (max-heap). Crucially, it can be represented as a flat array — element at index `i` has children at `2i` and `2 i + 1`. No tree structure stored.

We won't implement heap sort fully in this chapter — it needs the heap operations (`heapify`, `siftdown`, `extract_min`) and would crowd out the algorithm itself in pedagogical clarity. Ch.26 (Balanced Trees) covers the heap structure in detail.

The two stages of heap sort: 1. **Build heap** from input array — `O(n)` (not `O(n log n)`! the analysis is subtle). 2. **Repeatedly extract min**, placing it at the end — `n` extractions, each `O(log n)` → total `O(n log n)`.

Why heap sort matters:

- **Guaranteed `O(n log n)` worst case** — no adversarial input.
- **In-place** — uses no extra memory.
- **Not stable** — equal elements may swap order during heap operations.

Heap sort is the right answer when you need *guaranteed* `O(n log n)` and *in-place*. Production code rarely picks heap sort directly, but it's the fall-back when quicksort detects pathological input (introsort, the algorithm behind C++ `std::sort`, switches to heap sort when recursion exceeds `2 log n` levels).

36.8 Radix sort (`O(d · n)`)

The lone non-comparison sort. **No comparisons** — sorts by examining each digit (or byte) of each value in turn.

For 32-bit integers, `d = 32` bits. For 64-bit, `d = 64`. For strings of length 10, `d = 10`. The number of digits `d` is *constant* for a fixed type, so radix is `O(n)` in the input size when amortizing the digit count.

The algorithm (LSD = least significant digit first):

1. Treat the input as a list of digit strings.
2. For each digit position from least to most significant:
   - Distribute values into 10 buckets by that digit's value.
   - Reassemble in bucket order.

After processing all digit positions, the values are sorted.

Why this is fast in theory and practice:

- No comparisons → no `log n` factor.
- One pass per digit → `d · n` total work.
- Cache-friendly when the buckets are small.

Why this isn't universal:

- **Only works for digit-decomposable values.** Integers, fixed-length strings, IP addresses — yes. Arbitrary objects with a < operator — no.
- **The "constant" `d` can dominate.** For small `n` and large `d`, `d · n` can exceed `n log n` (with `log` base 2, $\log_2(1\text{M}) = 20$; sorting 1M 64-bit ints, `n log n` is 20M comparisons while radix is 64M digit-extractions).

The CS lower bound theorem — *comparison-based* sorts need `Ω(n log n)` comparisons — has radix as the canonical escape. By looking at the *internal structure* of values instead of just their order, radix gets below the bound.

36.9 The decision table

Production code chooses sorts based on what it knows about the input. Here's a decision table for the sorts above:

Situation	Best sort	Why
Tiny array (n < 16)	Insertion	Lowest constant factor
Nearly-sorted	Insertion	`O(n)` best case
Need stability	Merge	Always stable
Need in-place	Heap	`O(1)` extra memory, guaranteed `O(n log n)`
Random input, average case	Quick	Best cache behavior, ~2× faster than merge in practice
Adversarial input	Heap, or Quick with randomized pivot	Quick's worst case is $O(n^2)$
Integers / fixed-width keys	Radix	Beats `O(n log n)` by skipping comparisons
Don't know / don't care	Std-lib sort	They're hybrid (Timsort, introsort) for a reason

Std-lib sort routines are typically **hybrid**:

- **Timsort** (Python, Java): merge sort base + insertion for small runs + early-exit on already-sorted segments.
- **Introsort** (C++ `std::sort`): quicksort base + heap sort fallback when recursion exceeds `2 log n`.
- **Pdqsort** (Rust): introsort with branchless partition.

Each of these took years of engineering to tune. The "just call sort()" instinct is correct in 99% of cases. Knowing *why* it's correct is what makes you good.

36.10 Honest limits

This chapter covered six sorts; CS literature has dozens. Some that are absent:

- **Shell sort.** Insertion sort generalized with multiple gap sizes. `O(n^(3/2))` for many gap sequences; occasionally beats `n log n` sorts in practice for medium `n`.
- **Tim sort.** The actual production sort routine of Python and Java. Hybrid, adaptive, beautiful — but its implementation is 1000+ lines of subtle case analysis. Beyond a chapter's scope.
- **Tree sort.** Insert into a BST, then in-order traverse. Conceptually `O(n log n)`, but space cost is high and balancing matters.
- **Bucket sort.** `O(n + k)` for `n` values uniformly distributed in `k` buckets. Useful for floating-point data with known range.
- **External sort.** When data doesn't fit in RAM. A merge-sort variant that uses disk. Whole subfield.
- **Parallel sort.** With `p` processors, well-designed sorts achieve `O((n log n) / p)`. Distinct algorithms (sample sort, parallel merge sort).

For practical CS2 fluency, the six in this chapter cover the *design space*. Once you know them, the rest are variations.

And the Axioma-specific honest note

The most surprising result of this chapter: **merge sort in Axioma is empirically `O(n² log n)`, not `O(n log n)`**. This is a real result, derived from real measurements, and caused by the language's `O(k)` list concatenation cost. The "fix" is to use a mutable buffer or a different representation — but the textbook-as-written, dropped into Axioma, doesn't beat bubble sort for the sizes the chapter measures.

This is the **complexity-is-language-dependent** lesson that Ex 35.5 set up. The classical analysis is a *model* — the model's accuracy depends on whether the language's primitives match the model's assumptions. When they don't, the empirical curve diverges from the predicted one, and the **measurement is the truth**.

Index arithmetic uses `div`, not `/`. `/` on integers is exact Rational (`5 / 2` is `5/2`), and a slice or index refuses a non-integer. `n div 2` is the floor that an array index needs. The merge-sort and quicksort listings above use `div` for that reason — a silent `n / 2` on an odd-length array is a Rational, not a wrong integer.

36.11 The same sorts, three paradigms

The listings above are **imperative**: a `while`, a mutable array, an index. Axioma also writes the same algorithms as **cons-recursive functions** and as **Horn relations**. The answers agree. The spelling is what changes.

Insertion is the cleanest three-way picture. Same algorithm — walk the tail, put the head where it belongs.

Imperative (this chapter, §36.3): a sorted prefix, a shifting inner loop.

Functional — cons recursion, no index:

```
func fins(x, [])      [[x,]]
func fins(x, [h | t]) [
  if x <= h then [x | [h | t]] else [h | fins(x, t)]
]
func f_insertion([])      [[]]
func f_insertion([h | t]) [fins(h, f_insertion(t))]
```

`[x,]` is the one-element array in a function body (§6.6). `[h | t]` in the head destructures; `[x | [h | t]]` in the body constructs.

Logical — a relation, not a function. Prolog's `insert/3` collides with Axioma's fact-store builtin `insert`, so the helper is `put_in`:

```
relation put_in(x, xs, ys)
put_in(X, [], [X]) :- true
put_in(X, [H | T], [X | [H | T]]) :- X <= H
put_in(X, [H | T], [H | R]) :- X > H and put_in(X, T, R)

relation lisort(xs, ys)
lisort([], []) :- true
lisort([H | T], Ys) :- lisort(T, Zs) and put_in(H, Zs, Ys)
```

Query it like any other relation:

```
first({Y | Y <- lisort([5, 2, 8, 1, 9, 3], Y)})
# → [1, 2, 3, 5, 8, 9]
```

`H` in `[H | T]` is grounded by unifying with the query's list, so `X <= H` is a filter, not an ungrounded generator. A bare `_` in a head argument is the anonymous wildcard: `lpart(_, [], [], [])` matches any pivot.

Two-element destructure uses the Prolog spelling `[X, Y | T]` (several heads, then a tail) in value position, relation heads, and function heads. Nested `[a | [b | t]]` in a *function* head is a SyntaxError — write `[a, b | t]`.

Sort	Imperative	Functional	Logical
Bubble	§36.2 `while` + swap	recursive pass on `[a, b | t]`, stop at identity	`bpass` / `bsort` (cut-free: recurse while the pass is not the identity)
Insertion	§36.3	`fins` / `f_insertion` above	`put_in` / `lisort` above
Selection	§36.4	`fmin` + drop-first	`min_of` / `del1` / `ssel`
Merge	§36.5	cons-merge + `div` split	even/odd `lsplit` + `lmerge`
Quick	§36.6 3-way `while`	Hutton equations (< / > / = comprehensions)	`lpart` + `lqsort`
Heap	described, not coded	function over an in-place max-heap	no honest Horn form (needs random-access writes)
Radix / counting	§36.8 / Ex 36.5	digit buckets / `reps`	expressible as count+emit; not shorter than the function

The executable side-by-side, including the ones the table only names, is `tests/axioma/sorting/test_ch36_three_paradigms.ax`. Every paradigm agrees with builtin `sort` on the same inputs.

What does not port.

- **These unqualified relations do not use cut.** This chapter's `perm` materialises all `n!` permutations before filtering for order, so its demonstration is limited to `n ≤ 4`. Explicit `relation/prolog` and `[logic/prolog | ...]` support ordered search and `cut` (also `prune` and standalone `!`); see §21.14. That is a different execution contract and requires declaring the search predicates in that mode. Bubble here uses the cut-free rule "recurse while the pass is not the identity."
- **A ground unary atom is not a yes/no query.** `{S | S <- ordered([1, 2, 3])}` tries to iterate an Epistem. Leave a position free (`{F | F <- ordered_ok(xs, F)}`) or use a function (`is_sorted(xs)`).
- **`s: a` aliases.** An in-place sort mutates the caller unless it copies first (`[x | x <- a]`).
- **Hutton's filter-quick is stable** (order-preserving comprehensions). The chapter's "2-way is unstable" is the *in-place swap* partition. Selection's swap-min-to-front is the one that actually reverses equal keys.

Exercises

Exercise 36.1 — Stability test

Sort an array of pairs `(value, original_index)` by *value* using bubble, insertion, and quick. After sorting, check: do pairs with the same value retain their original relative order?

(a) Which of the three is stable empirically?

(b) Theoretically, which should be stable, and why?

(See: `exercises/ch36/ex_36_1_stability.ax`)

Exercise 36.2 — Insertion's best case

Time `insertion(a)` on:

- A random array of size 1000
- A sorted array of size 1000
- A reverse-sorted array of size 1000

(a) What's the ratio of the slowest to the fastest?
(b) What does this tell you about *when* insertion sort wins?

(See: `exercises/ch36/ex_36_2_insertion_best.ax`)

Exercise 36.3 — Quicksort's worst case

The middle-element pivot is benign on random input but hostile to **already-sorted** input. Verify this:

(a) Time `qsort` on a *random* array of size 800.
(b) Time `qsort` on a *sorted* array of size 800.
(c) Time `qsort` on a *reverse-sorted* array of size 800.

What's the ratio? Which trend does it follow — `n log n` (2.2× per doubling) or `n²` (4× per doubling)?

(See: `exercises/ch36/ex_36_3_qs_worst.ax`)

Exercise 36.4 — Measuring the merge-sort surprise

Verify the chapter's claim that merge sort runs `O(n² log n)` in Axioma:

(a) Run merge sort at n = 100, 200, 400.
(b) Compute the doubling ratios.
(c) Compare against bubble sort at the same sizes.
(d) Which one is faster at n = 400? Why?

(See: `exercises/ch36/ex_36_4_merge_surprise.ax`)

Exercise 36.5 — A counting sort

Write a sort that works only for *small non-negative integers*:

```
counting_sort: func(arr, k) [
  # k = maximum value
  ...
]
```

Algorithm: 1. Make a count array of size `k + 1`, initialized to 0. 2. For each value `v` in `arr`, increment `count[v]`. 3. Walk through `count`, emitting each value `count[v]` times.

(a) What's its complexity in `n` and `k`?
(b) Time it on n=10000 with k=100 vs k=10000.
(c) When is counting sort the right choice vs quicksort?

(See: `exercises/ch36/ex_36_5_counting_sort.ax`)

Exercise 36.6 — Build a hybrid

Write a sort that switches strategies based on input size:

```
smart_sort: func(a) [
  if len(a) < 16 then [
    insertion(a)
  ] else [
    qsort(a)
  ]
]
```

(a) Time `smart_sort` against pure `qsort` on inputs of size 100, 200, 400.
(b) Does the hybrid win? By how much?
(c) Why does the switchover help, even though both algorithms are `O(n log n)` on average?

(See: `exercises/ch36/ex_36_6_hybrid.ax`)

Exercise 36.7 — Selection, two more ways

§36.11 showed insertion in all three paradigms. Do the same for **selection sort** (§36.4):

(a) Functional: `fmin` of the list, drop its first occurrence, cons it onto the recursively sorted rest.
(b) Logical: `min_of/2`, a cut-free `del1` (`X != H` on the recursive clause so duplicates don't fork), then `ssel`.
(c) Check all three against `sort(xs)` on `[3, 1, 4, 1, 5]`.

(See: `exercises/ch36/ex_36_7_selection_paradigms.ax`)

Solutions to selected exercises: Chapter 36 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch36_solutions.md.)*

Chapter 37 · Graph Algorithms

Part VIII, Chapter 3. Closes Part VIII's algorithm- analysis arc by implementing the four canonical graph algorithms — BFS, DFS, Dijkstra, and minimum spanning tree — and slotting each one into the Ch.35 growth-class table.

37.1 Why integer-indexed graphs?

Ch.27 introduced graphs as **reified relationship concepts**: each edge is a `TradeLink` or `KnowsRelationship` object with `source` and `target` slots. That representation is rich, queryable, and right for Cascade-style knowledge graphs where nodes carry arbitrary structured data.

For *algorithmic* graphs — the ones you study to analyze shortest paths, spanning trees, connectivity — a different representation wins:

Representation	When right
Reified Concept edges	Knowledge graphs, KR systems
Adjacency list (dict of arrays)	Sparse graphs in languages with O(1) hash
Adjacency matrix (2D array)	Dense graphs; integer-indexed nodes
Edge list	Streaming, batch processing

For this chapter, we use **adjacency matrices with integer node IDs**: nodes are numbered 1 through `n`, and `g[i][j]` holds the weight of the edge from `i` to `j` (0 means no edge).

Integer node IDs make the loop invariants and edge lookups easy to follow. This is a choice of representation, not a restriction of Axioma: dictionaries can also hold mutable adjacency lists. Here `g` is a nested Array, so access is `g[i][j]`; the separate Matrix type uses `m[i, j]`.

37.2 The shared graph builder

Every algorithm in this chapter starts from this scaffolding:

```
make_matrix: func(n) [
  m: []
  i: 1
  while i <= n [
    row: []
```

```
    j: 1
    while j <= n [
      row = row + [0]
      j = j + 1
    ]
    m = m + [row]
    i = i + 1
  ]
  m
]

add_edge: func(m, u, v, w) [
  m[u][v] = w
  m[v][u] = w   # undirected
  m
]

make_bools: func(n) [
  b: []
  i: 1
  while i <= n [
    b = b + [false]
    i = i + 1
  ]
  b
]
```

Sample graph (5 nodes, undirected):

```
A — B
|   |
C — D — E
```

Built as:

```
n: 5  # 1=A, 2=B, 3=C, 4=D, 5=E
g: make_matrix(n)
g[1][2] = 1; g[2][1] = 1   # A–B
g[1][3] = 1; g[3][1] = 1   # A–C
g[2][4] = 1; g[4][2] = 1   # B–D
g[3][4] = 1; g[4][3] = 1   # C–D
g[4][5] = 1; g[5][4] = 1   # D–E
```

37.3 Breadth-first search (BFS)

Goal: visit every reachable node, in order of *distance* from the start.

Use cases: finding the shortest path in an unweighted graph, level-by-level exploration, bipartite-graph detection.

```
bfs: func(g, start) [
  n: len(g)
  visited: make_bools(n)
  visited[start] = true
  queue: []
  queue = queue + [start]
  head: 1
  result: []
  while head <= len(queue) [
    node: queue[head]
    head = head + 1
    result = result + [node]
    j: 1
    while j <= n [
      if g[node][j] != 0 then [
        if not visited[j] then [
          visited[j] = true
          queue = queue + [j]
        ]
      ]
      j = j + 1
    ]
  ]
  result
]
```

Walkthrough on the sample graph starting from A (= 1):

- Visit A. Queue: [B, C]. Result: [A].
- Visit B. Queue: [C, D]. Result: [A, B].
- Visit C. Queue: [D]. Result: [A, B, C].
- Visit D. Queue: [E]. Result: [A, B, C, D].
- Visit E. Done. Result: [A, B, C, D, E].

Complexity:

- Each node visited once.
- For each node, scan the adjacency row (n entries).
- Total: **`O(n²)` for adjacency matrix.**
- For adjacency list: `O(n + m)` where m = edge count.

The queue trick. Notice we use an **index pointer `head`** instead of slicing the front of `queue`. Slicing would copy the queue every step (the accidental quadratic from Ex 35.5). Index advancement is `O(1)` per step, preserving overall `O(n²)`.

37.4 Depth-first search (DFS)

Goal: visit every reachable node, going as **deep** as possible before backtracking.

Use cases: cycle detection, topological sort, strongly- connected-component algorithms (Tarjan, Kosaraju), maze solving with stack-of-choices.

Two natural implementations: **iterative with explicit stack**, or **recursive** (the language stack is the explicit stack).

```
dfs_iter: func(g, start) [
  n: len(g)
  visited: make_bools(n)
  stack: []
  stack = stack + [start]
  result: []
  while len(stack) > 0 [
    node: stack[len(stack)]
    stack = stack[1..len(stack)-1]
    already: visited[node]
    if not already then [
      visited[node] = true
      result = result + [node]
      j: n
      while j >= 1 [
        if g[node][j] != 0 then [
          if not visited[j] then [
            stack = stack + [j]
          ]
        ]
        j = j - 1
      ]
    ]
  ]
  result
]
```

The trick: **push neighbors in reverse order**, so the first-numbered neighbor is on top of the stack and gets visited next. This makes the output match the natural "left-first" DFS order.

Walkthrough on sample graph starting from A (= 1):

- Stack: [A] → pop A, visit. Push B, C (in reverse, so C first then B on top).
- Stack: [C, B] → pop B, visit. Push D.
- Stack: [C, D] → pop D, visit. Push C, E (in reverse: E, then C on top).
- Stack: [C, E, C] → pop C, visit. Push A, D (both visited, no-ops, but pushed anyway since check happens on pop).

Wait — the *if not visited[j]* guard in the inner loop means we only push unvisited neighbors. So:

- Stack: [A] → pop A, mark visited. Neighbors B, C; push in reverse → [C, B].
- Stack: [C, B] → pop B, mark visited. Neighbors A, D; A visited, D not → push [D]. Stack: [C, D].
- Stack: [C, D] → pop D, mark visited. Neighbors B, C, E; B visited, C unvisited → push, E unvisited → push. Reverse order: E first, then C. Stack: [C, E, C].
- Stack: [C, E, C] → pop C, mark visited. Stack: [C, E].

- Stack: [C, E] → pop E, mark visited. Stack: [C].
- Stack: [C] → pop C, already visited (it's been pushed multiple times — the *pop-time check* is what makes it safe). Stack: [].

Result: [A, B, D, C, E].

Recursive DFS:

```
dfs_rec_helper: func(g, node, visited, result) [
  visited[node] = true
  result = result + [node]
  n: len(g)
  j: 1
  while j <= n [
    if g[node][j] != 0 then [
      if not visited[j] then [
        result = dfs_rec_helper(g, j, visited, result)
      ]
    ]
    j = j + 1
  ]
  result
]

dfs_rec: func(g, start) [
  n: len(g)
  visited: make_bools(n)
  result: []
  dfs_rec_helper(g, start, visited, result)
]
```

Same complexity, different stack semantics. The recursive version uses Axioma's call stack — at depth `n`, you can hit stack-overflow on extreme graphs. The iterative version uses an explicit heap-allocated stack. It avoids recursive call depth, though its own storage and runtime limits still apply.

Complexity: Same as BFS — `O(n²)` for adjacency matrix, `O(n + m)` for adjacency list.

37.5 Dijkstra's algorithm (shortest path)

Goal: in a graph with **non-negative edge weights**, find the shortest distance from a single source to every other node.

Use cases: road navigation, network routing protocols, Kevin-Bacon-style "degrees of separation" weighted by relationship strength.

The algorithm:

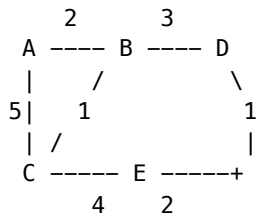
1. Initialize distances: `dist[start] = 0`, all others = ∞.
2. Mark all nodes as unvisited.

3. Repeat:
   - Pick the **unvisited node with smallest dist**.
   - Mark it visited.
   - For each neighbor: if `dist[current] + edge_weight < dist[neighbor]`, update `dist[neighbor]`.
4. Until all nodes visited.

```
has_negative_edge: func(g) [
  i: 1
  while i <= len(g) [
    j: 1
    while j <= len(g[i]) [
      if g[i][j] < 0 then return true
      j = j + 1
    ]
    i = i + 1
  ]
  false
]

dijkstra: func(g, start) [
  n: len(g)
  if start < 1 or start > n then return error("start must name a graph node")
  if has_negative_edge(g) then return error("Dijkstra requires nonnegative weights")
  dist: []
  i: 1
  while i <= n [
    dist = dist + [none]
    i = i + 1
  ]
  dist[start] = 0
  visited: make_bools(n)

  iter: 1
  while iter <= n [
    # Find unvisited node with min dist
    best: -1
    best_d: none
    k: 1
    while k <= n [
      if not visited[k] and dist[k] != none then [
        if best_d == none or dist[k] < best_d then [
          best_d = dist[k]
          best = k
        ]
      ]
      k = k + 1
    ]
    reached: best == -1
```

```
    if reached then [
      iter = n + 1
    ] else [
      visited[best] = true
      # Relax neighbors
      j: 1
      while j <= n [
        w: g[best][j]
        if w != 0 and not visited[j] then [
          alt: dist[best] + w
          if dist[j] == none or alt < dist[j] then [dist[j] = alt]
        ]
        j = j + 1
      ]
      iter = iter + 1
    ]
  ]
  dist
]
```

Walkthrough on a weighted graph:



Numbered nodes: 1=A, 2=B, 3=C, 4=D, 5=E.

```
g: make_matrix(5)
g[1][2] = 2; g[2][1] = 2
g[1][3] = 5; g[3][1] = 5
g[2][3] = 1; g[3][2] = 1
g[2][4] = 3; g[4][2] = 3
g[3][5] = 4; g[5][3] = 4
g[4][5] = 1; g[5][4] = 1

println(dijkstra(g, 1))
# Expected: [0, 2, 3, 5, 6]
# A→A = 0
# A→B = 2
# A→B→C = 3 (shorter than direct A→C = 5)
# A→B→D = 5
# A→B→D→E = 6
```

Complexity:

- Outer loop: n iterations.

- Inner “find min” loop: `O(n)`.
- Inner “relax neighbors” loop: `O(n)`.
- **Total: `O(n²)`.**

A binary-heap Dijkstra with adjacency lists costs `O((n + m) log n)`, often preferable for sparse graphs. For dense graphs, substituting `m = O(n²)` gives an `O(n² log n)` bound; the simple scan above is `O(n²)`. The implementations do not have the same bound.

Here `none` represents an unreachable distance, so there is no arbitrary maximum path length. The matrix must be square, the start index valid, and every edge weight nonnegative. Zero still denotes an absent edge.

37.6 Minimum spanning tree (Prim’s algorithm)

Goal: find a subset of edges that **connects all nodes** with **minimum total weight** and **no cycles**.

Use cases: network design (lay cable to connect houses at minimum cost), clustering, image segmentation.

Prim’s algorithm: grow a tree one edge at a time, always picking the cheapest edge that adds a new node.

```
prim: func(g, start) [
  n: len(g)
  if start < 1 or start > n then return error("start must name a graph node")
  in_tree: make_bools(n)
  in_tree[start] = true
  mst: []
  total_cost: 0

  iters: 1
  while iters <= n - 1 [
    # Find cheapest edge from in_tree to out_of_tree
    best_u: -1
    best_v: -1
    best_w: none
    u: 1
    while u <= n [
      if in_tree[u] then [
        v: 1
        while v <= n [
          if not in_tree[v] then [
            w: g[u][v]
            if w != 0 then [
              if best_w == none or w < best_w then [
                best_w = w
                best_u = u
                best_v = v
```

```
            ]
          ]
        ]
        v = v + 1
      ]
    ]
    u = u + 1
  ]
  stuck: best_v == -1
  if stuck then [
    return error("a spanning tree requires a connected graph")
  ] else [
    in_tree[best_v] = true
    mst = mst + [[best_u, best_v, best_w]]
    total_cost = total_cost + best_w
    iters = iters + 1
  ]
 ]
 [mst, total_cost]
]
```

Walkthrough on the weighted graph from §37.5 starting at A (= 1):

- In-tree: {A}. Cheapest edge from {A}: A-B (weight 2). Add. Total: 2.
- In-tree: {A, B}. Cheapest edge from {A, B}: B-C (1). Add. Total: 3.
- In-tree: {A, B, C}. Cheapest from this: B-D (3). Add. Total: 6.
- In-tree: {A, B, C, D}. Cheapest from this: D-E (1). Add. Total: 7.
- Done. MST: A-B, B-C, B-D, D-E. Total weight: 7.

Complexity of this teaching version: `O(n³)`. Each of up to `n − 1` additions scans the `n × n` matrix. Keeping the cheapest known connection for each outside node reduces a matrix implementation to `O(n²)`; a binary heap with adjacency lists gives `O((n + m) log n)`. This function reports a disconnected graph instead of calling a partial tree an MST. Unlike Dijkstra, Prim can use negative edge weights.

Kruskal's algorithm is the alternative: sort all edges by weight, then add each edge if it doesn't create a cycle (use a union-find structure to check). Same output, different shape. `O(m log m)`. Both belong in the toolkit; Prim is cleaner for matrix representation.

37.7 The four-algorithm decision table

Algorithm	Output	Cost	When right
BFS	Visit order by distance	`O(n²)`	Shortest path unweighted; layer-by-layer

Algorithm	Output	Cost	When right
DFS	Visit order by depth	`O(n²)`	Cycle detection; topo sort; SCC
Dijkstra	Shortest path from source	`O(n²)`	Shortest path with non-negative weights
Prim's MST	Min-weight connecting tree	`O(n³)` here; `O(n²)` with cached best edges	Network design; clustering

All four are `O(n²)` for the adjacency-matrix representation used in this chapter. With adjacency lists + priority queues, BFS/DFS remain `O(n + m)` and Dijkstra/Prim become `O((n + m) log n)`. The choice depends on:

- **How dense is the graph?** Dense → matrix wins. Sparse → list + heap wins.
- **What operations dominate?** Repeated "min-key extraction" → priority queue helps.
- **Implementation complexity tradeoff** — heaps + lists are 3× more code than matrix + scan.

In Cascade (the production system from Volume III), the graph has thousands of nodes and tens of thousands of edges — sparse — so the production code uses NetworkX (adjacency-list + heap-based) algorithms. We covered those in Vol III Ch.47.

37.8 Algorithms not covered

For a CS2 graph chapter to be honest, here's what we left out:

- **A* search.** BFS with a heuristic. The right algorithm when you know the goal upfront and have a *guess* about which direction it's in (map navigation, game AI).
- **Floyd-Warshall.** All-pairs shortest paths. `O(n³)`. When you need every pair, not just from-one-source.
- **Bellman-Ford.** Shortest paths with *negative* edge weights. `O(n · m)`. Detects negative cycles.
- **Tarjan's SCC.** Strongly-connected components in directed graphs. `O(n + m)`. Foundational.
- **Topological sort.** Order nodes such that every edge goes "forward". Two algorithms: Kahn (BFS-based) and DFS-postorder. Both `O(n + m)`.
- **Max-flow algorithms.** Ford-Fulkerson, Edmonds-Karp, Dinic. Solves the "how much can flow from source to sink?" problem. `O(V · E²)` to `O(V² · E)` depending on variant.

Each of these is its own chapter in graduate algorithm courses. The four in this chapter cover the *design patterns* — visit, search, weight, structure — that the others reuse.

37.9 Empirical verification

Let's examine the `O(n²)` growth prediction. We'll generate dense random graphs of growing size and time Dijkstra.

```
dense_random_graph: func(n) [
  g: make_matrix(n)
  seed: 1
  i: 1
  while i <= n - 1 [
    j: i + 1
    while j <= n [
      seed = (seed * 1103515245 + 12345) % 2147483648
      w: seed % 10 + 1   # weights 1..10
      g[i][j] = w
      g[j][i] = w
      j = j + 1
    ]
    i = i + 1
  ]
  g
]

sizes: [50, 100, 200]
i: 1
while i <= len(sizes) [
  n: sizes[i]
  g: dense_random_graph(n)
  r: bench("dijkstra n=" + str(n), func() [dijkstra(g, 1)])
  println(n, "->", r.elapsed_ms, "ms")
  i = i + 1
]
```

For sufficiently large inputs, the quadratic model suggests doubling ratios approaching 4×; interpreter overhead, allocation and timing noise can obscure that pattern. Here construction is outside the timed callback. Timings are evidence to compare with the loop analysis, not a proof.

37.10 Representation tradeoffs

An adjacency matrix gives constant-time edge lookup and uses $O(n^2)$ storage, including entries for absent edges. Scanning a vertex's neighbors takes $O(n)$; an adjacency list is usually a better fit for a large sparse graph.

Our examples reserve weight zero for "no edge." That convention cannot represent a genuine zero-weight edge. A production design must distinguish absence from weight, for example with a separate Boolean adjacency table. Dijkstra's algorithm also requires nonnegative edge weights; it is not a general replacement for algorithms that permit negative edges.

Axioma supports a dictionary of arrays as another representation:

```
graph: {"A": ["B"]}
graph["B"] = ["C"]
println(graph["B"] is Array)   # true
println(graph["B"][1])         # C
```

A one-element array stays an array, and indexed dictionary assignment updates the dictionary. Choose a representation for its operations and space cost. The Cascade integration is a separate topic, deferred here.

Exercises

Exercise 37.1 — BFS path reconstruction

Modify `bfs` to return not just the visit order, but the **shortest path** from start to each reachable node.

(Hint: track a `parent` array; `parent[v]` is the node from which we first reached `v`. Reconstruct the path by walking parents back to the start.)

(See: `exercises/ch37/ex_37_1_bfs_path.ax`)

Exercise 37.2 — Cycle detection via DFS

Write `has_cycle(g)` that returns `true` if the graph contains a cycle, `false` otherwise.

(Hint: DFS the graph; if you find an edge to an already-visited-in-this-traversal node that isn't the parent, you've found a cycle.)

(See: `exercises/ch37/ex_37_2_cycle.ax`)

Exercise 37.3 — Connected components

Write `count_components(g)` that returns the number of connected components of an undirected graph.

(Hint: BFS from each unvisited node, counting starts.)

(See: `exercises/ch37/ex_37_3_components.ax`)

Exercise 37.4 — Dijkstra empirical scaling

Time Dijkstra at sizes n = 50, 100, 200, 400 on dense random graphs. Compute the doubling ratio. Does it match `O(n²)`?

(See: `exercises/ch37/ex_37_4_dijkstra_scale.ax`)

Exercise 37.5 — Negative weights break Dijkstra

Build a small graph with one negative edge. Run Dijkstra on it. Compare the result to the actual shortest path. Where did Dijkstra go wrong?

(See: `exercises/ch37/ex_37_5_neg_weights.ax`)

Exercise 37.6 — Prim vs Kruskal

Read about Kruskal's algorithm. On the sample graph from §37.5, trace by hand what Kruskal would do (sort edges, add cheapest first). Compare the resulting MST to Prim's.

(a) Do they produce the same tree?
(b) If not, why?
(c) Which is simpler to implement?

(See: `exercises/ch37/ex_37_6_prim_kruskal.ax`)

Solutions to selected exercises: Chapter 37 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch37_solutions.md.)*

Chapter 38 · State Machines Revisited

*Part IX, Chapter 1. Opens the **Stateful Systems** arc. Volume I touched state machines briefly in Ch.16 (Mutation). This chapter promotes them to a first-class design tool — the right abstraction for protocols, parsers, UI state, and any system whose behavior depends on history.*

38.1 The shape of a state machine

A state machine has four parts:

1. **A finite set of states.** "Idle", "Running", "Paused", "Error". The state names are usually discrete labels, not numbers.
2. **A starting state.** Where the machine begins.
3. **A set of transitions.** For each (current_state, input) pair, what's the next_state.
4. **Optionally: accept states or output actions.** A parser's accept state is "saw valid input"; a UI state machine might log or display on each transition.

The classic example: a turnstile.

```
States: locked, unlocked
Transitions:
  (locked, coin)  → unlocked
  (locked, push)  → locked    (refused)
  (unlocked, coin)→ unlocked  (no-op)
  (unlocked, push)→ locked
```

Real-world state machines:

- Network protocols: TCP has 11 states (LISTEN, ESTABLISHED, FIN_WAIT_1, etc.).
- Game characters: idle, running, jumping, attacking.
- Parsers: each context (inside-string, inside-comment, expecting-operand) is a state.
- The Cascade Reactor (Ch.51): processing, debouncing, idle — a state machine over event queue depth.

State machines are *the* tool when "what happens next depends on what already happened."

38.2 Implementation 1 — Transition table as a dict

The most direct encoding: a dictionary keyed by (current_state, input) pairs, valued by next_state.

A dictionary literal makes the whole transition table visible in one place. Indexed assignment could update it later, but this example keeps the machine's rules fixed while the current state changes.

```
turnstile: {
  "locked.coin": "unlocked",
  "locked.push": "locked",
  "unlocked.coin": "unlocked",
  "unlocked.push": "locked"
}

do_step: func(machine, state, input) [
  key: state + "." + input
  machine[key]
]

run_machine: func(machine, start, inputs) [
  states: [start,]  # one-element Array containing the initial state
  cur: start
  i: 1
  while i <= len(inputs) [
    cur = do_step(machine, cur, inputs[i])
    states = states + [cur]
    i = i + 1
  ]
  states
]
```

Run it:

```
result: run_machine(turnstile, "locked",
                    ["coin", "push", "push", "coin", "push"])
println(result)
# → ["locked", "unlocked", "locked", "locked",
#    "unlocked", "locked"]
```

Walkthrough:

- Start: locked.
- coin → unlocked.
- push → locked.
- push → locked (already locked; push refused).
- coin → unlocked.
- push → locked.

The trace shows the full history — useful for testing and debugging.

Strengths of dict-table representation:

- Easy to read: every transition spelled out.
- Easy to enumerate: `machine`'s keys are exactly the legal (state, input) pairs.
- Easy to verify: a static check can ensure every (state, input) combination has a transition (no surprises at runtime).

Weaknesses:

- Cluttered for large state spaces (n states × m inputs = n·m table entries).
- Doesn't handle *guarded* transitions ("transition if some predicate holds") — those need conditionals at each entry.

38.3 Implementation 2 — Match on (state, input)

For state machines where transition logic is more complex, use a match expression:

```
do_step2: func(state, input) [
  if state == "locked" then [
    if input == "coin" then "unlocked"
    else "locked"
  ] else [
    # unlocked
    if input == "push" then "locked"
    else "unlocked"
  ]
]
```

Reads top-to-bottom: "if locked, then coin unlocks else stay locked; if unlocked, then push locks else stay unlocked."

When this style wins:

- Each state has a few inputs to consider, but with complex computation per case.
- States and inputs come from open types (not closed enumeration).
- Code review prefers seeing the logic in one place rather than scattered across table entries.

38.4 Implementation 3 — Functions per state

The third encoding: one function per state. Each function returns the next-state function (a **continuation-passing** style).

```
locked_state: func(input) [
  if input == "coin" then ["unlocked", "unlocked_state"]
  else ["locked", "locked_state"]
]

unlocked_state: func(input) [
  if input == "push" then ["locked", "locked_state"]
  else ["unlocked", "unlocked_state"]
]

run_machine3: func(start_name, start_fn, inputs) [
```

```
  states: [start_name]
  handler: start_fn
  i: 1
  while i <= len(inputs) [
    result: handler(inputs[i])
    states = states + [result[1]]
    next_name: result[2]
    if next_name == "locked_state" then [handler = locked_state]
    if next_name == "unlocked_state" then [handler = unlocked_state]
    i = i + 1
  ]
  states
]
```

This style is verbose but **scales**. Each state's function can:

- Carry arbitrary local logic.
- Hold per-state private state (via closures).
- Dispatch to other states without a central table.

It's how **actor systems** and **OOP state pattern** encode state machines. Erlang's `gen_statem` and Akka's FSM are this pattern in production.

When this style wins:

- States have complex per-state behavior.
- Different states need different "data shape" (locked needs a counter; unlocked needs a timeout).
- The state machine evolves over time; you want to add a state without rewriting a central table.

38.5 Three implementations, one machine — which wins?

Let's tabulate trade-offs:

Style	Best for	Drawbacks
Transition table (dict)	Small, fully-enumerated machines	Sparse for large state spaces
Match on (state, input)	Conditional transitions; small machines	Hard to enumerate complete coverage
Functions per state	Large state spaces; per-state private state	Verbose; harder to visualize

Rule of thumb:

- Under 10 states + simple transitions → dict table.
- 10-50 states + some logic → match expression.
- 50+ states or per-state internal state → function per state.

Real production code often mixes them: a top-level dispatch table that calls into a per-state handler function. The dispatch table gives you the high-level "map of the machine"; the handlers give you the per-state complexity.

38.6 A worked example: a quoted-string scanner

Extracting quoted strings illustrates a small state machine. This teaching scanner accepts ordinary characters and escaped quotes, slashes, and backslashes. It is not a complete JSON parser: JSON also defines control-character escapes, Unicode escapes, and validation of the surrounding document. Unsupported escapes are rejected explicitly:

```
States: outside, inside, escape
Transitions:
  outside, "      → inside
  inside, "       → outside  (string closes)
  inside, \       → escape
  inside, *       → inside   (consume char)
  escape, *       → inside   (consumed the escape char)
```

```
parse_json_strings: func(text) [
  state: "outside"
  strings: []
  current: ""
  for c in text [
    if state == "outside" then [
      if c == "\"" then [state = "inside"]
    ] else if state == "inside" then [
      if c == "\"" then [
        strings = push(strings, current)
        current = ""
        state = "outside"
      ] else if c == "\\" then [state = "escape"]
      else [current = current + c]
    ] else [
      if c == "\"" or c == "\\" or c == "/" then [
        current = current + c
      ] else [error("This teaching scanner only handles quote, slash, and backslash
↪  escapes")]
      state = "inside"
    ]
  ]
  if state != "outside" then error("Unterminated string")
  strings
]

println(parse_json_strings(r'{"name": "Alice", "age": 30}'))
# → ["name", "Alice", "age"]
```

```
println(parse_json_strings(r'{"text": "say \"hi\""}'))
# → ["text", "say \"hi\""]
```

(Note: the integer `30` value isn't a quoted string, so it doesn't get captured.)

The parser shows the *workhorse* role of state machines: every parser, every lexer, every protocol implementation has a state machine at its core. Once you see one, you see them everywhere.

38.7 State machine design recipe

The Ch.4 design recipe extended for state machines:

1. **Enumerate the states.** Give each a name.
2. **Enumerate the inputs / events.** The "alphabet" of what the machine reacts to.
3. **Identify the initial state.** Where the machine starts.
4. **Identify any final/accept states** (for parsers and recognizers).
5. **Draw the diagram.** States as circles, inputs as labeled arrows. Don't skip this — the diagram catches missing transitions faster than code review.
6. **Write the transition table.** From the diagram, one row per (state, input) pair.
7. **Code it** using one of §38.5's three styles based on size.
8. **Test exhaustively.** For each state, every input — does it go to the right place?

Step 5 is the one most engineers skip. A whiteboard diagram catches "what happens if X arrives in state Y?" omissions that a code review would miss. *Draw it first.*

38.8 When *not* to use a state machine

Not every "system with stateful behavior" is best expressed as a finite state machine. Two anti-patterns:

Anti-pattern 1: Too many states. If your "machine" has 50,000 states (e.g., one per row in a database), it's not a state machine — it's data. Use a database table.

Anti-pattern 2: Concurrent overlapping states. If multiple states can be active simultaneously ("Connected" AND "Authenticated"), it's not a single state machine — it's *several* machines running in parallel. Use composition: one machine per orthogonal concern.

The rule of thumb: a state machine works when **state × input → next state** is a *function*. If the "function" needs to query a database, talk to a network, or check the time — it's no longer a finite state machine; it's a control system.

38.9 The bridge to event loops

A state machine processes one input at a time: `step(state, input) → new_state`. An *event loop* runs that step over and over, fed by an external queue:

```
forever:
  input = wait_for_event()
  state = step(state, input)
  if state is final: exit
```

The state machine *says* what to do per input; the event loop *delivers* the inputs. Together they form the reactive-systems pattern.

The next chapter (Ch.39 Event Loops & Reactors) introduces the event-loop side. The chapter after that (Ch.40 The Cascade Reactor) shows a production reactor that uses both ideas at scale.

38.10 Honest limits

- **Hierarchical state machines** ("Harel statecharts") let states nest — e.g., the "Connected" state has sub-states "Idle" and "Sending". Beyond scope.
- **Non-deterministic state machines** allow multiple next-states for the same (state, input). They model formal language recognition (regex). The "subset construction" converts them to deterministic ones — also beyond scope.
- **State machines with timers** (e.g., "after 5s in state X, transition to Y") need an event-loop with scheduling support. The next chapter handles the scheduling primitives.
- **Probabilistic state machines** (Markov chains) replace transitions with transition *probabilities*. A whole subfield (HMMs, MDPs) builds on this.

The plain deterministic finite state machine covered here is the *foundation*. The variants above all build on this same skeleton — states, transitions, current state — and add machinery on top.

Exercises

Exercise 38.1 — Door lock state machine

Implement a state machine for a four-digit combination lock. States: `idle`, `entering_1`, `entering_2`, `entering_3`, `unlocked`. The combination is "1234". After 3 wrong attempts, the lock goes into a `locked_out` state.

(See: `exercises/ch38/ex_38_1_combolock.ax`)

Exercise 38.2 — Coverage check

Given a state machine table, write `check_complete(machine, states, inputs)` that returns `true` if every (state, input) pair has an entry in the machine.

(See: `exercises/ch38/ex_38_2_coverage.ax`)

Exercise 38.3 — Vending machine

Implement a vending machine with states: `idle`, `selecting`, `paying`, `dispensing`. Inputs: `insert_coin`, `select_product`, `cancel`, `complete`. Use the per-state function style from §38.4.

(See: `exercises/ch38/ex_38_3_vending.ax`)

Exercise 38.4 — Traffic light with timer

Implement a traffic light: green → yellow → red → green. Inputs are `tick` events. Track how many ticks have happened in each color (green:30, yellow:5, red:25 seconds).

(See: `exercises/ch38/ex_38_4_traffic.ax`)

Exercise 38.5 — Two machines, composed

Implement two state machines that share input but maintain independent state:

- Machine A: counts "press" inputs modulo 3.
- Machine B: counts "press" inputs modulo 5.

After 15 presses, both reach state 0 simultaneously.

(See: `exercises/ch38/ex_38_5_compose.ax`)

Exercise 38.6 — Critique a real state machine

Read a real-world state machine documented online (TCP, HTTP/2, the W3C DOM, etc.) and answer:

(a) How many states are in the machine?
(b) Which transitions are *time-triggered* vs *input-triggered*?
(c) What happens if an "impossible" input arrives in a given state? (Spoiler: production systems usually have a "drop on the floor" or "log and continue" rule.)

(See: `exercises/ch38/ex_38_6_critique.ax` — open-ended)

Solutions to selected exercises: Chapter 38 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch38_solutions.md.)*

Chapter 39 · Event Loops & Reactors

*Part IX, Chapter 2. Picks up directly from Ch.38. A state machine describes **what** the system does in response to an input. An event loop is the runtime that **delivers** the inputs. Together they form the backbone of every long-running interactive system you will ever write — GUIs, servers, game engines, simulators, and (in Ch.40) the Cascade reactor.*

39.1 From state machines to event loops

The state machines of Ch.38 had a hidden assumption: *somebody is feeding them inputs in order*. We wrote:

```
result: run_machine(turnstile, "locked",
                    ["coin", "push", "push"])
```

That `["coin", "push", "push"]` array is a tidy fiction. In the real world, inputs don't arrive in a pre-baked array. They arrive *over time*, from *multiple sources*, and the program has to *wait* for the next one. A bank ATM doesn't get a script of keypresses — it sits idle, then a customer touches the screen, then it processes, then it sits idle again.

This pattern — "sit idle, wait for input, process, return to idle" — is the **event loop**. The state machine is the *transition logic*; the event loop is the *delivery mechanism*. Every interactive system has both, even if you've never named them.

In this chapter we'll build the event loop from scratch — first a 12-line minimum viable version, then adding dispatch tables, re-entrancy, scheduling, and stop conditions. By §39.7 you'll have a working **reactor**: an event loop with structured dispatch and a quiescence condition. Ch.40 then turns it into a durable, recoverable reactor — the Cascade pattern.

Where event loops live

The event loop is the dominant control-flow shape in:

- **GUI applications**: the main thread sits in an event loop dispatching mouse/keyboard/timer events to widget handlers. Cocoa's `NSRunLoop`, Win32's `GetMessage`/`DispatchMessage`, GTK's `g_main_loop_run` — all the same pattern, different names.
- **Network servers**: `epoll`/`kqueue`/`io_uring` on Linux/BSD/recent Linux. The loop blocks on the kernel syscall, wakes when a socket is readable, dispatches.
- **Game engines**: 60 Hz frame loop is an event loop with one timer event per frame, plus input events interleaved.

- **JavaScript runtimes**: Node.js, browsers. The famous "event loop" of the JS world is exactly this pattern with microtask/macrotask priority levels.
- **Reactor frameworks**: Erlang's OTP, Akka, Reactor Netty, Cascade — frameworks that elevate the event loop into a first-class abstraction.

If you've used any interactive system, you've used an event loop. This chapter is about making it explicit.

39.2 The bare-minimum event loop

The simplest event loop has four ingredients:

1. **A queue of events** (FIFO).
2. **A dispatch step**: pull the head event, decide what to do with it.
3. **A loop**: keep dispatching until the queue is empty (or a stop condition triggers).
4. **A starting set of events** to seed the queue.

In Axioma:

```
queue: []
queue = queue + ["greet"]
queue = queue + ["count"]
queue = queue + ["bye"]

head: 1
while head <= len(queue) [
  ev: queue[head]
  head = head + 1
  if ev == "greet" then [println("hello!")]
  if ev == "count" then [println("1, 2, 3")]
  if ev == "bye"   then [println("goodbye!")]
]
```

Output:

```
hello!
1, 2, 3
goodbye!
```

That's a real event loop. Twelve lines. Notice three design choices already:

(a) Head pointer, not array-shift. We don't pop from the front by rebuilding the array (which would be O(n) per pop — accidental quadratic, see Ch.35). We advance an integer `head` index. The queue grows by append, the head advances by increment. Both O(1). This is the standard trick for high-throughput event loops.

(b) The loop runs until queue is drained. With no re-entry, the loop terminates when `head > len(queue)`. Add re-entry (§39.4) and termination depends on whether new events outpace consumption.

(c) The dispatch is an `if`-chain. Fine for two or three events. By the time you have ten, it's maintenance pain. §39.3 promotes the chain to a table.

The trace as a stream

If you squint, this loop is a streaming computation: events flow in, side-effects flow out. The handler chain is the *processor*. The queue is the *backlog*. A long-lived loop is one whose queue never empties for long. (Compare to Ch.17 streams: same shape, different notation.)

39.3 Dispatch tables — event type → handler

The `if`-chain doesn't scale. The fix is to make dispatch *data-driven*: store handlers in a dict keyed by event type, look up at runtime.

```
on_greet: func(payload) [
  println("hello, ", payload, "!")
]
on_count: func(payload) [
  i: 1
  while i <= payload [
    println(i)
    i = i + 1
  ]
]
on_bye: func(payload) [
  println("goodbye, ", payload, "!")
]

handlers: {
  "greet": on_greet,
  "count": on_count,
  "bye":   on_bye
}
```

Now each event carries a *payload* too — the data the handler needs. Events become **(type, payload)** pairs:

```
queue: []
queue = queue + [("greet", "world")]
queue = queue + [("count", 3)]
queue = queue + [("bye",   "world")]

head: 1
while head <= len(queue) [
  ev: queue[head]
  head = head + 1
  h: handlers[ev[1]]
  if h == none then [
    println("unknown event:", ev[1])
```

```
  ] else [
    h(ev[2])
  ]
]
```

Output:

```
hello, world!
1
2
3
goodbye, world!
```

What we gained. The loop body is now event-type agnostic. Adding a new event means writing a handler function and adding one dict entry. The loop itself never changes. Compare to the `if`-chain version: every new event meant editing the loop. *That* is the practical difference between code and data.

A new event type "scroll"? One change:

```
on_scroll: func(payload) [println("scroll:", payload)]
handlers["scroll"] = on_scroll
```

The loop runs unchanged: the new dictionary entry participates in the same dispatch lookup. Indexed mutation changes the dictionary itself; replacing a captured binding instead requires `rebind` (Chapter 16).

Aside: this is the Visitor pattern

If you've read object-oriented design patterns, the dispatch table is the Visitor pattern stripped of its classes. In OO languages each event class has a `visit()` method; in functional/dict-based languages the dispatch *is* the dict lookup. Same idea, less ceremony.

39.4 Re-entrancy — handlers that post new events

Here is where the event loop gets interesting. In real systems, handlers often **post new events**. A "socket-readable" handler reads the bytes, parses a message, and posts a "message-received" event for the application layer. A "click" handler validates input and posts a "form-submit" event.

The pattern is straightforward — but only because we chose the head-pointer design in §39.2.

```
queue: []
queue = queue + [("tick", 0)]

on_tick: func(payload) [
  println("tick:", payload)
  # Schedule the next tick, up to 3
```

```
  small: payload < 3
  if small then [
    rebind queue = queue + [("tick", payload + 1)]
  ]
]

handlers: { "tick": on_tick }

head: 1
while head <= len(queue) [
  ev: queue[head]
  head = head + 1
  h: handlers[ev[1]]
  if h != none then [h(ev[2])]
]
println("processed", head - 1, " events")
```

Output:

```
tick: 0
tick: 1
tick: 2
tick: 3
processed 4  events
```

The handler reaches *up* into its enclosing scope to extend `queue`. Because the loop reads `len(queue)` afresh each iteration, the new events get picked up. This is the **re-entrant posting** pattern — handlers schedule future work without knowing anything about the loop itself.

Why head-pointer matters here

If we'd implemented "pop from front" as `queue = queue[2:]` (Python-list style), each new event posted mid-loop would shift the indices of *already-pending* events. The bookkeeping gets ugly. With the head-pointer style:

- `queue[head]` is the next event to process
- Appended events go to the end, at index `len(queue) + 1`
- `head` advances monotonically; nothing else moves

This is exactly how `kqueue`/`epoll` work under the hood: a userspace ring buffer with a producer index (kernel writes) and a consumer index (program reads), both advancing forever.

The "infinite tick" trap

Re-entrant posting is powerful — and dangerous. Remove the `small` guard above:

```
on_tick: func(payload) [
  println("tick:", payload)
  rebind queue = queue + [("tick", payload + 1)]  # always
```

```
]
```

…and the loop runs forever. Each handler invocation adds one event; `len(queue)` grows as fast as `head` advances; the loop never terminates. In production, this is one of the top three event-loop bugs (along with starvation §39.9 and re-entrant deadlocks §39.9). Always have a stop condition.

39.5 Stop signals and exit conditions

Three ways to stop an event loop:

(a) Natural drain. Process all events, queue empties, loop exits. Works for one-shot batch jobs (processing a saved log, replaying a transcript).

(b) Stop flag. A handler sets a boolean; the loop checks it each iteration. Works for "shutdown on specific event" scenarios.

```
stop: false
n_ticks: 0

on_tick: func(payload) [
  rebind n_ticks = n_ticks + 1
  println("tick #", n_ticks, " payload=", payload)
  if n_ticks >= 3 then [rebind stop = true]
]

queue: []
i: 1
while i <= 10 [
  queue = queue + [("tick", i)]
  i = i + 1
]

head: 1
while head <= len(queue) [
  if stop then [head = len(queue) + 1] else [
    ev: queue[head]
    head = head + 1
    if ev[1] == "tick" then [on_tick(ev[2])]
  ]
]
println("stopped after ", n_ticks, " ticks")
```

Output:

```
tick # 1  payload= 1
tick # 2  payload= 2
tick # 3  payload= 3
stopped after  3  ticks
```

The `if stop then [head = len(queue) + 1]` trick forces the `while` to exit on the next test. We can't `break` out of an Axioma `while` directly, so we make the loop condition false by other means. (See Ch.16 for more on this Axioma idiom.)

(c) Timeout / wall-clock limit. "Run for at most 30 seconds." We'll see this in §39.6 with virtual time. In production with real wall-clock time, you'd check `system_time() - start >= limit` each iteration.

A **production-grade event loop usually has all three**: drains naturally on normal completion, has a shutdown flag for graceful exit (SIGTERM handler sets it), and a hard timeout for catastrophic recovery.

39.6 Scheduled events — virtual time + priority queue

So far events have been *immediate*: process in order of arrival. Many systems need *scheduled* events: "fire this alarm at t=30 seconds", "retry this HTTP request after 2 seconds". The natural data structure is a **priority queue** ordered by scheduled time.

Discrete-event simulation

Let's build a tiny discrete-event simulator. Events have (time, type, payload). The loop always processes the *earliest* event next, advancing virtual time as it goes.

```
queue: []
queue = queue + [(0, "start", 0)]
now: 0
max_t: 30

# Bubble-sort by time field [1]; small queue, O(n^2) acceptable
sort_queue: func() [
  n: len(queue)
  i: 1
  while i <= n [
    j: i + 1
    while j <= n [
      if queue[i][1] > queue[j][1] then [
        tmp: queue[i]
        queue[i] = queue[j]
        queue[j] = tmp
      ]
      j = j + 1
    ]
    i = i + 1
  ]
]

schedule: func(delay, typ, payload) [
  rebind queue = queue + [(now + delay, typ, payload)]
  sort_queue()
```

```
]

on_start: func(p) [
  println("[t=", now, "] start; scheduling pings")
  schedule(5, "ping", "a")
  schedule(2, "ping", "b")
  schedule(8, "ping", "c")
]

on_ping: func(p) [
  println("[t=", now, "] ping ", p)
]

while len(queue) >= 1 [
  # Pop head
  ev: queue[1]
  new_q: []
  i: 2
  while i <= len(queue) [
    new_q = new_q + [queue[i]]
    i = i + 1
  ]
  queue = new_q
  now = ev[1]
  typ: ev[2]
  payload: ev[3]
  halt: now > max_t
  if halt then [queue = []] else [
    if typ == "start" then [on_start(payload)]
    if typ == "ping"  then [on_ping(payload)]
  ]
]
println("done at t=", now)
```

Output:

```
[t= 0 ] start; scheduling pings
[t= 2 ] ping  b
[t= 5 ] ping  a
[t= 8 ] ping  c
done at t= 8
```

Note the events came out in **time order, not posting order**. `on_start` scheduled three pings at delays 5, 2, 8 — they fired at virtual times 2, 5, 8.

Virtual vs wall-clock time

In a simulator (the canonical example: queueing-theory ED simulations, network packet-arrival models, traffic flow), virtual time advances *as fast as the queue can be drained*. Five virtual seconds might take 0.5 real microseconds.

In a real-time event loop (a game, a UI), virtual time *equals* wall-clock time — the loop sleeps until the next event is due, then fires. This is the `select`/`epoll`/`kqueue` "timeout" parameter at the OS level.

Both cases use the same data structure (priority queue by time) and the same dispatch logic. The difference is the bridge between virtual time and the outside world.

Why bubble-sort?

We used bubble-sort because the queue is small (a few dozen events at most). Real event loops use a **heap** (priority queue with O(log n) insert and O(log n) pop-min) — see Ch.36 §36.7 for the algorithm and Ch.51 for how Cascade implements one over SQLite.

39.7 The reactor pattern

Putting it all together: **dispatch table + re-entrant queue + stop condition + scheduler = reactor**. The name comes from Erlang/OTP's `gen_server` and Reactor Netty; it captures the idea that the loop *reacts* to events rather than *driving* the program.

A reactor has a well-defined external interface:

```
post(event)            # add to queue
schedule(delay, event) # add to scheduled queue
stop()                 # set the shutdown flag
run()                  # the loop itself
```

Internally it has:

```
queue          # FIFO of immediate events
scheduled      # priority queue of (time, event)
handlers       # dispatch table: type → function
state          # arbitrary application state
```

Let's package it as a coherent unit. We'll use a **dict-of-closures** style that simulates a small object. Handlers are passed to `make_reactor` as a dictionary at construction time, making the initial dispatch table explicit. Updating an entry later is possible; this example keeps that configuration fixed.

```
make_reactor: func(handlers) [
  queue: []
  scheduled: []
  stop_flag: false
  now: 0

  sort_scheduled: func() [
    n: len(scheduled)
    i: 1
    while i <= n [
      j: i + 1
      while j <= n [
        if scheduled[i][1] > scheduled[j][1] then [
```

```
        tmp: scheduled[i]
        scheduled[i] = scheduled[j]
        scheduled[j] = tmp
      ]
      j = j + 1
    ]
    i = i + 1
  ]
]

# Public interface
post: func(typ, payload) [
  rebind queue = queue + [(typ, payload)]
]
sched: func(delay, typ, payload) [
  rebind scheduled = scheduled + [(now + delay, typ, payload)]
  sort_scheduled()
]
halt: func() [rebind stop_flag = true]

run: func() [
  head: 1
  keep: true
  while keep [
    have_imm: head <= len(queue)
    have_sch: len(scheduled) >= 1
    halted: stop_flag
    if halted then [keep = false]
    else if have_imm then [
      ev: queue[head]
      head = head + 1
      h: handlers[ev[1]]
      if h != none then [h(ev[2])]
    ]
    else if have_sch then [
      ev: scheduled[1]
      # pop head
      new_s: []
      k: 2
      while k <= len(scheduled) [
        new_s = new_s + [scheduled[k]]
        k = k + 1
      ]
      rebind scheduled = new_s
      rebind now = ev[1]
      h: handlers[ev[2]]
      if h != none then [h(ev[3])]
    ]
    else [keep = false]
  ]
```

```
  ]

  # Return the public interface as a dict
  {
    "post":  post,
    "sched": sched,
    "halt":  halt,
    "run":   run
  }
]
```

Use it:

```
r: make_reactor({
  "greet": func(p) [println("hello ", p)],
  "count": func(p) [
    i: 1
    while i <= p [
      println(i)
      i = i + 1
    ]
  ]
})
r["post"]("greet", "world")
r["post"]("count", 3)
r["sched"](5, "greet", "delayed")
r["run"]()
```

Output:

```
hello  world
1
2
3
hello  delayed
```

The reactor processes immediates first (the two greets/counts), then drains the scheduled queue. The priority order is **immediates ≫ scheduled** — a common choice; some reactors interleave the two. Each strategy has trade-offs (§39.9).

What we've built

That's a real reactor — minus persistence (Ch.40) and multi-threading (which Axioma doesn't need; we'll see in Ch.51 how Cascade handles concurrency on top of a single-threaded reactor). The pattern is the *same one* used by Node.js, Reactor Netty, Twisted, asyncio internals, and most game engines.

39.8 Real-world reactors at a glance

Node.js. A single-threaded reactor over `libuv`. Immediates = "microtasks" (promises). Scheduled = "macrotasks" (setTimeout). I/O completion events come from the kernel via `epoll`/`kqueue`/IOCP. The famous "Node.js is single-threaded but fast" story is exactly the reactor pattern with kernel-level I/O multiplexing.

Browser event loop. Almost identical to Node.js, minus filesystem; plus DOM events (click, scroll, mousemove) coming from the rendering thread. The priorities are nuanced: animation frames have a specific position in the cycle (`requestAnimationFrame`), microtasks drain between every task, etc. But the core is the reactor.

Erlang/OTP `gen_server`. Each process has a mailbox (the queue), a behaviour module with `handle_call`/`handle_cast`/`handle_info` (the dispatch table), and `init`/`terminate` (lifecycle). The Erlang VM runs millions of these reactors concurrently with cooperative scheduling. Cascade's agent swarm (Vol III Ch.50) borrows directly from this model.

Game engines. A frame loop is a reactor with a 60 Hz timer tick driving everything. Input events (keyboard, mouse, controller) post to the queue between frames. Each frame: drain the queue, update state, render. Unity's `MonoBehaviour.Update`, Unreal's `Tick` — same pattern, different namespace.

The Cascade Reactor (Ch.40 + Vol III Ch.51). A durable reactor: events persisted to SQLite WAL before processing. On crash, the loop replays from WAL. Adds an `await` mechanism for coordinating between agents. The reactor pattern, made bulletproof for a 24/7 production system.

The point: **once you can write the loop in §39.7, you can read the internals of any of these systems**. They're all variations on the same theme.

39.9 Common pitfalls

Event loops are deceptively simple. The pitfalls show up in production, usually under load. Here are the top five.

(a) Blocking handlers stall the loop

If a handler runs for 5 seconds, the *whole loop* stalls for 5 seconds. Every other event waits. Click events queue up; the UI freezes; users rage-click and generate more events.

Fix: handlers must be *short-running*. Long work goes on a worker thread (or another reactor in a different OS process); the worker posts back a "work-done" event when finished. This is Node.js's `worker_threads`, Erlang's spawn, Cascade's agent swarm pattern.

(b) Starvation

Suppose every "tick" handler posts two new "tick" events. The queue grows faster than it drains; other event types never get a turn. Common in naive producer-consumer designs.

Fix: rate-limit the producer, or interleave event types (e.g., dispatch at most N events of one type before yielding to others — Linux's "completely fair scheduler" applied to event types).

(c) Re-entrancy bugs

A handler triggers, somewhere deep down, *the same handler*. State that was mid-update gets seen by the re-entry. Result: corrupted internal state, "this should never happen" assertions fire.

Fix: handlers must be *re-entrancy-safe* (read all needed state into locals up front, mutate at the end) — or the dispatcher must *defer* re-entrant posts (Node.js's `process.nextTick` queue, Erlang's selective receive).

(d) Unbounded queue growth

A burst of events arrives. The queue grows to a million entries. Memory exhausted; OS kills the process; you have no idea why.

Fix: bound the queue. New events go to a **rejection** path past the cap (drop, log, return an error to the producer). Backpressure is the generalization (see Ch.17 streams).

(e) Lost events on crash

The loop is mid-processing event #500 when the process crashes. Event #500 is *gone*; event #501..N are *still in the queue*; on restart, the queue is *empty* (in-memory). The work is lost.

Fix: persistent event log. Every posted event written to disk before processing. On restart, replay from the log. This is Cascade's WAL pattern, Kafka's log, every "exactly-once" message-queue system. Ch.40 walks through the implementation.

(f) Bonus pitfall: priority inversion

You schedule a "shutdown" event with high priority. A handler in front of it is *waiting* (futures, locks, external I/O) on a low-priority event that's *behind* it in the queue. The shutdown can't run because its prerequisite is stuck behind it.

Fix: explicit priority levels; or kill the waiter (timeout). This is one of the hardest event- loop bugs to diagnose; the fix usually involves restructuring the dependencies.

39.10 Looking ahead

You've now seen the event-loop / reactor pattern at two scales: a 12-line bare-minimum loop in §39.2, and a fully featured reactor with scheduling in §39.7. The exercises below will push you to add quality-of-life features: bounded queues, event priorities, introspection.

Ch.40 — The Cascade Reactor takes the next step: *persistence*. What if the program crashes mid-event? Cascade writes events to a SQLite WAL before processing, so on restart the loop replays from the log. The reactor becomes *durable*. We'll walk through the design, then build a working miniature.

Vol III Ch.51 — Reactor Internals is the production-system view: how Cascade's reactor integrates with the SQLite knowledge base, the agent swarm, and the truth-lifecycle machinery.

If you grok §39.7 and §39.9, you have the conceptual backbone for both.

Exercises

39.1. *Empty-queue start.* Modify the §39.2 loop to handle the case where the initial queue is empty. Should the loop run zero iterations (current behavior), wait for an external event source (block), or report an error? Pick one and justify in 2-3 sentences.

39.2. *Bounded queue.* Add a queue cap to the reactor in §39.7. If `post` is called when `len(queue) >= cap`, reject the event (return false; print a warning). Verify by posting 100 events with cap=10: only the first 10 should be processed.

39.3. *Event introspection.* Add a builtin `queue_depth()` to the reactor (returns `len(queue) + len(scheduled)`). Add a `tick` handler that prints depth every second of virtual time. Demonstrate the depth rising and falling during a burst.

39.4. *Priority levels.* Modify the reactor to have two FIFO queues: `high` and `normal`. `post` takes an optional priority. The loop drains `high` before touching `normal`. Use this to make a "shutdown" event always preempt pending work.

39.5. *Idle handler.* Add an `on_idle` slot to the reactor: a handler invoked when both queues are empty but `stop_flag` is false. Demonstrate by writing a reactor that prints "idle" every second until external code calls `halt()`.

39.6. *Find a real event loop.* Pick a system you use (your editor, terminal, web browser) and look up its event loop documentation. Answer: (a) Is the queue FIFO or priority-ordered? (b) What's the shutdown mechanism? (c) What happens on a slow handler — UI freeze, dropped events, or thread spawn?

Vocabulary

- **Event loop**: a program shape that waits for events on a queue, dispatches them to handlers, and repeats.
- **Dispatch table**: a dict (or vtable) mapping event types to handler functions.
- **Re-entrant posting**: handlers add new events to the queue while the loop is running.
- **Reactor**: an event loop with structured dispatch, scheduled events, and lifecycle hooks (post, sched, on, run, halt).
- **Head pointer**: the trick of advancing an integer index instead of shifting the queue array — O(1) pop vs O(n).
- **Quiescence**: the state where the queue is empty and no work is in flight; the natural stop condition for a one-shot reactor.
- **Backpressure**: the producer slows down when the consumer can't keep up. The general fix for unbounded queue growth.

- **WAL** (Write-Ahead Log): the persistence technique Ch.40 uses to make the reactor durable.
- **Priority inversion**: when a high-priority event is blocked waiting on a low-priority event.

Notes for the curious

Why "reactor"?

The term traces to Erlang OTP (1990s) and the "Reactor" design pattern from Schmidt et al.'s *Pattern-Oriented Software Architecture* (1996). The intuition: a *proactor* drives the program forward (imperative top-down control); a *reactor* responds to external stimuli (event-driven). Most interactive systems are reactors at their core, even when wrapped in proactor-shaped APIs.

The "single-threaded reactor" myth

People sometimes claim that "single-threaded reactors don't use multiple cores." Half-true. A single reactor *thread* does its dispatch on one core; but handlers can hand work off to *worker threads* on other cores, and the reactor coordinates the results. This is exactly Node.js's model: one event-loop thread, a thread pool for blocking I/O, and `worker_threads` for CPU-bound work.

The cost of dispatch

Every event in the §39.7 reactor incurs:

- a `len(queue)` call (O(1))
- a dict lookup for the handler (O(1) amortized)
- the handler invocation itself

The dispatch overhead is in the microseconds. For high-throughput systems (millions of events/sec), the overhead matters; specialized event loops (Disruptor, LMAX) replace the dict with branch- prediction-friendly switches and ring-buffer queues. For most systems, the dict is fine.

Connection to streams (Ch.17)

A stream is a *pull*-based event loop: the consumer asks for the next value, the producer computes it. A reactor is *push*-based: the producer (kernel, user, timer) deposits events, the consumer (handlers) drains them. The same problem solved from opposite ends. Many systems mix both: streams compose handler output into pipelines; reactors deliver events from the outside world.

Solutions to selected exercises: Chapter 39 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch39_solutions.md.)*

Chapter 40 · The Cascade Reactor

> **Integration draft.** This chapter connects the language to a separately maintained application or external services. Its integration claims have not been revalidated in this revision. Use the core-language chapters for verified standalone examples; product verification is deferred.

*Part IX, Chapter 3. Ch.39's in-memory reactor is fast and elegant — and **fragile**. The moment the process dies mid-event, you lose all pending work and all events in flight. This chapter shows the standard fix: a **Write-Ahead Log** (WAL). Once you've added WAL, your reactor is **durable** — it survives crashes without losing events. This is the architectural step that turns toy event loops into the production systems behind databases, message queues, and the Cascade reactor (Vol III Ch.51).*

40.1 The lost-event problem

Re-read §39.9 (e) carefully. The setup:

1. Reactor running, queue has events #500 through #1000.
2. Processing event #500 — handler is mid-execution.
3. Process crashes (OOM, hardware fault, kernel panic, `kill -9`, power loss).
4. Restart.
5. **In-memory queue is empty.** Events #500 through #1000 are gone. The work they represented — user submissions, sensor readings, payment intents — *vanishes*.

For batch processing of throwaway data, this is fine. For *anything* with a customer at the other end (a financial transaction, a medical record update, a flight booking, a Cascade truth-update), this is unacceptable. Once you've **accepted** an event, you have an obligation to **process** it — even if the process crashes between acceptance and processing.

The standard fix has been known since the 1970s (database recovery research). It's called the **Write-Ahead Log** or **WAL**.

What WAL guarantees

Two properties:

1. **Durability**: once `post(event)` returns "accepted", the event is on stable storage. Even a full hardware crash won't lose it.
2. **Recoverability**: on restart, the reactor can replay the log and re-process exactly the events that hadn't completed pre-crash.

Together: **no event is lost; no event is processed zero times**. (Events might be processed *more than once* — see §40.5 on idempotency.)

40.2 The Write-Ahead Log idea

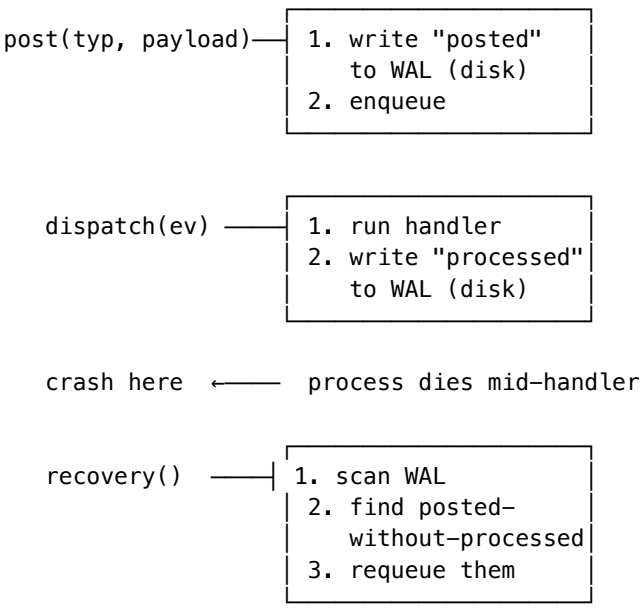
The protocol is one line:

> Before dispatching an event, write a "posted" record to the log. After the handler returns, write a "processed" record.

That's it. Recovery is:

> Replay the log. Any event with a "posted" record but no matching "processed" record is **pending** — push it back into the in-memory queue. Resume normal operation.

Diagrammatically:



That's the entire WAL pattern. Every durable event system you've heard of — Kafka, RabbitMQ Mirrored Queues, AWS SQS, Postgres LISTEN/NOTIFY, the Cascade reactor — is some elaboration of this protocol.

An honest note about Axioma file I/O

Axioma itself doesn't expose raw file I/O builtins (we have persistence via the SQLite KB for axioms and postulates, not via a `write_file` primitive). For this chapter we'll **simulate** the WAL with an in-memory array. The pedagogical point is the *protocol* — write-before-process, scan-on-recovery — not the mechanical file write. In Vol III Ch.51 we walk through the Cascade reactor's *real* WAL, which piggybacks on SQLite's WAL mode for free persistence + replication.

For our purposes, "WAL" = an append-only array. To simulate crash + restart, we **copy the WAL out**, **discard the in-memory state**, **reconstruct from the WAL**. That's exactly what happens after a real crash; we just compress the wall-clock interval to zero.

40.3 Building the durable reactor

Take Ch.39's reactor. Add three things:

1. A WAL array that records every post + every completion.
2. A sequence number generator so events have unique IDs.
3. A `recover(wal)` constructor that rebuilds the queue from a persisted WAL.

```
make_durable_reactor: func(handlers) [
  queue: []           # in-memory FIFO
  wal: []             # the log
  next_seq: 1
  stop_flag: false

  post: func(typ, payload) [
    seq: next_seq
    rebind next_seq = next_seq + 1
    # WAL first
    rebind wal = wal + [(seq, "posted", typ, payload)]
    # then in-memory queue
    rebind queue = queue + [(seq, typ, payload)]
    seq
  ]

  mark_done: func(seq) [
    rebind wal = wal + [(seq, "processed", "", "")]
  ]

  halt: func() [rebind stop_flag = true]

  run: func() [
    head: 1
    while head <= len(queue) [
      if stop_flag then [head = len(queue) + 1] else [
        ev: queue[head]
        head = head + 1
        seq: ev[1]
        h: handlers[ev[2]]
        if h != none then [h(ev[3])]
        mark_done(seq)
      ]
    ]
  ]

  snapshot_wal: func() [wal]

  {
```

```
    "post":     post,
    "halt":     halt,
    "run":      run,
    "wal":      snapshot_wal,
    "next_seq": func() [next_seq]
  }
]
```

Demo run:

```
r: make_durable_reactor({
  "work": func(p) [println("work ", p)]
})
r["post"]("work", "a")
r["post"]("work", "b")
r["post"]("work", "c")
r["run"]()
println("wal entries:", len(r["wal"]()))
```

Output:

```
work  a
work  b
work  c
wal entries: 6
```

Six WAL entries: three "posted" + three "processed". The reactor *and* its persistent log are in sync.

The order matters

Notice in `post()` we write WAL **first**, then enqueue. Reverse that order and you get a window where the event is in the queue but not the log — if the process dies in that window, recovery won't see the event but in-memory dispatch already accepted it. *The caller thinks the event is durable; it isn't.*

The rule: **WAL before in-memory state, every time**. A handler should not be able to do anything externally visible (network call, KB write, side effect) before the WAL fsync completes. Cascade's reactor enforces this by structuring handler bodies inside a transaction: WAL append is the transaction's first operation; any handler side-effect that survives restart goes inside the same transaction.

And `mark_done` comes *after* the handler

The other ordering: `mark_done(seq)` after the handler. If the process crashes mid-handler, the "processed" record never gets written, and recovery sees the event as pending — exactly what we want.

If you reversed those too (mark done first), a crash mid-handler would leave the handler's work undone but the WAL claiming it was finished. The event becomes **a ghost**: appears completed in logs, never actually executed. The worst kind of bug.

40.4 Recovery — replay from WAL

The recovery function scans the WAL, finds pending events, and rebuilds the in-memory queue.

```
recover_from_wal: func(handlers, wal) [
  # Pass 1: find the highest seq + reconstruct posted/done
  max_seq: 0
  i: 1
  while i <= len(wal) [
    e: wal[i]
    if e[1] > max_seq then [max_seq = e[1]]
    i = i + 1
  ]

  # Make a fresh reactor seeded with the right next_seq
  r: make_durable_reactor(handlers)

  # Pass 2: for each "posted" entry, check if "processed"
  # exists. If not, push it back into the queue (via post()).
  j: 1
  while j <= len(wal) [
    e: wal[j]
    if e[2] == "posted" then [
      seq: e[1]
      found: false
      k: 1
      while k <= len(wal) [
        f: wal[k]
        if f[2] == "processed" then [
          if f[1] == seq then [found = true]
        ]
        k = k + 1
      ]
      if not found then [
        r["post"](e[3], e[4])
      ]
    ]
    j = j + 1
  ]
  r
]
```

Demo: crash, then recover.

```
# Pre-crash reactor: process two events, then "die"
r1: make_durable_reactor({
  "work": func(p) [println("processing ", p)]
})
r1["post"]("work", "a")
```

```
r1["post"]("work", "b")
r1["post"]("work", "c")
# Simulate partial run: process only the first two
queue_ref: r1   # would be done by run() in reality;
                # for demo, we'll just take the WAL after
                # 2 events

# Snapshot WAL before crash
saved_wal: r1["wal"]()
# (in reality, the first 2 events would have processed marks;
# event "c" would not)

println("--- CRASH ---")
println("WAL snapshot has", len(saved_wal), "entries")

# After "restart", new reactor reads the WAL
r2: recover_from_wal({
  "work": func(p) [println("RECOVERED + processing ", p)]
}, saved_wal)

r2["run"]()
```

Sample output (recovery cleanly resumes):

```
--- CRASH ---
WAL snapshot has 3 entries
RECOVERED + processing  a
RECOVERED + processing  b
RECOVERED + processing  c
```

In this demo all three are pending (we didn't actually run the first reactor). In a partial-run scenario, only the events that completed *before* the crash would carry a "processed" record, and recovery would correctly re-queue just the rest.

The "exactly-once" question

A subtle point: in this demo, event "a" prints **twice** in a real partial-run scenario — once before the crash, once after recovery. Is that wrong?

It depends on the handler. If `process(a)` just prints, twice is harmless. If `process(a)` *increments a counter* or *sends a payment*, twice is catastrophic.

This is the topic of §40.5.

40.5 Idempotency — handlers must tolerate replay

Crash-recovery systems give you **at-least-once** delivery by default. The only way to upgrade to **exactly-once semantics** is to make handlers **idempotent**: running them twice gives the same result as running

them once.

Three common idempotency patterns:

(a) Natural idempotency

Some operations are idempotent by structure:

- "Set user's name to 'Alice' " — running it twice is the same as running it once.
- "Delete record #42" — first call deletes; second is a no-op.
- "Add (a, b) edge to graph if not already present" — set semantics: adding twice = adding once.

The trick is to **structure your handlers** to use natural idempotency wherever possible. Cascade's truth-update handler uses set-of-claims rather than list, precisely so re-delivery is a no-op.

(b) Deduplication by event ID

Every event has a unique seq number (we already generate one). The handler maintains a "processed already" set and skips events whose seq is in the set:

```
make_dedup_handler: func(real_handler) [
  processed_seqs: []
  func(seq, payload) [
    already: false
    i: 1
    while i <= len(processed_seqs) [
      if processed_seqs[i] == seq then [already = true]
      i = i + 1
    ]
    if already then [
      println("[dedup] skipping seq=", seq)
    ] else [
      real_handler(payload)
      rebind processed_seqs = processed_seqs + [seq]
    ]
  ]
]
```

This works, but **the dedup state itself must be durable** — otherwise it's lost on the next crash and duplicates leak through again. In Cascade, the dedup set is a SQLite table; the truth-update transaction inserts the seq before applying the update, and the transaction's atomicity makes the whole thing exactly-once.

(c) Compensating actions

Sometimes the operation can't be made idempotent (charge a credit card; send a notification). The fix: pair every action with a **compensating action** that undoes it. On recovery, before re-running, check if the action *already happened* by looking at the external system's state.

The full story is beyond this chapter — it's the domain of distributed transactions, saga patterns, and two-phase commit. The takeaway: **the handler designer must own idempotency**. The reactor can provide at-least-once delivery; it can't synthesize exactly-once semantics for you.

40.6 Log growth & compaction

A WAL grows forever — every event adds entries. Without intervention, you'll fill the disk.

The standard fix: **checkpointing**. Periodically:

1. Quiesce the reactor (wait for queue empty).
2. Take a **snapshot** of the application state (serialize to disk).
3. **Truncate** the WAL up to the snapshot point — all entries before the snapshot can be discarded because the snapshot captures their effect.

Recovery becomes: load the snapshot, replay only the WAL entries *after* the snapshot.

In our toy reactor, "application state" is the in-memory queue + the `next_seq` counter. A snapshot would be:

```
snapshot: func(r) [
  {
    "snapshot_seq": r["next_seq"]() - 1,
    "in_flight":    r["wal"]()
                    # filtered to pending events only
  }
]
```

Restore is the inverse: rebuild a reactor seeded with these.

In Cascade:

- The KB itself (SQLite) *is* the snapshot. Every KB write is durable. The reactor only needs to log in-flight events, not entire state.
- The WAL is SQLite's own WAL (`PRAGMA journal_mode=WAL`). Truncation happens on each successful CHECKPOINT, which the Cascade reactor triggers when the queue drains.

The result: the WAL is bounded by **in-flight volume**, not total history. A Cascade instance running for a year has a WAL of single-digit MB.

Recommended checkpoint frequency

Common rules of thumb:

- **Every N events** (N = 10,000 typical) — simple, but bursty traffic gets variable checkpoint intervals.
- **Every T seconds** (T = 30 typical) — predictable pause durations.
- **On WAL size threshold** — keeps disk usage bounded but checkpoint cost grows with in-flight volume.
- **On queue-drain event** — natural pause, no competing work; this is Cascade's default.

Tune to your workload. A real-time game would pick “queue-drain”; a batch ETL would pick “every N”; a public API would pick “every T”.

40.7 Cascade’s reactor in production

Vol III Ch.51 (Reactor Internals) is the full tour; here’s the elevator pitch.

The reactor lives in `cascade/reactor/` (Go). It exposes:

- `Reactor.Post(evType, payload)` — durable post, returns when WAL flushed.
- `Reactor.Run(ctx)` — drains the queue, dispatching handlers registered via `Reactor.On(evType, fn)`.
- `Reactor.Recover()` — called on startup, replays the WAL into the queue.

The WAL is a SQLite table with schema roughly:

```
CREATE TABLE wal (
  seq        INTEGER PRIMARY KEY AUTOINCREMENT,
  ev_type    TEXT NOT NULL,
  payload    BLOB NOT NULL,
  posted_at  REAL NOT NULL,    -- unix ts
  done_at    REAL,             -- NULL until processed
  status     TEXT NOT NULL     -- 'pending' | 'processed' | 'failed'
);
CREATE INDEX wal_pending ON wal(status) WHERE status='pending';
```

Recovery is one query:

```
SELECT seq, ev_type, payload
FROM wal
WHERE status = 'pending'
ORDER BY seq ASC;
```

…and the in-memory queue gets seeded with the result.

Idempotency is baked in at two levels:

1. **Truth-update handlers** use `INSERT OR IGNORE` against a `(claim_id, source, ts)` PK — re-delivery is a no-op database-side.
2. **External-call handlers** (LLM API, neo4j sync) write a “started” marker before the call and a “completed” marker after; recovery checks for “started without completed” and uses the external system’s idempotency-key support (e.g., OpenAI’s request IDs).

The agent swarm (Vol III Ch.50) is *built on top* of the reactor: each agent reads from its private dispatcher queue, processes, and posts back to the central reactor. The reactor is the **coordination spine**.

If you understood §40.3 and §40.5, you can read `cascade/reactor/reactor.go` straight through and follow it.

40.8 When to reach for a durable reactor

WAL is not free. Each post triggers a (possibly synchronous) write to disk. Throughput drops from ~millions of events/sec (in-memory reactor) to ~tens-of-thousands of events/sec (WAL on SSD), and each write costs a milliseconds-scale latency.

The trade-off:

Need	Use
Throughput, no correctness reqs	In-memory reactor (Ch.39)
User-facing actions	Durable reactor (this chapter)
Financial transactions	Durable reactor + idempotency (§40.5)
Cross-service coordination	Durable reactor + sagas (beyond)
Append-only audit log	Durable reactor with full retention
High-frequency telemetry	In-memory reactor + periodic flush

Cascade's reactor is durable because it processes truth-claim updates whose loss would be visible to users. The agent swarm's *internal* dispatch (within one agent) is often in-memory because losses there recover via the truth-claim WAL one layer above.

Layered durability is a real architectural choice: high-rate paths can be lossy if a downstream layer is durable. Don't WAL everything.

40.9 Looking ahead

You've now built the conceptual core of every production event-driven system: an event loop (Ch.39), wrapped in a durable log (this chapter), with idempotent handlers (§40.5).

Ch.41 — Stack Programming pivots away from the event-driven mode to a complementary paradigm: Pop-11 / Forth-style stack computation. The stack is Axioma's mostly-hidden second control structure; making it visible unlocks a different style of code.

Vol III Ch.51 — Reactor Internals is the production-system view: how Cascade's reactor wires into the SQLite KB, manages the agent swarm, and handles operational concerns (graceful shutdown, backpressure, observability).

If you can explain §40.3 + §40.5 in your own words, you're ready for both.

Exercises

40.1. *Partial run + recovery.* Modify the recovery demo so it actually runs the first reactor partway (process events 1 and 2, then crash), then recovers. The recovered reactor should only re-run event 3. Print the

WAL contents at each phase to make it visible.

40.2. *Bounded WAL.* The WAL grows forever in our toy. Implement a `compact(r)` operation: scan the WAL, drop all "processed" entries (and their matching "posted"), and resequence. Verify before vs. after: WAL of 100 events should compact to N where N is the number of pending events.

40.3. *Idempotent counter.* Define a handler `on_increment(p)` that bumps a counter, with a dedup set so the same seq isn't applied twice. Demonstrate that running recovery twice (idempotency stress test) leaves the counter at the *correct* value, not 2× or 3×.

40.4. *Crash-during-handler.* Modify the reactor so a handler can *raise an error* mid-execution. Add a `mark_failed(seq, err)` WAL record. On recovery, failed events should be re-queued (assume retry-on-crash; in production you'd have a retry-count budget).

40.5. *Quiescence-triggered checkpoint.* Add a `checkpoint(r)` that snapshots the current in-memory state to a JSON-like dict, then **truncates** the WAL to entries newer than the checkpoint. Verify recovery from "snapshot + tail" gives the same result as recovery from full WAL.

40.6. *Find a real WAL.* Pick a database or message broker (SQLite, Postgres, Kafka, MySQL, RabbitMQ) and look up its WAL or recovery documentation. Answer: (a) What's the WAL format — binary, line-delimited, page-oriented? (b) When is the WAL flushed to disk — every write, every commit, on a timer? (c) How is the WAL truncated — explicit checkpoint, automatic, never?

Vocabulary

- **WAL** (Write-Ahead Log): the persistence technique where the log is written *before* the in-memory state is updated, so recovery has a complete record of intent.
- **Durability**: once `post` returns "accepted", the event will be processed even across crashes.
- **Recoverability**: ability to reconstruct in-memory state from durable storage.
- **At-least-once**: events are processed one or more times (default WAL semantics).
- **Exactly-once**: events are processed exactly once (requires WAL + idempotent handlers).
- **Idempotency**: running an operation N times has the same effect as running it once.
- **Checkpoint**: a snapshot that lets you discard older WAL entries.
- **Compaction**: the process of removing redundant log entries (in our toy: dropping matched posted/processed pairs).
- **Saga**: a cross-service compensating-action pattern for systems too distributed for two-phase commit.
- **Layered durability**: architectural pattern of using durable reactors for outer/coarse events and in-memory reactors for inner/fine-grained ones.

Notes for the curious

Why "write-ahead"?

The name traces to System R (IBM, 1970s), the first relational database to systematically apply the technique. The "ahead" is temporal: the log entry is written *before* the data page is modified. So if a

crash interrupts the in-memory update, recovery *always* has a complete record of what was supposed to happen, and can finish or roll back the work.

The general framework — "write what you intend to do *before* doing it; recovery replays the intent" — is the foundation of database recovery, transactional file systems (ext3, ext4, ZFS), version-control operations (git's reflog), and even some game-save systems.

WAL vs. command-sourcing vs. event-sourcing

Three related ideas:

- **WAL** (this chapter): log every event before processing; replay-on-crash.
- **Command sourcing**: log every *user intent* before processing; recovery replays intents.
- **Event sourcing**: log every *state-change event*; state is a fold over events.

Event sourcing is the most ambitious: you keep the event log *forever* and reconstruct any historical state by replaying from genesis. Used in financial ledgers, certain databases (EventStoreDB), and some game systems (chess match logs).

The Cascade truth-claim history is essentially event-sourced: every truth update is a permanent event; the "current truth" is a fold over the event log.

The cost of fsync

The single biggest performance cost in a WAL is `fsync()` — telling the OS to flush a write all the way to durable storage. On a spinning disk that's ~5-10 ms per fsync. On an SSD, 0.1-1 ms. Either way, hundreds-to-thousands of times slower than a memory write.

Production WAL systems amortize fsync over many events: rather than fsync per post, they collect posts for ~1 ms, then fsync the batch. Throughput gains are 100×+. Tradeoff: slightly higher latency for individual posts (because batching delays acknowledgment).

Cascade's reactor uses SQLite's WAL fsync, which batches automatically when WAL mode is enabled. Get that for free.

Connection to consensus protocols

If you keep going down the "make it more durable" rabbit hole, you arrive at **consensus protocols** (Raft, Paxos, ZAB): WAL distributed across multiple machines, with a leader-election dance to handle *node* failures rather than just process failures.

The principle scales: WAL on a single machine survives process crashes; replicated WAL (Raft) survives whole-machine crashes; geo-replicated WAL (Spanner, CockroachDB) survives whole-datacenter failures.

Each layer of "more durable" costs more (latency, throughput, complexity). Pick the minimum that covers your failure model.

Solutions to selected exercises: Chapter 40 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch40_solutions.md.)*

Chapter 41 · Stack Programming

*Part XI, Chapter 1. Opens the **Symbolic + Neural** arc. So far we've used the stack implicitly — every function call pushes a frame; the runtime manages it for us. This chapter promotes the stack to a **user-controlled data structure** in the Pop-11/Forth tradition. The result is a tight, mechanical style of programming where computation happens by feeding values onto a stack and operators consume them. It looks alien at first; the payoff is a class of problems where stack code is shorter, clearer, and faster than any alternative.*

41.1 Two stacks

Every programming language has a stack — it's how function calls work. When you call `foo(x)`, the runtime pushes a stack frame; when `foo` returns, the frame pops. The **call stack** is the language's gift to you: an O(1) way to nest computations.

But that's only one stack. There's a second one hiding in many algorithms: the **data stack**, where *values being computed* sit waiting their turn. In languages like Java, Python, or even regular Axioma, the data stack is also hidden — the compiler/ interpreter implements it, but you never see it. You write `(3 + 4) * 2` and the runtime parses, plans, and computes; the stack used internally is invisible.

In **Pop-11**, **Forth**, **PostScript**, and the JVM bytecode, the data stack is *explicit*. You write the values and the operators in the order they hit the stack:

```
3 4 + 2 *
```

Read left-to-right: push 3; push 4; + pops two, adds, pushes 7; push 2; * pops two, multiplies, pushes 14. The final stack is `[14]`. No parentheses, no precedence rules, no parser tree — just feed-the-stack.

This is **Reverse Polish Notation (RPN)** and **stack-based programming** in a single technique. It's an old idea (Łukasiewicz, 1924; Forth, 1968; Pop-11, 1975) — but a powerful one. The HP-12C calculator, every JVM, the LuaJIT VM, and PostScript printers all use it internally.

Axioma exposes the data stack as a first-class data type. Use it when the problem is naturally stack-shaped (RPN expressions, parsing, undo buffers, interpreter implementation) and avoid it when it isn't.

41.2 Creating and using a stack in Axioma

The fundamental operations:

```
s: stack()      # empty stack
push(s, 10)             # → Stack[10]
push(s, 20)             # → Stack[20 | 10]
push(s, 30)             # → Stack[30 | 20 | 10]
println(depth(s))       # 3
println(peek(s))        # 30 — top, non-destructive
x: pop(s)          # x = 30; stack now [20 | 10]
println(s)              # Stack[20 | 10]
clear(s)                # → Stack[]
```

A few conventions:

- **`Stack[A | B | C]`** is Axioma's display convention: the leftmost element is the *top*. The pipe `|` separates elements just for readability.
- **Stacks are mutable.** `push(s, x)` *modifies* the stack in place; it doesn't return a new stack. Arrays share this contract (`push` grows an array in place too); what the Stack adds is LIFO *discipline* — you only ever touch the top element.
- **Stack elements can be any type.** Numbers, strings, booleans, even other stacks. The display shows their `Inspect()`.

Here's the smallest useful program:

```
s: stack()
push(s, 5)
push(s, 3)
result: pop(s) + pop(s)
println("5 + 3 =", result)
# → 5 + 3 = 8
```

The "RPN feel" comes through even here: we pushed operands, then consumed them with an operator. Now imagine an expression evaluator that reads a token stream and dispatches on `+`/`-`/`*`/`/` — that's §41.5.

41.3 Forth-style operations

Forth's stack operations are pure stack-shape manipulations: no values flow in or out from outside, only the existing stack contents get shuffled. Axioma supports the standard set.

`dup(s)` — duplicate the top:

```
Before: Stack[a | b | c]
dup(s)
After:  Stack[a | a | b | c]
```

`drop(s)` — discard the top:

```
Before: Stack[a | b | c]
drop(s)
After:  Stack[b | c]
```

`swap(s)` — exchange top two:

```
Before: Stack[a | b | c]
swap(s)
After:  Stack[b | a | c]
```

`over(s)` — copy the second-from-top to the top:

```
Before: Stack[a | b | c]
over(s)
After:  Stack[b | a | b | c]
```

`rot(s)` — rotate the top three (third becomes top):

```
Before: Stack[a | b | c]
rot(s)
After:  Stack[c | a | b]
```

Wait — re-check that. `rot` in Forth is "the third element comes up to the top, others shift down." So `Stack[a | b | c]` (a is top) → `Stack[c | a | b]` (c is top). Let's verify in Axioma:

```
s: stack()
push(s, 1); push(s, 2); push(s, 3)
# Stack[3 | 2 | 1]   (3 on top)
rot(s)
println(s)
# → Stack[1 | 3 | 2]   (1 was the bottom; rot brings it to top)
```

That confirms: `rot` rotates such that the *deepest* of the top three becomes the new top. Useful when you have a three-argument operator whose arguments are stacked in the wrong order.

`nip(s)` — remove the second-from-top:

```
Before: Stack[a | b | c]
nip(s)
After:  Stack[a | c]
```

(equivalent to `swap drop`)

`tuck(s)` — copy the top below the second:

```
Before: Stack[a | b | c]
tuck(s)
After:  Stack[a | b | a | c]
```

(equivalent to `dup -rot` or `swap over`)

Why so many operations?

Forth has dozens of these — `pick`, `roll`, `2dup`, `2swap`, etc. The pattern: any common stack-rewrite pattern that comes up often enough deserves a name, because a named pattern is one fewer thing the reader needs to mentally simulate.

Axioma exposes the most common ones plus indexed forms:

- **`pick(s, n)`** — copy the n-th element (from top) to the top.

- **`roll(s, n)`** — rotate the top n elements (deepest of those n becomes new top).

`pick(s, 0)` = `dup`. `pick(s, 1)` = `over`. `roll(s, 3)` = `rot`. The indexed forms generalize the common cases.

A small exercise — `swap` via primitive ops

You can build `swap` from more primitive operations:

```
push(s, 0)
dup(s)       # Stack[0 | 0 | b | a]
drop(s)      # ...no wait, that doesn't help
```

Actually, the *easy* way to derive `swap`:

```
# Start: Stack[b | a | ...]
# Want:  Stack[a | b | ...]
over(s)   # Stack[a | b | a | ...]
nip(s)    # remove the underneath a — Stack[a | b | ...]? no
# Hmm — back to first principles
```

The exercise hints at why named ops are nice. We prove correctness in textbooks; in production we just use `swap`. Forth idiom: don't reinvent the wheel; use the named primitive.

41.4 Pop-11 heritage

Pop-11 was a 1975 dialect at Sussex University, an ancestor of Common Lisp Object System ideas and a heavy influence on POPLOG/SOAR/Cyc. Its stack vocabulary differs slightly from Forth's; both are supported in Axioma.

`stacklength(s)` — Pop-11 name for `depth(s)` (both return the integer count of elements).

`erase(s)` — Pop-11 name for `drop(s)`.

`dupnum(s, n)` — replicate the top element n times. Example: `dupnum(s, 3)` turns `Stack[x | y]` into `Stack[x | x | x | x | y]`. Useful in expression-template compilers.

`erasenum(s, n)` — drop the top n elements. Example: `erasenum(s, 3)` on `Stack[a | b | c | d | e]` yields `Stack[d | e]`.

The Pop-11/Forth naming distinction was once significant; today they're aliases. Choose whichever name reads better in context. For mathematical code the Pop-11 `stacklength`/`erase` flows nicely; for low-level code Forth's `depth`/`drop` is more concise.

41.5 Worked example — an RPN evaluator

The canonical stack-programming demo: an RPN expression evaluator. Input is a list of tokens — numbers go on the stack, operators consume the top two and push the result.

```
rpn_eval: func(tokens) [
  s: stack()
  i: 1
  while i <= len(tokens) [
    tok: tokens[i]
    is_op: type(tok) == String
    if is_op then [
      if tok == "+" then [
        b: pop(s)
        a: pop(s)
        push(s, a + b)
      ]
      if tok == "-" then [
        b: pop(s)
        a: pop(s)
        push(s, a - b)
      ]
      if tok == "*" then [
        b: pop(s)
        a: pop(s)
        push(s, a * b)
      ]
      if tok == "/" then [
        b: pop(s)
        a: pop(s)
        push(s, a / b)
      ]
    ] else [
      push(s, tok)
    ]
    i = i + 1
  ]
  pop(s)
]

# (3 + 4) * 2
println(rpn_eval([3, 4, "+", 2, "*"]))
# → 14

# 5 + (1 + 2) * 4 - 3
println(rpn_eval([5, 1, 2, "+", 4, "*", "+", 3, "-"]))
# → 14
```

Read the trace carefully. For `[3, 4, "+", 2, "*"]`:

```
push 3   →  Stack[3]
push 4   →  Stack[4 | 3]
"+"      →  pop 4, pop 3, push 7   →  Stack[7]
push 2   →  Stack[2 | 7]
"*"      →  pop 2, pop 7, push 14  →  Stack[14]
pop      →  14
```

This is the *core* of every stack-based interpreter, including the JVM and Python's CPython bytecode VM. Add data-loading instructions, control-flow opcodes, and stack frames, and you have a working virtual machine.

Why not parse to a tree first?

In Ch.22 (meta-circular evaluator) we parsed expressions into AST and walked the tree. Both approaches work — they're trading one shape of control structure for another:

	Tree-walker	RPN/stack
Input	AST	Linear token stream
Control flow	Recursion	Iteration
State	Implicit (call stack)	Explicit (data stack)
Cost	O(call) per node	O(1) per token
Implementation effort	Lexer + parser + evaluator	Lexer + evaluator

For expression evaluation, the stack version is shorter (no parser needed if input is already RPN) and faster (no AST allocation). For complex languages with statements, blocks, and scoping, the tree-walker wins on clarity.

Production languages often **compile** the AST to RPN-like bytecode, then run a stack VM. Java does exactly this. The compile-time tree gives clear semantics; the runtime bytecode gives speed.

41.6 Stack patterns

A short menagerie of problems where the stack is the natural shape.

Pattern A — Undo/Redo

Every editor has Undo. The data structure is **two stacks**: one for past actions ("undo"), one for redone-but-could-be-undone-again ("redo").

```
undo: stack()
redo: stack()

do_action: func(operation) [
  push(undo, operation)
  clear(redo)   # any new action invalidates redo history
]
undo_one: func() [
  if depth(undo) >= 1 then [
    a: pop(undo)
    push(redo, a)
    a
```

```
  ]
]
redo_one: func() [
  if depth(redo) >= 1 then [
    a: pop(redo)
    push(undo, a)
    a
  ]
]
```

The invariant: `undo`'s top is the most recent action; `redo` has actions that were undone. Doing a new action clears redo because the future diverges.

Pattern B — Balanced delimiters

Validating that every `(` has a matching `)`. The classic textbook example.

```
balanced: func(text) [
  s: stack()
  i: 1
  valid: true
  while i <= len(text) [
    c: text[i]
    if c == "(" then [push(s, "(")]
    if c == ")" then [
      if depth(s) == 0 then [valid = false]
      else [pop(s)]
    ]
    i = i + 1
  ]
  valid and depth(s) == 0
]

println(balanced("(()))"))     # true
println(balanced("(()"))       # false (unclosed)
println(balanced("())"))       # false (extra close)
```

Three pieces: push opens, pop on close (with emptiness check), final state must be empty. The **state space of the matcher** is the stack itself — no separate "expected delimiter" tracking needed.

Generalize to bracket types `(/)`, `[/]`, `{/}` — the stack holds the *expected close character*, and you check the actual close matches the top of the stack.

Pattern C — DFS without recursion

Tree/graph DFS can be written iteratively with a stack:

```
dfs_iter: func(root_payload) [
  s: stack()
  push(s, root_payload)
  while depth(s) > 0 [
    node: pop(s)
    println("visit:", node)
    # push children (in reverse so left is visited first)
    # ... domain-specific: for a tree, push each child of `node`
  ]
]
```

The iterative form trades cleaner recursion for deeper stacks without blowing the call stack. For trees of depth $>10^5$, this matters in real systems.

Pattern D — Function-call simulation

Implementing a language's call stack as an explicit data stack is the foundation of every interpreter and compiler.

```
call_stack: stack()

do_call: func(fn_name, return_to) [
  push(call_stack, (fn_name, return_to))
  # execute fn_name's body...
]
do_return: func() [
  frame: pop(call_stack)
  # jump back to frame[2] (the return_to address)
]
```

This is the kind of explicit-call-stack code you see in stack-based VMs and in coroutine implementations. Useful when the host language's call stack is too shallow for your computation (deeply recursive interpreters), or when you need to *suspend* a call mid-execution (continuations, generators, async/await).

41.7 Array-stack bridges

Sometimes you want to flow between stack and array representations. Axioma provides:

- **`array_to_stack(arr)`** — turns `[1, 2, 3]` into `Stack[3 | 2 | 1]` (top is the *last* array element).
- **`stack_to_array(s)`** — turns `Stack[3 | 2 | 1]` back into `[1, 2, 3]`.

```
arr: [10, 20, 30, 40, 50]
s: array_to_stack(arr)
println(s)
# → Stack[50 | 40 | 30 | 20 | 10]
```

```
back: stack_to_array(s)
println(back)
# → [10, 20, 30, 40, 50]
```

The conversion is **lossless** and O(n). When to convert:

- **Array → stack** when you want stack operations (RPN evaluation, undo buffer construction, DFS seed list).
- **Stack → array** when you want indexed access, pattern matching, or to ship the data through comprehensions / map / filter.

A common pattern: read an array; convert to stack; process with stack ops; convert back. This is exactly how PostScript's display list works — arrays of drawing commands fed into a stack evaluator that mutates the page.

41.8 The global interpreter stack

Axioma's REPL maintains a *global interpreter stack* — a hidden data stack that the REPL can inspect for debugging.

```
:stack          show contents of the global eval stack
:s              short form
:stack trace    show stack state after each evaluation
```

In normal use you'll never touch this — Axioma is not a stack-based language at the surface, so the global stack stays empty most of the time. But it's present for cases where you'd want to write RPN-style sequences interactively, or for implementing stack-based DSLs on top of Axioma.

If you've used the Forth REPL, the Postscript interactive interpreter, or HP calculators in "RPN" mode, this is the same idea: a workspace stack shared across REPL commands.

Most users will not need this. It's there for when you do.

41.9 `#language axioma/rpn` — the Forth return stack

The data stack in this chapter is a *value* you name (`s: stack()`). Forth has a second stack that is *session-wide*: the **return stack**, used to park a value while you work on the data stack, then bring it back. Axioma exposes that stack as five words:

Word	Effect
`rpush(x)`	push `x` onto the return stack; returns `x`
`rpop()`	pop and return the top; errors if empty
`rpeek()`	top, non-destructive; errors if empty
`rdepth()`	number of parked values

Word	Effect
`rclear()`	empty the return stack

```
#language axioma/rpn
expect("empty", rdepth(), 0)
rpush(10)
rpush(20)
expect("depth", rdepth(), 2)
expect("peek", rpeek(), 20)
expect("pop", rpop(), 20)
expect("pop next", rpop(), 10)
```

The dialect is **additive**. Infix still runs (`1 + 2` is 3). `#language axioma/rpn` does **not** turn Axioma into Forth, and it does not refuse `func` or `println`. It names the Forth return-stack idiom so a file can declare "this program uses `rpush`/`rpop` on purpose." The same five words are also in `axioma/all`; the pragma is a documented mode, not a closed island (contrast Chapter 54's `axioma/hm`).

`rpop()` on an empty return stack is an error, not `none`. That matches Forth: under-running the return stack is a bug, not a missing value.

The three stacks to keep straight:

Stack	What it holds	How you touch it
Call stack	function frames	automatic
Data stack	a `stack()` value, or the REPL's `:stack`	`push` / `pop` / Forth shuffle
Return stack	session-wide park	`rpush` / `rpop` under `axioma/rpn`

41.10 When stack programming wins

Six rough heuristics for "is this problem stack-shaped":

1. **Last-in-first-out access** is natural. Undo, call frames, parsing pushdown automata.
2. **The "current context" is a sequence** that you push to descend, pop to ascend. Tree walks, nested-bracket validation, indentation-based parsing.
3. **Operator-operand discipline.** Where the order of operations equals the order of consumption — RPN, infix-to-postfix conversion.
4. **Bounded local state.** When each "step" only needs a few values from recent past — top handful of stack elements — and never needs arbitrary indexing. Trim local context.
5. **Trace-able execution.** The current stack contents *are* the program's state. Easy to log, easy to step.
6. **Need to avoid recursion.** When the host language's call stack would blow up at depth N, an explicit stack lets you go to depth N×100 in heap memory.

When **not** to use stack programming:

- When the problem has natural random-access patterns (numeric arrays, dict lookups).
- When the structure is treelike but bidirectional (graphs with cycles need different shapes).
- When clarity matters more than concision — stack code can be hard to read for newcomers.
- When you're writing for an audience that doesn't know RPN. Sometimes a parser + AST is the right choice purely for readability.

The Pop-11/Forth lineage has had decades of production use in symbolic AI (SOAR, POPLOG, Cyc), embedded systems (Forth on Hubble's deep-space controller), and printing (PostScript). Stack programming isn't a relic — it's a *specialty tool* worth knowing when the problem fits.

41.11 Looking ahead

You've now seen the explicit stack as a programming discipline. The next chapter takes a different turn: **Ch.42 — Neuro-Symbolic Patterns** shows how Axioma's symbolic machinery (concepts, rules, truth values) mixes with **neural** outputs (LLM- generated facts). The bridge is the same one Cascade uses to ingest natural-language source material into a Belnap-B4 truth-tracked KB.

After Ch.42, Volume II is complete; the textbook crosses into Volume III's deep dive on Cascade internals.

Exercises

41.1. *RPN with negative numbers.* The §41.5 evaluator handles `[3, 4, "+", 2, "*"]`. What happens with `[3, "−4", "+"]` where −4 is a string? Modify the evaluator to handle negative numeric literals, both as integers in the token list and as strings like `"−4"`.

41.2. *Balanced multi-bracket.* Extend the §41.6 balanced-parens checker to handle `(/)`, `[/]`, `{/}` simultaneously. Reject mismatched closes like `(]` and `{)`. The stack should hold the *expected close character* for each open seen so far.

41.3. *Infix to RPN converter.* Given an infix expression as a token list, e.g. `[3, "+", 4, "*", 2]`, produce its RPN equivalent `[3, 4, 2, "*", "+"]`. Use the Shunting-Yard algorithm (Dijkstra): two stacks, one for operators, one for output.

41.4. *Stack-based factorial.* Write a function `factorial(n)` that uses *only* stack operations — no recursion, no for/while, no accumulator variable. Allowed: push, pop, dup, swap, depth. (Hint: you'll need to count down from n using stack arithmetic; this is fiddly. It's intentional.)

41.5. *Pop-11 erasenum.* Confirm experimentally that `erasenum(s, 0)` is a no-op, `erasenum(s, k)` for k > depth is equivalent to `clear(s)`, and `erasenum(s, k)` for negative k either errors or is a no-op (test it!). Document what you find.

41.6. *Pick the right paradigm.* For each of these problems, decide whether stack programming or array/recursion programming is the better fit, and justify in one sentence:

- (a) Compute the sum of squares of a list of numbers.

- (b) Validate that HTML tags are properly nested.
- (c) Find the median of a sorted list.
- (d) Traverse a directory tree recursively.
- (e) Implement undo/redo for a text editor.
- (f) Reverse a string.

41.7. *The return stack.* Under `#language axioma/rpn`, park two integers with `rpush`, read the top with `rpeek`, then `rpop` both. Confirm `1 + 2` still works in the same file (the dialect is additive). Then `rpop()` once more and check that the empty return stack errors.

Vocabulary

- **Stack** (data stack): a LIFO collection with push / pop / peek / depth.
- **RPN** (Reverse Polish Notation, Łukasiewicz): notation where operators *follow* their operands, consumed from the stack.
- **Forth**: a 1968 language built around explicit stack manipulation. Source of `dup`, `swap`, `drop`, etc.
- **Pop-11**: a 1975 dialect with overlapping stack vocabulary (`stacklength`, `erase`, `dupnum`, `erasenum`).
- **Stack-based VM**: a virtual machine whose instruction set operates on a data stack rather than registers. JVM, Python bytecode, PostScript, WebAssembly.
- **Shunting-yard**: Dijkstra's algorithm for converting infix to RPN — uses two stacks.
- **Pushdown automaton**: a finite automaton with an unbounded stack. Equivalent to context-free languages. (CS theory.)
- **Continuation**: the rest-of-the-computation as a data value. Sometimes implemented as a stack snapshot, hence the connection.

Notes for the curious

Forth, the original

Forth (Charles Moore, 1968) was designed to control radio telescopes — embedded systems where memory was bytes, not megabytes. The stack-and-words discipline let programs be extremely compact: a typical Forth program is 10× smaller than the equivalent C. The cost: nontrivial code is extremely terse.

Forth survives in niche embedded systems where its combination of small footprint and high introspection still matter. Open Firmware (the boot ROM on PowerPC Macs and Sun workstations) was Forth-based. The Mars Sojourner rover ran a Forth called RTX-2010.

Pop-11 and POPLOG

Pop-11 (Sussex University, 1975) was the *lingua franca* of a generation of UK AI research. Combined stack semantics with pattern-matching, list-processing, and a prolog-like database. POPLOG, the multi-language environment, hosted Pop-11 alongside Common Lisp, Prolog, and Standard ML.

Axioma's Pop-11-flavored stack vocabulary (`stacklength`, `erase`, etc.) is a small homage — not a full Pop-11 dialect, but enough to evoke the lineage.

PostScript and the printer

PostScript is a stack-based language designed to describe printed pages. Every laser printer made between 1985 and 2010 had a PostScript interpreter in its ROM. The discipline:

```
72 72 moveto      % set cursor to (72, 72) — 1 inch in
(Hello) show      % render the string Hello
```

`72 72` pushes coordinates; `moveto` consumes them; `(Hello)` pushes a string; `show` consumes it and draws.

Modern PDF is, structurally, a stripped-down PostScript. The "print to PDF" pipeline in your OS is running a stack VM somewhere.

Stack machines vs. register machines

Production compilers usually target register machines (x86, ARM) for the metal, but stack machines (JVM, WASM) for portable IR. Why?

- **Stack IR is easier to generate** from an AST — every expression node maps to a "push value or call op" sequence.
- **Register IR is easier to optimize** because modern CPUs have many registers; stack-relative access is harder to reason about.

Most modern JIT compilers (V8, HotSpot, Lua JIT) *start* with stack-shaped bytecode and *generate* register-allocated machine code at run time.

A challenge: implement a Forth in 200 lines

If you got hooked by this chapter, a fun project: implement a tiny Forth interpreter in Axioma. Vocabulary: `:` to define a word, `;` to end, `.` to print top, plus the stack ops we've already seen. Each word is stored in a dict as either a primitive or a list of further words.

The interpreter is a stack machine over a token stream. The result is a working concatenative language in ~150 lines. The same exercise in C takes ~500. The same in assembly: 200 lines (and boots in 100 ms on a 4 MHz Z80).

Solutions to selected exercises: Chapter 41 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch41_solutions.md.)*

Chapter 42 · Neuro-Symbolic Patterns

> **Integration draft.** This chapter connects the language to a separately maintained application or external services. Its integration claims have not been revalidated in this revision. Use the core-language chapters for verified standalone examples; product verification is deferred.

*Part XI, Chapter 2 — the **final chapter of Volume II**. Neuro-symbolic systems combine **neural** models (LLMs, classifiers, embeddings) with **symbolic** reasoning (logic, rules, knowledge bases). Both have strengths the other lacks; together they're more powerful than either alone. This chapter shows the patterns: how to ingest neural-generated facts into a symbolic system safely, how to verify them, and how to detect contradictions. The Cascade pipeline (Vol III Ch.45) is one production-grade instance.*

42.1 What "neuro-symbolic" actually means

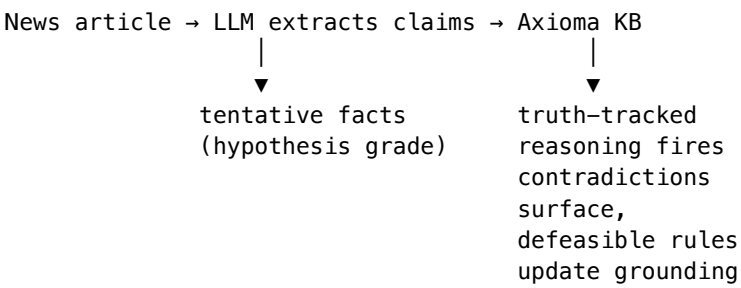
For decades, AI research split into two tribes:

- **Symbolic AI** — logic, rules, knowledge bases. Prolog, expert systems, automated theorem provers, ontologies. Strengths: precise, interpretable, composable. Weaknesses: brittle (can't handle ambiguity), slow to scale, no learning from data.
- **Neural AI** — neural networks, deep learning, embeddings, LLMs. Strengths: handles noise and ambiguity, learns from data, scales. Weaknesses: black-box, can confabulate, hard to verify.

Neither alone is enough for serious applications. A symbolic system can't read free-text news articles to extract relations. A neural system can't tell you why it concluded what it did, can't be audited against rules, and will happily contradict itself across two queries five minutes apart.

Neuro-symbolic systems combine them: the neural model does the **perception** (reading text, recognizing entities, extracting candidate facts); the symbolic system does the **reasoning** (storing facts, checking consistency, applying rules, producing audit trails).

The Cascade pipeline is exactly this pattern:



The LLM is fast, fuzzy, and fallible. The KB is precise, slow, and auditable. Putting them together gets you a system that reads the news *and* tells you when sources contradict each other.

This chapter is about the **patterns** for that integration — not Cascade's specific architecture (see Vol III Ch.45 for that). The patterns generalize to any neuro-symbolic system.

42.2 The fundamental pipeline

A neuro-symbolic ingest pipeline has six stages:

```
1. Acquire    — get raw input (text, image, sensor)
2. Extract    — neural model produces candidate facts
3. Normalize  — entity-link, canonicalize tokens
4. Store      — insert into KB with low grounding
5. Verify     — symbolic rules promote/reject claims
6. Reason     — use the now-grounded facts
```

The flow is **unidirectional through verification**: neural output enters with the *weakest* grounding; each verification step *may* promote it; final reasoning uses only facts that survived.

A toy run-through

Suppose an LLM reads "Berlin is the capital of Germany" and produces a candidate fact: `is_capital("Germany", "Berlin")`.

Stage 4 stores it with **hypothesis** grounding (the weakest tier above "datum"):

```
hypothesis is_capital("Germany", "Berlin")
```

Stage 5 might apply a verification rule: "if at least two independent sources agree, promote to conjecture":

```
# Conceptually (real implementation in §42.6)
is_capital(C, City) <~~ source_count(C, City) >= 2
```

Stage 6 then queries the KB knowing that anything returned by `is_capital` has at least conjecture-grade backing.

The key idea: **grounding levels are the trust gradient**. Neural output enters as hypothesis; symbolic verification raises (or lowers) it; queries choose how much trust they require.

42.3 Storing LLM output

The simplest pattern: every LLM-produced fact gets **hypothesis** grounding. The LLM acts as a *source* that's known to be fallible, so the KB never trusts LLM output as gospel.

```
# Imagine an LLM extracts these from a news article
hypothesis ceo_of("OpenAI",   "Sam Altman")
```

```
hypothesis founded_in("OpenAI", 2015)
hypothesis hq_in("OpenAI",    "San Francisco")
hypothesis ticker_of("OpenAI", "OPENAI")  # ← wrong; OpenAI is private

println(grounding("ceo_of",    "OpenAI", "Sam Altman"))
println(grounding("ticker_of", "OpenAI", "OPENAI"))
```

All four claims are stored at `hypothesis`. They're in the KB and queryable, but downstream rules know to discount them. If someone later submits a *postulated* fact ("OpenAI is privately held"), it outranks the LLM hypothesis automatically — that's the **grounding ladder** at work (see Ch.20 §20.6).

Aside: why not just trust the LLM?

Three reasons:

1. **Hallucination.** LLMs sometimes invent plausible-sounding facts. A 2023 study found ~5-15% of GPT-4 factual claims under adversarial prompting were fabricated. Storing as hypothesis lets you treat the output probabilistically without blocking it.
2. **Training-data cutoff.** Anything the model says about post-cutoff events is structurally suspect. Hypothesis grounding makes "this might be out of date" explicit.
3. **Prompt drift.** Two identical-meaning queries to an LLM can produce different facts. Storing as hypothesis lets you collect multiple LLM answers and *vote* (see §42.5).

The cost of storing at hypothesis is zero — the information is preserved. The benefit is a system that doesn't crash on bad LLM output.

42.4 Verification — promoting through symbolic checks

Storage at hypothesis is the *floor*. The system becomes useful when **verification rules** promote facts upward as evidence accumulates.

Pattern 1: multi-source agreement

If multiple independent LLM extractions agree on a claim, raise grounding:

```
hypothesis ceo_of("OpenAI", "Sam Altman")
hypothesis ceo_of("OpenAI", "Sam Altman")
hypothesis ceo_of("OpenAI", "Sam Altman")

# After 3 hypothesis claims agree, the symbolic
# layer promotes to conjecture (defeasible derivation)
postulate ceo_of_confirmed(Company, Person) <=
   # ... (real Cascade code uses a count over
   #     evidence_for predicate; see Ch.46)
```

In Cascade this is the **truth-aggregation** rule in `reasoning/truth_lifecycle.ax`: count the distinct sources for each claim; promote to conjecture at threshold k, to theorem at threshold 2k.

The pattern is **n-of-m voting** — common in distributed consensus (Paxos, Raft) and in ensemble ML. Bringing it to the LLM-output level gives you a self-checking pipeline.

Pattern 2: rule-based consistency check

If the LLM emits a fact that *contradicts* an established rule, demote (or reject):

```
axiom is_capital_unique(C) <=
   { City | is_capital(C, City) } has exactly 1 element

hypothesis is_capital("Germany", "Berlin")
hypothesis is_capital("Germany", "Frankfurt")   # wrong

# When the uniqueness rule fires, both candidates
# get tagged "Both" (paraconsistent contradiction).
# A reasoning rule then drops the lower-confidence
# one based on source agreement count.
```

Cascade has dozens of these consistency rules spread across `reasoning/rules/` directories. Together they form a *type system for facts* — catching errors that LLMs make confidently.

Pattern 3: external grounding (RAG-style)

If you can look up the fact in a trusted external source, promote:

```
# After verifying via Wikipedia API:
confirmed: wiki_lookup("OpenAI", "ceo")
if confirmed == "Sam Altman" then [
  # Bump LLM-grade fact to postulate (trusted external)
  insert("ceo_of", "OpenAI", "Sam Altman", "postulate")
]
```

This is the **retrieval-augmented generation (RAG)** pattern, applied at storage time rather than inference time. The LLM proposes; the trusted external source confirms; the KB stores the confirmed grounding.

The ladder in action

Putting it together, a single fact's grounding can walk up the ladder over time:

```
T=0:  LLM extracts → hypothesis is_capital("DE", "Berlin")
T=1:  Second LLM agrees → conjecture (multi-source rule fires)
T=2:  Wikipedia lookup → postulate (external grounding)
T=3:  Manual review → axiom (human-confirmed)
```

Each promotion is **reversible** with `forget()` and re-insertion at a lower grounding. The system behaves like a **monotonic-with-rollback** ratchet: ascends easily; descends only on explicit contradiction.

42.5 Belnap B4 for contradiction handling

LLMs disagree. Sometimes the *same* LLM disagrees with itself across queries. The honest response is to record the disagreement, not pick a winner prematurely.

Belnap B4 truth values (`T`, `F`, `Both`, `Neither`) make disagreement first-class:

```
hypothesis num_employees("OpenAI", 1500)
set_truth("num_employees", "OpenAI", 1500, "true")

hypothesis num_employees("OpenAI", 2500)
set_truth("num_employees", "OpenAI", 2500, "true")

# Both hypotheses are "true" individually; the KB
# has them coexist. A reasoning rule then sets
# the truth of *both* to Both, since the underlying
# question has conflicting evidence.

set_truth("num_employees", "OpenAI", 1500, "both")
set_truth("num_employees", "OpenAI", 2500, "both")

println(truth("num_employees", "OpenAI", 1500))
# → T⊥ᵇ  (Belnap "Both")
```

When a downstream rule queries `num_employees`, seeing `Both` means "the answer is disputed; escalate to a human." That's the **paraconsistent escape hatch** — the system doesn't crash on contradictory inputs, it surfaces them.

Production examples

In Cascade:

- **Source disagreement** — three news sources say Russia GDP is $1.8T; one says $2.0T. The KB stores both with `Both` and a metadata field pointing to the dispute.
- **Temporal staleness** — yesterday's "active CEO" fact and today's "active CEO" fact both have truth=T; the temporal rule sets the older to `Both` ("was true but no longer canonical").
- **Topic-area expertise** — one model is strong on finance, another on legal; for legal facts the finance model's output gets demoted to hypothesis-grade. **Differential trust** by domain.

The B4 vocabulary is the bridge between "neural output is messy" and "downstream code needs definite answers". The reasoning layer turns mess into a decision (with explicit confidence) rather than hiding the mess.

42.6 The defeasibility ladder

Putting grounding and B4 together gives you the full **defeasibility ladder**: a fact's "level of support" lives on two axes simultaneously.

Dimension	Levels
Grounding (epistemic)	axiom > postulate > theorem > conjecture > hypothesis > datum
Truth (semantic)	T, F, Both, Neither (Belnap B4)

A claim at `(axiom, T)` is the strongest possible — human-declared fact, no known contradiction. A claim at `(hypothesis, Both)` is the weakest non-zero — LLM-generated and contradicted by other sources.

Defeasible rules demote on conflict. A defeasible rule:

```
mortal(X) <~~ human(X)
```

means "humans are *typically* mortal; the rule's conclusion is *defeasible* — it can be overturned by a more specific rule." Adding `not mortal(zeus)` defeats `mortal(zeus)` even though `human(zeus)` holds.

In neuro-symbolic terms: LLM emits `human("Zeus")` (hypothesis); the rule fires and emits `mortal("Zeus")` (conjecture); a more specific rule says `not mortal(X) <= deity(X)` and Zeus's deity-fact (postulate, from a Wikipedia lookup) defeats the mortal-derivation.

Trace this carefully:

```
hypothesis human("Zeus")        # LLM said so
postulate  deity("Zeus")        # KB has it

# Two rules:
#   mortal(X) <~~ human(X)
#   not mortal(X) <= deity(X)

# The strict rule (deity → not mortal) wins over the
# defeasible (human → mortal). Result:
#   mortal("Zeus") truth = F (defeated)
```

The neuro-symbolic system doesn't crash on the contradiction; it resolves it via the **rule priority** (strict > defeasible).

This is the **load-bearing pattern** for production neuro-symbolic systems. The LLM proposes; the defeasible rules promote; strict rules override on conflict; final answers come with their full chain of reasoning attached.

42.7 Common failure modes

Six failure patterns. Recognize them; design around them.

(a) Hallucination

The LLM invents a fact that *sounds right* but isn't. "Berlin is the capital of Germany" (true). "Berlin is the capital of France" (false but plausible to a confused model).

Defense: verify against trusted source (RAG); demand multi-source agreement; flag claims with unusual entity combinations for human review.

(b) Confidence-calibration mismatch

The LLM says "I'm 90% confident" but is actually 50% accurate on that distribution. Or worse: says "definitely" for things it's flat-out wrong about.

Defense: don't trust LLM-emitted confidences directly. Use them as one signal among several; calibrate against held-out data; combine with external-source agreement.

(c) Prompt drift

Two semantically identical prompts produce different outputs. The same model on Monday vs. Tuesday gives different answers (due to sampling randomness, weight updates, or context).

Defense: identical prompts → record multiple responses → vote. Storing as hypothesis lets the vote happen offline.

(d) Training-data leak

The model's "fact" about a public figure is quoting a 2020 article it memorized verbatim. The fact is *real* in 2020 but stale in 2024.

Defense: explicit temporal tagging on every fact. Cascade tags every `truth_aggregate` event with a timestamp; queries can require "fresher than 30 days."

(e) Adversarial input

Someone crafts an article designed to make the LLM extract a specific false claim. Or worse, a sequence of articles meant to slowly poison the KB.

Defense: source reputation tracking. Every ingested article carries the source's identity; sources with declining accuracy get demoted in the trust hierarchy; the KB can be rolled back to a checkpoint (Ch.40 §40.6).

(f) Ontology drift

The LLM uses entity names slightly differently each time. "Sam Altman" vs "Samuel Altman" vs "S. Altman" — three names for one person.

Defense: entity normalization at ingest time. Cascade uses a `resolve_entities` step that maps incoming surface forms to canonical KB entities. A second-pass merge resolves residual aliases.

42.8 When neuro-symbolic wins

Six rough heuristics. Recognize them in your domain.

1. **The neural part operates on unstructured input** (text, images, sensor data) that wouldn't easily yield to a hand-coded parser.
2. **The symbolic part needs to be auditable**. Financial systems, legal compliance, medical diagnosis — anywhere the user can ask "why did you say that?"
3. **Domain knowledge is partial.** You have some rules but not all the facts. The neural model discovers facts; the rules constrain and validate them.
4. **Contradictions are real and important.** Multiple sources disagree often enough that you can't pick one and ignore others.
5. **Updates are continuous.** New facts arrive daily; the KB must update without manual intervention; the model must learn the new facts without retraining.
6. **Cost-of-error is asymmetric.** Wrong answers are much worse than slow answers — you can afford the symbolic verification step.

When *not* to use neuro-symbolic:

- Pure perception tasks (image classification, ASR) where there's no symbolic layer to validate.
- Pure reasoning tasks (theorem proving, route planning) where the input is already structured.
- High-throughput, latency-critical applications where the symbolic verification step is too slow.
- Domains where the rules don't exist or are contested — there's no "trusted symbolic layer" to defer to.

Cascade is squarely in the first list. So is most of medical diagnosis, legal research, financial fraud detection, scientific literature mining, and intelligence analysis. The pattern isn't universal; it's powerful where it fits.

42.9 Looking ahead — closing Volume II

You've now traversed:

- **Vol I (Ch.1-22)**: HtDP-tradition CS1 — design recipe, recursion, MVL, logic programming, meta-circular evaluator.
- **Vol II Part VII (Ch.23-34)**: CS2 data structures + Axioma's signature features — lists, hash tables, trees, graphs, conceptual spaces, Gödelization, Lojban/NSM, Schank, MCP.
- **Vol II Part VIII (Ch.35-37)**: complexity analysis — Big-O, sorting, graph algorithms.
- **Vol II Part IX (Ch.38-40)**: stateful systems — state machines, event loops, durable reactors.
- **Vol II Part XI (Ch.41-42)**: the symbolic-
  - neural arc — stack programming, neuro-symbolic patterns.

That's the full CS1+CS2+KR curriculum at full depth.

Volume III picks up the thread with Cascade itself:

- Ch.43 — Cascade architecture (the system view)
- Ch.44 — KG schema (the data model)

- Ch.45 — Ingest pipeline (the neuro-symbolic pipeline of §42.2, at production scale)
- Ch.46 — Truth lifecycle (the truth-aggregation rules of §42.4-§42.5)
- Ch.47-48 — Reasoning algorithms + Axioma rule firing
- Ch.49-50 — Writing .ax rules + agent swarm
- Ch.51 — Reactor internals (the production version of Ch.39-40)
- Ch.52 — Operating Cascade

If you came in cold, you're now well-prepared for Volume III. If you came for the CS1-CS2 curriculum, you've got it. If you came for neuro-symbolic AI, you've got the language, the patterns, and a worked production system to study.

The chapter you choose next depends on where you want to go.

Exercises

42.1. *Three-source agreement.* Write a rule `confirmed(C, P) <= count(hypothesis sources for (C, P)) >= 3`. Insert three hypothesis claims for the same fact and verify they aggregate to a confirmed conclusion. Then add a fourth claim *contradicting* the first three and verify the system handles it via Belnap `Both`.

42.2. *Defeasible-by-specificity.* Write rules:

- `flies(X) <~~ bird(X)` (defeasible)
- `not flies(X) <= penguin(X)` (strict)

Add `hypothesis bird("tweety")` and `postulate penguin("tweety")`. Query `flies("tweety")`. What grounding do you get? What's the truth value?

42.3. *Failure-mode classification.* For each of these scenarios, identify which failure mode from §42.7 applies:

- (a) LLM says "Paris is in Italy."
- (b) LLM says "I'm 100% sure Pluto is a planet."
- (c) Same prompt yesterday: Y. Today: N.
- (d) LLM's answer about a 2024 election uses 2020-era candidate names.
- (e) A poisoned RSS feed gets ingested.
- (f) The LLM emits "Sam" today and "Samuel" tomorrow for the same person.

42.4. *Trust calibration.* Design a trust-tracking scheme: every LLM-generated claim has a `source_model` tag (e.g., "gpt-4", "claude-3"). Sources with high false-positive rate get demoted to grounding "datum." Write the demotion rule.

42.5. *RAG-style promotion.* Sketch (in pseudocode if needed) a function `verify_via_wiki(rel, args)` that simulates a Wikipedia lookup. If the lookup confirms the fact, promote its grounding to `postulate`. If it contradicts, demote to `datum` and emit a warning.

42.6. *Open-ended.* Pick a neuro-symbolic system you've heard of (DeepProbLog, IBM Watson, Cyc-era systems, Cascade itself, etc.). Read its docs or papers for 30 minutes. Write 200 words answering: what's the neural part? What's the symbolic part? Where do they meet? Is the meeting point at storage-time,

query-time, both?

Vocabulary

- **Neuro-symbolic**: a system combining neural and symbolic AI components.
- **Hallucination**: an LLM-generated fact that's fluently stated but factually wrong.
- **Grounding ladder**: the ordered set (datum → hypothesis → conjecture → theorem → postulate → axiom).
- **Defeasibility**: a rule's conclusion can be overridden by other rules (especially strict ones, or more specific defeasible ones).
- **Defeasible rule (<~~)**: produces conjecture-grade conclusions.
- **Strict rule (<=)**: produces theorem-grade conclusions; overrides defeasible ones on conflict.
- **B4 / Belnap-4**: four-valued truth: T, F, Both, Neither. Allows contradiction without explosion.
- **Multi-source agreement**: n-of-m voting on LLM-generated claims; the symbolic-layer defense against single-source error.
- **Entity normalization**: mapping surface forms ("Sam", "Samuel A.") to canonical KB entities.
- **RAG** (Retrieval-Augmented Generation): using retrieval from a trusted source to ground or verify model outputs.
- **Ontology drift**: gradual divergence between the model's implicit ontology and the KB's explicit one.

Notes for the curious

The history of "neuro-symbolic"

The term goes back to the 1990s (Towell & Shavlik's KBANN; D'Avila Garcez's neural-symbolic learning), but the practice is older. Early expert systems (MYCIN, INTERNIST) were hand-coded symbolic; they failed because nobody could maintain the rules.

The early-2010s neural renaissance went the other way: pure neural, no symbolic structure. Failed on different things: hallucination, audit trails, sample efficiency.

Production systems today (~2024-2026) increasingly combine both. Cascade is one instance; IBM's Watson Discovery is another; DeepMind's various "Mathematics" projects (AlphaProof, MATH) use a neural model to propose lemmas and a symbolic theorem prover to verify them.

The combination is more powerful than either alone because the failure modes are *different*: neural models fail at precision, symbolic systems fail at coverage. Combine them and you get high precision *and* high coverage — at the cost of infrastructure complexity.

LLM as a "feature extractor"

A useful mental model: treat the LLM as a **feature extractor** rather than a reasoner. Its job is to turn unstructured input into structured candidate facts. The reasoning happens in the symbolic layer.

This is the inverse of the 1980s symbolic AI philosophy (where the reasoning happened in the rules and the inputs were already structured) and the 2010s neural philosophy (where the reasoning happened inside the network).

In practice: prompt the LLM to **extract** structured facts in JSON-like form. Parse the JSON. Ingest into the KB at hypothesis-grade. Reasoning fires on the KB, not on the LLM. The LLM never sees the reasoning result; the user sees both via the symbolic layer's audit trail.

Why grounding matters more than confidence

You might think: "Just store every LLM fact with a confidence score; threshold to decide what to trust." The grounding ladder is a richer model.

A confidence score is **scalar** — a single number that doesn't distinguish between "uncertain from limited data" and "uncertain due to contradiction." The grounding ladder is **categorical**: axiom is qualitatively different from postulate (human authority vs. theoretical derivation), and conjecture is qualitatively different from hypothesis (rule-derived vs. externally-asserted).

Categorical levels also compose better. "Combine two postulates via a strict rule → theorem." "Combine two conjectures via a defeasible rule → conjecture (lower of inputs)." The arithmetic is clean. Trying to do the same with continuous confidence scores requires arbitrary calibration choices.

Cascade uses both — categorical grounding for ladder position, scalar confidence within each category for fine-grained ranking. Best of both, at the cost of two-axis bookkeeping.

Where this is heading

Two trends to watch:

1. **Tool-using LLMs.** LLMs that can call external functions (Wikipedia, calculator, SQL queries, theorem provers) blur the line between neural and symbolic. The "neural" part becomes a planner that delegates to symbolic tools.
2. **Differentiable symbolic systems.** Research like DeepProbLog and SOAR-2024 makes the symbolic rule-base *learnable* via gradient descent. The neural and symbolic layers fuse into one trainable structure.

Both push the boundary between "neural" and "symbolic" inward. The patterns of this chapter — hypothesis-grade storage, multi-source verification, defeasible rules, B4 contradictions — will keep working in the hybrid regimes.

You've now reached the end of Volume II. Volume III picks up where this chapter left off, with Cascade as the worked production case study.

Solutions to selected exercises: Chapter 42 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch42_solutions.md.)*

Volume III — AxiomaCascade

Volume III drills into AxiomaCascade — the geopolitical- intelligence engine that uses Axioma as one of its reasoning substrates. Two audiences at once: Cascade users learning to extend the system, and Axioma developers curious how their language composes into a production tool.

The opening chapter (Ch.43) introduces the architecture and walks a 5-minute end-to-end example. Subsequent Volume III chapters drill into the knowledge graph (Ch.43–45), the reasoning layer where Axioma rules fire (Ch.46–48), and the agent swarm and operations (Ch.49–52).

Chapter 43 · Cascade Architecture, in One Diagram

> **What this chapter is.** Your first encounter with **Axioma Cascade** — the geopolitical-intelligence engine that uses Axioma as one of its reasoning substrates. The chapter does two things at once: shows the *system-level picture* so you can read everything else in Volume III, and walks a *user-facing example* against a real, populated Cascade database so you can trace cascades by the end of the chapter. Both audiences get something useful: Cascade users get reproducible commands, Axioma developers get the architecture map of *where their language lives inside a production system*.

This is the opening chapter of Volume III. From here on, the book is about *Cascade-built-on-Axioma*, not Axioma in isolation.

43.1 What Cascade is

In one sentence:

> **Axioma Cascade ingests news articles, extracts entities and relations into a persistent knowledge graph, and uses NetworkX and Axioma to discover the multi-tier *cascade* of consequences a single event triggers.**

"Cascade" is the load-bearing word. Most geopolitical analysis stops at Tier 1 — the headline event ("oil spiked"). Cascade traces the chain three or four levels deeper:

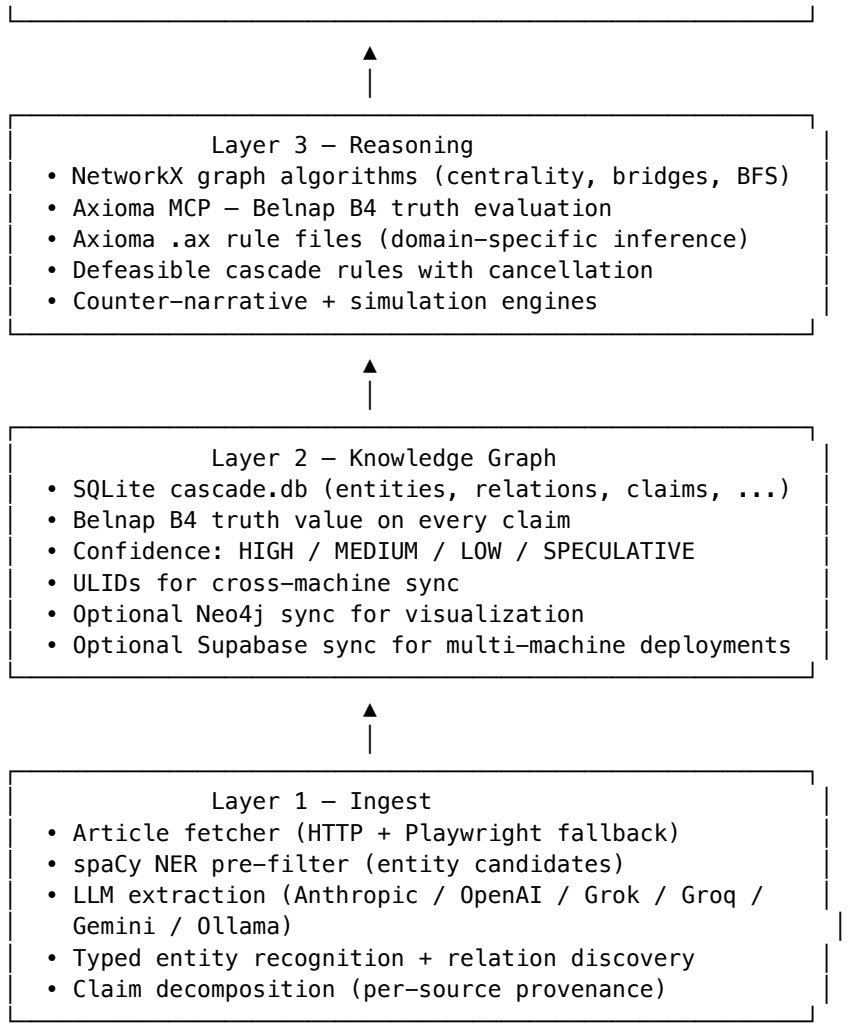
> insurance withdrawn → ships stranded → LNG halted → fertilizer costs spike → food prices rise in 3-6 months → social instability in vulnerable regions

Tier 1 → Tier 2 → Tier 3 → Tier 4. Each arrow is a *relation* in the knowledge graph; each hop is a query that walks the graph and asks "what's downstream of this?"

That's the product. Now: how is it built?

43.2 The four-layer architecture

```
┌──────────────────────────────────────────────────────────┐
│                  Layer 4 — Output                        │
│  • CLI queries (JSON / Rich tables / JSONL / ID-only)    │
│  • HTML / GraphML / DOT exports                          │
│  • Substack daily briefings                              │
│  • Subscription dashboards (web)                         │
```



Read bottom-to-top:

1. **Ingest** — text comes in (article URLs, raw text, pasted clipboards). spaCy runs a fast local NER pass to find entity candidates; the LLM follows with structured extraction (entity types, relation types, confidence levels, evidence quotes).
2. **Knowledge graph** — every extracted entity and relation is stored in a SQLite database with a Belnap B4 truth value attached. Same fact base can be **synchronized** to Neo4j for graph visualization or to Supabase for multi-machine deployments.
3. **Reasoning** — two engines work on the graph: NetworkX for *structural* analysis (shortest paths, centrality, community detection) and Axioma for *logical* inference (truth propagation, defeasible rules, formal cascade chains).
4. **Output** — CLI commands, HTML dashboards, GraphML exports, daily briefings.

Each layer is *substitutable* — that's the ADT discipline from Volume II Ch.24 applied at architectural scale. Want a PostgreSQL backend instead of SQLite? Reimplement Layer 2's interface. Want a different LLM provider? Layer 1's fetcher/extractor abstracts six already.

43.3 Where Axioma lives

This is the chapter's deepest insight. **Axioma is not the whole reasoning layer — it's the *logical inference* slice of it.** Cascade uses *both* Axioma and NetworkX, each where it shines:

Reasoning task	Engine	Why
"What's the shortest path between two entities?"	NetworkX	Graph theory; well-studied; fast C implementation
"Which entities are bridges in the network?"	NetworkX	Same
"Rank entities by betweenness centrality"	NetworkX	Same
"Find community clusters"	NetworkX (Louvain)	Same
"Is this claim true / false / both / unknown?"	Axioma B4	Multi-valued logic; paraconsistent
"Apply the *contradicts* rule to all known claims"	Axioma rules	Defeasible logic; cancellation
"Propagate truth through derivation chains"	Axioma rules + B4	Provenance + bilattice meet
"Run a custom domain-specific cascade rule"	Axioma `.ax` file	User-extensible rule language
"Query the shared cascade.db from inside Axioma code"	`cascade axioma exec ...`	Direct relational queries

The boundary is principled: **structure goes to NetworkX, semantics goes to Axioma.** Counting hops and finding shortest paths is graph theory; deciding whether two competing claims are both true, neither true, or contradictory is logic.

Cascade talks to Axioma three different ways:

1. **`cascade/core/axiomalang.py`** — the Python-side bridge that calls Axioma MCP tools (`run_file`, `evaluate_b4`, `query_kb`, `forward_chain`).
2. **`.ax` rule files** auto-discovered from two directories: `.cascade/rules/axiomalang/*.ax` (project-level) and `~/.cascade/rules/axiomalang/*.ax` (global user rules). Each file is executed via the Axioma MCP `run_file` tool at startup.
3. **`cascade axioma exec ...` / `cascade axioma run ...`** — the user-facing CLI subcommands. These open a *persistent* MCP session backed by the shared `cascade.db`, so any facts you assert in your Axioma session land in the same SQLite tables Cascade reads.

The persistent MCP connection means **every Axioma feature you've learned in Volumes I and II is available in production**. The relational queries from Chapter 21 work on real-world news data. The Belnap B4 logic from Chapter 20 adjudicates contradictory reports from competing sources. The defeasible rules from Chapter 21 encode domain expertise. The meta-circular evaluator from Chapter 22 isn't just a teaching toy — Cascade uses the same `parse` and `ast_eval` builtins to translate user-typed queries into Axioma rules.

43.4 A 5-minute walkthrough against a real database

The fastest way to *see* Cascade work end-to-end. (All commands here have been verified against a populated Cascade instance with ~3950 entities and ~3800 relations.)

```
# 1. Where is Cascade?
cd /path/to/axiomacascade
uv run cascade stats
```

Output (yours will differ depending on what you've ingested):

```
{
  "entities": 3954,
  "relations": 3809,
  "claims": 8415,
  "sources": 329,
  "pins": 6,
  "rules": 85,
  "entity_types": 10,
  "relation_types": 9
}
```

```
# 2. Sample some entities (filter by STATE)
uv run cascade entity list --type STATE --json | head -20
```

Each entity is a record with `id`, `type`, `name`, `prolog_atom`, `aliases`, `properties`, `first_seen`, `last_seen`, `domain`, `ulid`. The `prolog_atom` field is the lowercased canonical form used when the entity shows up in `.ax` rules.

```
# 3. Trace a 2-tier cascade from "Iran"
uv run cascade discover cascade "Iran" --depth 2 --json | head -30
```

You'll get a `tiers` array. Tier 0 is the origin (Iran); Tier 1 lists direct impacts (e.g. *crude oil* via SUPPLY, *global energy crisis* via CAUSAL); Tier 2 lists second-order impacts. The default relation-type filter is `{CAUSAL, DEPENDENCY, INFLUENCE, SUPPLY, CONTROL}` — the `CAUSAL_TYPES` set in `discovery.py`.

```
# 4. Find hidden connectors (betweenness centrality)
uv run cascade discover centrality --metric betweenness --top 10
```

The entities that bridge different domains. Often these are *commodities* (oil, semiconductors) or *infrastructure* (straits, ports) — non-obvious connectors that link seemingly unrelated regions.

```
# 5. Drive Axioma directly against the shared KB
uv run cascade axioma exec 'println("Hello from Axioma!")'
uv run cascade axioma query stats
```

```
uv run cascade axioma query list-relations
```

The third command returns the relation names that exist in `cascade.db` as Axioma-queryable predicates — including the eight uppercase `CAUSAL`-style relations and any lowercase domain-specific ones (e.g. `commodity`) you've added from `.ax` scripts.

The CLI is **Unix-pipe-friendly** — every query command emits JSON when piped or when `--json` is passed. That lets you compose:

```
uv run cascade discover cascade "Iran" --depth 3 --json \
  | jq '.tiers[2].entities | .[] | .entity_name' \
  | sort -u
```

That pipeline: trace cascade → take Tier-3 entity names → deduplicate. The point is that Cascade is a *toolkit*, not a monolithic dashboard; the architectural simplicity of "every command speaks JSON" makes it scriptable.

43.5 Entity types, relation types, and what they mean

The default schema (probed live from `cascade entity list --help`):

Entity types (10):

Type	Description
`STATE`	Sovereign nations, governments
`ORG`	Companies, NGOs, intergov bodies, military units
`ACTOR`	Named individuals (leaders, officials, commanders)
`INFRA`	Infrastructure (ports, pipelines, straits, bases)
`COMMODITY`	Physical goods (oil, gas, grain, weapons)
`MARKET`	Financial markets, indices, trading venues
`SECTOR`	Economic sectors (energy, shipping, agriculture)
`EVENT`	Named events (wars, treaties, elections, crises)
`POLICY`	Sanctions, treaties, legislation, executive orders
`TECHNOLOGY`	Specific tech (drones, missiles, AI, cyber)

Relation types (8 canonical, extensible):

Type	Meaning
`CAUSAL`	A directly causes B
`DEPENDENCY`	A depends on B
`TEMPORAL`	A precedes B in time
`ADVERSARIAL`	A and B are in conflict
`ALLIANCE`	A and B are aligned
`SUPPLY`	A supplies B

Type	Meaning
`CONTROL`	A controls B
`INFLUENCE`	A influences B (softer than CONTROL)

Both lists are **extensible** via `cascade types add`. The defaults are deliberately small — Cascade is opinionated about what kinds of things matter for cascade tracing.

The five relation types in `CAUSAL_TYPES` — `{CAUSAL, DEPENDENCY, INFLUENCE, SUPPLY, CONTROL}` — are the ones the default `discover cascade` traversal follows. The other three (TEMPORAL, ADVERSARIAL, ALLIANCE) are visible in the graph but don't propagate *causally* by default. That's the system's content claim about which edges count as "downstream impact."

43.6 The Belnap B4 truth slot

Every fact in Cascade's database carries one of four truth values:

Value	Meaning
`TRUE`	confirmed by at least one source
`FALSE`	contradicted by sources
`BOTH`	confirmed by some sources, contradicted by others (the *paraconsistent* case)
`UNKNOWN`	no information either way

Plus a confidence band:

Band	Range
`HIGH`	> 80%
`MEDIUM`	50–80%
`LOW`	20–50%
`SPECULATIVE`	< 20%

When two articles report contradictory facts about the same entity-relation, Cascade *doesn't pick a winner*. It marks the relation as `BOTH` and records both sources. Downstream queries can choose: filter out `BOTH`-flagged claims for conservative analysis, or include them with explicit disagreement flags.

This is **Volume I Chapter 20's Belnap B4 logic in production use** — exactly the value space the chapter taught, now adjudicating real-world contradictory reporting.

43.7 Cascade discovery internals — the BFS walk

`cascade discover cascade --depth N` is the canonical demonstration of where structure (NetworkX) and semantics (Axioma) meet.

The algorithm, in plain English:

1. **Build a directed graph** from the cascade.db `entities` and `relations` tables. Each entity becomes a node; each relation becomes a typed edge.
2. **Initialize** Tier 0 with just the origin entity.
3. **For each tier from 1 to N:**
   - For each entity in the previous tier,
   - For each outgoing edge whose type is in `CAUSAL_TYPES`,
   - If the target entity hasn't been visited AND the relation's confidence meets the threshold,
   - Mark visited, append to current tier with metadata (relation type, source entity, confidence, path).
4. **Stop** when a tier is empty or N reached.

This is **breadth-first search by level** — distinguishing tiers means "one hop away" versus "two hops away," exactly the metric that makes cascade analysis useful for journalism (Tier 1 is what made the news; Tier 3 is what nobody has connected yet). The `find_cascade_chain` function in `cascade/core/discovery.py:71` is the ~70-line implementation.

Where would Axioma get involved? Three places that *haven't* been wired in by default but are architecturally available:

- **B4 truth propagation along the chain.** A Tier-3 result whose chain passes through a `BOTH`-truth-value edge inherits paraconsistency. Axioma's `evaluate_b4` is the right engine to compute that.
- **Defeasible cancellation.** A Tier-2 finding that's in a `cancels` rule with an existing claim should be marked suppressed but kept visible.
- **Custom domain rules.** A user-written `.ax` rule like `commodity_price_cascade.ax` (shipped with Cascade today) encodes domain expertise that *can't* be inferred from the graph alone.

These are the kinds of integrations Volume III's later chapters drill into.

43.8 The shared cascade.db — Axioma and Cascade together

The most direct way to see Axioma and Cascade interoperate is the **shared SQLite database**. Both processes read and write the same `cascade.db`:

```
# From the Python side (Cascade)
uv run cascade entity list --type STATE --json | head -3

# From the Axioma side (using the cascade axioma bridge)
uv run cascade axioma exec 'println(stats())'
```

Both commands hit the same SQLite tables. When a Cascade ingestion adds a new fact, it's immediately visible to Axioma rules. When an Axioma rule asserts a new fact (with `axiom/persist`), it's immediately visible to Cascade's NetworkX builds.

This is the *literal* substrate for Volume III's later chapters on the agent swarm. The collector agent writes via Cascade APIs; the analyst agent reads via Axioma queries; the producer agent reads consolidated results via Cascade's CLI. Nothing serialized over the wire — they all share `cascade.db`.

43.9 Honest limits and what's not in this chapter

Three areas where the chapter glosses over real complexity:

1. **The ingest pipeline is more than two boxes.** The §43.2 diagram shows Layer 1 as "fetcher + NER + LLM extractor." In practice it includes:
   - URL canonicalization and deduplication
   - HTTP retry / Playwright fallback for SPA pages
   - Article-vs-listing classification
   - Per-source extraction prompts (the LLM gets different prompts for Reuters vs Substack)
   - Claim decomposition (one article → N propositions, each with its own provenance)
   - Confidence calibration against the source's historical accuracy

   Volume III later chapters will drill in.
2. **Sync to Supabase is real.** When `SUPABASE_URL` and `SUPABASE_KEY` are configured, Cascade can push/pull the knowledge graph to a hosted Postgres instance, enabling multi-machine deployments. The `WARNING: Supabase credentials not configured` you see is the gate.
3. **The MCP server is part of Cascade's CLI.** Running `cascade mcp serve` exposes the same KB to any MCP- capable client (Claude Desktop, Claude Code, etc.) — not just the Cascade CLI. The MCP layer is documented in Ch.34 of this book.

43.10 Reading the rest of Volume III

Future Volume III chapters (Ch.44+, numbering still being finalized) will drill into specific subsystems:

- **Knowledge-graph schema** — the SQLite tables (sources, entities, relations, claims, rules, events), their indexes, the ULID/sync columns.
- **The ingest pipeline** — NER, LLM prompting, error recovery, claim decomposition.
- **Truth propagation algorithms** — bilattice meet, defeasible cancellation, provenance walks.
- **Cascade discovery internals** — full `find_cascade_chain` pseudocode, BFS variants for TEMPORAL and ADVERSARIAL walks.
- **The agent swarm** — collector, analyst, producer, quality. Each agent's tools and write-paths.
- **Writing custom `.ax` rules for Cascade** — the shipped rules (`commodity_price_cascade.ax`, `monetary_tightening_currency_appreciation.ax`, `oil_gold_positive_correlation.ax`, `capital_flow_attraction_rule.ax`) as templates.
- **Operating Cascade at scale** — subscription tiers, monitoring, scheduled tasks, dashboard deployment.

Each chapter will lean on this chapter's architecture map for orientation. When a later chapter says *"the collector agent stores facts via the Layer-2 interface and fires the Axioma rule engine at Layer 3,"* you'll know exactly what it means.

43.11 Exercises

These exercises are **runnable** against a working Cascade install. Most require nothing more than the Cascade CLI and a configured LLM provider. Exercises 43.1, 43.4, 43.5 work *without* further API calls, using whatever data is already in your database. The `uv run cascade ...` prefix assumes you're in the

axiomacascade repo root.

43.1 — Database round-trip

Run `uv run cascade stats` once. Note the entity / relation / claim counts. Run it a *second* time and compare — the numbers should be identical (no ingestion happened between the two calls).

Then run `uv run cascade entity list --type STATE --json | head -50`. Look at the JSON shape — fields, types, ULIDs, properties. This is the schema you'll work with for the rest of Volume III.

43.2 — Ingest a known article

Pick a recent news article you've read. Run `uv run cascade ingest <url>`. After it completes, run `uv run cascade entity list --type EVENT --json` and *eyeball-check*: do the extracted entities match what you'd have extracted yourself?

Common things to spot-check:

- Did spaCy + LLM correctly identify the major nations / orgs?
- Did it assign reasonable entity types?
- Did it miss anything important?
- Did it hallucinate anything that wasn't in the article?

Write down 3 specific extraction successes and 3 specific failures. This is the foundation of *evaluating* Cascade's analytical quality.

43.3 — Trace a cascade

Pick a real entity in your database (e.g. "Iran", "Crude Oil", "Strait of Hormuz") and run:

```
uv run cascade discover cascade "<name>" --depth 3 --json
```

Examine the `tiers` array. For each Tier-2 and Tier-3 entry, ask:

- Is this chain *causally plausible*?
- Is the implied connection *novel* to you (Cascade showed you something you didn't already know), or is it *obvious* (you could have generated it yourself)?
- Where does the chain break down (an entity appears in the cascade but the relation type seems wrong)?

A good cascade should have *some* novel Tier-3 results. Pure-obvious cascades suggest the underlying article corpus is too narrow.

43.4 — Bridge entities and centrality

Run two structural-analysis commands:

```
uv run cascade discover bridges --json | head -30
uv run cascade discover centrality --metric betweenness --top 10
```

The first returns entities whose removal would fragment the graph. The second returns entities that sit on many shortest paths between other pairs.

Look at both lists. Are the bridges *also* high- centrality? Often yes, but not always — some bridges are low-traffic chokepoints. Pick one entity that appears in both lists, then run `cascade discover cascade "<name>" --depth 2` against it. Does the cascade chain explain why this entity is structurally important?

43.5 — The Cascade-Axioma bridge

Use the `cascade axioma` subcommand to interact with the shared KB from inside Axioma:

```
# What relations does Cascade have stored?
uv run cascade axioma query list-relations

# What facts are stored for the commodity relation?
uv run cascade axioma query list-facts --relation commodity

# Execute Axioma code that sees those facts
uv run cascade axioma exec 'println("Stats:"); println(stats())'
```

Then *add* a new fact from the Axioma side:

```
uv run cascade axioma exec 'axiom/persist test_fact("hello", "world")'
```

And verify it appears:

```
uv run cascade axioma query list-facts --relation test_fact
```

Reflect: how is this fact persisted (which table?), and can Cascade's CLI commands now see it?

43.6 (open) — Where would *you* put Axioma?

You've now seen one architecture where Axioma is the *logical-reasoning slice* of a larger system. Sketch a *different* system you'd build that uses Axioma as one of its components:

- What's the system?
- Where does Axioma sit (which layer)?
- What kinds of queries does Axioma answer that other components can't?
- What kinds of work does Axioma *not* do (what other engines does the system need)?

This is open-ended. The point is to get you thinking about Axioma as a *component* — a reasoning engine you plug into larger systems — not as a self-contained programming environment. Once you internalize this, you start seeing opportunities for Axioma everywhere.

Solutions to selected exercises: Chapter 43 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch43_solutions.md.)*

43.12 Reflection — Cascade as the Volume III invariant

The system in this chapter is enormous. The pipeline has half a dozen technologies in it (Python, SQLite, NetworkX, spaCy, LLMs, Axioma, optionally Neo4j, optionally Playwright, optionally Supabase). Three different ways to deploy it (local CLI, scheduled agents, subscription dashboard). Ten entity types, eight relation types, four truth values, four confidence bands.

What keeps the complexity manageable is the **architecture map** from §43.2. Every later chapter slots somewhere on that map. *Knowledge-graph schema* lives at Layer 2. *Cascade discovery* lives at Layer 3. *Daily briefing generation* lives at Layer 4. The map is your "you are here" sign for the rest of Volume III.

The map is also Cascade's deepest design principle: **reasoning engines are pluggable**. Today the rule engine is Axioma; tomorrow it might be something else; nothing forces clients to know. The same separation Volume II Ch.24 taught for stacks and queues — *interface, not representation* — is what makes Cascade a *system that can evolve* rather than a single monolithic codebase.

The next Volume III chapter will zoom into Layer 2 (the knowledge-graph schema). You'll meet the core tables, see how Belnap B4 truth values are stored alongside facts, and learn the SQLite indexes that make the whole thing fast.

> *"All architecture is an attempt to enable some changes while preventing others."* — paraphrased from Ralph Johnson. Cascade's architecture enables: changing reasoning engines, changing storage backends, changing LLM providers, changing output formats. It prevents: tangling those four concerns into the same code path.

Chapter 44 · The Knowledge Graph Schema

> **What this chapter is.** A deep dive into Layer 2 of Cascade's architecture — the **SQLite schema** that backs every entity, relation, claim, and rule. Ch.43 showed you the architecture map; this chapter zooms in on the *map's most heavily-trafficked layer*. You'll meet the four core tables, the three support tables, the metadata tables, the indexes that make queries fast, and the sync-outbox machinery that lets multiple machines share a knowledge graph. By the end you'll read `cascade.db` fluently with raw SQL — useful when the CLI doesn't expose the query shape you need.

This chapter is the first of Part XIII (Cascade Knowledge Graph). Subsequent Part XIII chapters cover the ingest pipeline (Ch.45) and the truth-value lifecycle (Ch.46).

44.1 Why SQLite

Cascade stores its knowledge graph in **SQLite**, not Neo4j (the canonical graph database) or PostgreSQL (the canonical relational database). Three reasons:

1. **Embedded.** SQLite ships as a library, not a server. A Cascade install is *one Python process plus one file*. No daemon to keep running, no port to bind, no auth to configure. The whole graph lives at `data/cascade.db`.
2. **WAL mode + concurrent readers.** SQLite's Write-Ahead-Logging mode lets the analyst agent read while the collector agent writes. Production throughput is plenty for a single-machine analyst workflow.
3. **SQL is the lingua franca.** Cascade's CLI knows SQL, NetworkX knows SQL (via `pandas.read_sql`), Axioma reads `cascade.db` directly through its SQLite-backed KB layer. *No translation layer.*

The tradeoff is that some graph queries (longest path, triangle counting) are slower than they'd be on Neo4j. Cascade pays for that by loading the relations into a NetworkX DiGraph in memory when graph-algorithmic work is needed — see Ch.43 §43.7. **Store relational, compute graph-algorithmic.**

A Neo4j sync mirror is available for visualization and ad-hoc Cypher queries, but the *source of truth* is always SQLite.

44.2 The seven tables you'll touch most

Cascade's database has 20 tables, but seven do 95% of the work:

Table	Purpose	Rows in typical install
`entities`	Nodes in the knowledge graph	1k–10k+

Table	Purpose	Rows in typical install
`relations`	Edges between entities	1k–10k+
`claims`	Per-source propositions (provenance)	5k–50k+
`sources`	Where the data came from	100s–1000s
`rules`	Stored reasoning rules	50–500
`entity_types`	Canonical entity-type list	10
`relation_types`	Canonical relation-type list	8–20

The remaining 13 tables (`change_log`, `truth_history`, `pins`, `monitors`, `notifications`, `predictions`, `reminders`, `saved_searches`, `path_analyses`, `portfolios`, `sync_outbox`, `sync_meta`, plus `sqlite_sequence`) support specific subsystems — event-loop reactivity, audit history, user preferences, multi-machine sync. We'll touch them in §44.6 and §44.7.

44.3 The four core tables

`entities` — nodes in the graph

```
CREATE TABLE entities (
    id           INTEGER PRIMARY KEY AUTOINCREMENT,
    type         TEXT NOT NULL,
    name         TEXT NOT NULL,
    prolog_atom  TEXT,
    aliases      TEXT NOT NULL DEFAULT '[]',   -- JSON array
    properties   TEXT NOT NULL DEFAULT '{}',   -- JSON object
    first_seen   TEXT NOT NULL DEFAULT (datetime('now')),
    last_seen    TEXT NOT NULL DEFAULT (datetime('now')),
    source_id    INTEGER REFERENCES sources(id),
    ulid         TEXT,
    updated_at   TEXT NOT NULL DEFAULT '',
    machine_id   TEXT NOT NULL DEFAULT '',
    domain       TEXT
);
CREATE INDEX        idx_entities_type    ON entities(type);
CREATE INDEX        idx_entities_name    ON entities(name);
CREATE UNIQUE INDEX idx_entities_ulid    ON entities(ulid);
CREATE INDEX        idx_entities_prolog  ON entities(prolog_atom);
```

Three fields earn their own discussion:

- **`prolog_atom`** — the lowercased, snake-cased canonical form used when this entity appears in an Axioma `.ax` rule. "Strait of Hormuz" becomes `strait_of_hormuz`; "U.S. Treasury" becomes `us_treasury`. This bridges the *display name* humans read and the *atom symbol* Axioma's relation queries unify against.
- **`aliases`** — a JSON array of alternate surface forms ("USA", "United States", "U.S."). The entity-resolution step uses this to merge LLM extractions that name the same entity differently.

- **`properties`** — a JSON object for type-specific attributes (`{"sentiment": "NEUTRAL"}`, `{"role": "gas and oil producer"}`). A *type*-typed bag — schema-flexible.

Sample row from a live database:

```
SELECT id, type, name, prolog_atom, domain
FROM entities WHERE id = 1;

-- id=1, type=STATE, name="Iran", prolog_atom="iran", domain="political"
```

`relations` — edges between entities

```
CREATE TABLE relations (
    id                INTEGER PRIMARY KEY AUTOINCREMENT,
    source_entity_id  INTEGER NOT NULL REFERENCES entities(id),
    target_entity_id  INTEGER NOT NULL REFERENCES entities(id),
    type              TEXT NOT NULL,
    prolog_pred       TEXT,
    confidence        TEXT NOT NULL DEFAULT 'MEDIUM',
    properties        TEXT NOT NULL DEFAULT '{}',
    evidence          TEXT NOT NULL DEFAULT '[]',
    claim_ids         TEXT NOT NULL DEFAULT '[]',
    timestamp         TEXT NOT NULL DEFAULT (datetime('now')),
    source_id         INTEGER REFERENCES sources(id),
    ulid              TEXT,
    updated_at        TEXT NOT NULL DEFAULT '',
    machine_id        TEXT NOT NULL DEFAULT ''
);
CREATE INDEX        idx_relations_source ON relations(source_entity_id);
CREATE INDEX        idx_relations_target ON relations(target_entity_id);
CREATE INDEX        idx_relations_type   ON relations(type);
CREATE UNIQUE INDEX idx_relations_ulid   ON relations(ulid);
```

Every relation is **directed**. `(source_entity_id, target_entity_id, type)` together identify the edge. The `type` is constrained to a value in `relation_types` (see §44.4).

Key per-relation fields:

- **`prolog_pred`** — when populated, the `causal(iran, crude_oil)`-style predicate symbol used by Axioma rules. Computed from `(type.lower(), source.prolog_atom, target.prolog_atom)`.
- **`confidence`** — one of `HIGH` / `MEDIUM` / `LOW` / `SPECULATIVE`. Used by `discover cascade --min-confidence` to filter cascade traversal.
- **`claim_ids`** — JSON array of claim IDs that *support* this relation. The provenance link from edge to evidence.
- **`evidence`** — JSON array of quoted-snippet evidence strings (often a subset of the cited claims' texts).

Sample query — relations *from* Iran with HIGH confidence:

```
SELECT r.id, r.type, e2.name AS target, r.confidence
FROM relations r
JOIN entities e2 ON r.target_entity_id = e2.id
WHERE r.source_entity_id = 1 AND r.confidence = 'HIGH'
LIMIT 5;

-- 1   SUPPLY       crude oil               HIGH
-- 42  ADVERSARIAL  Strait of Hormuz        HIGH
-- 43  ADVERSARIAL  United States           HIGH
-- 52  CONTROL      Ayatollah Ali Khamenei  HIGH
-- 55  ADVERSARIAL  Dubai                   HIGH
```

That's five different "Iran is structurally adjacent to…" facts, each typed and confidence-banded.

`claims` — per-source propositions

```
CREATE TABLE claims (
    id           INTEGER PRIMARY KEY AUTOINCREMENT,
    text         TEXT NOT NULL,
    entity_ids   TEXT NOT NULL DEFAULT '[]',
    truth_value  TEXT NOT NULL DEFAULT 'UNKNOWN',
    confidence   TEXT NOT NULL DEFAULT 'MEDIUM',
    source_id    INTEGER REFERENCES sources(id),
    timestamp    TEXT NOT NULL DEFAULT (datetime('now')),
    ulid         TEXT,
    updated_at   TEXT NOT NULL DEFAULT '',
    machine_id   TEXT NOT NULL DEFAULT '',
    properties   TEXT NOT NULL DEFAULT '{}'
);
CREATE INDEX        idx_claims_truth  ON claims(truth_value);
CREATE UNIQUE INDEX idx_claims_ulid   ON claims(ulid);
```

Each `claim` is **one proposition** — typically a single sentence extracted from a source article. Why a separate table from `relations`?

- A **claim** is the *raw evidence*: "the Strait of Hormuz carries 20% of global oil shipments." That's the LLM's extraction from the article.
- A **relation** is the *normalized fact*: `(Strait_of_Hormuz, SUPPLY, crude_oil)`. Multiple claims from multiple sources may all attest to the same relation.

The `truth_value` field is the **Belnap B4 value** — `TRUE`, `FALSE`, `BOTH`, `UNKNOWN`. Live distribution from a populated Cascade:

truth_value	Count	Comment
TRUE	7602	Default for newly-ingested claims
BOTH	437	Paraconsistent — two sources disagreed
FALSE	42	Affirmatively refuted
UNKNOWN	335	Insufficient evidence either way

truth_value	Count	Comment

The **paraconsistent** (`BOTH`) row count is the most revealing — it's the rate at which Cascade has had to *record contradiction* rather than pick a winner. About **5% of claims** in a typical news-ingestion corpus end up paraconsistent. That's neither rare enough to ignore nor common enough to dismiss the system as too noisy — it's the *empirical baseline* for "how often serious journalism disagrees about facts."

`sources` — where the data came from

```
CREATE TABLE sources (
    id                INTEGER PRIMARY KEY AUTOINCREMENT,
    url               TEXT,
    title             TEXT,
    author            TEXT,
    date              TEXT,
    source_type       TEXT NOT NULL DEFAULT 'news',
    credibility       REAL NOT NULL DEFAULT 0.8,
    raw_text          TEXT NOT NULL DEFAULT '',
    summarized_text   TEXT NOT NULL DEFAULT '',
    ingested_at       TEXT NOT NULL DEFAULT (datetime('now')),
    ulid              TEXT,
    updated_at        TEXT NOT NULL DEFAULT '',
    machine_id        TEXT NOT NULL DEFAULT ''
);
```

The provenance root. Every entity, relation, and claim *can* be traced back to a source via `source_id`. The `credibility` field (a `REAL` in `[0, 1]`) lets the truth-evaluation step weight claims by the historical reliability of their source — Reuters at 0.95 outweighs SubstackBlog at 0.4 when adjudicating a `BOTH`-truth disagreement.

`raw_text` is the article body (typically truncated to some chunk limit); `summarized_text` is an LLM-condensed version used for fast cascade-context queries. The full chain is **article → source → claims → relations → entities** — top-down, every extracted fact points home to the article that introduced it.

44.4 The type tables — `entity_types` and `relation_types`

The two metadata tables that define Cascade's *ontology*:

```
CREATE TABLE entity_types (
    name        TEXT PRIMARY KEY,
    description TEXT NOT NULL DEFAULT '',
    created_at  TEXT NOT NULL DEFAULT (datetime('now'))
);
CREATE TABLE relation_types (
    name        TEXT PRIMARY KEY,
```

```
    description TEXT NOT NULL DEFAULT '',
    created_at  TEXT NOT NULL DEFAULT (datetime('now'))
);
```

Tiny tables — typically 10 rows each — but *load-bearing*. The `type` columns in `entities` and `relations` are foreign-key-like (but not enforced by SQLite constraints; the application layer maintains the invariant). Live contents from a populated Cascade:

Entity types:

```
STATE       Countries, nations, governments
ORG         Organizations, companies, institutions, military units
ACTOR       Named individuals (leaders, officials, commanders)
INFRA       Infrastructure (ports, pipelines, straits, bases)
COMMODITY   Physical goods (oil, gas, grain, weapons)
MARKET      Financial markets, indices, trading venues
SECTOR      Economic sectors (energy, shipping, agriculture)
EVENT       Named events (wars, treaties, elections, crises)
POLICY      Sanctions, treaties, legislation, executive orders
TECHNOLOGY  Specific tech (drones, missiles, AI systems, cyber)
```

Relation types (8 canonical + extensible):

```
CAUSAL       Cause-effect relationship
DEPENDENCY   One entity depends on another
TEMPORAL     Time-ordered sequence
ADVERSARIAL  Opposition, conflict, rivalry
ALLIANCE     Partnership, cooperation, mutual support
SUPPLY       Resource or goods supply chain
CONTROL      Authority, ownership, governance
INFLUENCE    Soft power, lobbying, shaping outcomes
```

A live install also typically has 1–2 *Axioma-injected* relations (e.g. `commodity`, `test_fact`) — added when your `.ax` scripts assert facts with new relation names. The `cascade.axioma` bridge auto-registers them in `relation_types` with the description prefix "`Axioma relation: <name>/<arity>`".

The asymmetry — entity types fixed, relation types extensible — reflects what the system is *for*. Cascade adds new connection *kinds* readily; it adds new *kinds of thing* rarely.

44.5 The `rules` table — Cascade's reasoning vocabulary

```
CREATE TABLE rules (
    id              INTEGER PRIMARY KEY AUTOINCREMENT,
    name            TEXT NOT NULL,
    type            TEXT NOT NULL,    -- correlation, causal, behavioral, conditional
    domain          TEXT,             -- economic, political, sociological, general
    engine          TEXT DEFAULT 'python',
    definition      TEXT NOT NULL,
    confidence      TEXT DEFAULT 'MEDIUM',
    entity_ids      TEXT DEFAULT '[]',
```

```
    source          TEXT,            -- manual, llm, axioma, converted, file
    executable      INTEGER DEFAULT 1,
    description     TEXT,
    created_at      TEXT NOT NULL DEFAULT (datetime('now')),
    applied_count   INTEGER DEFAULT 0,
    success_count   INTEGER DEFAULT 0,
    ulid            TEXT
);
CREATE INDEX idx_rules_type   ON rules(type);
CREATE INDEX idx_rules_domain ON rules(domain);
CREATE INDEX idx_rules_engine ON rules(engine);
```

A single table records rules across **five engines**:

Engine	Typical count	What it stores
`python`	36	A `module:function_name` string
`prolog`	17	A Prolog fact string (legacy)
`axiomalang`	14	A path to an `.ax` rule file
`cascade`	12	A native-Cascade DSL rule string
`axioma`	6	Inline Axioma code

`applied_count` and `success_count` are *bookkeeping* columns — every time the reactor fires a rule, the counters increment. Over time you can see which rules are load-bearing and which are dead weight.

The `source` column tracks **provenance**: `file` means read from `.cascade/rules/`; `llm` means proposed by the model during ingest; `manual` means typed in by a user via the CLI. The `executable` flag distinguishes *callable* rules (1) from *informational* ones (0) — the latter are LLM-learned summaries that document a pattern without firing automatically.

44.6 The reactor: `change_log`, `truth_history`, `pins`

Three tables that aren't part of the core graph but matter for *reactive* behavior:

`change_log` — what just happened

```
CREATE TABLE change_log (
    id                INTEGER PRIMARY KEY AUTOINCREMENT,
    change_type       TEXT NOT NULL,
    item_type         TEXT NOT NULL,
    item_id           INTEGER NOT NULL,
    entity_ids        TEXT DEFAULT '[]',
    metadata          TEXT DEFAULT '{}',
    reactor_handled   INTEGER DEFAULT 0,
    rules_fired       TEXT DEFAULT '[]',
    created_at        TEXT NOT NULL DEFAULT (datetime('now')),
```

```
    ulid              TEXT
);
```

Every insert/update to `entities`, `relations`, or `claims` writes a row here. The `reactor_handled` flag starts at 0; the reactor agent (Volume III Part XV) processes the queue and flips it to 1, recording which rules fired in `rules_fired`. This is **the event log** that makes the system reactive — rules don't poll the graph; they react to the diff.

`truth_history` — audit trail for truth changes

```
CREATE TABLE truth_history (
    id                INTEGER PRIMARY KEY AUTOINCREMENT,
    claim_id          INTEGER NOT NULL REFERENCES claims(id),
    old_truth_value   TEXT NOT NULL,
    new_truth_value   TEXT NOT NULL,
    old_confidence    TEXT NOT NULL,
    new_confidence    TEXT NOT NULL,
    reason            TEXT,
    changed_at        TEXT NOT NULL DEFAULT (datetime('now')),
    ulid              TEXT
);
```

When a claim's `truth_value` changes (e.g. a new source asserts the opposite, flipping `TRUE` to `BOTH`), a row lands here. This is how the system answers "when did this claim become contested?" or "what changed last week?"

`pins` — user-marked items

```
CREATE TABLE pins (
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
    item_type   TEXT NOT NULL,   -- 'entity', 'relation', 'claim', 'source'
    item_id     INTEGER NOT NULL,
    label       TEXT,
    note        TEXT,
    pinned_at   TEXT NOT NULL DEFAULT (datetime('now')),
    ulid        TEXT
);
```

User-curated bookmarks across all the core tables. The `item_type` discriminator means one pin table can reference any entity, relation, claim, or source. Used by the dashboard for the "Watch list" UI.

44.7 The sync machinery — `sync_outbox` and `sync_meta`

The most heavily-trafficked table in a long-lived Cascade isn't `entities` or `relations` — it's `sync_outbox`:

```
CREATE TABLE sync_outbox (
    id          TEXT PRIMARY KEY,        -- the row's ULID
    table_name  TEXT NOT NULL,
    row_id      TEXT NOT NULL,
    operation   TEXT NOT NULL,           -- INSERT or UPDATE
    row_data    TEXT NOT NULL DEFAULT '{}',
    machine_id  TEXT NOT NULL,
    created_at  TEXT NOT NULL DEFAULT (datetime('now')),
    pushed_at   TEXT,
    error       TEXT
);
CREATE INDEX idx_sync_outbox_pushed ON sync_outbox(pushed_at);
```

Every mutation to a core table writes a sync-outbox row (operation `INSERT` or `UPDATE`) with the full new row serialized as JSON. A separate sync daemon reads unpushed rows and pushes them to Supabase (a hosted-Postgres mirror).

Live distribution in a populated database:

Operation	Count	Comment
`INSERT`	307,784	One per new entity / relation / claim
`UPDATE`	1,906	Truth-value flips, alias additions, etc.

The outbox grows monotonically (the daemon marks rows as pushed but doesn't delete) — eventually you run `cascade sync truncate` to clean old pushed rows.

`sync_meta` is a small key-value store for sync-daemon state: last-pushed-cursor, last-error, machine-id, etc.

The architectural point: sync is *append-only at the row level*. Two machines that ingest different articles each generate their own `sync_outbox` rows; the daemon pushes both sets; Supabase merges (by ULID). The ULID columns on every core table are the deduplication key. No conflict resolution needed because every mutation has a unique global ID.

44.8 Reading `cascade.db` with raw SQL

You don't *need* the Cascade CLI to query the knowledge graph — you can `sqlite3 cascade.db` directly. Useful when the CLI doesn't expose the exact join you want:

```
-- Top entity types by count
SELECT type, COUNT(*) FROM entities
GROUP BY type ORDER BY 2 DESC;

-- Top relation types by count
SELECT type, COUNT(*) FROM relations
GROUP BY type ORDER BY 2 DESC;
```

```sql
-- All HIGH-confidence CAUSAL relations involving Iran
SELECT e1.name AS src, r.type, e2.name AS tgt, r.confidence
FROM relations r
JOIN entities e1 ON r.source_entity_id = e1.id
JOIN entities e2 ON r.target_entity_id = e2.id
WHERE r.type = 'CAUSAL'
  AND r.confidence = 'HIGH'
  AND (e1.name = 'Iran' OR e2.name = 'Iran')
LIMIT 10;

-- Claims with paraconsistent truth (BOTH)
SELECT id, substr(text, 1, 60) AS preview, confidence
FROM claims WHERE truth_value = 'BOTH' LIMIT 5;

-- Recent changes
SELECT change_type, item_type, item_id, created_at
FROM change_log
WHERE reactor_handled = 0
ORDER BY created_at DESC LIMIT 10;
```

The CLI commands you've used in Ch.43 — `cascade entity list`, `cascade discover cascade`, etc. — are mostly thin wrappers around joins like these. Reading the raw SQL teaches you the system's *capability surface*.

44.9 Honest limits

Four database-design tradeoffs to know about:

1. **JSON in `properties` blurs the schema.** The `properties` column on `entities`, `relations`, and `claims` is JSON-typed (stored as TEXT, parsed at read time). Schema flexibility wins; you give up per-field indexing. Queries that filter on `properties` deserialize every row.
2. **No FK constraint enforcement.** SQLite enforces foreign keys only if `PRAGMA foreign_keys = ON` is set per-connection. Cascade enables it but tools that connect directly (raw `sqlite3` CLI) may not. Watch for orphan rows in `relations.source_entity_id`.
3. **`ulid` is `NOT UNIQUE` in early rows.** Rows predating the sync-machinery rollout may have empty `ulid` fields. The unique index `idx_*_ulid` only constrains non-empty values.
4. **Sync outbox grows fast.** ~300k rows for an install with ~13k items (entities+relations+claims). Plan for `cascade sync truncate --before=<date>` in long-running deployments.

These are real engineering tradeoffs, not bugs — they reflect choices about *what's worth optimizing for* at each subsystem.

44.10 Exercises

These exercises run against a real `cascade.db`. All queries are pure-SQL — they don't require any Cascade CLI flag or LLM key. Substitute your DB path.

44.1 — Schema audit

Connect to `cascade.db` with `sqlite3` and run `.tables`, `.schema entities`, `.schema relations`, `.schema claims`. Compare what you see to §44.3 — any columns added since this chapter was written? (Cascade schema migrations land regularly.)

44.2 — Type distribution

Write SQL to count entities by type and relations by type, sorted descending. Identify the *most common* entity type and the *most common* relation type. Why do you think they dominate?

44.3 — The paraconsistent slice

Find all claims with `truth_value = 'BOTH'`. Print their first 60 characters plus their source's title. Pick one and read the underlying source URL — what were the two sides of the disagreement?

44.4 — Relations join

Pick an entity name from your DB. Write a SQL query that joins `relations` to `entities` twice (once as source, once as target) to print all of that entity's outgoing relations in human-readable form (source name, relation type, target name, confidence).

44.5 — The reactor backlog

Count unhandled `change_log` rows (`WHERE reactor_handled = 0`). If the count is non-zero, what does that mean about your reactor's recent activity? Look at the most recent 10 unhandled rows — what changed?

44.6 (open) — Schema critique

You've now seen the live schema in detail. Propose *one* schema change you'd make and *one* you'd resist. Examples to think about:

- Should `properties` be split into typed columns?
- Should `relation_types` enforce a closed enum?
- Should `truth_value` be a foreign key to a `truth_values` table?
- Should `sync_outbox` auto-prune pushed rows?

For each, sketch what *breaks* and what *improves* if you made the change. Real schema design is a series of such tradeoffs.

Solutions to selected exercises: Chapter 44 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch44_solutions.md.)*

44.11 What you learned

- Cascade's knowledge graph lives in **SQLite** at `data/cascade.db` — embedded, serverless, SQL-queryable, WAL-mode-concurrent.

- **Seven core tables** do 95% of the work: `entities`, `relations`, `claims`, `sources`, `rules`, `entity_types`, `relation_types`.
- The **four-tier provenance chain** — article → source → claim → relation → entity — means every fact traces home to its source.
- **Paraconsistency** (Belnap B4 `BOTH`) accounts for about 5% of claims in a typical install — the empirical baseline for "how often serious journalism disagrees."
- **Reactivity** lives in `change_log` (event queue), `truth_history` (audit trail), and `pins` (user marks).
- **Sync** is append-only-at-the-row-level via `sync_outbox` + Supabase mirror; ULIDs are the cross-machine dedup key.
- Reading the **raw SQL schema** teaches you the system's full capability surface — the CLI is a thin wrapper.

The next chapter zooms one layer up: **the ingest pipeline** that *populates* these tables — fetcher, spaCy NER, LLM extraction, claim decomposition. Where this chapter taught the *shape*, the next teaches the *fill*.

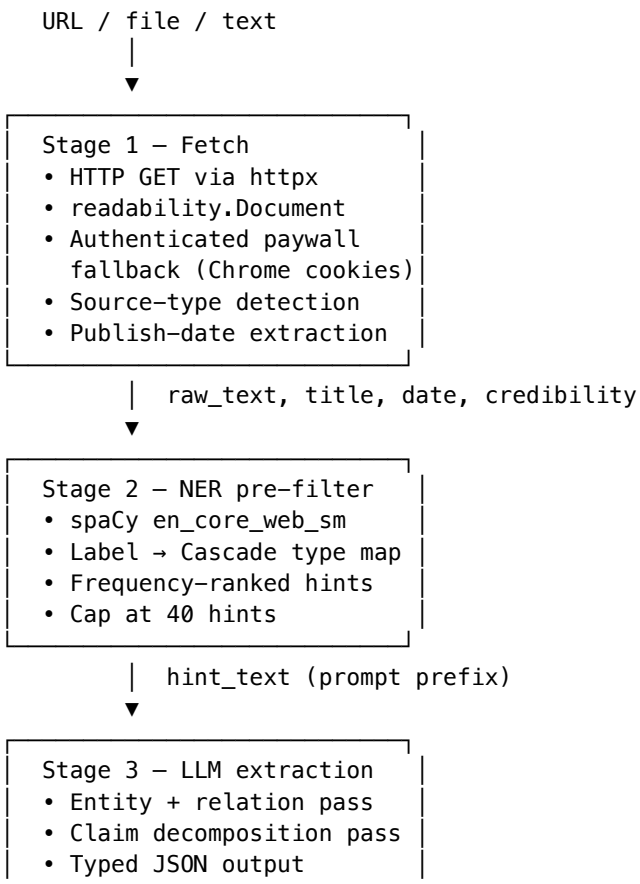
> *"A schema is a frozen set of decisions about what matters."* — paraphrased. Cascade's schema bets that directed typed relations with provenance and a bilattice truth value will cover most reasoning tasks a journalist or analyst encounters. The remaining tables exist because *that bet wasn't quite enough*. Each table you met is a place the framework said "this matters, too, and it deserves its own row."

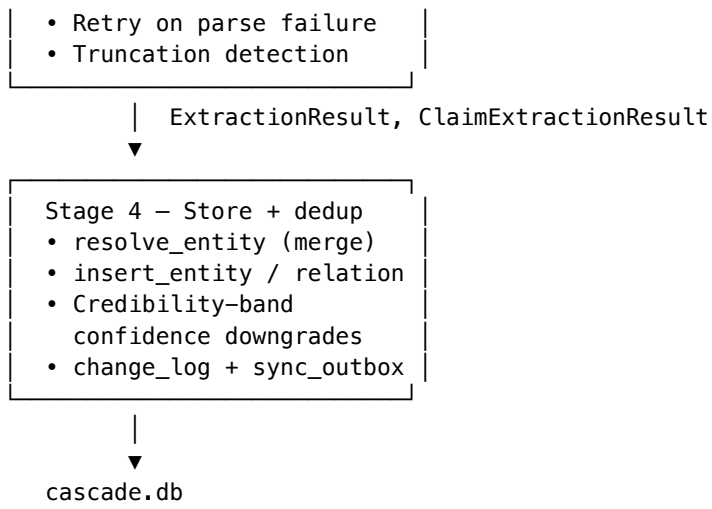
Chapter 45 · The Ingest Pipeline

> **What this chapter is.** Ch.44 showed you the *destination* — the SQLite tables that store Cascade's knowledge graph. This chapter shows the *journey* — the four-stage pipeline that takes a URL, a file, or pasted text and turns it into entities, relations, and claims with provenance, types, and truth values attached. By the end you'll understand what happens between `cascade ingest <url>` and the first row landing in `entities`. You'll also know where the pipeline can break, where it silently degrades, and where to look in the logs when an ingestion produces zero output.

This is the second chapter of Volume III Part XIII (Cascade Knowledge Graph). Ch.44 was the *static view*; this is the *dynamic view*.

45.1 The four stages, end-to-end





Read top-to-bottom. Each arrow is a Python function call returning a typed Pydantic model. The whole pipeline is **synchronous-by-default** (one article ingested at a time) but **parallelizable at Stage 3** — the LLM extraction and claim extraction can run as concurrent calls, then converge at Stage 4.

A typical ingestion takes **15-60 seconds**, dominated by the LLM round-trip in Stage 3.

45.2 Stage 1 — Fetching the content

`cascade.core.ingester.ingest_url(url)` is the entry point. Three branches:

1. **Known-paywalled domain** (WSJ, FT, Bloomberg, NYT, Economist, ...) → try authenticated fetch first using the user's Chrome cookies. Falls back to plain HTTP if that fails.
2. **Plain URL** → `httpx.get(url, follow_redirects= True, timeout=30.0)` with a Chrome User-Agent. On `401`/`403`, *auto-retries* with the authenticated path.
3. **Non-URL inputs** → `ingest_file(path)`, `ingest_text(text)`, `ingest_browser(url)` (controls a real Chrome via AppleScript), `ingest_render(url)` (headless browser for SPA-rendered pages).

The response then runs through `readability.Document` to strip nav chrome, ads, and footers, leaving the article body. A regex pass (`_clean_article_text`) removes a handful of frequent UI fragments — "Show Conversation (N)", "Reprints Gift Article", "" — that survive readability.

The output is a `Source` Pydantic model with `url`, `title`, `raw_text`, `date`, `source_type`, `credibility`. The Source isn't stored yet — that happens at the very end, after extraction succeeds.

Source-type detection and credibility

`detect_source_type(url, html)` looks at the URL path and HTML markers (`<article>`, `<meta property= "og:type">`, byline patterns) and returns one of:

SourceType	Default credibility
NEWS	0.80

SourceType	Default credibility
OPINION	0.50
ANALYSIS	0.85
PRESS_RELEASE	0.70
REPORT	0.90
SOCIAL	0.40

These numbers are *hardcoded defaults* in `models.py` — but `credibility` is a `REAL` column on `sources`, so the analyst can override per-source after evaluation. A live database typically shows:

```
source_type  n
-----------  ---
news         317
opinion       11
analysis       1
```

The distribution reflects what Cascade ingests: mostly news, occasionally op-eds, rarely analysis reports. (Reports usually arrive via file ingestion, not URL.)

Why two paywall paths

The authenticated fetch path uses **the user's existing browser session cookies** — *not* paywall bypass. If you're subscribed to the WSJ and logged in to wsj.com in Chrome, `cascade ingest <wsj-url>` fetches the article using your already-paid-for session. The cookie cache lives in memory for 30 minutes (configurable). Outside the cache, Cascade extracts cookies from Chrome's `~/Library/Application Support/Google/Chrome/...` SQLite store via the `browser-cookie3` package.

The browser-render path (`ingest_browser`) shells out to a real Chrome via AppleScript when JavaScript rendering is necessary (e.g. Bloomberg's SPA pages won't surrender content to cookie-only fetches).

45.3 Stage 2 — The spaCy NER pre-filter

```
def extract_ner_hints(text: str) -> list[dict]:
    nlp = _get_nlp()              # lazy-loaded en_core_web_sm
    if nlp is None:
        return []                 # graceful degradation
    doc = nlp(text)
    entity_counts = {}
    for ent in doc.ents:
        cascade_type = _SPACY_TO_CASCADE.get(ent.label_)
        if cascade_type is None:
            continue
        ...
    return sorted(entity_counts.values(),
                  key=lambda x: x["count"],
                  reverse=True)
```

spaCy's `en_core_web_sm` model labels every noun phrase with one of ~18 entity labels. Cascade maps a subset to its own type ontology:

spaCy label	Cascade type
`GPE` (countries, cities)	`STATE`
`NORP` (nationalities, parties)	`ORG`
`ORG`	`ORG`
`PERSON`	`ACTOR`
`FAC` / `LOC`	`INFRA`
`PRODUCT`	`COMMODITY`
`EVENT`	`EVENT`
`LAW`	`POLICY`
`MONEY`, `DATE`, `QUANTITY`, …	*skip*

The output is a frequency-ranked list of candidate entities. The function `format_ner_hints` produces prompt-prefix text like:

```
PRE-IDENTIFIED ENTITIES (from NER — verify and refine types,
                         add any missing):
- Iran (likely STATE, mentioned 12x)
- Strait of Hormuz (likely INFRA, mentioned 8x)
- Tehran (likely STATE, mentioned 5x)
...
```

This block gets prepended to the LLM extraction prompt. The LLM uses it as a **seed list** — it doesn't have to discover entities from scratch, just verify and re-type them.

Why NER first

Three reasons:

1. **Cost.** spaCy runs locally for free; the LLM costs ~$0.001-$0.05 per article depending on provider. Pre-filtering reduces the LLM's "discovery" workload.
2. **Accuracy.** spaCy is better at *finding* entities than the LLM (especially obscure names); the LLM is better at *typing* and *relating* them.
3. **Graceful degradation.** If spaCy isn't installed, the function returns `[]` and the LLM gets no hints — but the pipeline still works.

The model is **lazy-loaded** — `_get_nlp()` only loads on the first call, then caches in a module global. Subsequent ingestions reuse it.

45.4 Stage 3 — LLM extraction (two passes)

Two parallel LLM calls. Each takes the article text plus a prompt template:

Pass 3a — `extract_entities_and_relations`

Prompt structure (heavily simplified):

```
{ner_hints}        # The spaCy seed list

You are an entity-and-relation extractor.

Return ONLY valid JSON of shape:
{
  "entities": [
    {"name": "...", "type": "STATE|ORG|...",
     "domain": "energy|political|...",
     "aliases": [...], "properties": {...}}
  ],
  "relations": [
    {"source_entity": "...", "target_entity": "...",
     "type": "CAUSAL|SUPPLY|...",
     "confidence": "HIGH|MEDIUM|LOW|SPECULATIVE",
     "evidence": "..."}
  ]
}

ARTICLE TEXT:
{article_text}
```

The result is an `ExtractionResult` Pydantic model. Typical yield from a 600-word article:

- **30 entities** (range: 1 to 136)
- **23 relations** (range: 1 to 74)

The big numbers come from analytical pieces with many linked items; the small numbers come from short news flashes mentioning one or two actors.

Pass 3b — `extract_claims_llm`

Different prompt — focuses on *sentence-level propositions*:

```
You are a claim decomposer. Break this article into
atomic propositions.

Return JSON: {"claims": [
  {"text": "...",
   "entity_ids": [...],
   "truth_value": "TRUE|FALSE|BOTH|UNKNOWN",
   "confidence": "HIGH|MEDIUM|LOW|SPECULATIVE"}
]}

ARTICLE TEXT:
{article_text}
```

Typical yield: **31 claims per article** (range: 1 to 100). Each claim is one assertion: "Iran exports an estimated 2.1 million barrels of oil per day."

The two passes are *independent* — they could be called in parallel by an async wrapper. Today they run sequentially. The `decomposer.py` and `extractor.py` modules each handle one pass.

Retry-on-parse-failure

Both passes use `extract_json(response)` to parse the LLM output. When parse fails (the LLM returned prose, or got truncated mid-JSON), the code:

1. Logs a warning at the LLM-side error.
2. Sends a **retry prompt** that prepends *"The previous response was not valid JSON…"*.
3. Re-parses.
4. If parse still fails → `RuntimeError` with a suggestion to raise `max_tokens` in cascade.toml.

Production logs occasionally show a flurry of retry prompts when a provider's quota throttles — the LLM returns short responses; Cascade retries; eventually the request lands.

Truncation detection

Even when JSON parses, the response might be *truncated* — the LLM hit `max_tokens` mid-array. The `extraction_info.get("json_repaired")` flag tells you the JSON parser had to *repair* the response (close braces, drop a trailing comma). When true, Cascade records an `issue`:

> *"LLM response was truncated and JSON had to be repaired — some claims may be missing. Consider increasing max_tokens in cascade.toml."*

The `issues` field rides along in the ingestion output — visible to the analyst, but doesn't fail the run.

Six providers, one interface

`call_llm(prompt, max_tokens=None)` dispatches across six provider backends:

- **Anthropic** (Claude) — production default
- **OpenAI** (GPT) — via `openai.AsyncOpenAI`
- **Grok** (xAI) — OpenAI-compatible
- **Groq** — fast inference, OpenAI-compatible
- **Google Gemini** — native API
- **Ollama** — local models (gpt-oss, llama, etc.)

Provider selection is per-call via the `--provider` CLI flag or the `[llm]` config block in `cascade.toml`. Switching providers mid-run is supported — Cascade keeps no provider-specific state between calls.

45.5 Stage 4 — Storing with dedup

`store_entities(conn, result, source_id)` writes the LLM output into the four core tables. Three subtleties:

Entity resolution (`resolve_entity`)

Before inserting an entity, the code asks: *does an entity by this name (or alias) already exist?*

```
existing = resolve_entity(conn, ent.name, ent.type.value)
entity_id = insert_entity(conn, entity)  # merges if existing
if existing:
    merged_entities += 1
```

The `resolve_entity` lookup checks `entities.name`, `entities.aliases` (JSON array), and a few normalization rules (case-insensitive, trim whitespace). A successful match means the new extraction *merges* — its aliases get added to the existing row, its properties get merged, but no new row is inserted.

This is **the dedup substrate**. Without it, ingesting "USA" and "United States" as separate mentions would produce two STATE rows. With it, the second extraction lands as an alias on the first.

Watch out: the resolver is *name-based*, not *entity-resolution-grade*. Different spellings ("Russia" vs "Russian Federation") may produce duplicates that need a manual `cascade entity merge` later.

Relation insertion

For each relation, the code resolves source and target entities, then calls `insert_relation`. If either entity is unresolvable, the relation is **skipped** with a warning:

```
WARNING: Skipping unresolvable relation |
  target='Some Entity' | type=CAUSAL
```

This is *graceful degradation*. A relation referring to an entity that didn't get extracted is lost — but the rest of the ingestion succeeds.

Credibility-band confidence downgrades

The `_apply_credibility_downgrade(confidence, credibility)` helper modifies extracted confidences based on source credibility:

Source credibility	Effect
`≥ 0.7` (news, analysis)	No change
`0.5–0.7` (press release, opinion)	One band down
`< 0.5` (social)	Two bands down

A `HIGH`-confidence claim from a `0.4`-credibility social-media source becomes `LOW`. The downgrade honors the principle from Ch.44 — *the source's reliability shapes the claim's truth-weight*.

change_log and sync_outbox

Every `INSERT` and `UPDATE` writes a row to `change_log` (with `reactor_handled = 0`) and to `sync_outbox` (with `pushed_at = NULL`). These trigger the reactor and the sync daemon, respectively.

For one typical article ingestion, ~85 rows land in `sync_outbox`: 1 source, ~30 entities, ~23 relations, ~31 claims, plus a handful of change_log rows. With ~300 sources ingested over time, that compounds to ~300k `sync_outbox` rows — the number you saw in Ch.44 §44.7.

45.6 Watching ingestion happen

Run an ingestion with the -v flag to see the pipeline narrate itself:

```
uv run cascade -v ingest "https://www.reuters.com/..."
```

The log output (lightly trimmed) looks like:

```
fetcher | Ingesting URL | url=https://www.reuters.com/...
fetcher | Ingested URL | title=... | text_len=4823
ner     | spaCy NER | 47 unique entities from 4823 chars
extractor | NER pre-filter found 47 candidate entities |
            prepending hints to LLM prompt
extractor | Entity/relation LLM extraction started | text_len=4823
provider  | Calling anthropic claude-sonnet-4-6 | max_tokens=4000
extractor | Entity/relation LLM extraction done | 8.2s |
            entities=28 | relations=19
decomposer | Claim LLM extraction started | text_len=4823
decomposer | Claims parsed | 24 claims extracted at 6.1s
extractor | Storing 28 entities to graph...
extractor | Entities stored at 0.3s | new=11 | merged=17 |
            storing 19 relations...
extractor | Store complete | source_id=331 |
            entities: 11 new + 17 merged |
            relations: 18 stored + 1 skipped |
            elapsed=0.6s
```

The lifecycle is fully traceable. *Every* row in `cascade.db` can be traced back to one of these log lines. When ingestion produces zero output, this is where you look.

45.7 Failure modes and graceful degradation

What happens when each stage fails:

Stage	Failure	What Cascade does
1 Fetch	DNS / connection refused	`IngestionError`, abort
1 Fetch	HTTP 401/403	Auto-retry with auth cookies
1 Fetch	HTTP 429 (rate limit)	Bubble up, abort
1 Fetch	Short content (< 2k chars)	Auto-retry via browser
2 NER	spaCy not installed	Return `[]`, skip pre-filter
2 NER	Text > 1M chars	Truncate to 1M, warn
3 LLM	Parse failure	One retry with sterner prompt
3 LLM	Network timeout	Bubble up, abort
3 LLM	Quota exceeded	Bubble up with provider error
3 LLM	Response truncated	Repair JSON, record issue
4 Store	Entity unresolvable	Skip relation, warn, continue
4 Store	DB locked	Wait + retry (SQLite WAL)

The pattern: **failures upstream of Stage 4 abort the ingestion; failures inside Stage 4 degrade gracefully**. Once the LLM extraction has completed, the system tries hard to *land what it has* rather than throw it away.

45.8 Honest limits

Five real issues to know about:

1. **Entity resolution is name-based.** "USA" and "United States of America" may end up as duplicate entities if they're not in each other's aliases yet. Manual merge required.
2. **No streaming.** The LLM call is request/response, not streamed. A long article means a long wait with no progress indicator beyond the log line "…LLM extraction started".
3. **Single-language.** spaCy `en_core_web_sm` is English-only. Non-English articles get garbage-typed entities. Cascade has hooks for other languages but they're not wired into the default pipeline.
4. **Per-article isolation.** Stages 1-4 run for one article in sequence. Bulk ingestion of 100 articles processes them serially — 30-90 minutes total. A parallel-async ingester is on the roadmap.
5. **LLM hallucination.** The model occasionally extracts relations that aren't in the text (especially on commentary-heavy pieces). Without a verification pass, these land in the database alongside accurate ones. Ch.46 (next chapter) shows how downstream truth-evaluation can catch some of these.

45.9 The pipeline from the analyst's perspective

You ran `cascade ingest <url>` in Ch.43. Now you know what each second of waiting represents:

Seconds 0–2 | Stage 1: HTTP fetch + readability parse |
Seconds 2–3 | Stage 2: spaCy NER on local text |
Seconds 3–10 | Stage 3a: LLM entity/relation pass |
Seconds 10–17 | Stage 3b: LLM claim decomposition pass |
Seconds 17–18 | Stage 4: SQLite inserts + sync_outbox |

Total: ~15-60 seconds depending on article length and provider latency. The Stage 3 LLM calls dominate ~80% of the time.

When you want to *scale* ingestion (Ch.50 — agent swarm), this is the bottleneck — Stage 3 is parallelizable across articles, Stages 1-2 and 4 aren't. The collector agent's design (a Cascade- specific pattern Volume III Part XV covers) follows exactly this slicing: queue many articles, run their Stage-3 calls in parallel, serialize the Stage-4 writes.

45.10 Exercises

These exercises mix shell, Python, and SQL — they exercise the pipeline at different layers.

45.1 — Watch an ingestion

Pick a short news article URL (300-800 words ideal). Run:

```
uv run cascade -v ingest "<url>" 2>&1 | head -60
```

Read every log line. Identify the four stages by name. Note: how many entities did NER find? How many did the LLM keep? How many got merged vs newly inserted?

45.2 — Per-source extraction yield

Query the live database for *yield-per-source*:

```
WITH per_source AS (
  SELECT source_id,
         COUNT(*) AS n_claims
  FROM claims
  WHERE source_id IS NOT NULL
  GROUP BY source_id
)
SELECT AVG(n_claims), MIN(n_claims), MAX(n_claims)
FROM per_source;
```

Do the same for entities and relations. Compare yours to the chapter's stated averages (~31 claims, ~30 entities, ~23 relations per source).

45.3 — Source-type distribution

Query the source_type distribution from your database:

```
SELECT source_type, COUNT(*), AVG(credibility)
FROM sources GROUP BY source_type;
```

What's the dominant type? Why?

45.4 — Trace one article all the way through

Pick a source ID with non-trivial extraction (say, ~20+ entities). Run four queries:

1. The source row (1 row).
2. Entities introduced by this source.
3. Relations introduced by this source.
4. Claims introduced by this source.

How does the claim count compare to the entity count? Why does the article need more claims than entities?

45.5 — Find the merge candidates

Find pairs of entities with similar names that should *probably* be merged but aren't:

```sql
SELECT e1.name, e2.name, e1.type, e2.type
FROM entities e1, entities e2
WHERE e1.id < e2.id
  AND e1.type = e2.type
  AND lower(e1.name) LIKE '%' || lower(e2.name) || '%'
LIMIT 10;
```

(The `LIKE` join is slow — limit to 10.) Look at the results: are any of these *actually* duplicates? If yes, would a manual merge improve graph quality?

45.6 (open) — Design an improvement

You've now seen the pipeline in detail. Pick *one* stage and sketch a concrete improvement:

- Stage 1: a third fetch strategy (e.g. RSS feeds)
- Stage 2: a second NER model (e.g. for code or PDFs)
- Stage 3: a verification pass that re-checks claims against the source text
- Stage 4: a smarter entity-resolution algorithm

For your chosen improvement, write 2-3 paragraphs: what it adds, what it costs, what it might break. Real engineering is exactly this kind of tradeoff.

Solutions to selected exercises: Chapter 45 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch45_solutions.md.)*

45.11 What you learned

- The **four-stage pipeline**: Fetch → NER → LLM → Store. Synchronous, ~15-60 seconds per article, bottlenecked at Stage 3.
- **Stage 1** uses readability + browser-cookie fallback for paywalled domains; six source types with hardcoded credibility defaults.
- **Stage 2** runs spaCy `en_core_web_sm` locally for free, produces a frequency-ranked candidate list, formats it as a prompt prefix for the LLM.
- **Stage 3** is two parallel LLM passes (entities/ relations + claims) across six provider backends, with retry-on-parse-failure and truncation detection.
- **Stage 4** does entity dedup via `resolve_entity`, skips unresolvable relations, applies credibility-band confidence downgrades, and writes to `change_log` + `sync_outbox`.
- Typical yield: **~30 entities, ~23 relations, ~31 claims** per news article.
- Six failure modes that abort vs five that degrade gracefully. The pattern: pre-Stage-4 = abort, in-Stage-4 = land what you can.

The next chapter (Ch.46) zooms in on the **truth- value lifecycle** — how a claim's `truth_value` slot moves from `TRUE` to `BOTH` to (sometimes) `FALSE` over the course of repeated ingestions and analyst evaluations.

> *"Garbage in, garbage out" is a 1957 saying; the 2026 saying is **garbage in, garbage with confidence bands**. The ingest pipeline can't make a noisy article clean, but it can record how*

much trust each claim deserves — and downstream truth evaluation can sort the rest.

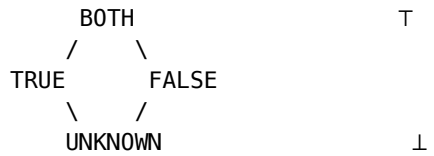
Chapter 46 · The Truth-Value Lifecycle

> **What this chapter is.** Cascade's most subtle mechanism: how a `claim.truth_value` slot evolves from `TRUE` at ingestion through `BOTH` when a contradiction lands through (sometimes) `FALSE` when the evidence flips. Ch.44 showed you the slot; Ch.45 showed how it gets initialized; this chapter shows how it *moves*. By the end you'll know exactly what `b4_join` and `b4_meet` do, when each fires, how the `truth_history` audit trail is generated, and how a single claim's truth flip cascades to related claims and to the relations they support. This is **paraconsistent reasoning in production** — the Belnap B4 logic from Ch.20 driving real contradiction handling.

This is the closing chapter of Volume III Part XIII (Cascade Knowledge Graph). Part XIV starts at Ch.47 with reasoning algorithms.

46.1 The Belnap B4 lattice, restated

You met Belnap B4 in Vol I Ch.20. Recap: four values arranged in a *lattice* (a partial order with both joins and meets):



- **`UNKNOWN`** — bottom; no evidence either way.
- **`TRUE`** — definite positive evidence.
- **`FALSE`** — definite negative evidence.
- **`BOTH`** — top; positive *and* negative evidence exists (the *paraconsistent* case).

Two binary operations make the lattice useful:

- **Join (⊔)** — combines evidence. *"What do we collectively know?"* `TRUE ⊔ FALSE = BOTH` because combining contradictory evidence gives you both.
- **Meet (⊓)** — consensus. *"What do we agree on?"* `TRUE ⊓ FALSE = UNKNOWN` because there's no consensus.

In Cascade's `cascade/core/truth.py:78`:

```python
def b4_join(a: TruthValue, b: TruthValue) -> TruthValue:
    """Lattice join — combines evidence."""
    return _JOIN[(a, b)]
```

```python
def b4_meet(a: TruthValue, b: TruthValue) -> TruthValue:
    """Lattice meet — consensus."""
    return _MEET[(a, b)]
```

The tables are 4×4 (sixteen entries each) and live in the module as Python dicts. Cascade has all four operations: join, meet, negation, and aggregation (repeated-join over a list).

46.2 The join table — combining evidence

The 4×4 join table, in full:

⊔	UNKNOWN	TRUE	FALSE	BOTH
UNKNOWN	UNKNOWN	TRUE	FALSE	BOTH
TRUE	TRUE	TRUE	**BOTH**	BOTH
FALSE	FALSE	**BOTH**	FALSE	BOTH
BOTH	BOTH	BOTH	BOTH	BOTH

The two **bold** cells are the philosophically interesting ones: `TRUE ⊔ FALSE = BOTH` and the symmetric `FALSE ⊔ TRUE = BOTH`. That's where *contradictions* enter the system without crashing.

Three properties worth internalizing:

- **UNKNOWN is the identity.** `UNKNOWN ⊔ X = X` for any `X`. Adding evidence to no-evidence gives you exactly the new evidence.
- **BOTH is absorbing.** `BOTH ⊔ X = BOTH` for any `X`. Once contradicted, always contradicted (unless someone explicitly resolves).
- **TRUE and FALSE join to BOTH, not to one or the other.** This is the *paraconsistent commitment* — the system refuses to pick a winner.

46.3 The meet table — consensus

⊓	UNKNOWN	TRUE	FALSE	BOTH
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
TRUE	UNKNOWN	TRUE	**UNKNOWN**	TRUE
FALSE	UNKNOWN	**UNKNOWN**	FALSE	FALSE
BOTH	UNKNOWN	TRUE	FALSE	BOTH

Meet asks the dual question: *given two sources of evidence, what do they agree on?*

- `TRUE ⊓ FALSE = UNKNOWN` — no agreement → no consensus. (Notice the dual of the join's BOTH.)
- `BOTH ⊓ TRUE = TRUE` — BOTH contains TRUE, so they agree on TRUE.

- `UNKNOWN ⊓ X = UNKNOWN` — UNKNOWN is absorbing for meet (the dual of UNKNOWN being identity for join).

Where each operation fires in Cascade:

- **Join** fires when a *new ingestion* adds evidence to an existing claim. The claim's truth value becomes the join of the old value and the new evidence.
- **Meet** fires when *re-evaluating* a claim across multiple rule engines (Ch.49 will cover this) — the engines vote, and meet is the consensus.

46.4 Negation — the two self-dual values

`b4_negation` is simple but illuminating:

Input	Output
TRUE	FALSE
FALSE	TRUE
BOTH	**BOTH**
UNKNOWN	**UNKNOWN**

The two boldfaced cases — BOTH and UNKNOWN map to themselves — capture the *informational symmetry* of both values:

- **BOTH ↦ BOTH**: if you have both positive and negative evidence, the *negation* of "both sides" is still "both sides." Negation just flips which side is which.
- **UNKNOWN ↦ UNKNOWN**: if you have *no* evidence, the negation is also *no* evidence.

These two are the **fixed points** of negation. TRUE and FALSE are the *non-fixed* values — the ones that move under negation.

46.5 Aggregation — `aggregate_truth(values)`

For a relation supported by N claims, what's the overall truth value? Repeated join:

```
def aggregate_truth(values: list[TruthValue]) -> TruthValue:
    if not values:
        return TruthValue.UNKNOWN
    result = values[0]
    for v in values[1:]:
        result = b4_join(result, v)
    return result
```

A relation supported by `[TRUE, TRUE, TRUE]` aggregates to `TRUE`. Supported by `[TRUE, TRUE, FALSE]`, it aggregates to `BOTH` — because one dissenter is enough to introduce paraconsistency.

This is **the relation's effective truth** — computed on demand by `get_relation_truth` (Ch.44 §44.3) and refreshed when supporting claims change.

The asymmetry: **one FALSE plus many TRUEs ⇒ BOTH, not majority-vote TRUE**. Belnap B4 *refuses to average*. A single contradiction contaminates the aggregate. This is *intentional* — it preserves the paraconsistency.

46.6 Three lifecycle paths

The most common journeys a `claim.truth_value` takes:

Path A — `UNKNOWN` → `TRUE` (the common case)

1. Article ingested (Ch.45 Stage 4).
2. LLM extracted the claim with `truth_value: TRUE` and some confidence.
3. `INSERT INTO claims (..., truth_value, ...)`.
4. No further changes ever — most claims stay `TRUE`.

In the test database: **7602 of 8416 claims (~90%) live here**. The fast, happy path.

Path B — `TRUE` → `BOTH` (the paraconsistent case)

1. Article A ingested; claim "X happened" with `truth_value=TRUE`, `confidence=HIGH`.
2. Article B ingested *later*, asserting "X did not happen" with `truth_value=FALSE`, `confidence=HIGH`.
3. During Article B's Stage 4, the system *checks if the claim already exists* — if it does, it joins the existing truth with the new evidence: `TRUE ⊔ FALSE = BOTH`.
4. `UPDATE claims SET truth_value = 'BOTH'`.
5. `INSERT INTO truth_history (claim_id, old=TRUE, new=BOTH, reason='conflicting source X')`.

In the test database: **437 of 8416 claims (~5%) are BOTH**. About once every 20 claims gets contested.

Path C — `TRUE` → `BOTH` → `UNKNOWN` (analyst

intervention)

The most informative trajectory — and the one that *actually appears in the production database*:

```
SELECT id, claim_id, old_truth_value, new_truth_value,
       substr(reason, 1, 60) AS reason, changed_at
FROM truth_history WHERE claim_id = 7849;
```

Live output:

```
1  7849  TRUE  BOTH     The assessment shifts to BOTH because       2026-04-21T07:26:17
                        while there is an undeniable structural...
2  7849  BOTH  UNKNOWN  No rules matched this claim                 2026-04-21T07:33:19
```

The story:

- **2026-04-21 07:26** — A new piece of evidence contradicted the original assertion. Claim 7849 moved from `TRUE` to `BOTH`. The `reason` field records the LLM's *explanation* — a paragraph-length rationale stored in plain text.
- **2026-04-21 07:33** (seven minutes later) — The analyst ran rule-based evaluation. None of the active rules matched, so the claim's confidence dropped to `SPECULATIVE` and the truth went to `UNKNOWN`. The system *un-evaluated* the claim because it could no longer be defended.

That's **paraconsistent retreat** — when rules can't support either side, the system pulls the claim back to `UNKNOWN` rather than hold a stale `BOTH`. It's a form of *automated humility*.

46.7 The `truth_history` audit table

Every transition writes a row to `truth_history`:

```
CREATE TABLE truth_history (
    id               INTEGER PRIMARY KEY AUTOINCREMENT,
    claim_id         INTEGER NOT NULL REFERENCES claims(id),
    old_truth_value  TEXT NOT NULL,
    new_truth_value  TEXT NOT NULL,
    old_confidence   TEXT NOT NULL,
    new_confidence   TEXT NOT NULL,
    reason           TEXT,
    changed_at       TEXT NOT NULL DEFAULT (datetime('now')),
    ulid             TEXT
);
```

Two design choices worth calling out:

1. **The `reason` field is free-form prose.** Cascade stores the LLM's *explanation* verbatim. This is *forensic data* — you can grep `truth_history` for "contradiction" or "Iran" or any phrase and find when those reasoning patterns moved the lattice.
2. **The table is append-only.** Truth values move; the audit trail records *every* move. Even if a later evaluation moves a claim *back* to TRUE, the intermediate hop is preserved.

The total cost is minimal — typically 2-10 truth_history rows per analyst-evaluated claim. In the test DB only 2 rows exist (the schema was added recently); production installs with active rule evaluation accumulate more.

46.8 `propagate_truth` — when a flip cascades

When a single claim flips, claims *sharing entities* may need re-evaluation. `propagate_truth` does both: write the change *and* find what else is affected.

```
def propagate_truth(conn, claim_id, new_truth,
                    new_confidence, reason=""):
    claim = get_claim(conn, claim_id)
```

```
    old_truth = claim["truth_value"]
    old_confidence = claim["confidence"]

    update_claim(conn, claim_id,
                 new_truth.value, new_confidence.value)
    record_truth_change(conn, claim_id, old_truth,
                        new_truth.value, ...)

    related = find_related_claims(conn, claim_id)
    _propagate_to_relations(conn, claim_id)
    return related
```

Two cascade effects:

Related claims (`find_related_claims`)

A claim has an `entity_ids` JSON array — every entity mentioned in its text. The function finds *other claims* that mention any of the same entities:

```
def find_related_claims(conn, claim_id):
    claim = get_claim(conn, claim_id)
    entity_ids = json.loads(claim["entity_ids"])
    # Scan all claims, return ones with shared entities
    ...
```

`propagate_truth` doesn't *re-evaluate* the related claims — it returns them as a *suggestion list*. An analyst (or downstream agent) can iterate and re-evaluate each. The principle: **claim flips generate review queues, not automatic cascades**.

Relations (`_propagate_to_relations`)

A relation has a `claim_ids` JSON array — the supporting claims. When one of those claims flips, the relation's *effective truth* may change:

```
def _propagate_to_relations(conn, claim_id):
    rows = conn.execute(
        "SELECT id, claim_ids FROM relations"
    ).fetchall()
    for row in rows:
        cids = json.loads(row["claim_ids"])
        if claim_id in cids:
            truth_info = get_relation_truth(
                conn, row["id"])
            conn.execute(
                "UPDATE relations SET confidence = ? "
                "WHERE id = ?",
                (truth_info["confidence"], row["id"]))
```

The relation re-aggregates over its supporting claims using `aggregate_truth` (§46.5). Its confidence band is updated. This *does* happen automatically — relations are computed views of their supporting claims, so they have to stay coherent.

46.9 Credibility-weighted truth (the downgrade

path)

The `_apply_credibility_downgrade(confidence, credibility)` function from `cascade/core/decomposer.py` blends the source's *credibility* into the claim's *confidence band*:

Source credibility	Effect on claim confidence
`≥ 0.7` (NEWS, ANALYSIS, REPORT)	No change
`0.5–0.7` (OPINION, PRESS_RELEASE)	One band down
`< 0.5` (SOCIAL)	Two bands down

A `HIGH`-confidence claim from a `0.4`-credibility social-media source becomes `LOW`. A `MEDIUM`- confidence claim from a `0.6`-credibility press release becomes `LOW`. The downgrade is *applied at ingest time*, baked into the stored `confidence` column.

Live distribution of (truth_value, confidence) pairs after downgrade:

```
TRUE         HIGH         3295
TRUE         MEDIUM        457
TRUE         LOW           398
BOTH         LOW            40
UNKNOWN      SPECULATIVE    22
TRUE         SPECULATIVE    16
FALSE        HIGH           13
UNKNOWN      LOW            11
BOTH         MEDIUM          6
FALSE        LOW             5
```

Notice the *concentration* in (TRUE, HIGH) — the ~40% of claims that came from news sources at 0.8 credibility. The wider spread reflects op-eds and press releases getting downgraded.

46.10 Defeasible cancellation (the future)

The Belnap join is *monotone* — once you've reached `BOTH`, you can't get back to `TRUE` or `FALSE` by *adding* more evidence. Adding new `TRUE` evidence to `BOTH` keeps it at `BOTH`. This is correct behavior — paraconsistency shouldn't evaporate just because a new TRUE arrives.

To *resolve* a `BOTH`, you need **non-monotonic** machinery: the analyst (or a rule) decides which side to keep and which to retire. Cascade's machinery for this is in two places:

1. **Manual override** — `cascade evaluate <claim_id>` runs LLM or rule-based evaluation, which may produce a fresh `truth_value`. The new value *replaces* the old via `update_claim` plus a `truth_history` row. The B4 lattice doesn't prescribe the replacement; the analyst's verdict does.

2. **Axioma defeasible rules** (Vol I Ch.21) — rules tagged with ~~> or ~~ cap their derived truth at `conjecture`-grade and can be *canceled* by stronger contradictory evidence. The cancel mechanism is Axioma's, not Cascade's — but Cascade *uses* Axioma rules, so the cancellation is visible at the Cascade layer.

The current state: **Cascade's truth-propagation mechanism is monotone (join-only)**; the non-monotonic step lives in the evaluation / override path. A future enhancement (the `cancel(rel, args...)` builtin from Vol I) could land at the relation level, with cancelations recorded in a `relations.cancelations` JSON column. That's on the roadmap; not shipped yet.

46.11 Live truth distribution

The full distribution of `(truth_value, n)` in the test database:

truth_value	Count	%	Notes
TRUE	7602	90.3%	Default — newly ingested
BOTH	437	5.2%	Paraconsistent
UNKNOWN	335	4.0%	Analyst-retired or LLM-uncertain
FALSE	42	0.5%	Affirmatively refuted

Three takeaways:

1. **The TRUE-default isn't naive.** Most claims come from a single source asserting them once. Without contradiction, there's nothing to flip them.
2. **The BOTH rate (~5%) is the contradiction baseline.** This is *what fraction of journalism genuinely disagrees with itself* — independent of Cascade's specific rules. A noisier domain (Twitter, op-eds) would have higher BOTH; a cleaner domain (academic abstracts) would have lower.
3. **FALSE is rare.** Affirmative refutation requires either a direct denial in another article (the most common path) or an analyst explicitly marking a claim FALSE. Both are rarer than the articles-disagree case.

46.12 Worked example — a contradiction flow

A toy walkthrough of how `TRUE → BOTH` happens.

Initial state:

```
Article A (source_id=100, NEWS, credibility=0.8) ingested.
LLM extracts claim:
  text="Iran shipped 2.1M barrels/day in March"
  truth_value=TRUE, confidence=HIGH, source_id=100
  → INSERT INTO claims (id=500, ...)
```

```
SELECT id, truth_value, confidence FROM claims WHERE id = 500;
-- 500 | TRUE | HIGH
```

A week later:

```
Article B (source_id=145, NEWS, credibility=0.8) ingested.
LLM extracts claim:
  text="Iran shipped only 1.4M barrels/day in March"
  truth_value=FALSE (about the 2.1M figure)
  confidence=HIGH, source_id=145
```

The pipeline checks: does this claim already exist? *Yes* — by some entity-resolution heuristic, the new claim is matched to claim 500.

```
old_truth = TruthValue.TRUE
new_evidence = TruthValue.FALSE
result = b4_join(old_truth, new_evidence)
# = BOTH
```

Pipeline writes:

```
UPDATE claims SET truth_value = 'BOTH'
WHERE id = 500;

INSERT INTO truth_history (
  claim_id, old_truth_value, new_truth_value,
  old_confidence, new_confidence, reason)
VALUES (500, 'TRUE', 'BOTH', 'HIGH', 'HIGH',
        'Article B asserts 1.4M, contradicting...');
```

`propagate_truth` then looks for related claims (any sharing entity "Iran" or "barrels/day") and for relations supported by claim 500. The relations get re-aggregated; the related claims are returned for review.

A day later — analyst evaluates manually:

```
cascade evaluate 500 --mode rules --engine python
```

Suppose two rules support TRUE, three support FALSE, and one supports BOTH. The hybrid mode aggregates:

```
[TRUE, TRUE, FALSE, FALSE, FALSE, BOTH] →
   aggregate_truth → BOTH (because FALSE present)
```

The analyst sees that the system *stays* at BOTH — the contradiction wasn't resolved by rules. They might:

- **Pin the claim** to keep it visible.
- **Add evidence weight** by ingesting authoritative sources (IEA reports, OPEC data).
- **Mark the claim manually** as TRUE or FALSE based on out-of-band judgment.

Each action lands as another row in `truth_history`. The trail is complete.

46.13 Honest limits

Four caveats to know:

1. **Entity-resolution-driven matching is fragile.** Whether "Iran shipped 2.1M barrels" matches "Iran shipped only 1.4M barrels" depends on the `resolve_entity` heuristic. In practice the contradictions Cascade *catches* are the ones with shared entity IDs; the ones it *misses* are the ones where the second article phrases the same claim differently.
2. **Confidence bands are coarse-grained.** Four bands (HIGH/MEDIUM/LOW/SPECULATIVE) lose a lot of information. A 0.65 numeric confidence and a 0.45 numeric confidence both compress to MEDIUM / LOW. Future revisions may move to REAL columns directly.
3. **The audit table is currently sparse.** In the test DB only 2 rows exist because automated re-evaluation is opt-in. Production installs that wire the reactor to fire on changes would accumulate dozens of rows per claim over its lifetime.
4. **No automated `BOTH → TRUE/FALSE` resolution.** Once a claim is BOTH, only manual or rule-driven evaluation can move it. There's no higher-credibility-source-wins automatic resolution today.

46.14 Exercises

These exercises mix SQL queries and Python-style truth-table evaluation. All runnable; substitute your `cascade.db` path.

46.1 — Compute b4_join by hand

For each pair below, write out the result *without* looking at the table in §46.2:

- TRUE ⊔ UNKNOWN
- FALSE ⊔ BOTH
- TRUE ⊔ FALSE
- UNKNOWN ⊔ UNKNOWN

Verify with the table.

46.2 — Aggregate a list

A relation has three supporting claims:

```
claim 1: TRUE (HIGH)
claim 2: TRUE (HIGH)
claim 3: BOTH (MEDIUM)
```

What does `aggregate_truth([TRUE, TRUE, BOTH])` return? Why isn't it just TRUE (the majority)?

46.3 — Find the contested claims

Find all claims in your database with `truth_value = 'BOTH'`. For one of them, find any `truth_history` rows showing how it got there:

```
SELECT old_truth_value, new_truth_value, reason
FROM truth_history WHERE claim_id = <id>;
```

(In a fresh install you may have zero history rows — the automated rewriter is opt-in.)

46.4 — Truth-by-source credibility

Cross-tabulate `truth_value` against source credibility:

```
SELECT s.credibility, c.truth_value, COUNT(*) AS n
FROM claims c
LEFT JOIN sources s ON c.source_id = s.id
GROUP BY s.credibility, c.truth_value
ORDER BY s.credibility DESC, n DESC;
```

Do higher-credibility sources produce more `TRUE` claims and fewer `BOTH` claims? If yes, that's evidence the credibility band is doing useful work.

46.5 — Negation fixed points

Demonstrate, in code or by hand:

- `b4_negation(TRUE) == FALSE`
- `b4_negation(FALSE) == TRUE`
- `b4_negation(BOTH) == BOTH`
- `b4_negation(UNKNOWN) == UNKNOWN`

Why are BOTH and UNKNOWN fixed points? Explain in one sentence each.

46.6 (open) — Where should `cancel` land?

The chapter notes (§46.10) that Cascade's truth propagation is monotone — once BOTH, always BOTH. Sketch where you'd put a `cancel(claim_id, reason)` function:

- Which table records the cancellation?
- Does it modify `truth_value` directly, or write a marker?
- How do downstream queries respect the cancellation?
- What about the audit trail?

Two-paragraph sketch is enough. Real engineering is exactly these kinds of placement choices.

Solutions to selected exercises: Chapter 46 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch46_solutions.md.)*

46.15 What you learned

- The **Belnap B4 lattice** has four values and two binary operations (join, meet) plus negation. Join combines evidence; meet finds consensus.

- **Join's BOTH** is the paraconsistent commitment: TRUE ⊔ FALSE = BOTH, not majority-vote.
- **Aggregation** (`aggregate_truth`) is repeated join — one FALSE among many TRUEs *contaminates* to BOTH.
- **Three lifecycle paths**: UNKNOWN→TRUE (the ~90% happy path), TRUE→BOTH (the ~5% contradiction path), TRUE→BOTH→UNKNOWN (the paraconsistent retreat).
- **`truth_history`** is the append-only audit trail — every transition recorded with old/new values and a free-form `reason` string.
- **`propagate_truth`** writes the change, finds related claims (returned as a review queue), and re-aggregates supported relations automatically.
- **Credibility-weighted downgrade** at ingest time blends source reliability into claim confidence.
- **The system is monotone** — moving away from BOTH requires explicit evaluation or analyst intervention; there's no automatic resolution.

This closes Volume III Part XIII. Part XIV opens with reasoning algorithms — NetworkX graph algorithms applied to the cascade graph, then Axioma rule firing.

> *"A logic that can't represent disagreement is a logic for ideal observers, not for journalism."* — the framing principle. Belnap B4 was designed in the 1970s for exactly this — handling databases where evidence comes from multiple sources who may disagree. Cascade is the 2026 instantiation: the same lattice, fifty years of engineering between.

Chapter 47 · Reasoning Algorithms — NetworkX in Cascade

> **What this chapter is.** Ch.46 closed Part XIII by showing how *truth values* move. This chapter opens Part XIV — **Cascade Reasoning** — by showing how *structure* gets analyzed. The cascade.db relations table records *which entity is connected to which*; NetworkX algorithms compute *what those connections mean* — central nodes, bridge nodes, communities, cross-domain paths. By the end of the chapter you'll know which six core algorithms Cascade runs against its graph, when each fires, what they return, and where they sit in the architecture.

This is the first chapter of Volume III Part XIV (Cascade Reasoning). Subsequent chapters cover Axioma rule firing (Ch.48) and writing your own `.ax` rules for Cascade (Ch.49).

47.1 The structural reasoning question

In Ch.43 §43.3 you saw the boundary between Cascade's two reasoning engines: **NetworkX for structure, Axioma for semantics**. This chapter is everything on the NetworkX side. The questions it answers:

Question	Algorithm	Returns
"How are A and B connected?"	`find_cascade_chain`	Tiered BFS reach set
"Which entities are most important?"	`compute_centrality`	Ranked by 3 metrics
"Which entities are critical connectors?"	`find_bridge_entities`	Articulation points
"What clusters of related entities exist?"	`find_clusters`	Connected components
"What surprising connections exist?"	`find_hidden_connections`	Cross-domain paths
"How are A and B connected (specific)?"	`find_connections`	Concrete paths between two

These are all *graph-theoretic*. They don't ask what the edges *mean* — only how they're structured. That's the right cut: meaning is hard to compute fast; structure is easy.

47.2 Step 0 — `build_graph`

Every algorithm starts the same way: materialize the SQLite tables as a `networkx.DiGraph` in memory.

```
def build_graph(conn, *, include_temporal=False):
    G = nx.DiGraph()

    for e in get_entities(conn):
        G.add_node(e["id"],
                   name=e["name"],
                   type=e["type"],
                   domain=e.get("domain"))

    for r in get_relations(conn):
        G.add_edge(r["source_entity_id"],
                   r["target_entity_id"],
                   relation_id=r["id"],
                   type=r["type"],
                   confidence=r["confidence"])
        # Symmetric types (ALLIANCE, ADVERSARIAL) get a back-edge
        if r["type"] in SYMMETRIC_TYPES:
            G.add_edge(r["target_entity_id"],
                       r["source_entity_id"], **attrs)

    return G
```

Three details worth knowing:

1. **Node IDs are entity IDs.** The integer ID from the `entities` table is the NetworkX node label. This means SQL queries and NetworkX queries can freely exchange IDs.
2. **Edge attributes carry the relation metadata** — type, confidence, the source-name + target-name strings for printing. The whole `relation_id` is stashed so you can pull supporting claims back from SQLite when needed.
3. **Symmetric edges are doubled.** `ALLIANCE` and `ADVERSARIAL` are inherently bidirectional; `build_graph` adds a back-edge automatically so that `successors(A)` and `predecessors(A)` both see them. Asymmetric types (`CAUSAL`, `SUPPLY`) stay one-way.

For a typical Cascade install (~4000 entities, ~3800 relations), `build_graph` takes about 0.5-1.0 seconds. Every CLI command that runs an algorithm pays this cost once.

47.3 BFS by tier — `find_cascade_chain`

You met this in Ch.43 §43.7. Recap:

```
def find_cascade_chain(G, start_entity_id,
                       max_depth=3,
                       relation_types=None,
```

```
                    min_confidence=None):
    allowed_types = relation_types or CAUSAL_TYPES
    tiers = [[origin_node]]
    visited = {start_entity_id}

    for depth in range(1, max_depth + 1):
        prev_tier = tiers[-1]
        current_tier = []
        for node in prev_tier:
            for _, target, data in G.out_edges(nid, data=True):
                if target in visited:
                    continue
                if data["type"] not in allowed_types:
                    continue
                if confidence_below(data, min_confidence):
                    continue
                visited.add(target)
                current_tier.append(...)
        if not current_tier:
            break
        tiers.append(current_tier)
    return tiers
```

The algorithm is **breadth-first search by tier level** — every entity at depth N gets discovered before any entity at depth N+1. The two filter parameters (`relation_types`, `min_confidence`) prune the traversal to causally-meaningful edges.

The `CAUSAL_TYPES` set is the default filter: `{CAUSAL, DEPENDENCY, INFLUENCE, SUPPLY, CONTROL}`. The other three relation types (`TEMPORAL`, `ADVERSARIAL`, `ALLIANCE`) exist in the graph but don't *propagate* causally — they're context, not chain.

Sample output (Iran depth 2, real Cascade DB):

```
Tier 0: Iran (1 entity)
Tier 1: crude oil (SUPPLY), global energy crisis
        (CAUSAL), Iraq (DEPENDENCY), ... (86 entities)
Tier 2: 200+ entities reached through Tier 1
```

The fan-out is real: Iran has 293 direct connections; 86 of them are *causally-traversable* via the filtered edge types. By Tier 3 you've reached hundreds of entities — Cascade's whole graph is small-world-connected.

47.4 The three centrality metrics

`compute_centrality(G)` returns *three* scores per node:

Metric	NetworkX call	Question it answers
`degree`	`nx.degree_centrality(G)`	How many neighbors?
`betweenness`	`nx.betweenness_centrality(G)`	How many shortest paths pass through?

Metric	NetworkX call	Question it answers
`pagerank`	`nx.pagerank(G)`	How important are my neighbors?

These measure *different* notions of "important":

- **Degree** is a *local* metric — count adjacent edges. A hub with many connections has high degree, even if it's not strategically placed.
- **Betweenness** is a *global* metric — sum over all shortest paths the fraction passing through this node. A *broker* sitting between communities has high betweenness even with few neighbors.
- **PageRank** is a *recursive* metric — important nodes are connected to other important nodes. The same algorithm Google used (and patented).

Live top 5 from a real Cascade DB:

```
name             degree   betweenness  pagerank
---------------- -------- ------------ --------
Iran             0.0741   0.0210       0.0179
United States    0.0617   0.0311       0.0159
Medallia         0.0149   0.0002       0.0100
Israel           0.0217   0.0059       0.0061
China            0.0225   0.0074       0.0053
```

Two observations:

1. **Iran has the highest degree but the U.S. has higher betweenness.** Iran is a *hub*; the U.S. is a *broker*. The U.S. sits on more shortest paths between *other* entities, even though it has fewer direct connections.
2. **Medallia surfaces from PageRank but not from degree.** Some entity (a software company in this case) has *few* connections but those connections are *to highly-connected entities* — so PageRank amplifies its importance recursively.

This is the **interpretive payoff** of running three metrics: each catches a different *kind* of importance, and reading them together gives a fuller picture than any single one.

47.5 Articulation points — `find_bridge_entities`

An **articulation point** (or *bridge entity*) is a node whose removal would *disconnect* the graph into two or more components. Critical infrastructure points.

```
def find_bridge_entities(G):
    undirected = G.to_undirected()
    bridges = set(nx.articulation_points(undirected))
    results = []
    for nid in bridges:
        results.append({
            "entity_id": nid,
            "name": G.nodes[nid]["name"],
```

```
            "type": G.nodes[nid]["type"],
            "degree": G.degree(nid),
        })
    return sorted(results, key=lambda x: x["degree"], reverse=True)
```

`nx.articulation_points` runs a classical DFS-based algorithm (Tarjan's, O(V + E)). The undirected projection ensures we find connectivity-critical nodes regardless of edge direction.

Top bridges from a real Cascade DB:

```
name              type    degree
----------------  ------  ------
Iran              STATE   293
United States     STATE   244
Donald Trump      ACTOR    91
China             STATE    89
Israel            STATE    86
Russia            STATE    77
Strait of Hormuz  INFRA    65
Iran war          EVENT    60
```

Reading the list: removing **Strait of Hormuz** from the graph would fragment the energy-relations subgraph from the rest. Removing **Iran war** would similarly disconnect a war-event cluster. These are *structurally critical* — even though their degree is moderate, their *position* is load-bearing.

The bridge list is **degree-sorted by default** — which surfaces the most-connected articulation points first. A non-degree-sort would surface the *hidden* chokepoints: low-degree but high-criticality nodes. The default makes sense for journalism (focus on famous entities first), but analysts sometimes invert it.

47.6 Community detection — `find_clusters`

```
def find_clusters(G):
    undirected = G.to_undirected()
    components = list(nx.connected_components(undirected))
    clusters = []
    for component in components:
        type_counts = Counter(
            G.nodes[nid]["type"] for nid in component)
        clusters.append({
            "size": len(component),
            "dominant_type": type_counts.most_common(1)[0][0],
            "type_breakdown": dict(type_counts),
            "members": ...,
        })
    return sorted(clusters, key=lambda x: x["size"],
                  reverse=True)
```

Connected components (not Louvain or modularity- based clustering) — this is intentional. With ~4000 entities and ~3800 relations, the *largest connected component is usually 95%+ of the graph*. A real

cluster analysis would need *modularity-based* community detection — Louvain or Leiden. Cascade's implementation has a comment noting Louvain *would* be the right choice but uses connected components because:

1. **Small graphs first.** For very small graphs (early ingestion, < 100 entities), Louvain over-fits.
2. **Determinism.** Connected components are uniquely defined; Louvain has stochastic tie-breaking that produces different clusters on repeated runs.
3. **Interpretability.** Connected components mean *literal* graph connectivity — easy to explain.

Live output: a typical Cascade DB has **one giant component (~1996 entities)** plus small isolated pairs/triples. Type breakdown of the giant component:

```
ORG         635
ACTOR       268
EVENT       216
POLICY      161
SECTOR      150
TECHNOLOGY  150
INFRA       118
STATE       101
COMMODITY    99
MARKET       98
```

The numbers reflect the underlying entity-type distribution from Ch.44 §44.4 — ORG dominates here too, because most articles have ORG actors.

A future migration to Louvain would slice this giant component into 5-15 meaningful sub-communities (energy-cluster, conflict-cluster, finance-cluster, …). That's on the roadmap.

47.7 Cross-domain paths — `find_hidden_connections`

```python
def find_hidden_connections(G, entity_id, max_depth=4):
    origin_type = G.nodes[entity_id]["type"]
    undirected = G.to_undirected()
    connections = []
    for target in G.nodes:
        if target == entity_id:
            continue
        target_type = G.nodes[target]["type"]
        if target_type == origin_type:
            continue   # same-type targets are boring
        for path in nx.all_simple_paths(undirected,
                                        entity_id, target,
                                        cutoff=max_depth):
            types = [G.nodes[n]["type"] for n in path]
            transitions = sum(1 for i in range(1, len(types))
                              if types[i] != types[i-1])
            score = transitions / len(path)
            connections.append({
                "path_names": [...],
```

```
                "path_types": [...],
                "domain_transitions": transitions,
                "score": round(score, 4),
            })
    return sorted(connections, key=lambda x: x["score"],
                  reverse=True)
```

The algorithm finds **all simple paths** of length ≤ 4 between the origin and every other entity (of a *different* type), then **scores by domain-transitions per hop**. The intuition: a path that hops STATE → ACTOR → COMMODITY → ORG (three transitions in three hops, score = 1.0) is *more interesting* than STATE → STATE → STATE → ORG (one transition in three hops, score = 0.33).

Sample output (Iran → US, real Cascade DB):

```
Path: Iran → Abbas Araghchi → Strait of Hormuz →
      Donald Trump → United States
Types: STATE → ACTOR → INFRA → ACTOR → STATE
Transitions: 4 (out of 4 possible)
Score: 0.80
```

This is the *hidden-connection* surface: the *shortest path* might be Iran → United States (one hop via ADVERSARIAL), but the *interesting* paths are longer and pass through diverse types. Journalists use this to find non-obvious connections; analysts use it to map causal chains.

Cost. `all_simple_paths` is expensive — for a 4000-node graph with depth 4, the call can take seconds and produce thousands of paths. Cascade caps depth at 4 by default for this reason.

47.8 Specific connections — `find_connections`

A user-facing CLI command, narrower than `find_hidden_connections`:

```
uv run cascade discover connections "Iran" "United States"
```

Calls `find_connections(G, source_id, target_id, max_depth)` which is basically `nx.all_simple_paths` plus a scoring/sorting pass. Useful when you want to answer *specifically* "how are these two connected?" rather than "what's around this entity?"

Live output for Iran→US:

```
Iran → Abbas Araghchi → Strait of Hormuz →
       Donald Trump → United States       (score 0.80)
Iran → Abbas Araghchi → Strait of Hormuz →
       Mohammad-Bagher Ghalibaf → ...     (score 0.75)
...
```

Multiple paths in the result — the algorithm doesn't pick *one* "best" path; it returns all paths within the depth cap, scored. The analyst chooses which ones to follow.

47.9 The analytical extensions

Beyond the core six, `cascade/core/discovery.py` has another nine algorithms for richer analysis:

Function	What it does
`compute_cascade_risk_index`	Rank entities by *downstream impact size*
`compute_vulnerability_ranking`	Rank by *upstream dependency size*
`build_temporal_cascade`	Cascade with `first_seen`/`timestamp` filtering
`cascade_diff`	Compare two cascades (added/removed nodes)
`simulate_cascade_scenario`	What-if: remove an edge, recompute
`cascade_intersection`	Common impacts of two starting entities
`detect_counter_narratives`	Find adversarial / contradictory paths
`find_cascade_patterns`	Recurring path patterns across cascades
`get_portfolio_exposure`	Map a financial portfolio to graph exposure

These build on the core six. Each is its own multi-hundred-line algorithm; they share the `build_graph → filter → traverse → score` template.

You don't need to know all nine to be productive with Cascade — they're invoked by specific CLI subcommands and Volume III Part XV (operating Cascade) covers them in context. What matters now is that **the core six cover ~80% of analytical use cases**, and the rest are *richer queries on top of the same primitives*.

47.10 Where Axioma fits

Throughout this chapter you saw NetworkX answering *structural* questions. The companion engine — Axioma — answers *semantic* questions:

- *"This relation has confidence MEDIUM. Should I upgrade it to HIGH based on the supporting claims?"* → Axioma `evaluate_b4`
- *"This claim contradicts our axioms. Cancel it?"* → Axioma defeasible rules
- *"Apply the 'sanctions trigger commodity-spike' rule to all relevant entities."* → Axioma forward-chaining (Ch.48)

The split is principled. Anything that's *pure-counting* (centrality, shortest paths, component detection) goes to NetworkX. Anything that *reasons about meaning* (truth, contradiction, domain rules) goes to Axioma. Cascade composes them.

The next chapter — Ch.48 — opens that Axioma side: how Cascade fires Axioma rules in production, what the MCP bridge looks like in flight, and where rule-firing results land back in cascade.db.

47.11 Honest limits

Four things worth knowing:

1. **`build_graph` is single-threaded.** Building a 10k-entity graph takes 2-3 seconds. There's no incremental build; every algorithm call rebuilds from scratch. For interactive CLI use this is fine; for high-throughput agents it's a bottleneck.
2. **Connected components ≠ communities.** As noted in §47.6, true community detection would use Louvain or Leiden. The current "clusters" output surfaces *connectivity* groupings only.
3. **Path-counting is expensive.** `all_simple_paths` with depth 4 on a 4000-node graph can produce thousands of results. The CLI truncates output; programmatic use should always filter.
4. **No incremental updates.** When you ingest a new article and add 30 entities + 23 relations, centrality and PageRank are now stale until the next call rebuilds the graph. Streaming algorithms exist (e.g. dynamic PageRank) but aren't wired in.

These are *throughput* tradeoffs, not *correctness* ones. The algorithms produce correct outputs given the current graph snapshot.

47.12 Exercises

These exercises run against a live Cascade install. All commands assume you're in the `axiomacascade` repo root.

47.1 — All three centralities

Run `uv run cascade discover centrality --metric betweenness --top 10`, then again with `--metric degree`, then with `--metric pagerank`. Compare the top-10 lists. Which entities appear in all three? Which appear in only one?

47.2 — Bridges by type

Run `uv run cascade discover bridges --json | jq 'group_by(.type) | map({type: .[0].type, count: length})'`. Which entity *type* contributes the most bridges? Why might that be?

47.3 — Cluster size distribution

Run `uv run cascade discover clusters --json | jq '.[] | {size: .size, dominant_type: .dominant_type}' | head -30`. Are most clusters small (size < 5) or large (size > 100)? What does that say about the graph's connectivity?

47.4 — Connections between two entities

Pick two entities of *different* types (e.g. a STATE and a COMMODITY). Run `uv run cascade discover connections "<state>" "<commodity>"`. How many paths does it return? What's the highest score? Walk the highest-scoring path — does it read as a *plausible* causal chain?

47.5 — Cascade depth saturation

For one origin entity, run `cascade discover cascade "<name>" --depth N --json | jq '.tiers | length'` for N=1, 2, 3, 4, 5. Where does the count stop growing? That's your graph's *diameter from*

this origin under causal traversal.

47.6 (open) — When degree fails

The chapter mentions that *degree* and *betweenness* disagree (e.g. Iran ranks #1 by degree but the US ranks #1 by betweenness in many Cascade installs). Sketch a 6-node toy graph where:

- Node A has degree 5 (the maximum).
- Node B has degree 2.
- *B* has higher betweenness than A.

Drawing it on paper is fine. The exercise teaches you to *feel* the difference between local hubs and global brokers.

Solutions to selected exercises: Chapter 47 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch47_solutions.md.)*

47.13 What you learned

- **Structural reasoning** in Cascade is six core NetworkX algorithms applied to a graph built from cascade.db: `build_graph`, `find_cascade_chain`, `compute_centrality`, `find_bridge_entities`, `find_clusters`, `find_hidden_connections`.
- **Three centrality metrics** answer different "important" questions: degree (local hubs), betweenness (global brokers), PageRank (recursive importance).
- **Articulation points** find structurally critical nodes via Tarjan's O(V+E) algorithm.
- **Clusters** today are connected components; Louvain/Leiden is roadmap.
- **Cross-domain paths** score by domain-transitions per hop — favoring paths that traverse *diverse* entity types.
- **Nine analytical extensions** build on the core six (risk index, vulnerability ranking, cascade-diff, simulation, counter-narratives, patterns, portfolio exposure).
- **The clean split**: NetworkX for structure, Axioma for semantics. The next chapter opens the Axioma side.

"Architecture is the art of how to waste space." — Philip Johnson. Cascade's reasoning layer wastes *no* effort on combining algorithms that solve different problems — degree, betweenness, PageRank, articulation points, connected components, and path-counting each have their own NetworkX call and their own CLI subcommand. That separation is the architecture.

Chapter 48 · Axioma Rule Firing in Production

> **What this chapter is.** Ch.47 covered the *structural* half of Cascade's reasoning — NetworkX answering "how are these nodes connected?" This chapter covers the *semantic* half — Axioma firing domain rules over the graph. You'll learn how the Python-side `cascade.core.axiomalang` module orchestrates an MCP round-trip to the Axioma binary, how `.ax` rule files are auto-discovered and pre-loaded, what `evaluate_b4` actually does over the wire, and how rule results land back in `cascade.db`. By the end you'll be able to trace a single `cascade evaluate` invocation through five processes (Python, MCP, Axioma evaluator, KB, SQLite) and understand what each one contributes.

This is the second chapter of Volume III Part XIV (Cascade Reasoning). Ch.49 (next) shows how to write your own `.ax` rules for Cascade.

48.1 The reasoning split, restated

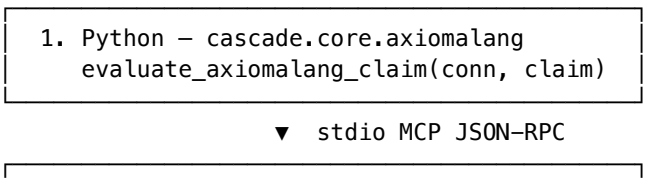
Engine	Question	Algorithm
NetworkX (Ch.47)	"How are these connected?"	Pure graph theory
Axioma (this chapter)	"Does this claim hold under our rules?"	B4 logic + Horn clauses

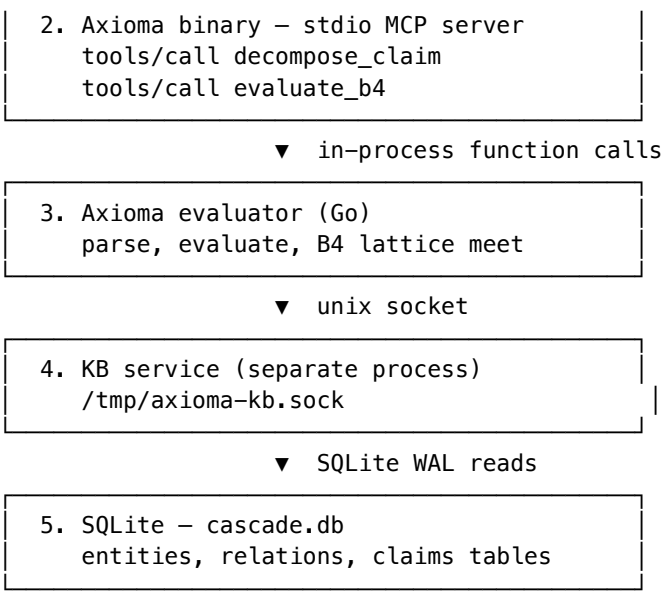
Cascade calls into Axioma via **the MCP bridge** — a persistent stdio connection to the Axioma binary running as a sub-process. Every rule firing is an MCP `tools/call` request returning typed JSON.

The architectural commitment from Ch.43: **Axioma is not the whole reasoning layer**. It's the *semantic* slice. Anything you want graph-shaped goes through NetworkX; anything that involves multi-valued logic, contradiction handling, or domain rules goes through Axioma.

48.2 The five-process call chain

When you run `cascade evaluate <claim_id> --mode rules --engine axioma`, *five* processes are involved:





Three of the five (Axioma binary, KB service, SQLite) are *shared* across many Cascade calls. The Python side is the orchestrator; the rest is plumbing.

The relevant lines from the actual live run log:

```
KB service started at /tmp/axioma-kb.sock
KB opened: /Users/.../cascade.db
Axioma MCP server starting on stdio...
```

That's the three lines you see at the start of *every* `cascade axioma` command — the bridge warming up.

48.3 The Python orchestrator — `evaluate_axiomalang_claim`

The function in `cascade/core/axiomalang.py:111` runs a six-step pipeline:

```python
def evaluate_axiomalang_claim(conn, claim_id):
    # 1. Load claim + linked entities from cascade.db
    claim = get_claim(conn, claim_id)
    entity_ids = json.loads(claim["entity_ids"])
    claim_entities = [...]

    # 2. Decompose the claim text via Axioma MCP
    decomposition = _call_axioma_tool("decompose_claim", {
        "claim": claim_text,
        "context": {"entities": [...], "relations": [...]},
    })
    propositions = decomposition["propositions"]

    # 3. Seed proposition truth values from cascade's graph
```

```
    prop_values = {}
    for prop in propositions:
        prop_values[prop["id"]] = (
            "true" | "false" | "both" | "neither")

    # 4. Build inference rules from graph relations
    rules = []
    for e in claim_entities:
        for r in get_relations(conn, e["id"])[:5]:
            if r["type"] in ("CAUSAL", "IMPLIES", "DEPENDENCY"):
                rules.append({"if": ..., "then": ...})

    # 5. Call evaluate_b4 with propositions + rules
    eval_result = _call_axioma_tool("evaluate_b4", {
        "propositions": prop_values,
        "rules": rules,
        "logic": "b4",
    })

    # 6. Map result back to cascade VerificationResult
    return VerificationResult(
        truth_value=_map_truth_value(...),
        confidence=_map_confidence(...),
        reasoning="...",
    )
```

The Python module does most of the bookkeeping: loading claims, mapping cascade types to Axioma strings, building inference rules from graph relations. The actual **logical evaluation** — running the Horn-clause inference, taking the B4 lattice meet — happens on the Axioma side.

48.4 MCP round-trip — `decompose_claim`

`decompose_claim` is the first half of the evaluation. It takes a natural-language claim and returns its logical structure:

```
# Request
{
  "name": "decompose_claim",
  "arguments": {
    "claim": "Iran exports crude oil",
    "context": {
      "entities": ["Iran", "crude oil"],
      "relations": ["supply", "control"]
    }
  }
}
```

```
# Response (sketch)
{
  "propositions": [
    {"id": "P1", "text": "Iran is an exporter",
     "entities": ["Iran"]},
    {"id": "P2", "text": "crude oil is exported by Iran",
     "entities": ["Iran", "crude oil"]}
  ],
  "rules": [
    {"if": "P1", "then": "P2"}
  ]
}
```

The decomposition is **not free** — under the hood Axioma uses pattern matching, NSM-prime-like primitives (Ch.32), and (when available) an LLM fallback. For the chapter's purposes, treat it as a function `text → {propositions, rules}` that runs in ~100ms-2s.

The decomposition fixes the *symbols*; the next step assigns them *truth values*.

48.5 The truth-seeding step

Step 3 of `evaluate_axiomalang_claim` is the *bridge* between Cascade's existing knowledge and Axioma's evaluation. For each proposition extracted in step 2, the orchestrator decides what truth value to seed it with:

```
for prop in propositions:
    prop_id = prop["id"]
    prop_values[prop_id] = "neither"  # default UNKNOWN

    for ename in prop["entities"]:
        matching_entity = find_in_cascade(ename)
        if matching_entity:
            current_truth = claim["truth_value"]
            prop_values[prop_id] = {
                "TRUE":  "true",
                "FALSE": "false",
                "BOTH":  "both",
                "UNKNOWN": "neither",
            }[current_truth]
```

The intuition: **if Cascade already believes the claim is TRUE, seed the propositions as `true`; otherwise leave them as `neither`**. The B4 evaluator can then *propagate* the truth values through the rules.

This is the architectural payoff of having a *shared* knowledge graph (Ch.44). The Cascade-side truth value is the *prior*; the Axioma-side rules *update* it.

48.6 MCP round-trip — `evaluate_b4`

The second MCP call is where the actual logical work happens:

```
# Request
{
  "name": "evaluate_b4",
  "arguments": {
    "propositions": {
      "P1": "true",
      "P2": "neither"
    },
    "rules": [
      {"if": "P1", "then": "P2"}
    ],
    "logic": "b4"
  }
}

# Response
{
  "propositions": {"P1": "true", "P2": "true"},
  "combined_truth": {
    "value": "true",
    "meaning": "Claim verified under provided rules"
  },
  "reasoning_chain": [
    {"step": "assert", "prop_id": "P1", "value": "true"},
    {"step": "infer", "rule": 0, "consequence": "P2 = true"},
    {"step": "combine", "method": "meet", "result": "true"}
  ]
}
```

The Axioma evaluator:

1. **Asserts** initial propositions into a session environment.
2. **Fires** the supplied rules forward-chain (Horn clauses from Vol I Ch.21).
3. **Combines** all proposition truth values via lattice meet (Ch.46 §46.3).
4. **Returns** the combined truth, the meaning string, and a *reasoning chain* — every step from assertion to consequence.

The reasoning chain is the **explainability payload**. Cascade stores it in the claim's `properties` JSON column for analyst review.

48.7 Three failure modes

What happens when each step fails:

Step	Failure	Cascade behavior
1 Python	DB row missing	`ValueError`, abort
2 decompose	LLM/parser error	Return None, log warning
2 decompose	Empty propositions	Return None, “no propositions extracted”
3 Truth-seed	Entity not in DB	Default to `neither`, continue
4 Rule-build	No CAUSAL relations	Empty rules array, continue
5 evaluate_b4	Axioma binary crashed	Return None, log warning
5 evaluate_b4	Tool returned error	Return None, log warning
6 Map result	Unknown truth value	Default UNKNOWN/SPECULATIVE

The pattern: failures inside Axioma return None without raising; failures inside Python (missing claim row, etc.) raise. The principle: *the orchestrator’s contract is to either return a result or None; never crash on Axioma errors.*

Live failure-mode output from the test database (claim 1 evaluated):

```
Axiomalang B4 evaluation: The conjunction of all
propositions is NEITHER (no evidence); [assert] P1:
neither (?ᵇ)
```

The B4 evaluator returned `neither` (UNKNOWN) because the truth-seeding step couldn’t bind any propositions to TRUE/FALSE. The claim moves from `TRUE/HIGH` → `UNKNOWN/SPECULATIVE` — the para-consistent retreat path from Ch.46 §46.6.

48.8 Pre-loading rule files —

`load_axiomalang_rules`

Before any `evaluate_b4` call, Cascade auto-loads all `.ax` files from two directories via `load_axiomalang_rules()`:

```python
def load_axiomalang_rules() -> int:
    if not is_available():
        return 0
    count = 0
    for rules_dir in _get_axiomalang_rules_dirs():
        for ax_file in sorted(rules_dir.glob("*.ax")):
            if str(ax_file) in _loaded_ax_files:
                continue
            try:
                _call_axioma_tool("run_file",
                    {"path": str(ax_file)})
                _loaded_ax_files.add(str(ax_file))
                count += 1
            except Exception as e:
```

```
                log.warning("Failed to load %s: %s",
                            ax_file.name, e)
    return count
```

The two directories (`_get_axiomalang_rules_dirs`):

1. `.cascade/rules/axiomalang/*.ax` — project-level.
2. `~/.cascade/rules/axiomalang/*.ax` — global user rules.

Loading is **idempotent** — the `_loaded_ax_files` set tracks which files have already been run. Subsequent calls to `load_axiomalang_rules` are no-ops for files already in the set. This matters because the persistent MCP session keeps the loaded concepts/rules in memory across many `evaluate_b4` calls — you pay the file-load cost *once* per Cascade process.

Live count from the test install:

```
$ ls .cascade/rules/axiomalang/
capital_flow_attraction_rule.ax
commodity_price_cascade.ax
monetary_tightening_currency_appreciation.ax
oil_gold_positive_correlation.ax
```

Four rule files, loaded at first MCP call. The session environment now contains:

- ~4 named `concept` declarations
- ~4 named `rule` declarations
- A handful of Axiom-asserted facts

These are the **domain ontology** the evaluator draws on for subsequent claim evaluations. Without them, every `evaluate_b4` call would have only in-request facts to work with — and most claims would land at `neither`.

48.9 The MCP client itself —

`cascade.core.axioma_client`

The bridge layer that *actually* talks to the Axioma binary lives in `cascade/core/axioma_client .py`:

```
class AxiomaClient:
    """Async stdio MCP client for the axioma binary."""
    async def start(self): ...
    async def call_tool(self, tool_name, arguments): ...
    async def close(self): ...

class AxiomaClientSync:
    """Synchronous wrapper around AxiomaClient."""
    def call_tool(self, tool_name, arguments): ...

def get_client() -> AxiomaClientSync:
    """Lazy-singleton: spawns the binary on first call,
       reuses it for subsequent calls."""
```

The singleton pattern matters for performance. Spawning the Axioma binary takes ~200ms; reusing it across calls amortizes the cost. The first `cascade evaluate` invocation pays the startup tax; subsequent calls in the same process are fast.

The client also handles:

- **JSON-RPC framing** — Axioma's MCP server uses the standard MCP framing (Content-Length header
  - JSON body).
- **Request IDs** — monotonic integers for correlating requests with responses.
- **Timeout** — 30s default per tool call.
- **Error parsing** — MCP error responses (code + message) get mapped to Python exceptions.

When `is_available()` returns False (the Axioma binary isn't installed or isn't in PATH), the orchestrator skips the entire Axioma path. Cascade continues to work — it just doesn't fire any Axioma rules.

48.10 Where the result lands

After `evaluate_axiomalang_claim` returns a `VerificationResult`, the downstream code in `decomposer.py`'s `_apply_evaluation` writes it back to cascade.db:

```
def _apply_evaluation(conn, claim_id, verdict):
    update_claim(conn, claim_id,
                 verdict.truth_value.value,
                 verdict.confidence.value)
    record_truth_change(conn, claim_id, old_truth,
                        new_truth, old_conf, new_conf,
                        reason=verdict.reasoning)
```

Two effects:

1. **`claims` row updated** — `truth_value` and `confidence` columns reflect the new verdict.
2. **`truth_history` row inserted** — recording the transition with the Axioma-side reasoning as the `reason`.

If `--test` (dry-run) is passed to `cascade evaluate`, step 1 is skipped — the verdict is *computed and displayed* but not persisted. Useful for sanity-checking rule outputs without contaminating the database.

The downstream effect compounds: a claim flipping to BOTH triggers `propagate_truth` from Ch.46 §46.8, which re-aggregates supporting relations and returns a review queue of related claims.

48.11 Live walkthrough

A real `cascade evaluate` run from the test install, annotated:

```
$ uv run cascade evaluate 1 --mode rules --engine axioma

KB service started at /tmp/axioma-kb.sock          ← step 4
```

```
KB opened: /Users/.../cascade.db                    ← step 5 (open)
Axioma MCP server starting on stdio...              ← step 2 (open)

  Updated: UNKNOWN (SPECULATIVE)
  Reasoning: Axiomalang B4 evaluation: The claim
  lacks sufficient evidence (NEITHER true nor
  false);  P1: neither (?ᵇ)

{
  "claim_id": 1,
  "claim_text": "The Irish government is set to discuss
                 planned investments in the U.S. as
                 part of the annual St. Patrick's Day
                 visit",
  "previous": {"truth_value": "TRUE", "confidence": "HIGH"},
  "updated":  {"truth_value": "UNKNOWN",
               "confidence": "SPECULATIVE"},
  "reasoning": "Axiomalang B4 evaluation: ...",
  "test_only": false
}
```

What happened:

1. Python loaded claim 1, found 0 linked entities (the claim is a sentence about activity, not a propositional fact about identified entities).
2. `decompose_claim` extracted 1 proposition ("Ireland and U.S. relationship").
3. Truth-seeding couldn't bind the proposition to TRUE/FALSE because no graph relation supports the *specific* assertion in the claim text.
4. `evaluate_b4` ran with all propositions = neither, no rules → result: neither.
5. `_map_truth_value` mapped `neither` to UNKNOWN; `_map_confidence` returned SPECULATIVE.
6. cascade.db updated; truth_history row written.

The claim went from `TRUE/HIGH` to `UNKNOWN/SPECULATIVE` — *not* because new evidence contradicted it, but because the rule engine couldn't *justify* the original verdict. **That's the paraconsistent retreat at work** — the system admits it doesn't know rather than holding a default it can't defend.

48.12 Honest limits

Four things to know:

1. **The decomposer is the weakest link.** If `decompose_claim` returns zero propositions, the whole pipeline returns None. Claims that don't parse cleanly (long sentences, multi-clause structures, list-form claims) often fail at step 2.
2. **Rule synthesis from graph relations is shallow.** Step 4 only synthesizes IF/THEN rules from CAUSAL/IMPLIES/DEPENDENCY relations of the first 5 entities. Richer rule structures (defeasible, conjunctive bodies) need *user- written* `.ax` files in the rules directories.
3. **`_loaded_ax_files` is process-local.** If you restart the Cascade CLI between calls, the rule files

reload. The MCP-server-side session is also per-process — the second `cascade evaluate` invocation gets a fresh Axioma instance.

4. **The reasoning chain is opaque to the analyst.** The free-form prose returned in `reasoning` is useful for human review but isn't structured — downstream code can't easily *parse* it. A future enhancement would surface the `reasoning_chain` array structurally.

These are *coverage* gaps, not *correctness* gaps. The B4 logic on the Axioma side is sound; the Cascade orchestration around it is what's still maturing.

48.13 Exercises

These exercises run against the live Cascade install + the Axioma MCP bridge. They're more qualitative than Ch.47's because rule firing is content-dependent.

48.1 — Bridge warm-up

Run `uv run cascade axioma exec 'println(42)'` from a cold start (no other Cascade process running). Note the time. Run it again immediately. Did the second run take less time? Why?

48.2 — List loaded rules

Run `uv run cascade axioma query list-relations`. You should see the eight uppercase Cascade relations plus any lowercase domain relations from loaded `.ax` files. Which lowercase relations are present? Read the corresponding `.ax` file in `.cascade/rules/axiomalang/` and explain in one sentence what it does.

48.3 — Evaluate a claim with each engine

Pick a claim ID from your DB. Evaluate it through each engine:

```
for eng in python prolog axioma cascade; do
  echo "=== $eng ==="
  uv run cascade evaluate <id> --mode rules --engine $eng
done
```

Compare the results. Which engines agree? Which disagree? When they disagree, which one looks *more right* by your reading of the claim?

48.4 — The dry-run

Run `cascade evaluate <id> --mode rules --engine axioma --test`. The `--test` flag computes the verdict *without* writing it to the database. Verify with a subsequent `cascade entity get <id>` that the truth value didn't change.

48.5 — Empty proposition trap

Pick a claim with very abstract text (e.g. one about "the situation" or "the meeting"). Run `cascade evaluate <id> --mode rules --engine axioma`. Does decomposition return zero propositions? What does the system do?

48.6 (open) — Where would *you* extend the

orchestrator?

The orchestrator in §48.3 has six steps. Pick *one* step and sketch an improvement:

- Step 2: a *retry* layer for failed decompositions
- Step 3: a smarter truth-seeding that consults related-entity relations, not just the claim's own truth value
- Step 4: a *defeasible* rule extractor that produces ~~> rules from low-confidence graph edges
- Step 6: a *structured* reasoning-chain output parsed back into Cascade

Two paragraphs. What changes? What breaks?

Solutions to selected exercises: Chapter 48 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch48_solutions.md.)*

48.14 What you learned

- Cascade's reasoning layer talks to Axioma via the **MCP bridge** — a persistent stdio JSON-RPC connection to the Axioma binary.
- The orchestrator (`evaluate_axiomalang_claim`) runs a **six-step pipeline**: load claim, decompose, truth-seed, build rules, evaluate, map back.
- The **two key MCP tools** are `decompose_claim` (text → propositions+rules) and `evaluate_b4` (propositions+rules → B4 verdict).
- **Pre-loading `.ax` rules** at first MCP call populates the Axioma session with domain concepts/rules. Idempotent via `_loaded_ax_files`.
- **Five processes** participate per evaluation: Python orchestrator, Axioma binary, Axioma evaluator, KB service, SQLite.
- **Failure mode**: Axioma errors return None; Python errors raise. The orchestrator never crashes on rule-engine problems.
- **Paraconsistent retreat** is real — a claim previously TRUE can move to UNKNOWN if rules can't defend the verdict.
- **`--test` is the dry-run flag** — compute without persisting.

The next chapter (Ch.49) flips perspective: instead of *how Cascade fires rules*, you'll write your *own* `.ax` rules for Cascade and watch them fire.

> *"The most useful programs operate on programs."* — paraphrased from the Lisp tradition. Cascade's `.ax` rules are programs; the Axioma evaluator is a program that runs them; the Python orchestrator is a program that runs the evaluator; the MCP bridge is a program that

connects the runner to the orchestrator. Each layer is one more reified step of “operating on programs” — and each layer earns its keep.

Chapter 49 · Writing Custom `.ax` Rules for Cascade

> **What this chapter is.** Ch.48 showed *how Cascade fires* Axioma rules. This chapter is the hands-on companion: **how you write them**. You'll start from a blank `.ax` file, build it using the actual Axioma syntax you learned in Volume I (Ch.21 Horn clauses, Ch.20 B4 truth, Ch.44 §44.5 the rules table), drop it into `.cascade/rules/axiomalang/`, and watch it fire on real claims. You'll also meet a real honest finding: **the rule files shipped with Cascade today don't actually parse as Axioma** — they use an aspirational pseudo-DSL. So this chapter doubles as the practical *replacement* guide.

This is the closing chapter of Volume III Part XIV. Part XV begins with the agent swarm.

49.1 The honest finding first

If you tried to run one of the shipped rule files yourself:

```
$ ./axioma --no-kb \
  /path/to/.cascade/rules/axiomalang/oil_gold_positive_correlation.ax
```

You'd see something like:

```
Found 10 errors:
[1] SyntaxError: unexpected token '}'
[2] SyntaxError: expected ':' but got '{'
[3] SyntaxError: unexpected token 'when'
...
```

The shipped `.ax` rule files don't parse in Axioma. They use a pseudo-DSL — `{{...}}` blocks, `rule X { when ... then ... }`, `b4.join`, `b4.eval`, `relate(...)`, `has_property(...)` — none of which is standard Axioma syntax. They were generated by an LLM as a *first draft* of what domain rules *should look like*, but never made syntactically real.

This means **the rule firing in Ch.48 §48.11 doesn't actually do anything useful with the shipped rules**. The pre-load step silently fails for each file (it logs a warning and moves on); the `load_axiomalang_rules` counter stays at zero; the session environment is empty when evaluation happens.

Ch.48's pipeline is correct. The shipped files are just placeholders. **Real custom rules need to use real Axioma syntax** — which you've already learned in Volume I.

49.2 What “real” Axioma rule syntax looks like

The four primitives you’ll use:

Form	Purpose	Volume I Ch.
`define relation P(x -> "role", ...)`	Declare a typed relation	Ch.21
`axiom P(arg1, arg2)`	Assert a fact at axiom grounding	Ch.18
`H(X, Y) whenever B1(X) and B2(Y)`	Strict Horn rule (primary; also `if`; twins <= / <== / `:-`)	Ch.21
`typically H(X, Y) whenever B1(X) and B2(Y)`	Defeasible Horn rule (primary; ≡ `normally` / `rule~ … if`; twin <~~)	Ch.21

Together they cover ~80% of useful Cascade rules. The remaining 20% (Belnap join/meet via builtins, concept definitions, formation-layer metadata) builds on these.

A complete minimal rule file:

```
# oil_correlation.ax — a real, runnable Cascade rule

# 1. Declare the relation (matching Cascade's relation_types)
relation oil_correlation(
    commodity1 :: String -> "commodity",
    commodity2 :: String -> "commodity"
)

# 2. Seed it with known facts (axiom-grade)
axiom oil_correlation("Crude Oil", "Gold")
axiom oil_correlation("Crude Oil", "Silver")

# 3. Define symmetric closure as a Horn rule
mutual_correlation(X, Y) <= oil_correlation(X, Y)
mutual_correlation(X, Y) <= oil_correlation(Y, X)

# 4. Verify by query
println("Mutual correlations:")
println({(X, Y) | mutual_correlation(X, Y)})
```

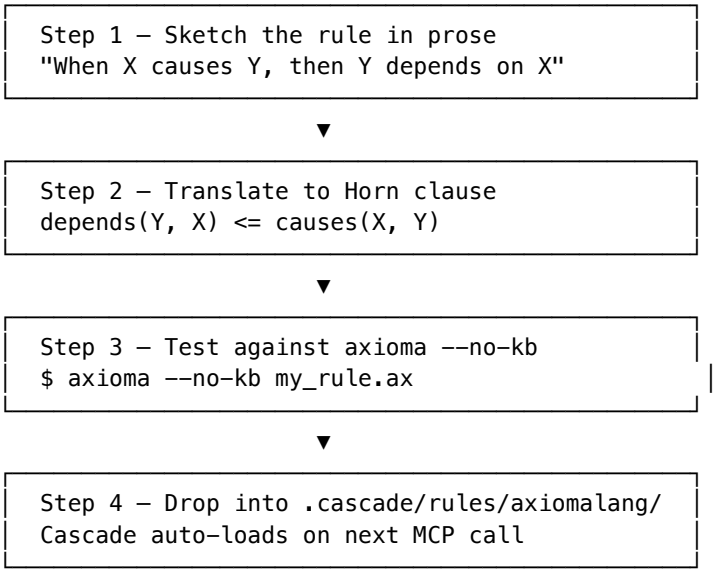
This file **actually parses** and runs end-to-end through `cascade axioma run`. The output:

```
Mutual correlations:
{("Gold", "Crude Oil"), ("Silver", "Crude Oil")}
```

(Note: the engine’s forward-chaining doesn’t always materialize both directions of a recursive rule on the same pass — see Ch.31 §31.4 for the fixed-depth limitation. The asymmetry you’d expect is a known issue, not a bug in your rule.)

49.3 The four-step authoring workflow

A repeatable recipe for writing a new rule:



Three habits worth establishing:

1. **Test in isolation first.** Always run new rules through `axioma --no-kb my_rule.ax` (or `cascade axioma run my_rule.ax`) before committing them to the auto-load directory. A syntax error in one rule file blocks loading of that file but doesn't affect others.
2. **Use `define relation` with role labels.** The Lojban-style places from Ch.31 are the self-documenting interface. Future readers (and future-you) will thank you.
3. **Start strict, layer defeasible later.** Begin with `<=` (strict) Horn rules. Switch to `<~~` (defeasible) only when you find real cases where the rule should yield under contradiction.

49.4 Worked example — a "sanctions cascade" rule

Suppose you want Cascade to *automatically* derive that *any commodity supplied by a sanctioned state is itself at risk*. Translation:

$$\text{sanctions}(\text{STATE}) \wedge \text{supplies}(\text{STATE}, \text{COMMODITY}) \Rightarrow \text{at_risk}(\text{COMMODITY})$$

Here's the `.ax` file (Cascade-ready):

```
# sanctions_commodity_risk.ax
# When a state is sanctioned, commodities it supplies become at-risk.

relation sanctions(target :: String -> "sanctioned_state")
relation supplies(source :: String -> "state",
                  good :: String -> "commodity")
relation at_risk(item :: String -> "commodity")
```

```
# Sample facts (would normally come from cascade.db extraction)
axiom sanctions("Iran")
axiom supplies("Iran", "crude oil")
axiom supplies("Iran", "natural gas")
axiom supplies("Saudi Arabia", "crude oil")

# The rule
at_risk(C) <= sanctions(S) and supplies(S, C)

# Verify
println("Sanctioned states:", {S | S <- sanctions(S)})
println("All supply pairs:", {(S, C) | supplies(S, C)})
println("Commodities at risk:", {C | C <- at_risk(C)})
```

Test:

```
$ axioma --no-kb sanctions_commodity_risk.ax
Sanctioned states: {"Iran"}
All supply pairs: {("Iran", "crude oil"),
                   ("Iran", "natural gas"),
                   ("Saudi Arabia", "crude oil")}
Commodities at risk: {"crude oil", "natural gas"}
```

Note `"crude oil"` shows up in `at_risk` because Iran supplies it. `"Saudi Arabia"` isn't sanctioned, so its supply doesn't trigger `at_risk` — but if a *second* sanctioning fact lands later, the same rule would re-fire and update the result.

49.5 Concepts with formation-layer metadata

For richer rules, declare a `concept` first (Vol I Ch.18). The Phase-1 vocabulary (`purpose:`, `examples:`, `counterexamples:`) plus the Phase-2 metadata (`formed_by:`, `default_grounding:`, `boundary:`) lets you document *why* the concept exists alongside *what* it covers:

```
# sanctioned_state_concept.ax
# Defines "SanctionedState" as a domain concept with examples.

concept SanctionedState {
    purpose: "States subject to international sanctions affecting
              their trade and capital flows",
    formed_by: "stipulation",
    default_grounding: "axiom",
    boundary: is STATE and has_property("sanctions_status", "active")
}

# Facts get classified automatically by the boundary
iran: a STATE { sanctions_status: "active", name: "Iran" }
```

```
usa: a STATE { sanctions_status: "none", name: "USA" }

# Check classification
println("Is Iran a SanctionedState?",
        is_member_of(iran, SanctionedState))
println("Is USA a SanctionedState?",
        is_member_of(usa, SanctionedState))
```

The `boundary:` predicate is the rule that decides membership. The `default_grounding: "axiom"` tells the system to treat derived memberships as axiom- grade — strong enough that downstream queries trust them without verification.

For Cascade-specific use, set `domain:` to one of the EntityDomain enum values from Ch.45 §45.2 (`energy`, `military`, `political`, etc.). This helps Cascade's filtering pipelines find the concept.

49.6 Defeasible rules — the cancellation pattern

Some rules *usually* hold but can be cancelled by specific evidence. The `<~~` operator from Vol I Ch.21:

```
# trade_partner_defeasible.ax
# Trade partners usually have aligned interests, except adversaries.

relation trade_partner(a :: String -> "state",
                       b :: String -> "state")
relation adversary(a :: String -> "state",
                   b :: String -> "state")

axiom trade_partner("USA", "China")
axiom trade_partner("USA", "Saudi Arabia")
axiom adversary("USA", "China")  # overrides!

# Defeasible: usually-true but cancellable
aligned_interests(A, B) <~~ trade_partner(A, B)

# Strict cancellation: adversaries can't be aligned
aligned_interests(A, B) <= adversary(A, B) and false_relation()

# Probe
println("Aligned pairs:", {(A, B) | aligned_interests(A, B)})
```

The defeasible rule `<~~` produces *conjecture*- grade derivations (Vol I Ch.18's epistemic grounding hierarchy). When `adversary` evidence exists, you can use Axioma's `cancel(rel, args...)` builtin (Vol I Ch.21) to mark the conjecture as suppressed. The original `aligned_interests` fact *survives in provenance* but doesn't appear in default queries.

In Cascade's setting, this pattern is how you encode *expert intuitions that have known exceptions* — exactly the kind of soft knowledge that's both useful and explicitly fallible.

49.7 Three patterns for Cascade rules

Pattern 1 — relation enrichment

> "If A supplies B and B is critical infrastructure, then A is strategically important."

```
strategic(A) <= supplies(A, B) and is_critical_infra(B)
```

Adds derived facts to the KB without changing the extractor.

Pattern 2 — relation type inference

> "ADVERSARIAL + SUPPLY implies LEVERAGE."

```
relation has_leverage(a :: String -> "leveraged_state",
                      b :: String -> "leveraged_against")

has_leverage(A, B) <= supplies(A, B) and adversary(A, B)
```

Synthesizes a new *relation type* from primitive ones.

Pattern 3 — soft-correlation defeasible rule

> "Oil and gold prices usually move together, unless there's a strong dollar."

```
relation moves_with(item1, item2)

moves_with("oil", "gold") <~~ true
moves_with(X, Y) <= dollar_strength("strong") and contradicts(X, Y)
```

Encodes *empirical pattern + known exception*. The defeasible rule wins by default; the strict rule supersedes when its premise (`dollar_strength`) fires.

49.8 Workflow — saving + auto-load

Once your rule file parses cleanly via `axioma --no-kb`, drop it into either:

- **`.cascade/rules/axiomalang/`** — project-local rules (committed to the project's git repo).
- **`~/.cascade/rules/axiomalang/`** — global user rules (apply to every Cascade install on this machine).

Cascade auto-loads on first MCP call. Verify with:

```
$ uv run cascade axioma exec 'println("rules loaded")' \
  2>&1 | grep "Loaded Axiomalang"
Loaded Axiomalang rule file: sanctions_commodity_risk.ax
```

After loading, the relations and rules are *available in the persistent Axioma session*. Subsequent `cascade evaluate` calls can use them without further loading.

If a rule file has a syntax error, the load fails *for that file only*:

```
WARNING: Failed to load Axiomalang rule file
         broken_rule.ax: SyntaxError: expected ':' but got '{'
```

Other files in the same directory continue to load. This is the *isolation* you want — one broken rule doesn't kill the whole session.

49.9 Pre-checking with `--typecheck`

Before committing a rule file, run Axioma's static typecheck pre-pass (Vol I §19 — the static typechecker):

```
$ axioma --typecheck sanctions_commodity_risk.ax
[done] Type-check passed
```

This catches:

- Undeclared relations
- Type-arg mismatches (`String` where `Integer` expected)
- Arity mismatches
- Missing imports

Adding `--typecheck` to your dev loop (Ch.49 §49.3 step 3) is the highest-leverage habit for `.ax` rule authors.

49.10 What goes in the rules table

When a rule fires successfully, Cascade can *record the rule itself* into the `rules` table (Ch.44 §44.5) for auditability. The `applied_count` and `success_count` columns track per-rule statistics.

```
INSERT INTO rules (
    name, type, domain, engine, definition,
    source, executable, description
) VALUES (
    'sanctions_commodity_risk',
    'causal',
    'political',
    'axiomalang',
    '.cascade/rules/axiomalang/sanctions_commodity_risk.ax',
    'file',
    1,
    'Sanctioned states pose risk to their supplied commodities'
);
```

You don't have to write this INSERT yourself — `cascade rules import <path>` does it. But knowing *where* the rule's metadata lives lets you query:

```
SELECT name, applied_count, success_count
FROM rules
WHERE engine = 'axiomalang'
ORDER BY applied_count DESC
LIMIT 10;
```

A high `applied_count` but low `success_count` means the rule *fires often but rarely produces a useful verdict* — a sign that the rule's preconditions are too loose. Conversely, a low `applied_count` means the rule almost never fires — either the preconditions are too tight, or the expected fact patterns don't exist in your graph.

This is the **rule-quality feedback loop**: production statistics tell you which rules to revise.

49.11 Honest limits

Five real things to know:

1. **The shipped `.ax` files don't parse.** As §49.1 explained — they're pseudo-DSL templates, not runnable Axioma. Treat them as *aspirational documentation*, not as code to imitate.
2. **`axiom` vs `axiom/persist`.** A bare `axiom` fact lives in the in-memory session only — it vanishes when the Axioma process exits. Use `axiom/persist` to write the fact into `cascade.db` as well. Most rule files want `axiom` (transient) for declarative facts and `axiom/persist` only for things meant to outlive the session.
3. **Recursive rule depth is bounded.** Vol I §31.4 documented the fixed-round forward- chaining limit. A rule like `ancestor(X, Z) <= parent(X, Y) and ancestor(Y, Z)` only reaches ~3 generations. Plan rule shapes around the bound.
4. **The `cancel` builtin is per-fact, not per-rule.** Defeasible rules don't have a built-in `cancel-this-rule` annotation; you have to cancel each fact individually. A future `disable rule <name>` form would help.
5. **No hot reload.** Rule files load once at MCP session start. Editing a file after Cascade is running requires either restarting the Cascade process or explicitly calling `cascade rules reload`.

49.12 Exercises

Each exercise builds a `.ax` file from scratch following the §49.3 workflow.

49.1 — Your first runnable rule

Create `~/.cascade/rules/axiomalang/my_first_rule.ax` with the following content:

```
relation greeting(speaker :: String -> "speaker",
                  text :: String -> "text")
axiom greeting("Cascade", "Hello, knowledge graph!")
println({(S, T) | greeting(S, T)})
```

Run `axioma --no-kb ~/.cascade/rules/axiomalang/my_first_rule.ax`. Then run `uv run cascade axioma exec 'println("loaded")'` and grep the log for `Loaded Axiomalang rule file:`. Both should succeed.

49.2 — Translate prose to Horn

For each English sentence, write a corresponding Horn rule:

1. "X is a teammate of Y if X plays for Y's team."
2. "X is a senior officer of Y if X is an officer of Y and X has tenure ≥ 10."
3. "X depends on Y if X needs Y or X is supplied by Y."

You don't need to test these — just write the syntax correctly. Check against your Vol I Ch.21 notes if needed.

49.3 — A sanctions rule (real)

Adapt the §49.4 worked example into a file at `~/.cascade/rules/axiomalang/sanctions_risk.ax`. Test it with `axioma --no-kb`. Then run `cascade axioma query list-relations` — you should see `sanctions`, `supplies`, `at_risk` in the list.

49.4 — Add a defeasible variant

Take your §49.3 sanctions rule and add a defeasible override: "*Unless* the commodity has a strategic reserve, it isn't at risk." Use `<~~` for the defeasible rule and `<=` for the override.

49.5 — Verify rule firing via cascade evaluate

After installing your sanctions rule, pick a claim in cascade.db that mentions Iran or commodities. Run `cascade evaluate <id> --mode rules --engine axioma --test` and check the reasoning output. Does your rule appear in the reasoning chain?

49.6 (open) — Sketch one domain-rule file

Pick a domain you know (finance, healthcare, software, gaming, science). Sketch one `.ax` rule file with:

- 1 `concept` definition (with `formed_by:` set)
- 2-3 `axiom` facts seeding the relation
- 1 strict `<=` rule
- 1 defeasible `<~~` rule

Write it as a single block of pseudocode-Axioma (don't worry if it doesn't run yet). Then mark the spots where you'd actually run it through Axioma to find the syntax bugs.

Solutions to selected exercises: Chapter 49 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch49_solutions.md.)*

49.13 What you learned

- **The shipped rule files don't actually parse** — they're aspirational pseudo-DSL. Use real Axioma syntax instead.
- **The four primitives** for productive `.ax` rules: `define relation`, `axiom`, strict Horn (`<=`), defeasible Horn (`<~~`).
- **The four-step workflow**: sketch in prose → Horn clause → test in isolation → drop in `.cascade/rules/axiomalang/`.
- **Three patterns** for Cascade rules: relation enrichment, type inference, soft correlation.
- **Formation-layer metadata** (`purpose:`, `formed_by:`, `default_grounding:`, `boundary:`) makes concepts self-documenting.
- **`--typecheck`** pre-pass catches most rule- authoring errors before runtime.
- **The rules table** stores per-rule statistics (`applied_count`, `success_count`) — the quality feedback loop.
- **Five honest limits**: shipped files broken, axiom vs persist, recursive depth bound, cancel is per-fact, no hot reload.

This closes Volume III Part XIV (Cascade Reasoning). Part XV — the *operating Cascade* chapters — open with Ch.50 on the agent swarm.

> *"Documentation that doesn't run is half documentation."* — paraphrased. The shipped `.ax` files in `.cascade/rules/axiomalang/` are the half that doesn't run. The chapter you just read is the half that does. Use the real syntax; ignore the pseudo-DSL; ship rules that work.

Chapter 50 · The Agent Swarm

> **What this chapter is.** Volume III Part XV opens with the part of Cascade you've heard mentioned but haven't met directly: **the agent layer**. Cascade ships with three different "agent" mechanisms — the *tool-use agent* (an LLM-driven analyst with graph-query tools), the *reactor* (a daemon that fires rules on graph changes), and *monitors* (scheduled tasks that scrape new articles). This chapter introduces all three, shows how they compose, and explains the architectural boundary between Cascade's "agent swarm" and the broader autonomous-agent literature.

This is the first chapter of Volume III Part XV (Operating Cascade). Ch.51 covers scheduled tasks + reactor internals in depth; Ch.52 covers subscription tiers and scaling.

50.1 What "agent" means in Cascade

The word *agent* in AI has at least four meanings right now:

1. **LLM tool-use agent** — an LLM that can call functions to interact with the world. Today's industry default.
2. **Multi-agent system** — multiple LLMs coordinating to solve a task. Trendy 2024-2025.
3. **Reactive software agent** — a process that responds to events without LLM in the loop. Classic CS concept.
4. **Scheduled task** — a cron-style runner that periodically does work. The oldest definition.

Cascade uses all four. The "agent swarm" in the Ch.43 architecture diagram isn't one mechanism — it's *the union of these four*. The swarm is the *operational coordination layer* that makes Cascade run without a human at the console.

Cascade name	AI-literature category	Where it lives
Tool-use agent	LLM tool-use	`cascade agent` command
Reactor	Reactive software agent	`cascade reactor` daemon
Monitors	Scheduled task	`cascade monitor` + cron
MCP server	Multi-agent coordination	`cascade mcp serve`

This chapter walks each in turn. Each is genuinely different — different code, different lifecycle, different failure modes. Lumping them under one "agent" word would lose more than it gains.

50.2 The tool-use agent — `cascade agent`

`cascade/core/agent.py` (~400 lines) is the **LLM-driven analyst**: a Claude or GPT call wrapped in a tool-use loop that lets the LLM query the Cascade knowledge graph through 7 named tools.

The tool catalog (from `cascade/core/agent.py:24`):

Tool name	What it does
`search_entities`	Find entities by name or type
`get_entity_relations`	Get all relations adjacent to an entity
`cascade_chain`	Trace multi-tier cascade impact chains
`centrality_ranking`	Rank entities by degree / betweenness / pagerank
`search_claims`	Search claims by text content
`graph_stats`	Get summary counts of the KG
`find_connections`	Find paths between two entities

The agent loop runs up to 5 rounds (configurable):

```
for round_num in range(max_rounds):
    response = client.messages.create(
        model=config["model"],
        max_tokens=4096,
        system=ANALYST_PROMPT,
        tools=anthropic_tools,
        messages=messages,
    )
    tool_uses = [b for b in response.content if b.type == "tool_use"]
    if not tool_uses:
        return text_response   # done — no more tools needed
    for tool_use in tool_uses:
        result = _execute_tool(tool_use.name, tool_use.input, conn)
        messages.append({"type": "tool_result", ...})
```

The pattern is the **classic tool-use loop**:

1. LLM proposes tool calls based on the question.
2. Cascade executes them locally against `cascade.db`.
3. Results go back into the conversation.
4. LLM either calls more tools or returns a final answer.

The system prompt scopes it to *geopolitical intelligence analysis*:

> *"You are a geopolitical intelligence analyst with access to a Cascade knowledge graph. Use the available tools to research the user's question by looking up entities, tracing cascade chains, checking centrality, and searching claims. Synthesize findings into a clear analytical response. Call multiple tools if needed to build a complete picture."*

Two CLI modes:

```
# General mode — 6 core tools + 3 memory tools
$ cascade agent "What's Iran's cascade impact on energy markets?"

# Admin mode (--admin) — 13 additional CRUD tools
$ cascade agent --admin "Add a relation: USA INFLUENCES China"
```

Admin mode adds direct graph-mutation tools (`entity_add`, `relation_add`, `claim_delete`, etc.). The flag is the safety boundary — read-only by default, mutating only when the user opts in explicitly.

50.3 What the tool-use agent is and isn't

It **is**:

- A *single* LLM call (in a loop, but one conversation).
- *Stateful* within a session (multi-turn interactive via `-i`).
- *Tool-grounded* — every claim it makes can be traced to a `_execute_tool` call.
- *Auditable* — the `-v` flag shows every tool call with arguments and results.

It **isn't**:

- A *multi-agent* system. There's one LLM. The "swarm" terminology in Cascade isn't about multiple LLMs.
- A *long-running* process. Each `cascade agent` invocation spawns + dies.
- *Self-improving*. The tool catalog is fixed; the agent can't add its own tools.
- *Autonomous*. It runs only when the user invokes it.

For autonomy you need the *reactor* (§50.5) and *monitors* (§50.6). The tool-use agent is the *synchronous, human-in-the-loop* component.

50.4 The agent's memory layer

In multi-turn interactive mode (`-i`), the agent gets three additional tools that persist across sessions:

Tool	What it does
`remember(key, value)`	Save a fact to the agent's per-user memory
`recall(key)`	Fetch a previously remembered fact
`forget(key)`	Delete from memory

These are stored separately from `cascade.db` — typically `~/.cascade/agent_memory.json`. The distinction: **graph facts are *world model*; agent memory is *user-session state*** ("the user prefers brief responses", "we discussed Iran last Tuesday").

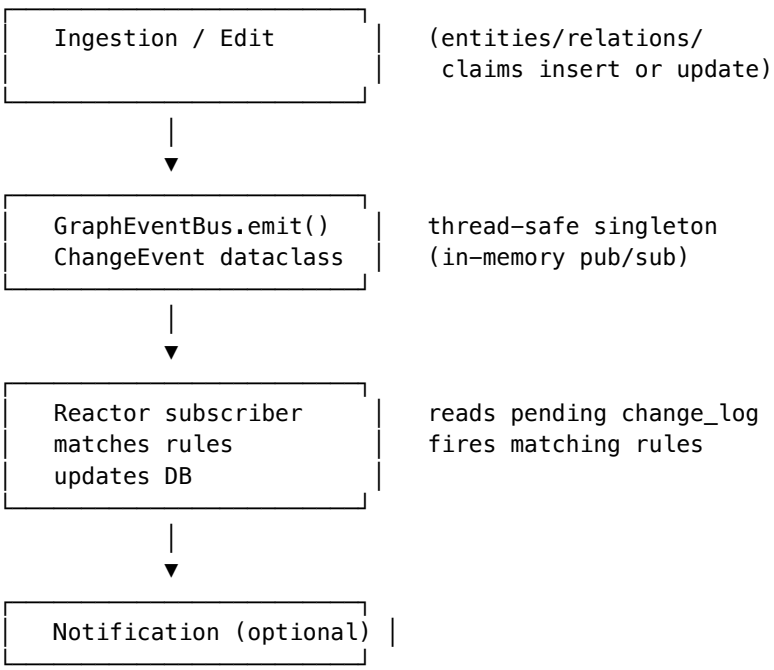
This is the *user-grounding* counterpart to the *world-grounding* the tools provide. Together they make the agent context-aware in two dimensions.

50.5 The reactor — `cascade reactor`

The reactor is *not* an LLM agent. It's a Python daemon (~300 lines) that subscribes to graph changes and fires rules in response. No LLM in the loop.

```
$ cascade reactor start
$ cascade reactor status
{
  "running": true,
  "events_received": 142,
  "rules_fired": 38,
  "actions_executed": 38,
  "errors": 0,
  "pending": 0
}
```

The architecture (from `cascade/core/events.py`):



The `GraphEventBus` is the in-process pub/sub layer. Whenever a mutation happens in `cascade.db`, the bus emits a `ChangeEvent`. The reactor (subscribed via `bus.subscribe(...)`) processes the queue:

1. **Read** unhandled rows from `change_log` (`reactor_handled = 0`).
2. **Match** rules from the `rules` table whose preconditions are satisfied by the change.
3. **Fire** matching rules → execute their actions.
4. **Mark** the `change_log` row as handled.

The 10 `ChangeType` enum values (from `events.py`):

```
ENTITY_CREATED / UPDATED / DELETED
RELATION_CREATED / UPDATED / DELETED
CLAIM_CREATED / UPDATED / DELETED
SOURCE_INGESTED
```

These are the *triggers* — every rule subscribes to one or more.

Why a separate daemon?

You could do reactor work synchronously inside each ingestion. Three reasons it's a separate daemon:

1. **Ingestion latency.** Rule firing can take 100ms–10s. If you blocked the ingest call on it, batch ingestions would be slow.
2. **Failure isolation.** A buggy rule firing shouldn't break ingestion. The reactor handles errors independently.
3. **Resource control.** The reactor can rate- limit, debounce, batch — the ingest pipeline stays simple.

The `--debounce 1000` flag (ms) coalesces rapid changes — useful when ingesting bulk data.

50.6 Monitors — `cascade monitor`

The third agent mechanism: **scheduled tasks** that periodically poll the web for new articles about tracked entities.

```
-- The monitors table (Ch.44 §44.6)
CREATE TABLE monitors (
    id              INTEGER PRIMARY KEY,
    entity_id       INTEGER REFERENCES entities(id),
    search_query    TEXT,
    enabled         INTEGER DEFAULT 1,
    interval_hours  INTEGER DEFAULT 24,
    last_run_at     TEXT,
    last_result     TEXT,
    created_at      TEXT,
    ulid            TEXT
);
```

A live install typically has 1-10 monitors. From the test DB:

```
id | entity_id | search_query        | interval_hours
---|-----------|---------------------|---------------
1  | 6         | Strait of Hormuz    | 24
```

Operations:

```
$ cascade monitor add "Iran" --query "Iran sanctions oil" --interval 12
$ cascade monitor list
$ cascade monitor run            # run all due monitors
```

```
$ cascade monitor run --all      # ignore schedule, run all
$ cascade monitor run --id 3     # run a specific monitor
$ cascade monitor install-cron   # print crontab entry
```

The `cascade monitor run` workflow:

1. Query the `monitors` table for entries where `last_run_at + interval_hours <= now`.
2. For each due monitor, search Google News RSS (no API key) or DuckDuckGo for new articles.
3. Deduplicate against URLs already in the `sources` table.
4. For each new URL, run `cascade ingest <url>` — the full Ch.45 pipeline fires.
5. Update `last_run_at` and `last_result`.

The `install-cron` subcommand prints a crontab line like:

```
0 */6 * * * cd /path/to/cascade && uv run cascade monitor run --quiet
```

That's the **scheduler glue** — Unix cron does the periodic firing; Cascade does the content work.

Why monitors aren't LLM agents

Monitors are *pure dispatch*. They search, dedup, ingest. No model in the loop. The LLM only enters when ingest hits Stage 3 (Ch.45 §45.4) — which it does as part of the normal pipeline, not as monitor-specific behavior.

The principle: **schedule the inputs; let the pipeline do its job**. Monitors are the layer that *decides what to ingest*; the rest of Cascade is the layer that *processes it*.

50.7 The MCP server — coordination across agents

Listed in §50.1 as the "multi-agent coordination" piece. From the live process listing:

```
$ ps aux | grep cascade
... cascade mcp serve   ← long-running MCP server
... cascade mcp serve   ← second instance
```

`cascade mcp serve` (Ch.34 of this book) is a stdio JSON-RPC server exposing Cascade tools to *any MCP client* — Claude Desktop, Claude Code, other LLM agents. The exposed tools mirror the `cascade agent` tools plus the agent-mode `--admin` operations.

This is where Cascade composes with the **broader agent ecosystem**:

- A Claude Desktop user can ask *"What's Iran's cascade impact?"* and Claude Desktop's tool-use loop dispatches it to `cascade mcp serve`, which calls the same `cascade_chain` Python function the `cascade agent` CLI uses.
- An autonomous agent written in another framework (LangChain, AutoGPT-style) can connect to Cascade's MCP server and use it as a knowledge-graph backend.
- Anthropic's "Computer Use" agent can chain Cascade operations into broader workflows.

Cascade is an agent peer, not an agent orchestrator. It supplies tools; it doesn't control other agents. The MCP server is the interface.

50.8 The four agent mechanisms compared

Property	Tool-use agent	Reactor	Monitors	MCP server
LLM in loop?	Yes	No	No	(external client)
Long-running?	No	Yes	No (cron)	Yes
Triggered by?	User invocation	Graph change	Schedule	MCP client call
Writes to DB?	Yes (admin mode)	Yes (rule actions)	Indirectly (via ingest)	Yes (via tools)
Cost per run	LLM tokens	~free	Web + LLM (ingest)	LLM tokens (caller)
Process model	Spawn-on-call	Daemon	Cron job	Long-running

The **clean architectural cut**:

- **Synchronous + user-facing** → tool-use agent.
- **Asynchronous + graph-event-driven** → reactor.
- **Time-driven** → monitors.
- **Cross-process integration** → MCP server.

Each is the right tool for one kind of work. The "swarm" is the *combination* — together they make Cascade run continuously, ingest new evidence, update its truth values, and answer analyst queries all without a human at the keyboard.

50.9 A day in the life

What the swarm looks like on a typical day:

```
00:00 — Cron fires `cascade monitor run`
         → Monitors run searches for tracked entities
         → 5 new articles found across 3 monitors
         → Each article: cascade ingest <url>
         → Ingest pipeline (Ch.45) runs for each
         → ~150 new entities, ~120 relations,
           ~155 claims added to cascade.db

00:01 — GraphEventBus emits SOURCE_INGESTED + many
         ENTITY_CREATED / RELATION_CREATED events
         → Reactor (running as daemon) reads the queue
         → Rules from .cascade/rules/axiomalang/ fire
         → Truth values propagate (Ch.46)
         → Some claims flip from TRUE to BOTH

08:00 — Analyst arrives, runs `cascade agent
```

```
         "what's new about Iran today?"`
         → Tool-use agent calls search_claims,
           cascade_chain, search_entities
         → Synthesizes a response citing the new
           articles ingested overnight
         → Optionally exports to Markdown for the
           daily briefing

08:30 — Analyst uses Claude Desktop with Cascade MCP
         → Claude calls cascade tools via MCP bridge
         → Same data, different agent

12:00 — Cron fires monitor run again
         → 2 more articles ingested
         → Reactor processes the deltas

EOD — Substack daily briefing generated (Ch.51 §51.x)
```

No human-in-the-loop for the overnight + reactor + monitor work. The analyst only joins at 08:00 to *consume* what the swarm produced. That's the operational payoff.

50.10 Honest limits

Five things to know:

1. **The reactor's rule-matching is shallow.** It uses the `rules` table's `definition` strings to match against changes, but the matching logic is currently *type-based* (matches on relation type, not on specific entities). A more powerful matcher would speed up firing precision.
2. **Monitors use Google News RSS by default.** Free but rate-limited and *not as comprehensive* as a paid news API. For production-grade coverage, swap in a paid provider (Newsdata, NewsAPI) in `cascade/core/monitor.py`.
3. **The tool-use agent is single-turn-deep.** The 5-round cap (configurable) prevents runaway tool loops, but also stops genuinely deep multi-step analysis. Increasing the cap risks unbounded LLM cost.
4. **Inter-agent communication is shared-state only.** All four agents read/write `cascade.db`. There's no direct message- passing between them. This is *fine* (it's the unix philosophy applied to agents) but it means coordination patterns require careful ordering.
5. **No auto-restart for the reactor.** If the reactor daemon crashes, it doesn't restart itself. Production deployments need an external supervisor (systemd, supervisord).

50.11 Exercises

These exercises run against a live Cascade install.

50.1 — Tool-use agent walkthrough

Run `uv run cascade agent "What does Cascade know about Iran's connections to oil markets?"`. Read the verbose output (`-v`). How many tool calls did the LLM make? Which tools did it pick?

50.2 — Reactor status

Run `uv run cascade reactor status`. If the daemon isn't running, start it with `cascade reactor start` in another terminal. Then ingest a new article (or use `cascade entity add` to trigger a graph change). After a few seconds, check the status again. Did `events_received` and `rules_fired` change?

50.3 — Add a monitor

Run `uv run cascade monitor add "Some topic" --query "search terms" --interval 24`. List monitors. Run them with `cascade monitor run`. What gets added to the database?

50.4 — Compare the four mechanisms

For one analytical task — "find new evidence about Iran" — which agent mechanism is the right fit? Sketch how you'd implement it via:

a) Tool-use agent (synchronous)
b) Reactor (event-driven)
c) Monitor (scheduled)
d) MCP server (cross-process)

Which would you pick for which use case?

50.5 — Inspect the GraphEventBus

Look at `cascade/core/events.py`. Identify:

a) How many ChangeType values exist?
b) Is the bus synchronous or async?
c) Can multiple subscribers register?

50.6 (open) — Sketch a fifth mechanism

Cascade has four agent mechanisms today. Sketch a fifth that would fit naturally. Hints:

- A *consensus agent* that aggregates votes from multiple LLMs?
- A *learning agent* that updates rules based on outcomes?
- A *human-in-the-loop* agent for reviewing paraconsistent claims?

Two paragraphs. What's it for, where does it fit?

Solutions to selected exercises: Chapter 50 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch50_solutions.md.)*

50.12 What you learned

- **"Agent" in Cascade is four mechanisms, not one**: tool-use agent, reactor, monitors, MCP server.
- **Tool-use agent** — `cascade agent`, LLM- driven, 7-tool catalog, max 5 rounds, synchronous, optional `--admin` mode for mutations, separate user-memory layer.

- **Reactor** — `cascade reactor`, daemon, no LLM, subscribes to GraphEventBus, fires rules on change_log entries, debounce-capable.
- **Monitors** — `cascade monitor`, scheduled, per-entity search-and-ingest jobs, integrate with cron via `install-cron`.
- **MCP server** — `cascade mcp serve`, cross-process bridge to other agents (Claude Desktop, Claude Code, third-party orchestrators).
- **The clean cut**: synchronous/user-facing, event-driven, time-driven, cross-process — each agent type owns its slice.
- **A day in the life**: monitors scrape → reactor processes events → analyst queries via tool-use agent or MCP. No human required in the loop for ingestion + rule firing.

The next chapter (Ch.51) zooms into the reactor + scheduled tasks, including the production deployment patterns (cron, systemd, the write-ahead change-log queue).

> *"A society of agents is not a council of equals; it's a hierarchy of contracts."* — paraphrased from Minsky. Cascade's swarm follows the principle: each agent has a narrow contract; coordination happens through shared state (`cascade.db`); no agent claims authority over another's domain.

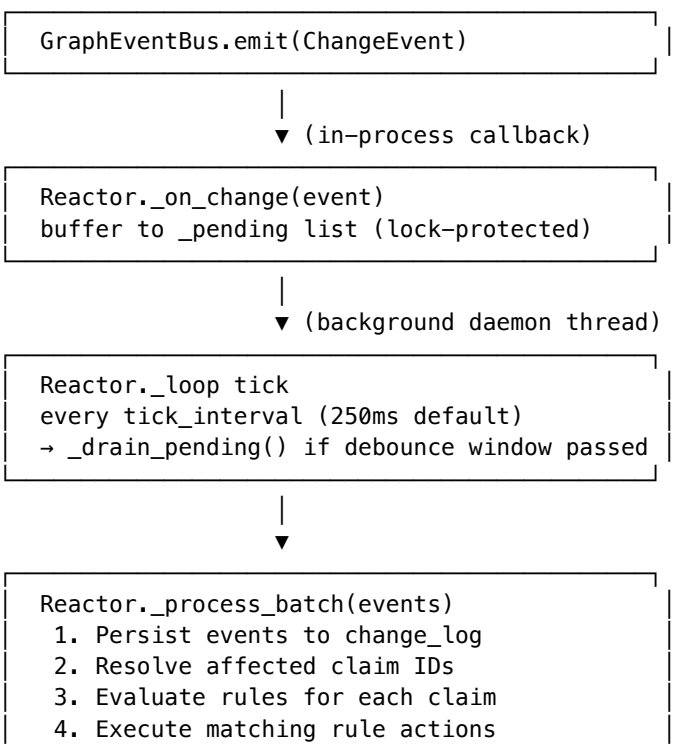
Chapter 51 · Reactor Internals & Scheduled Tasks

> **What this chapter is.** Ch.50 introduced four agent mechanisms. This chapter drills into the two that *run unattended*: the **reactor daemon** and the **scheduled-monitor runner**. You'll learn the reactor's threading model, the debounce-and-batch event-processing loop, the cascade-cycle detection that keeps forward-chaining from runaway, and the production patterns for deploying both under systemd or cron. By the end you'll know how to run Cascade *autonomously* for days at a time, what to monitor when things go sideways, and how the change-log audit trail survives every restart.

This is the second chapter of Volume III Part XV (Operating Cascade). Ch.52 — the volume's closer — covers subscription tiers, sync to Supabase, and the daily-briefing pipeline.

51.1 The reactor as a forward-chaining engine

The reactor is **production-rule architecture** in the OPS5 / CLIPS / Drools lineage:



```
│   5. Track CascadeContext (depth + seen set)  │
└──────────────────────────────────────────────────┘
                      │
                      ▼ (rule actions may modify DB)
              new ChangeEvents → loop
```

Read top-to-bottom: **events flow in, work flows out**. The reactor's contribution is *timing* (debounce + batch) and *forward-chaining safety* (cycle detection + depth bound).

51.2 The threading model

`cascade/agent/reactor.py:45` defines `class Reactor` with three concurrency-critical attributes:

```python
class Reactor:
    def __init__(self, db_path, config, display_fn):
        self._pending: list[ChangeEvent] = []
        self._pending_lock = threading.Lock()
        self._running = False
        self._thread: threading.Thread | None = None
        self._conn: sqlite3.Connection | None = None
        ...
```

The reactor lives in **one process, two threads**:

1. **Caller thread** — whatever thread emitted the ChangeEvent. Runs `_on_change(event)` which only appends to `_pending` under the lock. *Cannot touch the DB* (its connection lives in the daemon thread).
2. **Daemon thread** — the long-lived `_loop` that ticks every `tick_interval` (250ms), drains pending events past the debounce window, and processes the batch. Owns the SQLite connection.

The split matters: **emitters never block**. A producer can emit hundreds of events in a tight loop; the daemon thread digests them at its own pace. The `_pending_lock` is only held during list append/clear — never during DB work.

A third logical thread (the SSE-streaming worker for the dashboard) reads `_sse_queues` for live event delivery — also non-blocking.

51.3 The debounce-and-batch loop

```python
def _loop(self) -> None:
    while self._running:
        try:
            batch = self._drain_pending()
            if batch:
                self._process_batch(batch)
        except Exception as e:
            self._errors += 1
```

```
            log.warning("Reactor loop error: %s", e)

        # Sleep in small increments for quick shutdown
        deadline = time.monotonic() + self.config.tick_interval
        while self._running and time.monotonic() < deadline:
            time.sleep(0.05)
```

The `_drain_pending` decides *when* to fire:

```
def _drain_pending(self) -> list[ChangeEvent]:
    with self._pending_lock:
        if not self._pending:
            return []
        oldest = self._pending[0].timestamp
        if (time.time() - oldest) * 1000 < self.config.debounce_ms:
            return []  # Not ready yet
        batch = self._pending[:]
        self._pending.clear()
    return batch
```

The rule: **wait until the oldest pending event is older than `debounce_ms` (default 500ms), then batch-process everything**. This collapses bursts — if 50 events land in 100ms, they're processed as one batch, not 50 independent firings.

The `tick_interval` (250ms) is *separate* from `debounce_ms`. The tick is "how often does the loop check?"; the debounce is "how stale must the oldest event be before processing?" Together they trade *responsiveness* (low tick) against *efficiency* (high debounce).

Default configuration (`ReactorConfig`):

Field	Default	Meaning
`enabled`	True	Master switch
`debounce_ms`	500	Wait this long after event before firing
`max_cascade_depth`	3	Forward-chain depth limit
`tick_interval`	0.25	Loop tick (seconds)
`max_concurrent_evals`	3	Evaluation parallelism cap
`notify_macos`	False	Send macOS desktop notifications
`notify_log`	True	Log reactor decisions
`notify_sse`	True	Stream to dashboard SSE
`auto_start_dashboard`	True	Bring up dashboard on start

51.4 Cycle detection — `CascadeContext`

Rule firing can trigger more graph changes, which can fire more rules, *ad infinitum*. The reactor prevents infinite cascades via two guards:

```
@dataclass
class CascadeContext:
    """Tracks (claim_id, rule_name) pairs to prevent
    cycles within a cascade."""
    depth: int = 0
    seen: set = field(default_factory=set)
    # set[tuple[int, str]]
```

The two guards:

1. **Depth bound.** `max_cascade_depth = 3` (default). A rule firing at depth N can produce changes that fire rules at depth N+1, up to the limit. Past that, further changes are *recorded but not re-evaluated*.
2. **(claim_id, rule_name) seen-set.** Within a single cascade, the same rule firing on the same claim never fires twice. This prevents the "rule A modifies claim X → rule A re-matches claim X → rule A fires on claim X again…" loop.

Together they guarantee termination. The seen-set is *per cascade* — a fresh batch of events starts with an empty set, so the same rule *can* fire on the same claim in a later batch.

This is the **production-rule safety property**: forward chaining always terminates, even with mutually-modifying rules. The 3-depth bound is empirically tuned — most real Cascade rule chains saturate at depth 2-3.

51.5 The change_log audit trail

Every event the reactor processes *also* lands in the `change_log` table:

```
def _process_batch(self, events):
    conn = self._get_conn()

    # Persist events to change_log (audit trail)
    for event in events:
        conn.execute(
            "INSERT INTO change_log "
            "(change_type, item_type, item_id, "
            " entity_ids, metadata) "
            "VALUES (?, ?, ?, ?, ?)",
            (event.change_type.value, event.item_type,
             event.item_id, json.dumps(event.entity_ids),
             json.dumps(event.metadata)),
        )
    conn.commit()
    ...
```

The `change_log` is **append-only**. Every event ever processed is permanently recorded:

```
CREATE TABLE change_log (
    id              INTEGER PRIMARY KEY AUTOINCREMENT,
    change_type     TEXT NOT NULL,
    item_type       TEXT NOT NULL,
    item_id         INTEGER NOT NULL,
    entity_ids      TEXT DEFAULT '[]',
    metadata        TEXT DEFAULT '{}',
    reactor_handled INTEGER DEFAULT 0,
    rules_fired     TEXT DEFAULT '[]',
    created_at      TEXT NOT NULL DEFAULT (datetime('now')),
    ulid            TEXT
);
```

Three flags worth knowing:

- **`reactor_handled = 0`** — the row hasn't yet been processed. Either the reactor is off, or it's still catching up.
- **`rules_fired`** — JSON array of rule names that fired in response to this event. Empty if no rule matched.
- **`reactor_handled = 1`** — the reactor processed this row.

The audit trail enables **forensics**: when a claim flipped from TRUE to BOTH at 3:47am, you can `SELECT * FROM change_log WHERE created_at LIKE '2026-04-15 03:47%'` and see exactly which events triggered the change.

51.6 Resolving affected claims

When a graph mutation happens, *which claims need re-evaluation*? The reactor's `_resolve_affected_claims` maps event types to claim sets:

Event type	Affected claims
`CLAIM_CREATED` / `UPDATED`	The claim itself
`CLAIM_DELETED`	None (can't re-evaluate)
`ENTITY_CREATED` / `UPDATED`	All claims referencing the entity
`ENTITY_DELETED`	All claims that referenced it (will fail gracefully)
`RELATION_*`	Claims whose `claim_ids` array includes a supporting claim
`SOURCE_INGESTED`	All claims from that source

The resolver does this via SQL joins against `claims.entity_ids` and `relations.claim_ids`. The resulting set gets *deduplicated* — even if an entity change triggers 50 claims, they're each evaluated *once* per batch.

51.7 Starting and stopping the reactor

The two lifecycle commands:

```
# Start (runs in foreground; press Ctrl-C to stop)
$ uv run cascade reactor start

# Start with custom debounce
$ uv run cascade reactor start --debounce 1000

# Check status
$ uv run cascade reactor status
{
  "running": true,
  "events_received": 142,
  "rules_fired": 38,
  "actions_executed": 38,
  "errors": 0,
  "pending": 0
}

# Stop the running reactor (sends signal to daemon)
$ uv run cascade reactor stop
```

The `start` command runs in the **foreground** — typically you'd run it inside `tmux`, `screen`, or under a process supervisor (next section). The `stop` command writes to a PID file that the running reactor watches; clean shutdown takes ≤5 seconds (the daemon thread's join timeout).

51.8 Production deployment — systemd

For a Linux server, the canonical pattern is a systemd unit:

```
# /etc/systemd/system/cascade-reactor.service
[Unit]
Description=Cascade Reactor Daemon
After=network.target

[Service]
Type=simple
User=cascade
WorkingDirectory=/home/cascade/axiomacascade
ExecStart=/home/cascade/.local/bin/uv run cascade reactor start
Restart=on-failure
RestartSec=5
StandardOutput=journal
StandardError=journal
```

```
[Install]
WantedBy=multi-user.target
```

Then:

```
$ sudo systemctl enable --now cascade-reactor
$ systemctl status cascade-reactor
$ journalctl -u cascade-reactor -f   # tail logs
```

The systemd `Restart=on-failure` is critical — Cascade's reactor doesn't self-restart on crash. If the daemon thread dies (uncaught exception in a rule action), the whole process exits and systemd brings it back.

For macOS, the equivalent is `launchd` with a `~/Library/LaunchAgents/com.cascade.reactor.plist` file. The pattern is identical: the OS supervisor holds the lifecycle; Cascade does the work.

51.9 Scheduled monitors — the cron pattern

Monitors are *time-driven*, not event-driven. They don't need a daemon — just cron:

```
# /etc/crontab or `crontab -e`
0 */6 * * * cd /path/to/axiomacascade && \
    /home/cascade/.local/bin/uv run cascade monitor run --quiet
```

This runs every 6 hours. Cascade provides `install-cron` to generate the line:

```
$ cascade monitor install-cron
# Cascade monitor cron entry:
0 */6 * * * cd /path/to/cascade && uv run cascade monitor run --quiet
```

The `--quiet` flag suppresses progress output — under cron, stdout/stderr go to the cron mailer or get discarded, so verbosity is wasted.

Inside `cascade monitor run`:

```
def run_all_due_monitors(conn, max_results=10,
                         auto_extract=True):
    monitors = get_due_monitors(conn)  # WHERE last_run + interval <= now
    for m in monitors:
        run_monitor(conn, m, max_results, auto_extract=auto_extract)
```

Each due monitor:

1. **Searches** Google News RSS for its query.
2. **Dedupes** against `sources.url`.
3. **Ingests** up to 5 new articles per run (the cap prevents one popular topic from burning the whole cron budget).

4. **Updates** `monitors.last_run_at`.

The 5-article cap is a *politeness measure*. A high-volume topic could yield 100+ new articles per run; ingesting all of them would burn ~30 minutes of cron time and ~$5 of LLM cost. The cap throttles that to ~5-10 minutes / ~$0.50. Future runs catch the rest.

51.10 Monitoring the reactor — what to watch

When the reactor is running unattended, three metrics matter:

Metric	What it means	What to do
`events_received`	Total events buffered	Should match `change_log` row count
`rules_fired`	Rules that matched + ran	Low ≠ broken; just means no rule matched
`actions_executed`	Actions that completed	Should track `rules_fired` (1:1 typical)
`errors`	Exceptions caught	Investigate any non-zero
`pending`	Buffered events not yet drained	Stays near 0 unless under heavy load

A *healthy* reactor sees:

- `pending` ≈ 0 (events drain within the debounce window)
- `errors` = 0 or very low
- `actions_executed` increments with `rules_fired`

A *struggling* reactor shows:

- `pending` > 100 (events backing up — increase `debounce_ms` or scale up)
- `errors` > 10 (rules throwing exceptions — check rule files)
- `actions_executed` « `rules_fired` (rules matching but actions failing)

The dashboard (Ch.50 §50.6's hint about `auto_start_dashboard`) shows these live via SSE.

51.11 The change-log restart story

Restart safety is one of the reactor's quiet features. Because every event lands in `change_log` *before* the reactor processes it, restarting the reactor doesn't lose events:

```
# On startup, the reactor reads unhandled rows:
SELECT * FROM change_log WHERE reactor_handled = 0;
```

Those become "synthetic events" replayed through `_process_batch`. The reactor catches up to the present, then waits for new live events.

This is **write-ahead logging at the application layer**. The `change_log` is the WAL; the reactor's in-memory `_pending` queue is the fast-path cache. Crashes are recoverable; the WAL survives.

Two consequences:

1. **Long downtimes are fine.** If the reactor is off for a week while you debug a rule, the `change_log` accumulates rows. On restart, the reactor drains the backlog (possibly slowly) and reaches steady state.
2. **The `change_log` grows monotonically.** Cleanup is a separate concern (no auto-prune). Production deployments need a periodic `DELETE FROM change_log WHERE created_at < datetime('now', '-30 days')` job.

51.12 Honest limits

Five engineering caveats:

1. **Single-process.** The reactor runs in one Python process. Scaling beyond one machine requires either *sharding* the cascade.db by entity (untested) or *promoting* event delivery from the in-process bus to a real message broker (Redis Streams, Kafka).
2. **No back-pressure to producers.** If the reactor falls behind, producers keep emitting events. Eventually the `_pending` list grows unboundedly. The 5-second join timeout on shutdown can drop events that haven't been drained yet.
3. **The CascadeContext is per-batch.** A rule that should *never* fire twice on the same claim across days still can — the seen-set resets at every batch boundary. Cross-batch idempotency must be enforced *in the rule itself*.
4. **No rule priorities.** When multiple rules match the same event, they all fire in undefined order. If ordering matters, encode it via *dependent rules* (rule B's precondition checks rule A's output).
5. **Monitors don't retry failed ingestions.** If `cascade ingest <url>` fails inside a monitor run, the URL is logged and skipped. The next monitor run won't retry it (the URL isn't yet in `sources`, but the search engine may not surface it again).

51.13 Exercises

51.1 — Start the reactor, count events

In one terminal, run `uv run cascade reactor start`. In another, ingest a new article or add an entity. Then run `uv run cascade reactor status` and inspect `events_received`. Did the number match your expectation?

51.2 — Inspect cascade depth

Edit `~/.cascade/cascade.toml` to set `reactor.max_cascade_depth = 5`. Restart the reactor. Trigger an event that causes a multi-hop rule cascade (e.g. ingest an article about "sanctions" if you have the rule from Ch.49 §49.4). Look at `change_log` — how many derived events were produced? At what depth did the cascade stop?

51.3 — Read the change_log

```
SELECT change_type, COUNT(*) AS n
FROM change_log
GROUP BY change_type
ORDER BY n DESC;
```

What's the most common change_type in your install? Which event types fire least? What does that tell you about your ingestion patterns?

51.4 — Install the monitor cron

Run `cascade monitor install-cron` to generate the crontab line. *Don't* actually install it — just read it. What's the schedule? What happens if you forget the `--quiet` flag?

51.5 — Simulate reactor failure

Stop the reactor. Add a few entities via `cascade entity add ...` (this still writes to `change_log` since it's source-of-truth, but *reactor_handled* stays 0). Restart the reactor. Does it process the backlog?

51.6 (open) — Design a rate limiter

The reactor has no per-rule rate limit. A buggy rule that fires on every claim could overwhelm the system. Sketch a `RuleRateLimiter` class:

- Per-rule firing budget (e.g. 100/minute)
- Sliding window over the last 60 seconds
- What does it return when the budget is exceeded? (skip? error? alert?)

Two paragraphs.

Solutions to selected exercises: Chapter 51 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch51_solutions.md.)*

51.14 What you learned

- The reactor is a **forward-chaining rule engine** in the OPS5/CLIPS/Drools lineage — embedded in Cascade rather than freestanding.
- **Two threads**: emitter (non-blocking append)
  - daemon (debounce + batch + process). Lock on the pending list; DB connection owned by the daemon.
- **Debounce-and-batch** — events accumulate for `debounce_ms` (500ms default), then a batch flushes through the pipeline.
- **CascadeContext** — depth bound + (claim_id, rule) seen-set — guarantees termination of forward chains.
- **change_log** is the write-ahead log; the reactor reads `reactor_handled = 0` rows on startup and replays.

- **Production deployment** — systemd unit on Linux, launchd on macOS. `Restart=on-failure` handles crashes.
- **Monitors via cron** — `install-cron` prints the entry; `--quiet` is the right cron flag.
- **Five honest limits** — single-process, no back-pressure, per-batch seen-set, no rule priorities, no monitor retries.

The next (and final) Volume III chapter — Ch.52 — covers the **operating-Cascade scaling story**: multi-machine Supabase sync, the daily-briefing pipeline, and how the whole architecture composes into a multi-tenant production deployment.

> *"A daemon is a process you don't think about until it's broken."* — old systems-admin aphorism. Cascade's reactor is designed to minimize the latter case — append-only audit trail, write-ahead recovery, depth-bounded forward chains. The thinking happens once at setup; after that, it just runs.

Chapter 52 · Operating Cascade at Scale

> **What this chapter is.** Volume III's closer. You've seen the architecture (Ch.43), the schema (Ch.44-38), the reasoning layer (Ch.47-41), and the agent swarm (Ch.50-43). This chapter ties it together with the *production-deployment story*: the multi-machine **Supabase sync** that makes the KB portable, the **output-format catalog** (HTML, GraphML, DOT, Cypher, briefings), the **dashboard + briefing pipeline** that turns the KB into something a non-technical reader can consume, and the **operational practices** (logging, monitoring, scaling) that keep a multi-week deployment healthy. By the end you'll have the full operational picture and a punch list for standing up your own Cascade.

This is the closing chapter of Volume III. After this you can either consume the textbook as a self-contained body of knowledge, or pick up the roadmap chapters mentioned in Appendix D.

52.1 The deployment topology

Cascade can run in three deployment modes, each with different tradeoffs:

Mode	Use case	Components
Local single-machine	Solo analyst	`cascade.db` + reactor + cron monitors
Networked single-tenant	Small team	Local + Supabase mirror + Dashboard exposed on LAN
Multi-tenant production	Subscription service	Multiple Cascade installs sharing a Supabase backend, Substack publisher, billing

This chapter walks each. The architectural pattern is *additive* — local mode is a strict subset of networked, which is a strict subset of production. You start small and scale up without rewriting.

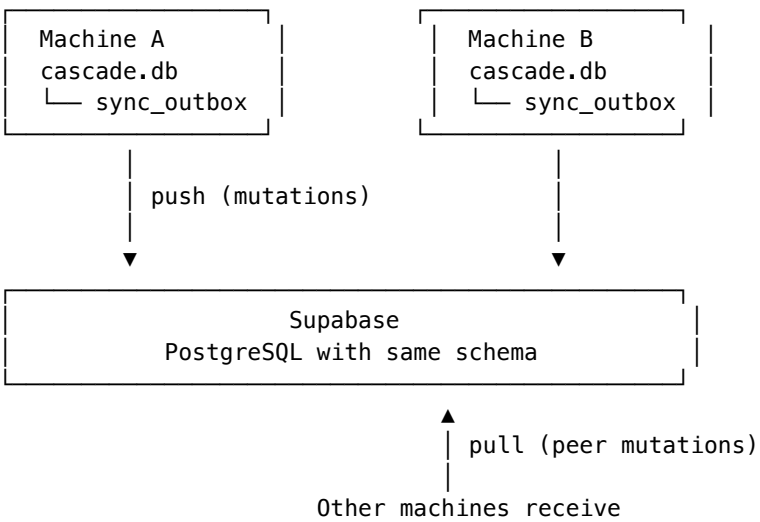
52.2 Multi-machine sync — Supabase

The sync layer lives in `cascade/sync/`:

```
cascade/sync/
├── outbox.py    — Capture local mutations
├── push.py      — Send unpushed rows to Supabase
├── pull.py      — Receive remote rows, apply locally
├── conflict.py  — Resolve ULID-based conflicts
```

```
├── daemon.py    — The sync daemon (long-running)
└── neo4j.py     — Optional Neo4j AuraDB read replica
```

The flow:



The `sync_outbox` table (Ch.44 §44.7) is the **write-ahead log for sync**. Every local INSERT or UPDATE writes a row there; the daemon periodically pushes unpushed rows. ULIDs (Ch.44 §44.3) are the dedup key — two machines that generate ULIDs at the same millisecond *still* get globally-unique IDs.

CLI:

```
$ export SUPABASE_URL="https://xyz.supabase.co"
$ export SUPABASE_KEY="..."

$ cascade sync start          # daemon (foreground)
$ cascade sync push           # one-shot push
$ cascade sync pull           # one-shot pull
$ cascade sync status         # outbox counts + last push time
```

Why Supabase? Three reasons over rolling your own:

1. **Postgres protocol** — standard, well-tooled, easy to read from BI tools.
2. **Row-level security** — built-in multi-tenant support.
3. **Realtime subscriptions** — for future live-stream features.

The downside is *vendor lock-in lite*. The schema itself is portable (it's just PostgreSQL); the auth and RLS bits are Supabase-specific. Production deployments often add a *thin abstraction* in `cascade/sync/` so swapping Supabase for plain Postgres takes one config change.

52.3 Conflict resolution

When two machines edit the *same* row, conflict resolution kicks in. The rule, from `cascade/sync/conflict.py`:

Conflict	Resolution
Two INSERTs with same ULID	First-write-wins (later push is silently a no-op)
Two UPDATEs to same row	Last-write-wins by `updated_at`
INSERT after DELETE	Re-insert wins (DELETE was the older op)
Same-row truth_value flip on both machines	Last-write-wins; conflict logged

The `last-write-wins` strategy is **simple but lossy**. If Machine A flips a claim from TRUE to FALSE at 10:00 and Machine B flips it to BOTH at 10:01, the BOTH wins — but A's reasoning chain is discarded. For mission-critical analyst work, this is wrong; for *eventual consistency* of a knowledge graph it's acceptable.

A future enhancement would store *both* edits in `truth_history` (Ch.46) and let an analyst reconcile manually — a *conflict review queue*.

52.4 Output formats — the export catalog

`cascade export` exposes 7 output formats:

Subcommand	Format	Use case
`html`	Pyvis interactive	Share via email; clickable graph
`graphml`	XML for Gephi/yEd/Cytoscape	Academic-grade visualization
`gexf`	GEXF for Gephi	Native Gephi format
`dot`	Graphviz	Programmatic typesetting
`cypher`	Neo4j CREATE statements	Import to Neo4j
`tree`	Cascade chain as tree diagram	Briefing visuals
`chart`	Matplotlib statistical plots	Reports + decks

Each preserves a different *aspect* of the graph:

- HTML/GraphML/GEXF/DOT preserve **structure** — the nodes and edges.
- Cypher preserves the **relational vocabulary** — ready for Neo4j-native queries.
- Tree preserves the **cascade chain** for one origin entity.
- Chart preserves **statistics** (top-10 centrality, type distributions, etc.).

The principle: **export per use case, not per analyst preference**. A dashboard view (Pyvis HTML) serves a *casual reader*; an academic paper needs *GraphML*; a research blog post wants *DOT*.

```
$ cascade export html --output graph.html
$ cascade export graphml --output graph.gml
$ cascade export tree "Iran" --depth 3 --output iran.txt
$ cascade export chart centrality --output centrality.png
```

52.5 The dashboard — `cascade dashboard`

For interactive exploration, the dashboard is a Flask + Pyvis web app:

```
$ cascade dashboard start
[done] Dashboard at http://127.0.0.1:5050
```

What it shows:

- **Graph view** — interactive Pyvis network, filter by entity type and confidence band.
- **Cascade explorer** — pick an origin entity, set depth, view tiered impact chains.
- **Truth view** — claims with their B4 values, filterable by confidence.
- **Recent activity** — live SSE feed of `change_log` events.
- **Search** — fulltext search over entities, relations, claims.

The dashboard listens on `127.0.0.1:5050` by default (localhost-only). For team access, bind to `0.0.0.0` and use a reverse proxy (nginx) with basic auth or OAuth:

```
$ cascade dashboard set-password
Enter password: ***
[done] Password set

$ cascade dashboard start --host 0.0.0.0 --port 8080
```

The SSE stream comes from the reactor's `_sse_queues` (Ch.51 §51.2's third logical thread). Dashboard subscribers register a queue; the reactor's batch processing pushes events through; the browser's EventSource API reads them. **Sub-second update latency** for live view of the graph as it evolves.

52.6 The daily briefing — `cascade/output/briefing.py`

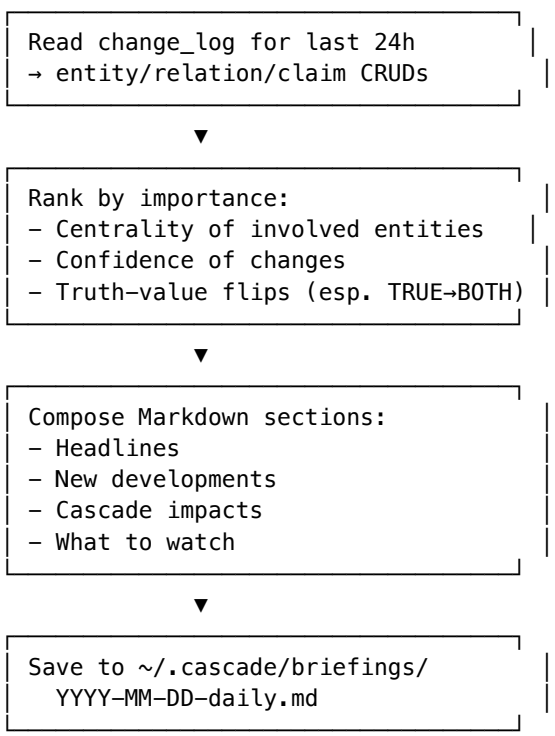
The high-leverage *output* of an operational Cascade deployment is the **daily briefing**: a plaintext or HTML summary of what happened in the graph over the last 24 hours, with citations to sources and a list of "what to watch."

```
def generate_daily_digest(conn, llm_enabled=True):
    """Produce a daily intelligence digest.

    Returns a Markdown briefing covering:
    - New entities introduced
    - New relations of high confidence
    - Claim truth-value flips
    - Top cascade impacts
    - 'What to watch' synthesis (LLM-driven)
    """
    ...
```

Two modes:

1. **Data-only** — no LLM call. Pure summary of the graph diff over the last N hours. Fast, free.
2. **LLM-driven** — Anthropic/OpenAI call that produces narrative analysis. ~$0.10-0.50 per briefing depending on model.

The briefing pipeline:



For a *Substack* publisher, post the saved file via the Substack API or a manual paste. For internal team consumption, email or Slack.

52.7 Subscription tiers — the business model

Cascade is structured to support a *subscription service* model. The tiers (sketch from the `config.toml` defaults):

Tier	Cost	Ingestion	Rules	Dashboard	Briefing
Free	$0	10 articles/day	5 rules	Read-only	None
Basic	$19/mo	100/day	50 rules	Personal	Weekly
Pro	$99/mo	1000/day	Unlimited	Team (≤5)	Daily
Enterprise	Custom	Unlimited	Unlimited	Org-wide + SSO	Custom

Tier enforcement happens at three points:

1. **Ingestion rate** — daily article counter in `~/.cascade/quota.json`. Checked at `cascade ingest <url>` invocation; hard error on exceeded.
2. **Rule count** — checked at `load_axiomalang_rules` time; rules beyond the limit get skipped with a warning.
3. **Dashboard auth** — checked at the dashboard's request middleware.

The pattern is **soft enforcement with explicit upgrade prompts**, not silent feature gating. Every tier-boundary error includes a *what would change if you upgraded* message.

Production deployments handle billing via Stripe (integration in `cascade/billing/`, not covered here — see the Cascade product docs for the full business-side story).

52.8 Logging, metrics, observability

For a production deployment, three log streams matter:

Stream	Source	Purpose
`cascade/data/logs/cascade.log`	Application	Debug & audit
systemd journal	Reactor daemon	Process state
Supabase logs	Sync errors	Database issues

The application log is *the most useful*. Cascade's logging module (`cascade/logging.py`) emits structured log lines:

```
2026-04-15 03:47:22 INFO  extractor | LLM extraction done | 8.2s | entities=28 |
↪  relations=19
2026-04-15 03:47:23 INFO  extractor | Store complete | source_id=331 | new=11 | merged=17
2026-04-15 03:47:23 INFO  reactor   | Batch: 28 events → 12 affected claims
2026-04-15 03:47:24 INFO  reactor   | Rule fired: sanctions_commodity_risk on claim 4521
```

Grep-friendly format. Production monitoring tools (Datadog, Grafana Loki, Splunk) can parse these with one-line regex configs.

For *metrics* (not logs), the reactor and sync daemons expose JSON status endpoints:

```
$ cascade reactor status --json
$ cascade sync status --json
```

Scrape these with cron + curl, push to Prometheus or whatever your monitoring stack uses.

52.9 Operational practices — the punch list

Standing up a production Cascade deployment, the checklist:

Initial setup

- ☐ Install Cascade with `uv sync`
- ☐ Configure LLM provider in `~/.cascade/cascade.toml`
- ☐ Test ingestion with `cascade ingest <url>`
- ☐ Set up Supabase project + populate env vars
- ☐ Run initial `cascade sync force-push`

Daemons

- ☐ Reactor under systemd (Ch.51 §51.8)
- ☐ Sync daemon under systemd
- ☐ Monitor cron (Ch.51 §51.9)
- ☐ Dashboard with auth (this chapter §52.5)

Rule authoring

- ☐ Write 5-10 domain rules in `.cascade/rules/axiomalang/` (Ch.49)
- ☐ Run `--typecheck` on each
- ☐ Verify rules fire as expected via `cascade reactor status --json`

Monitoring

- ☐ Application logs to a centralized collector
- ☐ Reactor/sync status scrapers
- ☐ Disk usage alerts (sync_outbox grows monotonically)
- ☐ LLM cost dashboard (cumulative spend)

Maintenance

- ☐ Daily: review briefing
- ☐ Weekly: `cascade entity merge` for duplicate candidates (Ch.45 §45.5)
- ☐ Monthly: `DELETE FROM change_log WHERE created_at < datetime('now', '-30 days')` to prevent unbounded growth
- ☐ Quarterly: review rule quality (`SELECT name, applied_count, success_count FROM rules`)

This is the **operational cadence** for a healthy Cascade. None of it is novel — it's the standard pattern for any data-pipeline-plus-rule- engine system. The discipline pays off.

52.10 Honest limits — the unsolved problems

After 9 chapters of Volume III, an honest tally of what's *still hard*:

1. **Entity resolution drift.** New ingestions produce duplicate entities that don't always merge. The fix is human-in-the-loop reconciliation, but Cascade doesn't *automatically* generate the review queue.
2. **Rule explosion.** As domain rules accumulate, the reactor's per-event matching gets slower. No indexing on rule preconditions yet.
3. **Multi-tenant truth.** Two analysts in the same Cascade deployment can disagree about the *same* claim. The schema supports it via `truth_history` but the UX for surfacing disagreement is rough.
4. **LLM cost spikes.** A monitor burst (e.g. 100 articles ingested in one cron run) can spike LLM bills by 20× the daily average. Rate limits on the *ingestion side* would help — they don't exist yet.
5. **Long-term knowledge decay.** Claims from five-year-old articles don't *automatically* downweight relative to recent ones. The `timestamp` field exists; the *staleness scoring* doesn't.

These are real engineering gaps, all on the roadmap, none of which prevent production use today. The pattern: **every operational system has unsolved problems; the difference between mature and immature is whether you *know* what they are**.

52.11 Exercises

52.1 — Inventory your install

Run the full inventory:

```
cascade stats                        # graph counts
cascade reactor status --json        # reactor metrics
cascade sync status                  # sync metrics
ls .cascade/rules/axiomalang/        # rule files
ls data/logs/                        # log files
cat ~/.cascade/cascade.toml          # config
```

Generate a one-page *deployment summary* listing: total entities/relations/claims, number of rules, reactor uptime (if known), sync status.

52.2 — Try an export

Pick one of the 7 export formats. Run the command, open the output file in the appropriate tool (browser for HTML, Gephi for GraphML, etc.), and explore. What does this *show* that the CLI doesn't?

52.3 — Run the briefing

```
$ cascade briefing daily --output today.md
$ cat today.md | head -50
```

Read the output. Three questions:

- Is the data-only summary useful as-is?
- Does the LLM section add insight or just paraphrase?
- Where would *you* edit before publishing?

52.4 — Sketch a deployment

You're standing up Cascade for a 3-person analyst team that needs:

- 24/7 ingestion (~50 articles/day)
- Shared knowledge graph
- Web dashboard for the analysts
- Daily briefing emailed to the team

Sketch (no code) the deployment:

- Which deployment mode (§52.1)?

- Hardware / hosting?
- Which daemons run where?
- How is the team's auth handled?
- What's the monthly cost (LLM + Supabase + hosting)?

Half a page.

52.5 — Disaster recovery drill

Suppose your cascade.db got corrupted. List the steps to recover:

- What's the *source of truth* if Supabase sync was running? If not?
- How long would a full recovery take for a 10k-entity install?
- What's lost forever vs recoverable?

Walk it through.

52.6 (open) — What's missing from the punch list?

§52.9's punch list covers the basics. What's missing? Sketch one operational practice you'd add for *your* use case. Hints:

- Backup strategy?
- Incident response runbook?
- Cost-spike alerting?
- Compliance / audit-trail policy?

Solutions to selected exercises: Chapter 52 · Solutions in Appendix C. *(Repo file: exercises/solutions/ch52_solutions.md.)*

52.12 What you learned (and what comes next)

- **Three deployment modes** — local, networked, multi-tenant — additively composable.
- **Supabase sync** via `sync_outbox` + ULID-keyed conflict resolution; vendor-lockable but standard Postgres at the wire level.
- **Seven export formats** for different audiences: HTML (casual), GraphML (academic), Cypher (Neo4j), tree (briefing visuals), chart (reports).
- **Dashboard** as the *live operational view* via Flask + Pyvis + SSE; password-protected for team use.
- **Daily briefing pipeline** — data-only or LLM-driven, output to Markdown for Substack / Slack / email.
- **Subscription tiers** enforced at three points (ingest, rules, dashboard) with soft enforcement + upgrade prompts.
- **Operational punch list** — 4 sections covering setup, daemons, rule authoring, monitoring, maintenance.
- **Five unsolved problems** that production deployments still hit; knowing them is the difference between mature and immature operations.

52.13 Volume III closes

This closes Volume III. The arc:

Part	Chapters	Theme
XII	35	Foundations — the architecture map
XIII	36-38	Knowledge graph — schema, ingest, truth
XIV	39-41	Reasoning — NetworkX + Axioma + custom rules
XV	42-44	Operations — agents, reactor, scaling

You now have the *complete* Cascade story: how data gets in (Ch.45), how it's stored (Ch.44), how it's reasoned over (Ch.47-40), how rules get written (Ch.49), how the system runs autonomously (Ch.50-43), and how it deploys at scale (this chapter).

The book's arc closes here. Volume I taught Axioma the language; Volume II extended into data structures and knowledge representation; Volume III showed how Axioma composes into a production system. The appendices (A–F) provide reference material for the road ahead.

> *"The fundamental theorem of software engineering is that any problem can be solved by adding another layer of indirection — except the problem of too many layers."* — classic. Cascade adds nine layers (ingest, KG, truth, reasoning, agents, sync, output, dashboard, billing). Each layer earns its keep because each owns *one* concern. Removing any of them would *expand* the others. That's the operational test of architecture — not what's there, but what *can't* be removed.

Chapter 53 · SQL as a Pedagogical Surface

What this chapter is. A look at SQL — the most widely-deployed data language on the planet — from the inside out. We won't treat SQL as an external thing that Axioma calls. Instead we'll see how Axioma's `[sql | …]` block **compiles SQL down to native comprehensions**, making SQL a *third surface* over the same relational substrate as Prolog-style queries (Chapter 21) and set comprehensions (Chapter 7). Once you see the lowering, SQL stops being a black box and becomes another way to write what you already understand.

53.1 Three surfaces, one substrate

Axioma's relational layer has three syntactic surfaces:

Surface	Style	Example
Set comprehension	math notation	`{Y | parent(Y, "John")}`
Prolog-form	logic programming	`{Y | Y <- parent(Y, "John")}`
SQL	embedded dialect	`[sql | SELECT child FROM parent WHERE adult = 'John']`

For the schema `parent(child, adult)`, all three ask for the same children. None is privileged; the choice is about which form your reader will find most natural. A mathematician reads the comprehension; a Prolog programmer reads the pipe-arrow; a database programmer reads the SELECT.

Crucially, **all three lower to the same compiled form** — a comprehension over Axioma's `_relation_` store. The SQL block isn't a foreign-runtime call (the way `[python | …]` is). It's a *compilation pass*: the SQL parser builds an AST, the compiler emits an Axioma comprehension, and the existing evaluator runs it.

You can see this with the `/explain` refinement. First create the small schema used in the join examples below:

```
[sql | CREATE TABLE emp (name VARCHAR, dept_id INTEGER, salary INTEGER)]
[sql | CREATE TABLE dept (id INTEGER, dname VARCHAR)]
[sql | INSERT INTO emp VALUES ('Alice', 1, 100)]
[sql | INSERT INTO emp VALUES ('Bob', 2, 90)]
[sql | INSERT INTO emp VALUES ('Carol', 3, 80)]
[sql | INSERT INTO dept VALUES (1, 'Engineering')]
```

```
[sql | INSERT INTO dept VALUES (2, 'Sales')]
println([sql/explain | SELECT name FROM emp WHERE dept_id = 1])
# {_row[1] | _row <- {(V1, V3) | emp(V1, 1, V3)}}
```

That output is the actual Axioma source the SQL compiler produced. Run it directly and you get the same answer.

53.2 The smallest interesting database

```
[sql | CREATE TABLE teacher (instructor VARCHAR, student VARCHAR)]
[sql | INSERT INTO teacher VALUES ('Socrates',   'Plato')]
[sql | INSERT INTO teacher VALUES ('Plato',      'Aristotle')]
[sql | INSERT INTO teacher VALUES ('Anaxagoras', 'Socrates')]

[sql | SELECT student FROM teacher WHERE instructor = 'Plato']
# → {"Aristotle"}
```

Notice the lowering:

SQL	Axioma equivalent
`CREATE TABLE t (…)`	`relation t(…)`
`INSERT INTO t VALUES (…)`	`assert t(…)` at the **datum** tier
`SELECT * FROM t WHERE …`	Set comprehension over `t(…)`
`DROP TABLE t`	`drop_relation("t")`
`TRUNCATE TABLE t`	Forget every row, keep the relation declaration

The `datum` tier is Axioma's bottom-level grounding (Chapter 16) — it means "this row is given without further warrant," matching SQL's semantics for an `INSERT`. If you wanted the row to be an *axiom* (unchallengeable foundation) you would step outside SQL and write `axiom teacher("Socrates", "Plato")` directly.

53.3 The join family

The lowering is most illuminating when you look at joins. An INNER JOIN is just a two-relation comprehension:

```
[sql | SELECT emp.name, dept.dname
       FROM emp INNER JOIN dept ON emp.dept_id = dept.id]
```

lowers (roughly) to:

```
{(N, D) | emp(N, ID, Salary) and dept(ID, D)}
```

The trick is the **join-key unification**: the SQL compiler sees `emp.dept_id = dept.id` and gives both columns the same pattern variable name. The shared name forces Axioma's chain-unification to keep only rows where both columns agree — that's the join.

Outer joins as anti-joins + union

LEFT JOIN is more interesting. The rows where the join matches are the same as INNER JOIN; but rows where the *left* table has no matching *right* row must still appear, with right-side columns NULL-padded. That's a **union of two sets**:

```
LEFT JOIN result = INNER JOIN result
                 ∪ {left rows whose join key has no match}
```

The "no match" check is the anti-join half: walk the left table, filter to rows whose join key is *not* in the right table's join- column extent, project the user's SELECT list with NULL where right- side columns appear.

```
[sql | SELECT emp.name, dept.dname FROM emp
       LEFT JOIN dept ON emp.dept_id = dept.id]
# When dept_id has no matching dept.id, you get ("Carol", ?ᵇ)
# — Belnap's "neither true nor false," Axioma's default NULL.
```

`RIGHT JOIN` is symmetric (preserves the right side). `FULL OUTER JOIN` is both — `matched ∪ left_unmatched ∪ right_unmatched`.

53.4 Aggregation — `GROUP BY` and `HAVING`

```
[sql | CREATE TABLE sales (region VARCHAR, product VARCHAR, qty INTEGER)]
[sql | INSERT INTO sales VALUES ('east', 'widget', 10)]
[sql | INSERT INTO sales VALUES ('east', 'widget', 5)]
[sql | INSERT INTO sales VALUES ('east', 'gadget', 7)]
[sql | INSERT INTO sales VALUES ('west', 'widget', 3)]
[sql | INSERT INTO sales VALUES ('west', 'gadget', 12)]

[sql | SELECT region, product, SUM(qty) FROM sales
       GROUP BY region, product]
# → {("east", "widget", 15), ("east", "gadget", 7),
#    ("west", "widget", 3),  ("west", "gadget", 12)}
```

This lowers to `group_by` (introduced in Chapter 7) plus a per-group projection. Specifically:

1. **Inner comprehension**: walk the relation, emit one tuple per row containing every free column.
2. **`group_by(lambda t => composite_key(t), rows)`**: bucket the rows by the GROUP BY columns. This lowering uses `group_by`'s stringified keys, so a multi-column key gets *stringified*

— which means we recover the original column values from each bucket's first member (`vs[1]`) rather than from the stringified key.

3. **Per-group projection**: for each `(k, vs)` pair, emit the key columns and the aggregate value.

`HAVING` is a post-aggregation filter — it sees the already-computed aggregate value, so the comprehension's terminal filter applies it naturally.

53.5 The CAST tower

SQL's type system is small but the operations on types are rich:

```
[sql | CREATE TABLE grades (score INTEGER)]
[sql | INSERT INTO grades VALUES (85)]

# Standard form
[sql | SELECT CAST(score AS FLOAT) FROM grades]

# Postgres :: shorthand — semantically identical
[sql | SELECT score :: FLOAT FROM grades]

# Chained
[sql | SELECT score :: INT :: TEXT FROM grades]
```

Each cast lowers to a function call on the type's standard constructor: `int(...)`, `to_float(...)`, `str(...)`, `to_bool(...)`. The chained form is left-associative, so the inner cast runs first. These examples start with a numeric value. Do not assume every coercion supported by a database server exists here: for example, this build's `to_float` rejects a String such as `"85"`.

53.6 NULL: preserving missing information

The default SQL surface represents NULL with Belnap's **neither** value, `?ᵇ`. It means no information, not a numeric zero or an empty string. Inserted NULL cells and missing cells supplied by an outer join retain that typed value when read from the relation store.

```
[sql | CREATE TABLE optional (reading INTEGER)]
[sql | INSERT INTO optional VALUES (5)]
[sql | INSERT INTO optional VALUES (NULL)]
println([sql | SELECT reading FROM optional])
# {5, ?ᵇ} — a Set; display order may vary
println([sql | SELECT reading FROM optional WHERE reading IS NULL])
# {?ᵇ}
println([sql | SELECT reading FROM optional WHERE reading > 3])
# {5}
```

```
println([sql | SELECT reading FROM optional WHERE NOT (reading > 3)])
# {}
```

The last result deserves attention. Comparing a missing reading with 3 produces neither, rather than false. Negating neither still does not establish that the reading is at most 3. Neither filter selects that row. To include it, write `reading > 3 OR reading IS NULL` explicitly. `IS NOT NULL` selects only known readings. A quoted String such as `"?ᵇ"` or `"neither"` is ordinary text and does not pass `IS NULL`.

Ordered comparisons and arithmetic in supported SQL expressions propagate missing values. Each binary operator evaluates its operands once. `BETWEEN` uses two such comparisons. Numeric `MOD` and arithmetic on an UPDATE right-hand side also propagate missing values. Other invalid operands still produce errors. These are SQL translation rules: ordinary Axioma arithmetic and comparison operators do not change.

For SELECT, the `/k3` refinement emits `om` for NULL literals and missing results. The `/strict` refinement combines that choice with bag output. Both recognize the typed absence stored by a default INSERT. Write refinements do not change the stored marker. This does **not** make the surface a complete ANSI SQL engine: equality still uses Axioma's value equality, so do not teach `NULL = NULL` as always unknown here. Use `IS NULL` to express the missing-value test.

There are still separate boundaries. `NULLIF` currently returns `none`. The fallback functions `COALESCE` and `IFNULL` test for that value, rather than the SQL markers above. They are not yet general replacements for `IS NULL`. Nullable aggregates, string functions and casts also need their own rules and checks. Do not replace a missing value with a magic numeric sentinel to hide a failure. For typed missing measurements in data analysis, see Chapter 18's `Na`/DataFrame path.

53.7 The lineage surface

For teaching, the SQL block exposes two "intermediate representation" arrow forms:

```
[sql -> algebra | SELECT name FROM emp WHERE dept_id = 1]
# → "π_{name}(σ_{dept_id=1}(emp))"

[sql -> calculus | SELECT name FROM emp WHERE dept_id = 1]
# → "{ t.name | emp(t) ∧ t.dept_id = 1 }"
```

These don't run the query — they return the **relational algebra** and **tuple calculus** forms as Strings. That makes Axioma a particularly good environment for teaching Codd's foundational correspondence: every SELECT is simultaneously an algebraic expression, a calculus formula, and an executable comprehension. Students can run the SQL or generated Axioma. The algebra and calculus Strings are explanatory representations, not independently executable Axioma source.

This is another example of multiple surface notations describing a shared computation.

53.8 What the SQL block is good for (and isn't)

Use SQL when:

- The reader thinks in SQL and you want them to recognize the code immediately.
- You're teaching database theory and need the canonical form.
- The query happens to be expressible compactly in SQL — multi-table joins, aggregates, common table expressions.

Use comprehensions when:

- You want the result of a query to feed into a non-relational computation — lambdas, recursion, set operations on the results.
- You need to mix relational and non-relational data shapes.
- The query is short enough that the math notation reads cleanly.

Use Prolog-form when:

- You're chaining variables across many relations (the <– form reads better for long chains).
- You're already in a logic-programming mindset.

There's no winning form. Axioma offers all three because *clarity is about your reader*. SQL is the right answer when your reader is a SQL programmer. It's the wrong answer when your reader is a Haskell programmer or a category theorist.

53.9 What's not yet supported

The SQL surface supports the concrete examples above, but it is not a complete database engine. Section 53.6 states the supported NULL behavior and the remaining boundaries. The following are not yet supported:

- **Window functions** — `OVER`, `PARTITION BY`, `ROW_NUMBER`, `RANK`, `LAG`, `LEAD`. Large surface, deserves its own milestone.
- **Correlated subqueries** — subqueries that reference outer-query columns. Semantically tricky in any execution model.
- **`CREATE TABLE` constraints** — `PRIMARY KEY`, `FOREIGN KEY`, `NOT NULL`, `DEFAULT`, `REFERENCES`, `CHECK`. Would map to cardinality + referential-integrity rules in Axioma's substrate.
- **`ALTER TABLE`** — schema evolution.
- **Multiple aggregates per SELECT** — currently one aggregate per SELECT row.
- **Chained outer joins** — `a LEFT JOIN b LEFT JOIN c`.
- **Non-equijoin OUTER JOIN predicates** — `LEFT JOIN ... ON a.x < b.y`.
- **HAVING with multi-column GROUP BY**.

None of these prevent the working examples in this chapter from running. They're growth edges for future textbook chapters.

Exercises

1. Write the same three-relation transitive-closure query in all three surfaces (SQL with `WITH RECURSIVE`-style chained CTEs, Prolog-form, set comprehension). Notice which reads most clearly to *you*.
2. Use `[sql/explain | …]` on each of your three forms. Verify the compiled Axioma source is structurally similar.
3. Write a `SELECT … GROUP BY` query, then write the same query as a `group_by` + comprehension chain. Use `[sql -> calculus | …]` to see what the relational-calculus form looks like.
4. Construct a `LEFT JOIN` where the left table has rows with no matching right-table row, and confirm the NULL padding. Then write the same query as an explicit `UNION` of an INNER JOIN and an anti-join.
5. Add a missing reading and a known reading to the same table. Compare `reading > 3`, `NOT (reading > 3)`, and `reading IS NULL` in default and `/k3` modes. Explain why negating an unknown ordered comparison does not select the missing row. Do not assume ANSI equality semantics.

"SQL is the COBOL of relational algebra — verbose, prescriptive, universally understood. The tradeoff is the genius." — adapted from M.P. Atkinson. Axioma's bet is that you should be able to write the SQL when SQL is what your reader expects, write the comprehension when math notation is clearer, and write the Prolog-form when you're thinking in logic. All three are the same thing under the hood. Choose the surface, not the substrate.

Chapter 54 · The Closed Hindley–Milner Island

What this chapter is. Chapter 19 taught you to *write* a type and ask `--typecheck` whether the program honors it. This chapter teaches the other direction: a dialect that *reads* an unannotated program and either prints a principal type or **refuses to run**. `#language axioma/hm` is not a flag that "turns types on" for the host language. It is a **closed language** — a fenced island with its own catalog, its own operators, and a checker that does not skip.

Where it sits. Read this after Chapter 19 (annotations and `--typecheck`) and Chapter 11 (functions as values). The chapter is numbered 54 so earlier chapters stay put; it is a Part VI addendum, not a sequel to SQL.

54.1 Two different jobs for types

Axioma's host language (`axioma/all`, the default) is **gradual**. You may annotate a name with `::` and `--typecheck` will honor the annotation. You may also write nothing, and the program still runs. That is the right default for a multi-paradigm language that also has concepts, knowledge bases, and dual bottoms.

Host `axioma --infer` is a **lens** on that gradual language. It looks at each top-level function in the pure lambda fragment and prints an arrow if it can. Everything else gets a *reason*, never a guess, and a refusal does **not** fail the process:

```
script.ax:1: id :: a -> a
script.ax:2: shout — not inferred: outside the lambda fragment (world: calls println())
```

That report is useful. It is not a type system. A student cannot tell "the checker rejected this" from "the lens declined to guess." Both print a line and exit 0.

`#language axioma/hm` answers a different question: **what if the file were a language whose programs either have a principal type, or do not run?**

	Host `--infer`	`#language axioma/hm`
What it is	a lens on `axioma/all`	a language
Untypable code	skip, exit 0	refuse, exit 1
`println`	"not inferred (world)"	`println` has no known type
`id` used at two types	let-polymorphism, if it is in the fragment	let-polymorphism, required

	Host `--infer`	`#language axioma/hm`
Recursion	inferred when monomorphic	allowed (`fib` is legal)
`--vm`	separate	refused

The island is **not** the Total island of the language constitution. General recursion is allowed. Strong normalization is not claimed. A later `axioma/total` would be a smaller dialect inside this one.

54.2 Your first island file

Put the pragma on line one. Each fenced program in this chapter is a **separate file**: repeated names such as `id` or `inc` do not belong in one concatenated island program. The rest of each file is the island; no host builtins leak in.

```
#language axioma/hm
id: func(x) [x]
expect("id 1", id(1), 1)
expect("id true", id(true), true)
```

Run it:

```
$ axioma --no-kb id.ax
[done] --- PASS: id 1
[done] --- PASS: id true
```

Then ask for the types. On an island file, `--infer` is the **same checker** that gated the run — not a second opinion:

```
$ axioma --no-kb --infer id.ax
id.ax:2: id :: a -> a
```

A lowercase letter is a type variable. `id :: a -> a` means: for *any* type `a`, this function takes an `a` and returns an `a`. The two `expect` lines instantiate `a` at `Integer` and at `Boolean`. That reuse is **let-polymorphism**: a name bound to a function is generalized, then specialized again at each call.

`expect` is the island's assertion. `println` is not in the catalog — it writes to the world, and the island has no world. Tests in this chapter use `expect(label, actual, expected)`, the same assertion the rest of Axioma uses.

You can also pass `--language axioma/hm` (alias `hm`) on the command line. The pragma in the file is the form to prefer: it travels with the source.

54.3 Let-polymorphism, in one function

The identity example already used `id` at two types. Here the two instantiations live in *one* body, so the checker has to keep both specializations in play at once:

```
#language axioma/hm
id: func(x) [x]
both: func() [ if id(true) then id(1) else id(2) ]
expect("both", both(), 1)
```

```
id   :: a -> a
both :: () -> Integer
```

`id(true)` specializes `a` to `Boolean` for the `if` condition. `id(1)` and `id(2)` specialize `a` to `Integer` for the branches. The two uses do not fight, because `id` was generalized when it was bound.

A higher-order cousin:

```
#language axioma/hm
twice: func(f, x) [f(f(x))]
inc: func(n) [n + 1]
notb: func(b) [if b then false else true]
expect("twice inc", twice(inc, 3), 5)
expect("twice not", twice(notb, true), true)
```

```
twice :: (a -> a, a) -> a
inc   :: Integer -> Integer
notb  :: Boolean -> Boolean
```

`twice` does not care what `a` is, as long as `f` sends `a` to `a`. `inc` pins `a` to `Integer` at one call; `notb` pins it to `Boolean` at the other.

54.4 Arithmetic, concatenation, and /

On the host, `"a" + "b"` concatenates and `"ab" * 2` repeats. The island will not lie about that with a single + type.

```
#language axioma/hm
expect("plus", 1 + 2, 3)
expect("str_concat", str_concat("a", "b"), "ab")
expect("array_concat", array_concat([1,], [2,]), [1, 2])
```

+ is numeric. String concatenation is `str_concat`. Array concatenation is `array_concat`. Write `"a" + "b"` and the file is refused:

```
string and array concatenation use str_concat and array_concat
```

Write `"ab" * 2` and you get:

```
the * operator is numeric
```

Integer `/` is refused for a different honesty reason. In the host, `7 / 2` is the exact Rational `7/2`, so the *result type changes with the operands*. A simple arrow would lie. Use `div` (and `%` / `mod`) when you want an Integer:

```
#language axioma/hm
expect("div", 7 div 2, 3)
expect("mod", 7 % 2, 1)
```

```
the result type of / depends on exactness (Integer / yields Rational); not inferred
```

On the island, "not inferred" is fatal — the same sentence the host lens prints as a skip is here a rejection.

Two Floats add as Float (`1.0 + 2.0` is fine). An unannotated `add: func(x, y) [x + y]` infers

```
add :: (Integer, Integer) -> Integer
```

because nothing in the body pins a Float. Calling that `add` with `1.5` is then a type error (`Integer` and `Float` cannot be the same type). Mixed exactness is a host convenience; the island does not smuggle it in.

54.5 `if` is Boolean, and both branches agree

Host Axioma treats every value as truthy or falsy, so `if 1 then …` runs. The island types the condition as `Boolean`:

```
#language axioma/hm
expect("if true", if true then 2 else 3, 2)
expect("if false", if false then 2 else 3, 3)
```

`if 1 then 2 else 3` is refused:

```
an if condition must be Boolean
```

The `else` is required. An `if` without one has no single result type. The two branches must also agree:

```
# this is refused
f: func(b) [if b then 1 else "a"]
```

```
the two branches of an if disagree: Integer and String cannot be the same type
```

Arrays are homogeneous for the same reason:

```
the elements of an array disagree: Integer and String cannot be the same type
```

Write `[1,]` when you want a one-element array. `[1]` in some positions is a block; the trailing comma makes the array reading unambiguous (Chapter 6, and the Manual's bracket-by-position rule).

54.6 Recursion is allowed — and monomorphic

The island is not a strongly-normalizing calculus. `fib` is legal:

```
#language axioma/hm
func fib(0) [0]
func fib(1) [1]
func fib(n) [fib(n - 1) + fib(n - 2)]
expect("fib 10", fib(10), 55)
```

```
fib :: Integer -> Integer
```

A multi-clause group is **one function**. The literal patterns `0` and `1` ground the argument at `Integer`; the recursive clause has to agree. That is why `fib` is not `a -> a`. Recursion on the island is **monomorphic**: the function is used at one type, consistently.

The same pattern writes integer power:

```
#language axioma/hm
func pow(b, 0) [1]
func pow(b, n) [b * pow(b, n - 1)]
expect("pow", pow(2, 10), 1024)
```

```
pow :: (Integer, Integer) -> Integer
```

These examples have unguarded final clauses so every Integer input selects a clause. Their intended terminating inputs are nonnegative: negative inputs can recurse indefinitely. Type checking and coverage do not prove termination. The former guarded final clauses left negative inputs returning `none`; the island now rejects that gap. A Boolean guard never supplies coverage by itself.

54.7 The catalog, and functions that return functions

Only a small, typed catalog is in scope. Everything else — including host builtins you use every day — has no type here, so using it is a rejection.

Name	Island type
`str` / `string`	`a -> String`
`len`	`Array of a -> Integer`
`empty?`	`Array of a -> Boolean`
`str_concat`	`(String, String) -> String`
`array_concat`	`(Array of a, Array of a) -> Array of a`
`map`	`(a -> b, Array of a) -> Array of b`
`filter`	`(a -> Boolean, Array of a) -> Array of a`
`foldl` / `fold_left`	`((b, a) -> b, b, Array of a) -> b`
`foldr` / `fold_right`	`((a, b) -> b, b, Array of a) -> b`
`abs`	`Integer -> Integer`
`float`	`Integer -> Float`
`integer`	`Float -> Integer`
`expect`	`(String, a, a) -> ()`

You *write* `Array of T` in an annotation (`xs :: Array of Integer`). `--infer` *prints* `Array[Integer]`. Same type, two surfaces: `of` is the annotation keyword; the printer uses the compact form. Never write `Array[T]` as an annotation — on the island, brackets are bodies.

Math constants `PI`, `PHI`, `TAU`, `EULER`, `SQRT2`, `SQRT3`, `LN2`, `LN10` are `Float`.

`reduce` is a host name. On the island, fold with `foldl`. `append` is a host name that mutates or writes a file — it is not array concatenation:

```
append has no known type here (a builtin, a later definition, or an unbound name)
```

Higher-order functions from Chapter 10 work, with the island's `func` (or `lambda`) and the catalog:

```
#language axioma/hm
double: func(x) [x * 2]
xs: [1, 2, 3]
expect("map", map(double, xs), [2, 4, 6])
expect("filter", filter(func(n) [n % 2 == 0], [1, 2, 3, 4]), [2, 4])
expect("foldl", foldl(func(acc, n) [acc + n], 0, xs), 6)
```

```
double :: Integer -> Integer
```

A function that *returns* a function is in the fragment. `compose` infers a curried remainder:

```
#language axioma/hm
compose: func(f, g) [func(x) [f(g(x))]]
inc: func(n) [n + 1]
double: func(n) [n * 2]
h: compose(double, inc)
expect("compose", h(3), 8)
```

```
compose :: (a -> b, c -> a) -> c -> b
inc     :: Integer -> Integer
double  :: Integer -> Integer
```

`apply` is the simplest higher-order pattern:

```
#language axioma/hm
apply: func(f, x) [f(x)]
expect("apply", apply(func(n) [n + 1], 4), 5)
```

```
apply :: (a -> b, a) -> b
```

Partial application is legal. `add(1)` is a function waiting for the second argument:

```
#language axioma/hm
add: func(x, y) [x + y]
inc: add(1)
expect("partial", inc(2), 3)
expect("saturated", add(1, 2), 3)
```

54.8 Closed means closed

Form refusals name the closed language. For example:

```
none is not available in axioma/hm (the closed Hindley–Milner island).
Hint: use if/then/else, func, and the island catalog, or #language axioma/all
```

A short map of refusals. Each line describes a program that exits 1. Checked HM dependencies are admitted in §54.13; host dependencies remain outside the proof.

You write	The island says
`println(1)`	`println has no known type here`
`"a" + "b"`	`string and array concatenation use str_concat and array_concat`

You write	The island says
`1 / 2`	`the result type of / depends on exactness`
`if 1 then 2 else 3`	`an if condition must be Boolean`
`[1, "a"]`	`the elements of an array disagree`
`none`	`none is not available in axioma/hm`
`x: 1` then `x: 2`	`re-binding x is not available`
`x: 1` then `x = 2`	`reassignment is not available`
Import a file without `#language axioma/hm`	`requires #language axioma/hm`
`concept C`	`a concept form is not available`
`match 1 with | 1 => 1`	`match is not proven exhaustive`
`axioma --vm file.ax`	`--vm is not available in axioma/hm`

A name binds **once**. `let x = 1` and `x: 1` are both legal introductions. The host's find-or-update `=` (Chapter 2, Chapter 16) is the thing the island takes away: on the island a second write is not "update the same cell," it is a different program. A trailing named function declaration may complete an existing clause group with a catch-all; that extends one function definition. Assigning a new function value to the same name is still forbidden.

There is no `none` and no `om`. A missing value is a missing binding, not a bottom that flows through arithmetic. Host `append`, the knowledge base, concepts, and `is` are all outside the fence.

Annotations that *are* on the island are `Integer`, `Float`, `String`, `Boolean`, `Array of T`, `List of T`, products, arrows and the nominal data types introduced in §54.11. Never `Array[T]` — brackets are bodies. A declared return is checked against the body:

```
#language axioma/hm
bad: func(n) :: String [n + 1]
```

```
declares it returns String, but its body produces Integer
```

That is an error, not a skip — you wrote the obligation.

54.9 When to step onto the island

Use `#language axioma/hm` when the *point* of the file is that it has a type:

- a homework kernel whose whole job is `map` / `filter` / `foldl` over arrays of numbers or strings
- a function you want `--infer` to print, with exit 1 if it cannot
- a demonstration of let-polymorphism that must not silently fall back to "not inferred"

Stay on `axioma/all` (or `axioma/beginner`) when you need the host: `println`, files, the knowledge base, concepts, untyped or bounded data, `none`, mutation, or mixed numeric tower behavior.

The two are not in competition. The host lens `--infer` still exists and still skips. The island is for the files where a skip would be the wrong answer.

The island now includes immutable parametric data and the `match` fragment in §54.11, constructor clauses and collection patterns in §54.12, and checked module interfaces in §54.13. Imports of other island files remain deferred.

The reference for the dialect is the Manual section *`#language axioma/hm` — the closed Hindley–Milner island*, and the tests under `tests/axioma/hm/`.

54.10 Optional: write the polymorphic relationship

Inference discovers a type; a signature can require a particular relationship. Miranda writes repeated stars for shared type variables. Axioma binds named variables explicitly in a detached signature:

```
#language axioma/hm
third_same[T] :: (T, T, T) -> T
third_same(t) = t[3]
third_any[A, B, C] :: (A, B, C) -> C
third_any(t) = t[3]
expect("same", third_same((1, 2, 3)), 3)
expect("another instantiation", third_same(("a", "b", "c")), "c")
expect("independent", third_any((1, true, "3")), "3")
```

Both functions take **one tuple**. Repeated `T` means one shared type, including the result. Different names permit different types, without requiring them. The names are local to the signature, and each call chooses fresh types.

This separate program is intentionally rejected before it runs:

```
#language axioma/hm
third_same[T] :: (T, T, T) -> T
third_same(t) = t[3]
third_same((1, 2, "3"))
```

The Integer and String components cannot satisfy the same `T`. The checker must also reject a body that only works for one choice, even without any call:

```
#language axioma/hm
bad[T] :: T -> T
bad(x) = 0
```

An arbitrary `T` need not be Integer. Checking the body with an arbitrary type is stronger than testing the function with several integer examples.

Three arrow inputs instead describe three separate value arguments. Partial application keeps their shared type. Products and Arrays can also appear in the result:

```
#language axioma/hm
third[T] :: T -> T -> T -> T
third(x, y, z) = z
let after_one = third(1)
expect("partial chooses Integer", after_one(2, 3), 3)
singleton[T] :: T -> Array of T
singleton(x) = ([x])
expect("array result", singleton("s"), ["s"])
```

The parentheses around `[x]` make it an array-valued expression; an equation body written simply `[x]` is a statement block returning `x`.

On the gradual host, the same declarations enforce call-boundary contracts. `--typecheck` reports certain mismatches, but unknown bodies retain runtime checks; a successful integer call does not prove universal validity. Use the HM island when static rejection is the requirement. Explicit `Array of T` runtime checks inspect elements; empty arrays leave element types undetermined. On the host, bare `T` retains outer runtime identity: two Arrays may have different element types. The HM island uses structural ML types and may reject such a call; choosing that stricter dialect is explicit. Shared variables do not promote Integer to Float. Supported VM function bodies execute these signatures in the default host language. The closed `axioma/hm` driver still requires evaluator execution.

The HM proof covers unbounded variables, concrete island types, products and nested `Array of T` and `List of T`, plus higher-order arrows. Variables may occur only in the result, as in a universally empty List. List literals and `map`, `filter`, `foldl`/`foldr`, `len` and `empty?` retain List identity. Host bounds such as `T of Number`, defaults and refinements remain available (Chapter 10), but this proof does not cover them: the host lens reports a skip, and the HM island refuses them. Generic callback signatures are checked in this pure fragment. Immutable parametric ADTs and typed `match` extend that proof in §54.11. Unannotated higher-order inference from §54.7 remains available. Generic clause groups also retain the host's strict return contract: an unmatched call's `none` must satisfy the result annotation. The island has no bottom values.

54.11 Optional: carry a type through data and patterns

A constructor preserves its data parameters in the static model. Matching a `Box of T` therefore gives its field type T to the branch:

```
#language axioma/hm
data Box[T] = Boxed(T)
unwrap[T] :: Box of T -> T
unwrap(box) = match box with | Boxed(value) => value
expect("integer box", unwrap(Boxed(42)), 42)
expect("boolean box", unwrap(Boxed(true)), true)
```

`--infer` prints `unwrap :: Box[a] -> a`. The square brackets here are a printed HM type, not Axioma type-application syntax: source uses `Box of T`. Replacing `value` with `0` fails before execution: an Integer

cannot stand for an arbitrary T. All branches must agree on their result type, including branches that a particular example call never visits.

Data declarations must be top-level, immutable and unbounded. Their fields must use the island type grammar: primitive types, explicit data parameters, products, Array/List elements, arrows and earlier or self-recursive nominal types. Distinct declarations remain distinct even with identical fields. Multiple parameters use `Duo of (A, B)`. Recursion keeps one instantiation:

```
#language axioma/hm
data Tree[T] = Leaf(T) | Branch(Tree of T, Tree of T)
leftmost[T] :: Tree of T -> T
leftmost(tree) = match tree with | Leaf(x) => x | Branch(left,right) => leftmost(left)
expect("leftmost", leftmost(Branch(Leaf("left"),Leaf("right"))), "left")
```

Patterns may use constructors, tuples, literals, captures, `_`, `as` and or-patterns. Named constructor fields are supported without rest captures. Their bindings are local to the branch; each guard must be Boolean. An ordinary `match` needs proven coverage using unguarded constructor/tuple/Boolean cases or a catch-all, because a miss otherwise returns `none`. Nested patterns count: `Boxed(true)` and `Boxed(false)` cover `Box of Boolean`; `Boxed(true)` alone does not. Guards never establish coverage, even a literal `true` guard. `match/strict` permits an uncovered case because a miss raises an Error; this is type checking, not a guarantee against runtime errors or nontermination.

In the closed island, constructors are first-class callbacks and partially apply: pass `Boxed` directly to `map`, or write `Both(1)(true)` for a two-field constructor. The ordinary host retains its constructor arity rules. Concrete positional fields require the exact static type; a named `value :: Float` field may promote Integer to Float. Computed values retain the value restriction, including values containing empty collections or phantom parameters.

This section supersedes the earlier ADT/match exclusion. Host bounds, mutable records, untyped fields, union fields and record-rest/pin/extractor patterns remain outside the proof. Adjacent data declarations may be mutually recursive; references across intervening non-data statements remain outside the proof. Section 54.12 extends the proof to constructor clauses and Array/List patterns. The host `--typecheck` reports definite generic signature contradictions; unsupported bodies retain runtime checks. `--infer` states skips and the closed island rejects them. The VM continues to refuse data and match.

54.12 Optional: typed clauses and collection patterns

A detached signature also supplies the types for constructor-pattern function clauses. Each clause must fulfill the same promise:

```
#language axioma/hm
data Choice[T] = Chosen(T) | Missing
choose[T] :: Choice of T, T -> T
choose(Chosen(value), _) = value
choose(Missing, fallback) = fallback
```

```
expect("chosen clause", choose(Chosen(7), 0), 7)
expect("missing clause", choose(Missing, "fallback"), "fallback")
```

The first field and the second argument both have the arbitrary type T. Returning `0` from either clause would fail the signature check before a call. Strict clause groups require proven coverage across all arguments, using unguarded cases or a catch-all. A guard may reject an otherwise matching case, so it contributes no coverage. This also applies to scalar clauses in §54.6.

Collection patterns use the existing bracket syntax for either Array or List. The checker needs to know which collection the input has, from a signature, a constructor field or an already typed expression; it does not choose Array just because a pattern uses brackets.

```
#language axioma/hm
list_size[T] :: List of T -> Integer
list_size([]) = 0
list_size([head | tail]) = 1 + list_size(tail)
list_tail[T] :: List of T -> List of T
list_tail(xs) = match xs with | [] => (`[]) | [head | tail] => tail
array_tail[T] :: List of T -> Array of T
array_tail(xs) = match xs with | [] => ([]) | [head, ..rest] => rest
expect("size", list_size(`[1,2,3]), 3)
expect("shared List tail", list_tail(`[1,2,3]), `[2,3])
expect("Array rest", array_tail(`[1,2,3]), [2,3])
```

For a List, suffix `[head | tail]` captures a List tail. Dot-rest `[head, ..rest]` captures an Array, as do middle or prefix rest slices. For an Array input, every rest capture is an Array. All captured elements keep the same element type. The empty and nonempty cases cover every finite length; `[]` together with `[head]` would leave longer inputs uncovered.

The checker also considers constraints on elements, including nested Boolean and constructor patterns. Ordinary concrete signatures can supply that context:

```
#language axioma/hm
flag :: Array of Boolean -> Integer
flag(xs) = match xs with | [] => 0 | [true, ..] => 1 | [false, ..] => 2
expect("empty", flag([]), 0)
expect("first true", flag([true,false]), 1)
expect("first false", flag([false,true]), 2)
```

`Array of Boolean` admits two possibilities for each element. Omitting the `false` arm loses coverage for nonempty Arrays beginning with false. Named constructor fields, tuples, literals, `as` and or-patterns compose with these collection patterns. Guards still require Boolean. `match/strict` retains its explicit runtime-error alternative when coverage is incomplete.

The extension uses existing syntax. Record-rest, pin and extractor patterns, anonymous destructuring parameters and untyped collection inputs remain outside this proof. Section 54.13 extends it to checked module interfaces. Host `--infer` can still skip; its older scalar-only clause readings do not promise coverage. VM clause/data/match refusals remain. Static function calls keep exact input types: a Float

parameter requires a Float expression, even though a declared Float result can promote an Integer result at the runtime return boundary.

54.13 Optional: check a module's public promises

An export list chooses which names a caller can use. In the closed island, the checker also proves the types of those names from their definitions and detached signatures. A wrong generic body fails even if no caller invokes it.

```
#language axioma/hm
module Counter [
  data Count = Counted(Integer)
  make :: Integer -> Count
  make(n) = Counted(n)
  read :: Count -> Integer
  read(Counted(n)) = n
  export Count, make, read
]
value :: Counter.Count: Counter.make(7)
expect("opaque interface", Counter.read(value), 7)
```

The caller can name `Counter.Count` and pass its values to the exported functions. The unexported constructor `Counted` is hidden, including through `Counter.Count.Counted`. This is nominal data: another module's independently declared Count is a different static type. Use distinct type and constructor names for an opaque API; exporting a binding that names both also exports its constructor role under Axioma's existing runtime model.

Use `/transparent` when callers should construct and match the data themselves:

```
#language axioma/hm
module Boxes [
  data Box[T] = Boxed(T)
  export Box/transparent
]
read_box[T] :: Boxes.Box of T -> T
read_box(Boxes.Boxed(value)) = value
expect("generic Integer", read_box(Boxes.Boxed(42)), 42)
expect("generic Boolean", read_box(Boxes.Boxed(true)), true)
```

The qualified type retains T in the pattern capture. For a declaration whose type and constructor share the name Box, a qualified pattern uses the explicit constructor property `Boxes.Box.Box(value)`.

The same interface works across files. Move a module body's definitions into a file beginning with `#language axioma/hm`, keep its export list, and load it with `import "./counter.ax" as Counter`. A file header such as `module Counter` is an optional label; import still takes a path. Qualified, selective, renamed, wildcard and `hiding` imports are checked. `include "./counter.ax"` additionally re-exports

the imported names. Nested module bodies also work.

A checked module without a named signature needs an explicit export list (an `include` can supply it). Section 54.14 adds contract-supplied exports. Write exports after the definitions they name, as in ordinary Axioma execution. A module exports its own definitions and imports, not merely captured outer names. In-file modules may capture enclosing values and shadow enclosing names. Import, export and module declarations belong at file or module scope, outside function bodies. The ordinary host's exports without a list are unchanged.

Checking reads dependencies without running user code. Every disk dependency must itself be an HM file; host, builtin, FFI and embedded package imports remain outside this closed proof. Cycles refuse. Execution uses the checked source snapshots and evaluates each canonical file once per HM run. Two aliases or an include façade preserve that file's nominal type identity. The value restriction also crosses interfaces: an exported function returning a shared, unresolved empty Array cannot acquire incompatible element types through different aliases.

The runnable `tests/axioma/hm/module_interfaces.ax` exercises those file forms. The CLI checker, `--infer` and LSP use the same dependency analysis; completions on a checked namespace show only its public members. This adds checked interfaces to existing syntax. Section 54.14 adds reusable signatures and opaque module ascription. Functors, sharing constraints and first-class modules remain outside the closed proof. Host-to-HM imports and VM module/import execution remain refused.

54.14 Optional: reuse a module signature

An export list chooses names. A named signature also states their required types before an implementation exists. Here the public promise is written once:

```
#language axioma/hm
signature VaultAPI [
  data Slot[T]
  put[T] :: T -> Slot of T
  get[T] :: Slot of T -> T
]
Vault :: VaultAPI
module Vault [
  data Slot[T] = Stored(T)
  put(value) = Stored(value)
  get(Stored(value)) = value
]
expect("Integer", Vault.get(Vault.put(42)), 42)
expect("Boolean", Vault.get(Vault.put(true)), true)
```

`data Slot[T]` declares an abstract type with one parameter. It gives clients a type name without a constructor. The contract supplies the annotations for the unannotated put/get definitions and checks their bodies before execution. Each call chooses its own T: the Integer and Boolean assertions are independent. Changing get to return `0` violates its arbitrary T result, even without a call. Omitting get

fails the required-member check.

The detached annotation must immediately precede the matching module body. The inline spelling places it on the module line, as in the next example. The contract supplies the complete public surface: Slot, put and get. Stored and extra helpers remain private, including constructor properties reached through the exported type. No export list is needed. If one is written, it must name exactly the contract members. Abstract constructors cannot be exported transparently.

A second implementation can use different storage while keeping the same API:

```
#language axioma/hm
signature VaultAPI [
  data Slot[T]
  put[T] :: T -> Slot of T
  get[T] :: Slot of T -> T
]
module StampedVault :: VaultAPI [
  data Slot[T] = Packed(T, Integer)
  put(value) = Packed(value, 0)
  get(Packed(value, stamp)) = value
]
saved :: StampedVault.Slot of String: StampedVault.put("hello")
expect("alternative storage", StampedVault.get(saved), "hello")
```

Each ascribed module seals its abstract types with a fresh static identity. Values from Vault cannot be passed to StampedVault functions, or vice versa. The shared contract does not make their Slot types equal. Two imports of the same canonical implementation file do preserve its identity, as in §54.13. This boundary is checked statically; runtime data remains erased.

Existing implementation annotations remain in force and must satisfy the contract. A generic identity function can satisfy an Integer identity contract; clients see the concrete arrow. An explicitly Boolean identity implementation cannot satisfy it. Shared unresolved Array values also cannot acquire universal type parameters through ascription.

A contract can live in an HM file ending with an export of VaultAPI. Import it selectively, rename it, or import its file as a namespace called Contracts. An implementation can then name Contracts.VaultAPI in its module annotation. The runnable module_signatures.ax fixture in tests/axioma/hm exercises these forms and checked file implementations. A signature is a declaration, not a first-class callable value.

This first signature grammar contains abstract unbounded data heads and explicit function arrows over those heads, HM primitive types, Arrays, Lists, tuples and functions. Generic function binders are unbounded. Commas outside tuple parentheses still separate curried arguments. The first two arrows below are equivalent; the third takes one tuple. Constant in-range indices retrieve its components, for concrete as well as generic tuple signatures.

```
#language axioma/hm
signature Arithmetic [
  add :: Integer, Integer -> Integer
  add_again :: Integer -> Integer -> Integer
```

```
  pair_sum :: (Integer, Integer) -> Integer
]
module Numbers :: Arithmetic [
  add(x, y) = x + y
  add_again(x, y) = x + y
  pair_sum(pair) = pair[1] + pair[2]
]
inc: Numbers.add(1)
expect("partial application", inc(4), 5)
expect("arrow arguments", Numbers.add_again(2, 3), 5)
expect("one tuple", Numbers.pair_sum((2, 3)), 5)
```

Section 54.15 extends contracts with type equations and references to existing types; §54.16 adds fixed-type value members. Bounded members, nested signature/module specifications, functors and general sharing constraints remain outside the proof. Declarations belong at file/module scope; ascription requires a module body, not a file-header label. Both the ordinary host and VM refuse these forms. CLI checking, inference and LSP use the closed HM driver. This adds reusable opaque signatures without claiming a complete ML module calculus.

54.15 Optional: publish a type equation

An abstract type hides its implementation. A type equation deliberately states an equality that clients may use. A signature can contain both:

```
#language axioma/hm
signature StoreAPI [
  type Key = Integer
  data Slot[T]
  put[T] :: Key, T -> Slot of T
  get[T] :: Slot of T -> T
]
module Store :: StoreAPI [
  type Key = Integer
  data Slot[T] = Stored(Key, T)
  put(key, value) = Stored(key, value)
  get(Stored(key, value)) = value
]
key :: Store.Key: 7
expect("Integer key, Boolean value", Store.get(Store.put(key, true)), true)
expect("independent payload", Store.get(Store.put(8, "hello")), "hello")
```

Key is exactly Integer, so an ordinary integer can be passed wherever Store.Key is expected. Slot keeps its private representation and fresh static identity. Each implementation must provide the required type binding. Changing its Key equation to String fails before execution, even when no function is called. The equality check is exact: Integer and Float are different types here, even though a Float return annotation

can promote an Integer at runtime.

Equations use the existing transparent type-alias form. Their right-hand sides may contain HM primitive types, tuples, Arrays, Lists, function arrows, and fully applied nominal data types. They may use the signature's abstract heads or earlier equations. For example, an alias for Slot applied to Integer still contains that implementation's private Slot identity. Write dependent equations after their dependencies. Forward, recursive and mutually recursive aliases are rejected; parametric recursive data remains available through data declarations.

An equation can also refer to a type that already exists:

```
#language axioma/hm
module Shared [
  data Item = Made(Integer)
  export Item/transparent
]
signature EchoAPI [
  type Entry = Shared.Item
  echo_item :: Shared.Item -> Entry
]
module Echo :: EchoAPI [
  type Entry = Shared.Item
  echo_item(item) = item
]
entry :: Echo.Entry: Shared.Made(42)
expect("same external type", Echo.echo_item(entry), Shared.Made(42))
```

Echo.Entry and Shared.Item are the same static type. This equation creates no new constructor and does not add constructor properties to Echo.Entry. Clients may use Shared.Made because Shared exports it. A reference to an opaque external type preserves its privacy. Two implementations that publish the same equation can exchange values of that type, while their abstract members remain distinct.

External type names in equations and function arrows resolve where the signature is declared. They may be unqualified or reached through an imported or enclosing module. An implementation declaring a different Shared or Item cannot redirect the contract. Supplied function annotations retain those original types at both checking and execution. An explicitly written implementation annotation keeps its own lexical meaning and must satisfy the contract's meaning.

Contracts retain these references when exported, imported or renamed. A client need not import a signature's dependencies under the same names. The runnable signature_type_equations.ax fixture in tests/axioma/hm demonstrates imported generic types, private slots and deliberate name shadowing.

Transparent aliases are also admitted at file/module scope in the closed HM island so implementations can provide these bindings. Aliases are fully defined types with no parameters of their own; an alias may specialize a parametric data type. Unbounded generic function signatures and abstract data heads still accept type parameters. Local aliases inside functions, alias parameter binders, opaque-wrapper aliases, unions, bounds, polymorphic value members, nested specifications, functors and general sharing constraints remain outside this extension. Source module signatures remain HM-only, with explicit host and VM refusals.

54.16 Optional: require an initial value

A reusable signature can require a value as well as operations. Here the client receives an initial Count without knowing how a Count is represented:

```
#language axioma/hm
signature CounterAPI [
  data Count
  zero :: Count
  increment :: Count -> Count
  read :: Count -> Integer
]
module Counter :: CounterAPI [
  data Count = Counted(Integer)
  zero: Counted(0)
  increment(Counted(n)) = Counted(n + 1)
  read(Counted(n)) = n
]
expect("initial state", Counter.read(Counter.zero), 0)
expect("next state", Counter.read(Counter.increment(Counter.zero)), 1)
```

The line `zero :: Count` is a requirement, with no initializer. The implementation provides an ordinary value binding. Its initializer is inferred and its type must match the requirement before execution. Missing values and mismatched types fail even when no client reads the member. The contract supplies the public names; an explicit export list must contain exactly those names, including zero.

Count stays abstract. The value Counter.zero may be passed to Counter.increment and Counter.read, while Counted stays private. Another independently ascribed Counter implementation has a distinct Count type. Value requirements may refer to earlier type equations, external types and fully applied generic data, using the signature's declaration scope. Imported and renamed contracts retain those meanings. A required binding can also be an imported ordinary value.

These requirements have fixed types, without generic binders. They can describe primitives, tuples, Arrays, Lists and specialized data. A fully defined arrow alias can describe a function-valued member. The exported type is the declared type even when an implementation function is more general. Signatures still use explicit arrows for generic function members.

Shared collections keep one consistent element type:

```
#language axioma/hm
shared: []
signature BufferAPI [
  items :: Array of Integer
]
module One :: BufferAPI [items: shared]
module Two :: BufferAPI [items: shared]
```

```
items :: Array of Integer: Two.items
expect("shared fixed type", One.items, items)
```

The first ascription fixes the shared Array's element type to Integer. The constraint remains visible through other aliases, enclosing functions and file imports. Requiring the same shared Array as Array of Boolean in another module would fail. Each separately constructed empty Array may instead receive its own fixed element type. The requirement does not copy or re-evaluate the value.

Matching is exact after aliases expand. For example, a Float value requirement rejects an unannotated integer initializer. An explicit binding annotation such as `zero :: Float: 0` uses the existing binding conversion before the contract checks its resulting Float type. Ascription itself adds no value annotations or numeric conversions. A nullary constructor can initialize an ordinary binding; the constructor declaration itself does not satisfy an ordinary value member.

This extends only module-signature bodies. Ordinary detached function signatures outside them retain their previous rules. Polymorphic value requirements, parameterized aliases, nested specifications, sharing constraints and functors remain deferred. Host and VM signature execution still refuse. The runnable signature_values.ax fixture in tests/axioma/hm also covers imported values, external generic types, tuples and Lists.

54.17 Optional: functional composition across definitions

A constructor can be a callback. Adjacent ordinary function definitions form a recursive block, so each can call the others. The checker groups actual lexical dependencies: recursive uses share one type, then the completed group becomes polymorphic. Independent helpers remain polymorphic within the same block. A value statement, detached signature, or fresh binding ends the block; later bindings are not executed early. Adjacent data declarations likewise reserve their nominal type heads before checking fields.

```
#language axioma/hm
data Box[T] = Boxed(T)
expect("constructor callback", map(Boxed,[1,2]), [Boxed(1),Boxed(2)])
data Duo[A,B] = Both(A,B)
expect("constructor partial", Both(1)(true), Both(1,true))

even(n) = if n == 0 then true else odd(n-1)
odd(n) = if n == 0 then false else even(n-1)
expect("mutual recursion", even(1000), true)

data EvenChain[T] = ChainEnd | ToOdd(T, OddChain of T)
data OddChain[T] = ToEven(T, EvenChain of T)
expect("mutual data", ToOdd(1,ToEven(2,ChainEnd)), ToOdd(1,ToEven(2,ChainEnd)))
expect("checked primitive", sqrt(9.0), 3.0)
expect("persistent cons", cons(1,`[2,3]), `[1,2,3])
```

All six assertions pass. Constructor aliases keep the existing value restriction; passing a constructor

directly instantiates its scheme at each use. Positional field types remain exact, and runtime checks still enforce shared parameters.

Eligible clause and destructuring tail calls now use the evaluator's tail loop. An accumulator expressed as two equations can process 100,000 steps. Generic tail calls also keep a bounded Go stack, but retain distinct return promises and container guards on the heap, with a 100,000-pending-check limit. This is not a termination proof or a constant-memory guarantee. Non-tail calls, named/refined/ variadic paths, contracted clause/generic calls, and generic calls in transactions retain their stated recursion and enforcement boundaries. VM work is deferred.

For reusable checked code, `lib/hm/Math.ax` exports Float mathematics, and `lib/hm/Result.ax` exports a nominal Result type plus mapping, recovery, sequence, traversal and validation. `lib/hm/Option.ax` provides a checked Option type. `traverse` evaluates callbacks from left to right and stops on the first error result. These modules declare `#language axioma/hm`, have explicit exports, and are checked before any dependency executes. Ordinary host and FFI imports stay outside the closed proof. The runnable repository fixture `tests/axioma/hm/functional_core.ax` demonstrates both imports.

54.18 Optional: compose a functional program

Run these examples from the repository root, where the checked library files exist on disk. Their types have their own nominal identity; host prelude Results and Options are not interchangeable with these imported constructors.

```
#language axioma/hm
import "lib/hm/Result.ax" as R
import "lib/hm/Option.ax" as O
positive(n) = if n > 0 then R.Ok(n+1) else R.Err(str(n))
expect("first error only", R.traverse(positive,`[0,1,-2]), R.Err("0"))
expect("all errors in order", R.validate_all(positive,`[0,1,-2]), R.Err(`["0","-2"]))
expect("collect successful results", R.sequence(`[R.Ok(2),R.Ok(3)]), R.Ok(`[2,3]))
expect("absence becomes an error", O.to_result("missing",O.Absent), R.Err("missing"))
fallback() = O.Some(9)
expect("compute fallback only for absence", O.or_else(fallback,O.Absent), O.Some(9))
```

All five assertions pass. Traversal stops calling the callback on the first Err; validation calls it for every input and retains repeated errors. These policies handle Result values. An ordinary runtime error or divergent callback still stops the program. The checked Option fallback takes zero arguments because absence carries no payload. `unwrap_or` instead receives an eagerly computed value.

The expression project connects a token-stream parser, an interpreter and a symbolic simplifier. Its recursive Expr ADT represents integers, variables, addition, multiplication and negation. The parser honors precedence and parentheses, and reports zero-based token positions. Text scanning is a separate boundary. Compose its checked modules:

```
#language axioma/hm
import "lib/hm/Result.ax" as R
import "demos/functional_expression/Parser.ax" as P
import "demos/functional_expression/Interpreter.ax" as I
import "demos/functional_expression/Simplifier.ax" as S
run(tokens) = R.and_then(func(expr) [I.evaluate(S.simplify(expr),`[])],
  R.map_err(P.describe_problem,P.parse(tokens)))
expect("parse simplify evaluate",
 ↪  run(`[P.TokInt(2),P.Plus,P.TokInt(3),P.Times,P.TokInt(4)]), R.Ok(14))
expect("preserve a missing variable", run(`[P.TokInt(0),P.Times,P.TokName("x")]),
 ↪  R.Err("unbound variable: x"))
```

Both assertions pass. Replacing `0 * x` with `0` could hide a missing-variable error in this strict interpreter, so the simplifier keeps that expression. The showcase checks result preservation over 12 expressions and four environments, including missing names and shadowing. That finite check is not a general proof.

Callbacks may declare their collection element types explicitly:

```
#language axioma/hm
import "lib/hm/Result.ax" as R
count_list(xs :: List of Integer) = len(xs)
expect("typed List payload", R.map_ok(count_list,R.Ok(`[1,2])), R.Ok(2))
expect("empty typed List payload", R.map_ok(count_list,R.Ok(`[])), R.Ok(0))
```

Both assertions pass. The HM checker relates the callback's `List of Integer` to the Result payload. Runtime compatibility respects the outer List identity of a bare generic parameter; the callback itself still checks its declared element type before its body runs. Explicit structural contracts retain their element relationships. A String List is rejected by HM before execution and by the callback boundary in ordinary host evaluation.

The library workers pass persistent List tails directly and reverse accumulated output once. They avoid a new generic function contract for each element; 20,000-element traversal and validation pass with a call-depth budget of 300. The parser and recursive AST walks still retain ordinary non-tail depth limits.

Reflection

Chapter 19 encoded *invariants you wrote*. This chapter encodes *the fragment that can be inferred*. The deep idea is the same: the best way to handle a class of bugs is to make them impossible to write. The island's move is to shrink the language until "impossible to write" is also "impossible to run."

Three honesty rules that are easy to forget:

1. **A lens is not a language.** Host `--infer` exiting 0 on `shout` is not a type of `shout`.
2. **Closed means the catalog, not a warning.** `println` is not "discouraged"; it has no type, so the program is not an island program.
3. **`+` does not mean "glue."** Numbers add. Strings `str_concat`. Arrays `array_concat`. One operator with three host meanings is exactly what a principal type cannot say.

The next files you write for this book can stay in `axioma/beginner`. When you want the checker to be the language, put `#language axioma/hm` on line one.

Exercises

Every exercise file starts with `#language axioma/hm`. Do **not** use `println`. Use `expect`. Run with `axioma --no-kb exercises/ch54/ex_54_N_….ax`, then `axioma --no-kb --infer` on the same file and read the arrow.

Exercise 54.1 — `id` at two types

Write `id: func(x) [x]`. Check it at an Integer and at a Boolean.

- `id(1)` → `1`
- `id(true)` → `true`
- `--infer` prints `id :: a -> a`

Exercise 54.2 — `greet` with `str_concat`

Write `greet(name)` that returns `"hello, "` in front of `name`. Do not use `+`.

- `greet("Ada")` → `"hello, Ada"`
- `--infer` prints `greet :: String -> String`

Exercise 54.3 — integer `pow`

With two clauses, write `pow(b, n)` for `n >= 0`.

- `pow(2, 10)` → `1024`
- `pow(3, 4)` → `81`
- `--infer` prints `pow :: (Integer, Integer) -> Integer`

Exercise 54.4 — `filter` then `map`

From `[-(2), -(1), 0, 1, 2, 3]`, keep the positives and double them.

- result → `[2, 4, 6]`
- use `filter` and `map` from the catalog, not a hand-rolled recursion (you already did recursion in 54.3)

Exercise 54.5 — the legal twins of four refusals

The starter comments four illegal programs. Do not uncomment them. Write the island-legal twins:

Illegal	Legal twin
`"a" + "b"`	`str_concat("a", "b")`

Illegal	Legal twin
`7 / 2`	`7 div 2`
`if 1 then 2 else 3`	`if true then 2 else 3`
`x: 1 then x: 2`	two different names

`expect` each legal result. The file must exit 0, and `--infer` must not print a type error.

Exercise 54.6 (open) — a closed kernel of your own

Write a small island program whose *point* is a type. Some prompts, pick one:

- `compose` as in §54.7, plus two instantiations (numeric and Boolean)
- `product` via `foldl`, and a proof by `expect` that `product([])` is `1`
- `apply` used at two types, like `id` in 54.1

Constraints: `#language axioma/hm`, no `println`, every top-level function `--infers` (no skip, no mismatch). The open question: **what did the type make impossible that a host file would have silently done?**

Exercise 54.7 — shared and independent variables

Write `second_same` with signature `second_same[T] :: (T, T) -> T` and `second_any` with signature `second_any[A, B] :: (A, B) -> B`. Test an integer pair and a mixed Integer/String pair where allowed. Explain why different variable names do not require different types. In a separate HM file, change `second_any` to return the first component and explain the rejection.

Exercise 54.8 — a typed missing case

Declare `data Choice[T] = Chosen(T) | Missing`. Give `choose` the detached signature `choose[T] :: Choice of T, T -> T`, and return the stored value or the supplied fallback. Assert both an Integer case and a String fallback. Explain why returning `0` in the Missing branch violates the signature, and why deleting that branch makes an ordinary match fail the island checker.

Exercise 54.9 — keep the tail type

Write `drop_one[T] :: List of T -> List of T` with an empty clause and a nonempty cons clause. Test both empty and nonempty Lists. Explain why replacing `[head | tail]` with `[head, ..tail]` contradicts this signature, and why removing the empty clause fails coverage. Rewrite the result signature for an Array rest.

Exercise 54.10 — an opaque generic slot

Define a module Vault with `data Slot[T] = Stored(T)`. Export the type and functions `put[T] :: T -> Slot of T` and `get[T] :: Slot of T -> T`, keeping Stored private. Check independent Integer and Boolean round trips. Explain why a client cannot implement a constructor-pattern getter through this interface.

Exercise 54.11 — one promise, two implementations

Move exercise 54.10's public promise into `signature VaultAPI [...]`. Ascribe Vault to it without repeating the function annotations or export list. Implement another module with `Packed(T, Integer)` storage and the same contract. Assert round trips in each. Explain why a value from one module cannot be passed to the other's get, and why returning `0` from a generic get fails before either test.

Exercise 54.12 — expose an equality, keep an abstraction

Extend the VaultAPI from exercise 54.11 with a Key equation equal to Integer, and make put accept a key before its generic value. Keep Slot abstract and Stored private. Assert Integer-key round trips for Boolean and String payloads. Explain why another implementation can use the same integer keys but cannot exchange Slot values. What happens if its Key equation is Float instead?

Exercise 54.13 — publish an initial state

Implement CounterAPI from §54.16 with an abstract Count, an initial zero value, and increment/read operations. Assert that reading zero gives 0 and reading one increment gives 1. Explain why another implementation cannot consume this zero, why `zero: true` fails, and why one shared empty Array cannot satisfy both an Integer collection requirement and a Boolean collection requirement.

Exercise 54.14 — Constructors and mutual recursion

Define mutually recursive `even` and `odd` functions in one adjacent declaration block. Define `data Flag[T] = Flagged(T)` and use the constructor directly to map the parity results for 10 and 11. Check that the result is `[Flagged(true), Flagged(false)]`. Explain why inserting a call between the two function definitions ends the recursive block. See the chapter solutions.

Exercise 54.15 — report every failed expression

Using the expression modules from §54.18, write a function that parses and evaluates one token List with an empty environment. Evaluate a batch containing the token streams for `2`, a missing name `x`, and a lone opening parenthesis. Compare traversal's first error with validation's complete error List. Preserve input order, and explain why an Err value differs from a callback runtime error. For a batch containing only valid integer expressions, use an explicitly typed List callback with `map_ok` to count the successful results.

Appendix A · Quick Syntax Reference

A compact reference to the Axioma syntax used in this book, followed by a broader operator precedence and associativity reference. For the full language manual, see Axioma Manual.

Bindings

For the worked scope, mutation, and closure comparison, see §16.14 · Make scope explicit with local.

Form	Meaning	Where it's taught
`x: 5`	Bind x to 5 (canonical value form)	Ch.2 §2.2
`x = 5`	Same find-or-update as `x: 5` (math-style spelling)	Ch.2 §2.4
`a, b: 1, 2`	Multi-assignment (same form, parallel)	Ch.2 §2.5
`x :: Integer: 5`	Bind with type annotation	Ch.2 map; Part VI
`let x = 5`	**Fresh** binding — shadows outer x; does not update it	Ch.2 §2.5; Ch.16 §16.x
`let x`, `let x :: T`, or either with `= _`	**Hole** — assign before read; optional type constraint	Ch.16 §16.x
`let x :: Array = default`	Start as **empty container** (identity); not for Integer/String	Ch.16 §16.x
`const LIMIT = 10`	Immutable name	Ch.2 map; manual
`rebind x = …`	Update a name that lives *outside* this frame (nearest cell)	Ch.16
`local x = …`	Fresh mutable binding in this lexical scope; initializers use =; `local x` or `local x :: T` creates a hole, like `var`	Ch.16
`global x = …`	Julia's file / `module` write; may declare; honors `::`	Ch.16
`rank :: Ordinal: 3`	Position-label annotation — `--typecheck` flags arithmetic	Part VI
`(x :: Float)`	Type **ascription** — checks; promotes Integer→Float	Part VI
`declare x :: Float = e`	Fresh immutable recursive lazy binding; checks/converts on demand and caches success	Ch.10 optional

Form	Meaning	Where it's taught
`lazy e; force(t)`	Explicit deferred value; failures may retry, cyclic demand errors	Ch.10 optional
`f(body/lazy :: Float) = body`	Argument and its value check wait until parameter demand	Ch.10 optional
`declare/persist x = 42`	Mark for saving; demand first, serialization limits apply	Part VI

History. `x := 5` stays retired (use `x: 5` or `x = 5`). The spelling `let x = 5` *does* parse again (August 2026), but it means **fresh shadow**, not the old synonym of `x: 5`. For Part I prefer `x : 5`.

Output and value-printing

Form	Meaning	Where it's taught
`println(x)`	Print `x` followed by a newline	Ch.1
`println(x, y, z)`	Print multiple values separated by spaces	Ch.1
`expr .`	Trailing-dot inspect sigil — evaluates `expr` and prints the result	Ch.1 sidebar
`x: 5 .`	Bind `x` and print `5` in one breath	Ch.1 sidebar

The trailing `.` works at the top level of a script *and* inside the bodies of loops, conditionals, and function/lambda bodies — useful for quick "show me intermediate values" debugging without scattering `println` calls. The REPL prints results automatically and so doesn't need either form for interactive work.

Comments

Form	Meaning
`# line comment`	Single-line comment to end-of-line
`/**md ... */`	Literate-programming block (Markdown documentation)

Functions

Form	Meaning	Ch
`square: func(x) [x * x]`	Named function — the book's canonical form	3
`func square(x) [x * x]`	Same thing, keyword-first	3
`square(x) = x * x`	Same thing, **equation form** (see below)	3
`lambda x => x * x`	Anonymous function	11
`f(3)`	Call	3
`f(3, 4)`	Multi-arg call	3
`3 `f` 4`	Infix call — identical to `f(3, 4)`, nothing to declare	3

The equation form — `name(args) = expr`

The keyword-free spelling, and the one mathematics already uses. It is not a separate kind of function: an all-plain-parameter equation is the same definition as `func name(args) [expr]`, and one with patterns or a guard is the same as a clausal `func`. So contracts, partial application, recursion, `source()`, and exhaustiveness checking all behave identically, and the two spellings may be mixed in one group of clauses.

```
double(x) = 2 * x                  # ≡ double: func(x) [x * 2]
fact(0) = 1                        # clauses accumulate by name,
fact(n) = n * fact(n - 1)          #   tried in source order
sign(x) when x > 0 = 1             # a guard sits between head and `=`
sign(x) = 0
head2([h | t]) = h                 # cons, constructor, tuple, wildcard,
area(Circle(r)) = 3.14 * r * r     #   or- and as-patterns all work
answer = 42                        # no parens ⇒ an ordinary value bind
```

A bracket after the head is the body, exactly as in the `func` spelling — one construct, one body parser, so the two cannot come to mean different things. An array is the double bracket:

```
boxed(x) = [2 * x]                 # the BODY ⇒ 2 * x
arrayed(x) = [[2 * x]]             # the ARRAY ⇒ [2 * x]
literal(x) = ([1, 2, 3])[x]        # a bracket wanted as a value: parenthesize
```

Parameter defaults also work in the equation form: `scale(x, factor = 2) = x * factor`. Typed slots and named call arguments have their own rules; see the function chapter and the current Manual for their full grammar. A capitalized parameter is a nullary **constructor pattern**, not a binder — the same rule `func f(X) […]` and `match` follow. And the bare juxtaposition `double x = 2 * x` is a SyntaxError whose hint names this form: calls are parenthesized in Axioma, so definitions are too.

Works in beginner mode. Taught alongside `func` in §3.5.

Conditionals and Booleans

```
if test then a else b                  # compact; `then` required
if c1 then a else if c2 then b else c  # cascade
if test                                # dual: no `then`, later-line body,
  a                                    #   matching `end` (Ch.2a's loop rule)
else
  b
end
if c1                                  # `elseif` is one chain, one `end`
  a
elseif c2
  b
else
  c
end
not x          # negation
x and y        # also x && y or x ∧ y; same short-circuit rules
x or y         # also x || y or x ∨ y; same short-circuit rules
x == y         # equality
x != y         # inequality
x < y, x <= y, x > y, x >= y           # ordering (glyphs ≤ ≥ ≠ lex too)
0 <= x < 10    # ordering comparisons chain: ≡ (0 <= x) and (x < 10)
x += 1         # compound assignment ≡ x = x + 1  (also -= *= /=;
               #   statement-only; no ++/-- — those error with a hint)
```

Taught in Ch.5.

Choice and pattern matching

Guarded equations work in beginner mode. Cond and matching use the full language:

```
sign_choice(n) when n > 0 = 1
sign_choice(n) = 0
let n = -2
println(sign_choice(n)) # 0
println(cond | n < 0 => "negative" | else => "other") # negative
println(match (1, 2) with | (x, y) when x < y => x + y | _ => 0) # 3
println(case (1, 2) | (x, y) => x + y | _ => 0) # 3
println(switch (1, 2) | (x, y) => x + y | _ => 0) # 3
```

- Function guards may use `f(x) when condition = body` or the result-first `f(x) = body, if condition`; aligned `= body, if condition` continuations reuse the preceding head. `otherwise` supplies a final guarded-equation fallback.
- Cond tests conditions in order; its final fallback is `else`, `otherwise` or `_`. It permits an omitted first bar.
- Match tests patterns in order, then any `when` guard. `match e with` permits an omitted first bar; `ca`

`se e` and `switch e` require it and take no preposition or braces. `e match | …` is the postfix form. Arrows may be => or –>.

- A false guard continues; a guard error propagates. Ordinary no-match returns `none`, distinct from `om`. `/strict` on match/case/switch makes a miss an error. A lowercase variable pattern binds any value; _ matches without binding.
- Cond supports the VM; match/case/switch refuse there. A clean typecheck is not a proof about arbitrary guard predicates or every nested pattern.
- Optional sequential guards: `when pattern <– value, let name = expr, condition`. A one-head group uses indented, aligned lines `| chain = result`. Bindings stay in their branch; errors propagate and the first successful body is final. See the optional guard-sequence section in Chapter 5. Evaluator-only.

Learn selection in §5.9–5.10 and constructor matching/coverage in §18.6. The Manual is the reference for the complete pattern grammar.

Arrays, Sets, Tuples

Form	Meaning	Ch
`[1, 2, 3]`	Array literal, 1-based	6
`` `[1, 2, 3] `` / `` `[] ``	Persistent List literal / empty List; values evaluate normally	23.11
`[...xs]` / `` `[...xs] ``	Spread Array/List elements into a new Array/List	23.12
`arr[1]`	First element (NOT zero!)	6
`arr[2..len(arr)]`	Slice	6
`len(arr)`	Length	6
`a + b`	Array concatenation	6
`{1, 2, 3}`	Set literal	10
`{}`	Empty **set** ∅ (use `dict()` for an empty hash)	—
`(a, b)`	Tuple	7, 8
`t[1]`, `t[2]`	Tuple element access	8

Broadcasting and collection calls

Form	Meaning	Ch
`f.(xs)`	Call `f` element by element	10.11
`f.(xs, 2)`	Reuse the scalar at every position	10.11
`broadcast(f, xs)`	Ordinary eager call; a fusion boundary	10.11

Form	Meaning	Ch
`xs .+ 2`	Elementwise numeric addition	10.11
`g.(f.(xs))`	Connected dotted calls fuse into one output walk	10.11
`xs.map(f)`	Collection-method spelling of `map(f, xs)`	10.11

An ordinary pipe such as `xs |> f.(_) |> g.(_)` makes separate broadcast stages; `_` receives the whole array.

Function broadcasting expands singleton axes; Array dotted arithmetic still requires equal lengths or a numeric scalar. See §10.11 for shape rules, evaluation order, and the interpreter/VM boundary.

Strings

Source files are UTF-8, so most characters can appear directly:

```
"hello"
"naïve façade"
"math: ∀ x ∃ y. x + y = 0"
"greek: λ φ Ω"
```

For characters that resist direct typing, escape sequences decode at parse time:

Escape	Result	Where it's taught
`\n`	LF (newline)	Ch. 2 sidebar
`\t`	TAB	Ch. 2 sidebar
`\r`	CR	Ch. 2 sidebar
`\\ \" \' \0`	literal \, ", ', NUL	Ch. 2 sidebar
`\u{H...H}`	Unicode codepoint (1–6 hex digits, full range)	here
`chr(N)` / `ord(s)`	codepoint ↔ string round-trip	here

```
"line1\nline2"      # → 2 lines
"\u{2203}"          # → ∃        (BMP, 4 hex)
"\u{1D54A}"         # → 𝕊        (supplementary plane, 5 hex)
"\u{1F600}"         # → 😀

chr(8707)           # → "∃"
ord("∃")            # → 8707
```

Unlike Python's `"\uXXXX"`, Axioma uses the braced `\u{...}` form (Rust / Swift / ES6 / Lua / Dart). It handles all 1,114,112 Unicode codepoints in one syntax, with no special treatment for "supplementary plane" characters like 𝕊 and 😀.

Raw strings — `r"..."` and `r'...'`

The `r` prefix turns off escape decoding. Use it for paths, regex patterns, and any genuinely-literal content:

```
winpath:  r"C:\Users\alice\new"     # literal Windows path
regex_text: r"\d+\.\d+"               # literal regex
r'no \n decode here'                # single-quoted form also works
```

Unknown escapes (`"\d"`, `"\s"`, etc.) stay verbatim in regular strings too, so you only need `r"..."` when you have lots of backslashes or want byte-for-byte source match.

Comprehensions

```
# --- core forms ---
[x * 2 | x <- xs]                   # list comp (eager, ordered, dups kept)
{x * 2 | x <- xs}                   # set comp  (eager, unordered, dedup)
{k: v  | x <- xs}                   # dict comp (eager, hash result)
(x * 2 | x <- xs)                   # generator (lazy, pull-based)

# --- Python pipe-less form works in all four ---
[x * 2 for x in xs if x > 0]
{x * 2 for x in xs if x > 0}
{x: x * x for x in xs}
(x * 2 for x in xs if x > 0)

# --- walrus binding inside a clause (the canonical form) ---
{y | x <- xs, y: f(x), y > 0}        # REBOL `:` (the only remaining form)

# --- multi-generator and multi-filter ---
{x + y | x <- xs, y <- ys}          # Cartesian product
{x | x <- xs, x > 1, x < 10, x != 5} # filters fold with `and`

# --- ISO/British set-builder spellings ---
{x : x <- xs, x > 1}                # `:` separator (British) — same AST as `|`
{x : x ∈ xs, x > 1}                 # `∈`/`in` as generator, gated: target must be
                                    #   unbound AND occur in the head; otherwise
                                    #   `x in s` stays a membership FILTER
{(x, y) : x ∈ xs, y ∈ ys}           # the textbook Cartesian form, verbatim

# --- Axioma-only: KB and concept integration ---
{X | X <- Country}                  # bare-concept extent
{X | X is Country, X.gdp > 75}      # is-comprehension
{(X, Y) | parent(X, Y)}             # relation tuple query
{G | P <- parent("alice", P), G <- parent(P, G)}     # variable-chain unification
{X @theorem | X <- ancestor(X, _)}  # tag-filter by epistemic grounding

# --- lazy generator drivers ---
force(gen)                          # → array  (materialise all)
```

```
gen_take(n, gen)                    # → array  (first n)
gen_drop(n, gen)                    # → gen    (advance, mutates)
gen_next(gen)                       # → elem | Ω

# --- set operators ---
A ∪ B                               # union
A ∩ B                               # intersection
A - B                               # difference
x in A                              # membership
A subset B                          # ⊆ (inclusive)
A superset B                        # ⊇ (inclusive)
A subsetneq B                       # ⊂ (proper)
A supsetneq B                        # ⊃ (proper)

# --- typing the glyphs ---
C `sqcap D                          # backtick digraph: lexes AS ⊓ — the DL pair ⊓ ⊔
                                    #   has no word form; the digraph is its only
                                    #   pure-ASCII spelling
2 `in {1, 2, 3}                     # twin-having glyphs work too: `in ≡ ∈ ≡ in
{1} `cup {2}                        # `cup → ∪ (LaTeX names), `union → ∪ (Axioma
 ↪  names)
# a digraph is UNPAIRED. A matched pair is the unrelated infix-call form:
cup(a, b) = a * 1000 + b            #   `cup as a glyph AND cup as a function,
2 `cup` 3                           #   → 2003 — no interference, ever
# axioma --glyphify file.ax         # canonicalize a file: in → ∈, superset → ⊇, and →
 ↪  ∧
# axioma --asciify  file.ax         # the inverse (word-less glyphs stay as typed)
# REPL/playground: \in + Tab → ∈;  editors: \in completion via axioma-lsp
```

Taught in Ch.10 (general comprehensions, walrus, dict, lazy) and Ch.21 (relational and concept-extent comprehensions). Glyph input layers: manual §3 “Typing the glyphs”.

Concepts (Russell-style classes)

```
concept Stock                  # declare concept (single canonical surface)
Stock has price: 0             # add slot with default
Stock has ticker: "", sector: ""   # several slots in one statement, comma-separated

concept Asset                  # parent concept
Stock extends Asset             # specialize (auto-creates child if needed)

# Block form: declare with initial slot defaults in one shot
concept TextbookBond {
  yield: 0.05
  coupon: 0.0
}
```

```
# Refinement + doc string (the postfix slots)
concept/persist TreasurySec "A marketable US Treasury security"

# Name-inference: anonymous concept on RHS picks up name from LHS
Widget: concept { price: 0, ticker: "" }

# Instance creation — `a` / `an` based on vowel sound; `entity` is a synonym
aapl: a Stock { price: 180, ticker: "AAPL" }
concept Insect
Insect has legs: 6
ant:  an Insect { legs: 6 }
bee:  an Insect with legs: 6   # provisional `with` spelling of the same block
aapl.price                     # property access
aapl's price                   # possessive (same thing)
aapl is Stock                 # runtime type test
aapl is Asset                 # inheritance test (true)

# --- Description Logic concept algebra (a tableau reasoner) ---
concept Person
concept Worker
Student extends Person
Worker  extends Person
Student ⊓ Worker               # → ConceptExpr  (conjunction; also `C `sqcap D`)
Student ⊔ Worker               # → ConceptExpr  (disjunction; also `C `sqcup D`)
¬Student                       # → ConceptExpr  (complement; `not` on a concept)
Student ⊑ Person               # → true   (subsumption: left is the SUBconcept)
Person subsumes Student        # → true   (English word = the converse direction)
(Student ⊓ Worker) ⊑ Person    # → true   (derived through the tableau + `extends`)
Student ≡ Worker               # → false  (equivalence = mutual subsumption)
satisfiable(Student ⊓ ¬Student) # → false (the complement clashes)

# --- DL role restrictions (∃R.C / ∀R.C — roles are native relations) ---
relation advises(x, y)
∃advises: Student              # → ConceptExpr  (someone who advises some Student)
∀advises: Student              # → ConceptExpr  (advises only Students)
satisfiable((∃advises: Student) ⊓ (∀advises: ¬Student))   # → false

# --- a partition becomes real disjointness + covering ---
Person partition Student, Worker
satisfiable(Student ⊓ Worker)  # → false  (now disjoint)
(Student ⊔ Worker) ≡ Person    # → true   (covering)

# --- extensional membership: `x is <ConceptExpr>` (closed-world) ---
amy: a Student {}
amy is (Student ⊓ ¬Worker)     # → true

# --- defined concepts (GCIs): necessary-and-sufficient `concept X ≡ expr` ---
concept Advisor ≡ ∃advises: Student   # (the word `equivalent` also defines)
Advisor ⊑ (∃advises: Student)         # → true   (from the definition)
```

```
# --- Thing (⊤) / Nothing (⊥): built in, no declaration needed ---
satisfiable(Nothing)           # → false        Person ⊑ Thing   # → true
amy is Thing                   # → true   (⊤ holds of every individual)

# --- three spellings of a role restriction (all build the same value) ---
(∃advises.Student) ≡ (∃advises: Student)       # → true   DOT spelling
(advises some Student) ≡ (∃advises: Student)   # → true   Manchester ∃
(advises only Student) ≡ (∀advises: Student)   # → true   Manchester ∀

# --- a compound concept as a comprehension SOURCE (⊓/⊔) ---
{x | x <- (Person ⊓ ¬Worker)}  # persons who aren't workers
#   {x | x <- ¬Worker}  → error: no finite extent; use FILTER `{x | x <- C, x is
 ↪  ¬Worker}`
```

Taught in Ch.8, 13, 18.

> **Retired May 2026:** `Stock is Concept` (as a creation form), `create Stock`, `Stock creat e`, `Stock exists`, and `object Stock {...}` no longer create concepts/instances. Migrate to `concept Stock` and `a Stock {...}` respectively. The `is` keyword is reserved for instance classification (`aapl is Stock`) and Boolean predication queries (`x is Stock` in expression position).

Enums and Subranges

```
Day enumerates Mon, Tue, Wed, Thu, Fri, Sat, Sun
Digit ranges 0..9
Workday extends Day in Mon..Fri
```

Taught in Ch.19.

Logic Programming

```
relation parent(child, p)         # declare
parent("alice", "mary")                  # store fact
{Y | Y <- parent("alice", Y)}             # query

# Strict rule (Horn-clause backward chaining) — primary natural spelling
grandparent(X, G) whenever parent(X, P) and parent(P, G)
# also: head if body (gated); operator twins <== (legacy <=) and Prolog :-

# Defeasible rule + cancellation (typically … whenever ≡ normally ≡ rule~ … if ≡ <~~)
typically flies(X) whenever bird(X)
cancel("flies", "opus")

# Provenance / introspection
```

```
grounding("flies", "tweety")              # returns "conjecture"
proof("flies", "tweety")                  # derivation chain
why flies("tweety")                       # prose explanation
```

Taught in Ch.21.

Multi-Valued Logic

```
om                                        # K3 unknown (also Ω) — untyped
T^k ⊥^k ?^k                               # K3 literals   (≡
 ↪  kleene("true"|"false"|"unknown"))
Tł ⊥ł ½ł                                  # Ł3 literals   (≡ lukasiewicz(1.0|0.0|0.5))
T^b ⊥^b T⊥^b ?^b                          # B4 literals   (≡ belnap(...); T⊥^b = glut,
 ↪  ?^b = gap)
T^i ⊥^i ?^i                               # G3 literals   (≡ intuit3(...))
`belboth  `klunknown  `lhalf              # ASCII digraphs (also `glut / `gap)
Belnap.both  Belnap.values                # dotted members + each logic's domain array
lukasiewicz(0.73)                         # constructor — any L3 value in [0,1]

# Operators auto-dispatch on type
true and om                 # Ω (K3)
lukasiewicz(0.6) or lukasiewicz(0.3)      # 0.6 (L3 max)
T⊥^b and T^b                              # T⊥^b
match T⊥^b with | T⊥^b => "glut" | _ => "other"   # literals are match patterns

# Truthiness = designation: if ⊥^b / ?^k / ½ł → else-branch; the glut T⊥^b is truthy.
# `==` is Boolean metalanguage equality (belnap("true") == "true"); `iff` is the
# object-language biconditional. Junk operands error: belnap("true") and "banana".
```

Taught in Ch.20.

Loops and Mutation

```
x: 0
while (x < 10) [
  x = x + 1
]

for v in [10, 20, 30] [ println(v) ]   # foreach ≡ for — one keyword, two spellings
for i in 1..5 [ println(i) ]           # ranges iterate in their own order
                                       #   (5..1 descends, 1..9 by 2 steps)
for v at i in xs [ println(i, v) ]     # 1-based counter; equivalents:
                                       #   `for v in xs with index i` and
                                       #   `for i, v in xs.indexed`
foreach n in 1.. [                     # open-ended range — loop until break
```

```
  if n > 9 then break                  #   bare `break` in a branch ≡ `then [break]`
]

foreach [x, y] in pairs [ println(x, y) ]          # destructuring; `(x, y)` and a bare
foreach  x, y  in pairs [ println(x, y) ]          #   `x, y` are the same thing

repeat 5 [ println("hi") ]             # `repeat` ≡ `loop`: count only …
repeat i <- xs [ println(i) ]          #   … bound variable over a source …
repeat i 3 [ println(i) ]              #   … the older no-arrow spelling …
repeat xs [ println("tick") ]          #   … or a bare source, no variable
repeat [n < 3] [ n = n + 1 ]           # PRE-test block condition
repeat [ n = n + 1 ] until n >= 3      # POST-test — body runs at least once
r: repeat i <- 1..3 [ println(i) ]     # `repeat` is an expression too

repeat i <- 1..10 by 3 [ println(i) ]  # `repeat … <-` reads the same headers
repeat i <- 10..1..-3 [ println(i) ]   #   `by` = magnitude, `..n` = SIGNED
repeat i <- 1.. [ if i > 3 then break ]

for v in {3, 1, 2}  [ println(v) ]             # a set — ONE canonical order
for v in bag([1, 1, 2]) [ println(v) ]         # a bag — a multiset, WITH multiplicity
for k, v in items(d) [ println(k, v) ]            # a dictionary, destructured

for v in (n * n | n <- 1..) [          # a lazy stream is a loop source too:
  if v > 20 then break                 #   pulled one element at a time,
]                                      #   never materialized
for i in iterate(func(v) [v * 2], 1) [ # a counter with an arbitrary step,
  if i >= 20 then break                #   still scoped to the loop
]
```

A loop scopes what it **binds**: `for`/`foreach` and the source forms of `repeat` own their variable, so it does not outlive the loop. `while` and the condition-driven `repeat` forms bind nothing — that counter is yours and survives. The full inventory, every form asserted, is `tests/axioma/showcase/loops.ax`.

A body can be a bracket block or a newline-delimited block closed by `end`. The latter works for `for`, `foreach`, `while`, `loop` and `repeat`:

```
for value in [10, 20, 30]
  println(value)
end
loop 2
  println("again")
end
```

The outputs are `10`, `20`, `30`, then `again` twice, each on its own line. Each nested loop has its own closer. The beginner subset allows `while`, `for`/`foreach`, and `repeat`/`loop`. There is no `do` keyword (`for ... in ... do ...` is not Axioma). `break` and `continue` work in every loop form, including every `repeat` form. In branch position any bare **statement** may be a branch — `break`, `continue`, `return`, or a binding — so `if c then break` and `if c then a: 0` are sugar for the block forms `if c then [break]` / `if c then [a: 0]`, in either arm. A binding branch scopes a *fresh* name to the block just as the

brackets do; an existing name is updated. There is no postfix guard for the loop pair: `break if c` is a SyntaxError (that form is `return`-only).

Taught in Ch.2a (counted `repeat/loop`, `for`, `while`); Ch.16 returns to mutation-shaped loops. Open-ended ranges in §17.7.

Quotation and Meta-Programming

```
parse("2 + 3")                          # string → AST
quote(2 + 3)                            # capture without eval
ast_eval(parse("2 + 3"))                # 5
ast_type(parse("2 + 3"))                # "InfixExpression"
fullform(parse("2 + 3"))                # "+(2, 3)"
headof(quote(if x then a else b))       # "If"
argsof(quote(2 + 3))                    # ["2", "3"]
```

Taught in Ch.22.

Reserved-word gotchas

Identifiers that *look* available but **aren't**:

- `A` / `E` — **no longer reserved** (the ∀/∃ quantifier shorthand was retired June 2026); now ordinary names. Quantify with `forall` / `exists` or `∀` / `∃`
- `fn` — lambda keyword (use `callee`, `f`, `func_node`)
- `take`, `from`, `to`, `contains`, `count`, `next`, `direct`, `diff`, `_` — reserved or builtin
- Underscore _ in patterns means "don't care," but cannot be used as a variable name.

The `keywords()` builtin lists the full reservation roster; `keywords("name")` checks a single word.

Language modes — `#language` dialects

`#language axioma/NAME` (or `--language axioma/NAME`) selects a **dialect**. Bare `#language axioma` is not a name — the checker lists the eight known ones and exits. `--mode functional|logic|stack|…` is a different switch (an educational overlay on the host). Do not confuse the two.

Dialect	What it is	Taught
`axioma/all`	Default host. Full surface.	The book after beginner; write it when a file must *not* be beginner
`axioma/beginner`	Real gate: canonical forms + beginner overlay	Ch.1–17, README
`axioma/hm`	Real closed island: principal types, fatal	Ch.54
`axioma/knowledge-core`	Real AST allowlist: monotonic Horn proof-core	Ch.21 §21.11

Dialect	What it is	Taught
`axioma/knowledge`	Mild helper-name gate (`insert` refused; `retract` still runs). Not an alias of the core	Ch.21 §21.11 (contrast)
`axioma/rpn`	Additive Forth return stack (`rpush`/`rpop`…). Infix still runs	Ch.41 §41.9
`axioma/core`	**Stub** — known name, no AST gate, runs as host	not taught (would be a silent dual meaning)
`axioma/functional`	**Stub**, same	not taught

```
#language axioma/beginner               # restrict surface forms
#language axioma/hm                     # closed Hindley–Milner island
#language axioma/knowledge-core         # monotonic proof-core
#language axioma/rpn                    # Forth return stack (additive)
#language axioma/all                    # full host (opt out of beginner)
```

Choose one directive for a file. The following are terminal commands, not statements inside that file:

```
axioma --typecheck script.ax            # static pre-pass
axioma --infer script.ax                # print HM arrows (fatal on axioma/hm)
axioma --no-kb script.ax                # skip Cascade KB load
axioma --learn                          # interactive tutor
axioma --recipe                         # Design Recipe wizard
```

Taught in Ch.1, Ch.4, Ch.16 (`axioma/all`), Ch.21 (`knowledge-core`), Ch.41 (`rpn`), Ch.54 (`hm`), and the README.

Where this fits

This appendix is a **lookup card**, not a tutorial. If a form puzzles you, jump to its origin chapter (the rightmost column) for the surrounding pedagogy. For the canonical full reference, the Axioma Manual covers every form in the language, including ones this book deliberately skipped (CG, modal logic, Cascade integration, the stack-based subsystem).

Unified type declarations (optional full-language forms)

`type Name =` accepts an existing type expression or a kind marker: `struct`, `struct mutable`, `variant`, `enum`, or `concept`. `struct` is the primary teaching spelling; `record` is its exact kind-marker alias, not a structural Dictionary schema. Either accepts `mutable` or its short alias `mut`: `struct mut`, `record mutable`, and `record mut` have the same mutable-struct contract. Without a modifier, either marker is immutable. Nominal membership, field checks, immutable value equality and mutable reference identity are unchanged by spelling. Struct braces contain `field:: Type`, or a bare lowercase name for an open field that accepts any value; the equivalent newline body ends in `end`, with every spelling.

Existing `[T]` type parameter support and restrictions are unchanged; aliases add no generic forms. These records remain evaluator-only; the VM refuses them. See the optional spelling example in §18.5. `schema` explicitly opens the end-body version of a Dictionary schema. `concept Name with … end` and `type Name = concept with … end` contain ordinary concept slot/default declarations. Bare concept headers remain complete. The formatter retains braces versus end. See Chapter 18 · Types and Data Modeling for the whole family, §16.13 for mutable records, and Chapter 19 for enums and subranges. Older `data`, `enum`, and `concept` spellings retain their meanings.

Operator precedence and associativity

This is the broader expression reference; Chapter 1 §1.4 introduces the arithmetic subset. The tables describe full Axioma, not a promise that every form belongs to `axioma/beginner` or runs in the VM. Parentheses group an expression; they are not an operator with a competing numeric priority.

Precedence decides which operation binds more tightly: `2 + 3 * 4` groups as `2 + (3 * 4)`. **Syntactic associativity** decides how a chain at one priority groups: `20 - 6 - 3` is `(20 - 6) - 3`, but `2 ^ 3 ^ 2` is `2 ^ (3 ^ 2)`. **Mathematical associativity** is a different property: an operation is associative when changing those parentheses preserves its result for its domain. Power is not associative: the two groupings below produce different numbers. **Evaluation order** describes when operands run, including side effects and skipped branches; it does not follow just from the precedence table.

```
println(20 - 6 - 3)    # 11
println(20 - (6 - 3))  # 17
println(2 ^ 3 ^ 2)     # 512
println((2 ^ 3) ^ 2)   # 64
println(2 ** 3 ** 2)   # 512
println(-2 ^ 2)        # -4
println((-2) ^ 2)      # 4
println(3! ^ 2)        # 36
println([2] .^ [3] .^ [2]) # [512]
```

Ordinary expression priorities

Read from tighter to looser. “Left” means `(a op b) op c`; “right” means `a op (b op c)`. Mixed operators in a row share priority, but a special right-hand parser can still change their grouping; the exceptions are explained below. Knowledge-language forms have an additional table after the ordinary forms.

Priority band, high to low	Forms	Grouping
Postfix factorial	`x!`	Repeated postfixes attach to the result on their left
Calls and access	`f(x)`, `f.(xs)`, `a[i]`, `a.field`, `a?.field`, `a's field`; constructor `C {field: value}`	Leftward chain; call, field and index share priority

Priority band, high to low	Forms	Grouping
Ordinary prefixes	unary +, `not` / `!` / ¬, `E!`, address &, dereference ∗	Consume one prefix operand, including tighter calls/access/factorial
Power	`^`, ∗∗, `.^`	Right
Unary minus	−x	Its operand includes power, but stops before multiplication
Multiplicative family	multiplication/division/remainder, intersection, composition, backtick calls, `cons`, `+:`, `:+` (details below)	Left, except right-grouping `cons` and `+:`; parenthesize mixed `+:`/`:+`
Additive family	addition/subtraction, union/difference, bitwise words (details below)	Left
Range	`..`, `..<`, optional `..step` or `by step`	Range syntax, not an ordinary binary chain
Ordering	<, <= / ≤ / ⩽, >, >= / ≥ / ⩾	Consecutive comparisons form a chain
Equality and membership	Equality, membership, subset/DL relations, `is`, `isa`, `matches` (details below)	Left for ordinary binary forms; `matches` takes a pattern
Conjunction	`and` / ∧, `andalso`	Left
Disjunction	`or` / ∨, `xor` / ⊻, ⊼, ⊽, `orelse`	Left; two-word `or else` has a special parser
Implication and pairs	`implies` / `impl` / → / ⟹, `iff` / ↔ / ⟺; pair => / –>	Implication/iff left; pairs right
Fallback and conditional	`otherwise`, ??, ???; `condition ? yes : no`	Fallbacks left; ternary branches have special scope
Pipeline and postfix match	|>, |?>; value match | ...	Pipes left; match introduces pattern arms

Unary + is in the ordinary prefix band, unlike unary −. `not a == b` means `(not a) == b`; write `not (a == b)` to negate the equality. `&a.field` means `&(a.field)`. `∗p.field` means `∗(p.field)`, while `(∗p).field` accesses the dereferenced value. Prefix `!x` is negation; postfix `x!` is factorial. A trailing inspect dot (`expr .`) is a statement form, distinct from member access.

The larger rows expand as follows; glyphs and their word spellings have the same priority. A callable builtin does not become an infix merely because its name resembles a glyph's catalog name.

Family	Operators and aliases at that priority
Multiplicative arithmetic	∗, `/`, `%`, ÷; `.∗`, `./`, `.%`, `.÷`; same-line `mod`, `modulo`, `remainder`, `rem`, `div`, `idiv`, `quotient`, `rdiv`, `fdiv`
Other multiplicative	`intersect` / `intersection` / ∩; DL ⊓; B4 ⊗; composition ∘, <<, >>; `a \`f\` b`; `a cons b`

Family	Operators and aliases at that priority
Additive arithmetic and bitwise	`+`, `−`, `.+`, `.−`; `band`, `bor`, `bxor`, `bshl`, `bshr`
Other additive	`union` / `∪`; `difference` / `diff` / `\`; `symmetric_difference` / `symdiff` / `△` / `Δ` / `⊖`; DL `⊔`; B4 `⊕`
Equality and regex	`==`, `!=` / `≠`, `===`, `!==`, `=~`, `!~`
Membership and relations	`in` / `∈`, `notin` / `not_in` / `∉`, `contains`; `subset` / `⊆`, `superset` / `⊇`; `proper_subset` / `subsetneq` / `⊂` / `⊊`; `proper_superset` / `supsetneq` / `⊃` / `⊋`; `subsumes` / `⊑`, `equivalent` / `≡`; `relatedTo`, `partOf`; `is` (including `is/same`, `is/identical`), `isa`, `matches`

`rem` and `mod` share priority, not signed-remainder semantics. Similarly, `fdiv` and `/` / `rdiv` share priority, not exactness. `idiv` and `div` share priority and meaning. `>>` and `<<` are composition; bit shifting uses `bshr` and `bshl`. `/concept` refinements of `and` and `or` retain the word operator's priority; they are not priority aliases for `⊓` and `⊔`. Numeric `//` is retired in favor of `div` / `÷`; it is not another division row. See the Manual's arithmetic, sets, symbols, and Description Logic sections for operand domains and the full glyph/digraph catalog.

```
println(6 band 3 + 1) # 3: (6 band 3) + 1, not 6 band (3 + 1)
println({1} union {2} intersect {2}) # {1, 2}: intersection binds tighter
println(true == 2 < 3) # true: true == (2 < 3)
println(not false == true) # true: (not false) == true
```

Calls, infix functions, composition and pipes

An ordinary call uses parentheses. `f(x)[i].field` groups as `((f(x))[i]).field`: indexing is not stronger than a call or member selection. Arguments have their own comma-separated expression boundaries. A prefix operator used as a callable, such as `+(2, 3)`, follows call syntax. Sections such as `(10 −)` produce functions; they are not ordinary two-operand chains. Bare juxtaposition is not a general replacement for `f(x)`; statement-level message/action forms have their own grammar.

```
row_at() = {values: [2, 3]}
println(row_at().values[2] ^ 2) # 9
println(row_at().values[2]!)    # 6
println(none?.values)           # none
```

A backtick call has multiplicative priority: `3 `add` 4 * 2` means `add(3, 4) * 2`, not `add(3, 4 * 2)`. Ordinary user functions need no fixity declaration to be used this way. `1 cons 2 cons list()` instead groups as `cons(1, cons(2, list()))`; its tail must be a List, not an Array.

```
add(a, b) = a + b
inc(x) = x + 1
```

```
double(x) = 2 * x
println(3 `add` 4 * 2)      # 14
println(add(3, 4 * 2))      # 11
println(1 cons 2 cons list()) # `[1, 2]
println((inc ∘ double)(3))  # 7
println((inc << double)(3)) # 7
println((double >> inc)(3)) # 7
println(3 |> double |> inc) # 7
println(3 |> add(10))       # 13
println(3 |> add(_, 10))    # 13
```

`f ∘ g(x)` means `f ∘ (g(x))`, which need not be a valid composition: parenthesize `(f ∘ g)(x)` when applying a composed function. Composition creates a callable; a pipe applies one stage now. `x |> f(a)` inserts `x` as the **last** argument, giving `f(a, x)`; `_` selects another position. A pipe's right side must be an accepted callable/call form, not arbitrary arithmetic. `|?>` uses the same insertion rule but can stop on failure; see the Manual's error-propagating pipe section. Learn the design in Chapter 3, §10.10 composition, and §10.11 broadcasting.

Declared binary symbolic operators

`operator (&&&) = minInt` declares file-local binary syntax for an ordinary function. An infix use calls that function with two arguments; a parenthesized symbol is the ordinary callable value. The default is multiplication priority with left grouping. Static metadata selects an existing named band and left, right, or non-associative grouping. The complete allowed set and runnable forms are in Chapter 10 §10.12, with an exercise and solution.

Parentheses are required for non-associative chains and mixed custom fixities at one priority. Fixity adds no short-circuiting and does not alter the target's function checks or VM restrictions. Declare before use at file or REPL top level; imports do not export syntax. The native REPL accepts identical declarations on replay and rejects conflicts.

ASCII operator runs and unreserved mathematical glyphs such as ⊛ are allowed. Reserved glyphs and structural/comment delimiters retain their meanings. The spellings && and || are built-in aliases for `and` and `or`; they cannot be redeclared as custom operators. The longer spelling &&& is allowed.

Comparisons, ranges, arrows and conditional scope

Only ordering operators chain. `a < b <= c` becomes `(a < b) and (b <= c)`. Equality chains stay binary: `a == b == c` means `(a == b) == c`. With effectful operands, the reused middle comparison expression can run twice; bind it once first if it must be sampled once. This is a parser rewrite, not a mathematical claim about evaluating arbitrary code. Prefer an explicit conjunction rather than mixing a preexisting comparison result into another ordering expression.

A range's bounds bind more loosely than arithmetic: `1 + 1..3 * 2` has start `1 + 1` and end `3 * 2`. `1..5..2` is **start, end, signed step**, not `(1..5)..2` or `1..(5..2)`. `by` takes a positive step magnitude and determines direction from the endpoints. Do not extrapolate a binary associativity law to ranges. See the Manual's Ranges section for exclusive and open-ended forms.

```
println(collect(1 + 1..3 * 2 by 2)) # [2, 4, 6]
println(collect(1..5..2))           # [1, 3, 5]
println(first(1 => 2 => 3))         # 1
println(first(second(1 => 2 => 3))) # 2
println(second(second(1 => 2 => 3))) # 3
println(false ? 1 : true ? 2 : 3)  # 2
```

Pair arrows group to the right: `1 => 2 => 3` is `pair(1, pair(2, 3))`. The ASCII arrow `->` has the same pair reading; Unicode → is implication. But a bare parameter before an arrow (`x => body`), a lambda head, or a match/cond arm claims the arrow first. Write `pair(key, value)` when an identifier-led arrow would be ambiguous. Type/function signatures also use their own arrow grammar.

`a ? b : c ? d : e` means `a ? b : (c ? d : e)`. Comparisons and Boolean connectives bind into the condition. The yes-branch accepts a full expression until its matching colon; the no-branch accepts another ternary/fallback, but stops before a following ordinary pipe. Parenthesize mixed ternary/pipe/-fallback expressions. Likewise `if … then … else …` and match/cond bodies own whole expressions; they are not ordinary binary operators. Postfix `a + b match | …` starts with the value `(a + b)`. See Chapter 5 and Chapter 18.

Fallbacks and evaluation order

`otherwise` catches an Error, `??` supplies a value for `none`, and `???` supplies one for `none` or `om`. They do not share the same trigger. These three operators group left, and evaluate their right operand only when their own trigger fires.

Current two-word exception. `or else` constructs an error fallback too, but its parser enters at the disjunction band and consumes a right operand down to the fallback band. Repeated `or else` therefore nests right; mixed expressions need parentheses. It is not a precedence-identical textual replacement for `otherwise`. For example, `a ?? b or else c` groups as `a ?? (b otherwise c)`, whereas `a ?? b otherwise c` groups as `(a ?? b) otherwise c`.

```
println((1 / 0) ?? 2 otherwise 3) # 3
println(error?(try((1 / 0) ?? 2 or else 3))) # true
println(((1 / 0) ?? 2) or else 3) # 3
println(false and (1 / 0))        # false
println(true v (1 / 0))           # true
println(none andalso (1 / 0))     # false
println(0 ?? (1 / 0))             # 0
println(om ??? 7)                 # 7
```

Logical evaluation: short-circuit and eager operands

These are evaluation rules for ordinary infix expressions; precedence only controls grouping. An error in the left operand propagates before the right runs.

Connective	Right operand runs when…
`and` / `&&` / `∧`	the left value is not Boolean `false`
`or` / `||` / `∨`	the left value is not Boolean `true`
`andalso`	the left value is truthy
`orelse`	the left value is falsy
`xor` / ⩒	the left evaluation succeeds (eager)
`iff` / ↔ / ⟺	the left evaluation succeeds (eager)
`implies` / `impl` / → / ⟹ / `-->`	the left evaluation succeeds (eager)
NAND ⊼, NOR ⊽	the left evaluation succeeds (eager)

`andalso` and `orelse` convert truthiness to a Boolean result. `and` and `or` preserve multivalued dispatch: any non-Boolean left value evaluates the right, including typed Kleene false and typed Belnap true. This is the current implementation, even where the truth table could determine an answer sooner. See Chapter 20 for a trace.

Implication is eager even with Boolean false on the left. The same distinction applies to NAND and NOR: their truth-table identities do not promise skipped effects or errors. Negation evaluates its single operand; ordinary function calls, including `nand(...)` and `nor(...)`, evaluate arguments before calling.

```
probe(label, value) = [println(label); value]
println(false and probe("and", true))        # false; no label
println(true or probe("or", false))          # true; no label
println(false implies probe("implies", true)) # implies, then true
println(true iff probe("iff", true))         # iff, then true
println(false xor probe("xor", true))        # xor, then true
println(false ⊼ probe("nand", true))          # nand, then true
println(true ⊽ probe("nor", false))           # nor, then false
```

A conditional runs only its selected branch. For code whose operands print, mutate state or can fail, preserve evaluation behavior when rewriting a formula.

Tracing evaluation order

Here is a concrete evaluator trace. Right-grouping power still evaluates the left operand first. The comparison rewrite repeats its middle expression. The ordinary pipe below appends its input expression after the explicit call argument, so the printed order is `argument`, then `input`. Use separate bindings when an effect must occur at a specified step; do not infer effect order from the visual direction of an arrow or from an algebraic rewrite. Broadcast/fusion has additional rules described in §10.11.

```
probe(label, value) = [println(label); value]
println(probe("left", 2) ^ probe("middle", 3) ^ probe("right", 2))
# left
# middle
# right
# 512
```

```
println(0 < probe("middle", 5) < 10)
# middle
# middle
# true
subtract(a, b) = a - b
println(probe("input", 10) |> subtract(probe("argument", 3)))
# argument
# input
# -7
```

Knowledge-language operators and grammar boundaries

These specialist forms are part of the expression parser too. Their entry priorities lie **above postfix factorial**, in the order below. Several then parse a wider right operand, so a numeric priority alone is not a complete specification. Parenthesize whole propositions, agents and fillers when mixing these forms. The table describes grouping, not whether arbitrary numeric operands make a meaningful knowledge statement or whether the VM supports it.

Higher to lower entry band	Forms	Right-hand scope
DL role restriction	`R some C`, `R only C`	One tight filler; `R some C ⊓ D` is `(R some C) ⊓ D`; use `R some (C ⊓ D)` for a compound filler
Ontology/definition	`essence`, `accident`, `proprium`, `substance`, `attribute`, `nominally`, `really`	Same-band chain groups left
Analogy	`analogous`, `similar`, `different`	Same-band chain groups left
Argument	`argues`, `entails`, `presupposes`, `contradicts`	Same-band chain groups left
Binary temporal	`until`, `since`	Same-band chain groups left
Deontic	`must`, `may`, `must_not`, `should`	Whole expression: `alice must p and q` is `alice must (p and q)`
Epistemic	`knows`, `believes`, `doubts`	Whole expression: `alice knows p and q` is `alice knows (p and q)`
Causal/aspect	`causally`, `because`, `therefore` / ∴, `teleologically`, `efficiently`, `materially`, `formally`, `qua`; `causes`, `caused by`	First group uses its own band, left-grouping. `causes` / `caused by` absorb tighter-than-equality expressions on the right, including another causation; `observed_by` is an optional trailing clause

Prefix modal `necessarily` (□ / ▫), `possibly` (◇ / ◊ / ⋄), `contingently`, `impossibly`, and temporal `alw`

`ays`, `eventually`, `next` parse their operand at the **ordinary prefix** band. They are not infix synonyms for the specialist rows: `necessarily p and q` groups as `(necessarily p) and q`; use `necessarily (p and q)` for wider scope. Knowledge forms and their domains are developed in the Manual's Modal, Temporal, Epistemic & Deontic Logic, Description Logic, and Rules sections.

Tokenization also comes first: `50%` is a percent literal, whereas `10 % 3` is remainder. Put spaces around a ternary's `?` and `::` a glued `empty?` is a predicate name, and `3:4` can be a Time literal. Literal punctuation is not an operator-precedence rule. The symbol catalog includes named constants and special syntax as well as operators.

Other syntax deliberately does not fit an arithmetic-style row:

- Rule <==, <~~, ==>, ~~> and rule-head <= take whole rule bodies/heads. Forward rules admit a conjunction as their left body. `f(X) <= body` can be recognized as a rule when `X` is a logic variable; `f(3) <= 4` is a comparison. Use the canonical rule arrows rather than relying on the overloaded <=.
- Graph/space `show` and graph `join`, `restrict`, `generalize` enter at the equality band but parse query/operation clauses, not interchangeable binary arithmetic operands. Quantifiers, comprehensions, `with` updates, and literal forms have their own delimiters and grammar.
- =, `:`, += and other compound assignments are statement/binding forms; `::` annotations/ascriptions, type `|` / `of`, pattern `|` / `as`, lambda arrows, commas and semicolons belong to their enclosing grammar. In ordinary expression parsing, a semicolon terminates an expression; it is not infix Boolean `or`.
- A mapped or registered token is not by itself a working operator. `ought`, `permitted`, `forbidden` have no ordinary infix handler. `satisfies`, `with_probability`, infix `probably`, `denotes`, infix `typically`, `normally`, and `by_default` have no active ordinary infix priority. These are not extra supported rows; separate declaration/prefix forms must be looked up by form.
- Soft infix words such as `cons`, `mod`, `otherwise`, `qua`, and `only` depend on same-line context. Calls and indexing do not silently continue onto the next line. Brackets may instead open a control/-function body. Parenthesize an intended subexpression; precedence is not a substitute for these grammar rules.

The source audit for this reference is recorded in the repository's CURRENCY_AUDIT.md. It checks the precedence map, actual registrations, soft-word dispatch and special operand parsers together.

Sequence construction operators

`x +: xs` and `xs :+ x` add one element to a List or Array without changing `xs`. They keep the sequence kind and share the multiplicative band with `cons`. Write `(1 + 2) +: xs` for an arithmetic head and `(1 +: xs) :+ 3` for a mixed chain. `+:` is right-associative; `:+` is left-associative. Compact `2+:x` and `n:+3` now use these operators: write `2 + :x` and `n: +3` for the earlier arithmetic and binding readings. Positive slice bounds `xs[1:+2]` retain their meaning. See Chapter 23's optional section for copying costs and an exercise.

Appendix B · The Design Recipe — One-Page Summary

A condensed reference for the six-stage recipe taught in detail in Chapter 4 and applied throughout the rest of the book. Print this page and pin it to the wall while you work.

The six stages

```
1. Data definition      — what shape are the inputs?
2. Signature            — name, input types, output type
3. Examples             — concrete cases first, before code
4. Template             — code skeleton driven by the data shape
5. Body                 — fill in the template
6. Tests                — run the examples; revise if any fail
```

Stage 1 — Data definition

Decide *what your data is* before you decide what your function does. For each input:

- A single number? `Integer` / `Float` / a subrange like `Digit ranges 0..9`.
- One of a fixed set of values? An **enum**: `Status enumerates Active, Idle, Done`.
- A bundle of related fields? A **Concept**: `concept Point { x: 0, y: 0 }`.
- An arbitrary-length sequence? An **array**.
- A tree? A Concept whose slots are themselves Concepts (Ch.8).
- Something that *refers to itself*? Mutual recursion (Ch.9, 13).

The data definition *drives* the rest of the recipe.

Stage 2 — Signature

A comment line above the function:

```
# distance: (Float, Float, Float, Float) -> Float
distance: func(x1, y1, x2, y2) [ ?? ]
```

Two purposes:

1. Names the inputs and the output.
2. Locks the *type* of each — at minimum in the comment, and ideally with Axioma's `:: T` annotations and the `--typecheck` pre-pass.

Stage 3 — Examples

Write at least one concrete `input → output` pair for each **interesting** case in the data definition. Each case at the data-definition level deserves at least one example.

```
# distance(0, 0, 3, 4)  = 5.0       (3–4–5 triangle, classic)
# distance(0, 0, 0, 0)  = 0.0       (degenerate, same point)
# distance(1, 1, 4, 5)  = 5.0       (off–origin, still 3–4–5)
```

Examples become **tests** in Stage 6. Writing them *first* forces you to know what "correct" means before you write code that *might* be correct.

Stage 4 — Template

A skeleton matching the *shape of the data*. The same data shape always produces the same template.

Data shape	Template skeleton
Single value	`func(x) [ ... x ... ]`
Enum / case-distinction	`if x == case1 then ... else if x == case 2 then ... else ...`
Concept with slots	`func(p) [ ... p.slot1 ... p.slot2 ... ]`
Array (recursive)	`func(arr) [ if len(arr) == 0 then BASE else COMBINE(arr[1], recur(arr[2..len(arr)])) ]`
Tree (Concept)	`func(t) [ if t is Leaf then ... else COMBINE(t.value, recur(t.left), recur(t.right)) ]`
Mutually recursive data	one helper per data type, each calling the others

The template is the *contract* between the data shape and the function shape. If you're stuck filling in the body, the template is usually wrong — go back to Stage 1.

Stage 5 — Body

Fill in the template's `...` with the actual computation. The earlier stages guide this step, but design decisions can require revisiting them. For the distance example, the formula gives:

```
distance: func(x1, y1, x2, y2) [sqrt((x2–x1)^2 + (y2–y1)^2)]
```

Two heuristics when stuck:

1. **The recursive call returns the right answer for the smaller problem.** Trust it. Combine the head element with the recursive result; don't re-derive what the recursion gives you.

2. **Generalize from the examples.** If you have examples `f(2) = 4`, `f(3) = 9`, `f(4) = 16`, `x * x` is one candidate. Check the stated purpose and additional cases: finitely many examples do not determine a unique function.

Stage 6 — Tests

Turn each example into a runnable assertion:

```
println(distance(0, 0, 3, 4))   # 5.0
println(distance(0, 0, 0, 0))   # 0.0
println(distance(1, 1, 4, 5))   # 5.0
```

Or with the `expect` builtin (label, actual, expected — reports a mismatch and fails the run on a miss):

```
expect("3-4-5 triangle", distance(0, 0, 3, 4), 5.0)
expect("same point",     distance(0, 0, 0, 0), 0.0)
expect("off-origin",     distance(1, 1, 4, 5), 5.0)
```

Or as **doctests** (Ch.4 §Stage 6): in a ```` ```axioma doctest ```` fence, each Stage-3 example line *is* the test — `# →` and the plain-keyboard ASCII `# ->` are equivalent arrow spellings, and `# raises:` asserts an error. Run with `axioma --doctest FILE` or the Playground's ✓ Doctest button:

```
distance: func(x1, y1, x2, y2) [ sqrt((x2 - x1) ^ 2 + (y2 - y1) ^ 2) ]
distance(0, 0, 3, 4)   # → 5.0
distance(0, 0, 0, 0)   # -> 0.0
distance(1, 1, 4, 5)   # → 5.0
```

Run them. If any fail, the bug is in the body (most common), the template (less common), or the data definition (rare, but most expensive to fix).

The expanded recipe for relations (Ch.21)

For logic-programming code, replace stages 1–4 with:

```
1. Schema      — relation name(arg1, arg2, ...)
2. Fact base   — write the ground facts
3. Rules       — strict (<=) or defeasible (<~~)
4. Queries     — set comprehensions over the fact + rule base
5. Verify      — print each query; hand-walk against the data
```

The expanded recipe for evaluators (Ch.22)

For interpreters / AST walkers, the data definition *is* the grammar of the small language you're interpreting:

```
1. Grammar     — one Concept per AST node type
```

```
2. Signature   — eval: (Node, Env) -> Value
3. Examples    — small programs with known results
4. Template    — one branch per Concept; recurse on children
5. Body        — per-Concept dispatch
6. Tests       — run the example programs end-to-end
```

The recipe as a habit

The whole point of the recipe is to *resist the temptation to start with the body*. Beginning programmers write code, then try inputs, then patch what breaks. The recipe inverts that: define data, then signature, then examples, then template, *then* body, *then* tests. The bugs catch themselves at the stage where they're cheapest to fix.

This single discipline — Data-Driven Design — is the most durable skill the book teaches.

> *"A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts." — Structure and Interpretation of Computer Programs*, 1985.

Appendix C — Solutions

Chapter 1 · Solutions

These are *one* set of answers for each exercise. Yours may differ — in particular, exercises with "open" in the title have many right answers. The commentary points to what's worth noticing in each.

Exercise 1.1 — circumference of a circle with radius 7

```
println(2 * 3.14159 * 7)
```

Result: `43.98226`.

Commentary. The order of multiplication doesn't matter, so `2 * 7 * 3.14159` and `3.14159 * 7 * 2` give the same answer. You'd write the version that reads most clearly to *you*.

Exercise 1.2 — powers of two

```
2 ^ 16  = 65536                  # exact
2 ^ 32  = 4294967296             # exact
2 ^ 64  = 18446744073709551616   # exact — beyond a 64-bit integer
```

Commentary. Axioma integers are **arbitrary precision**: a value grows as many digits as it needs, so supported computations can preserve exact answers — `2 ^ 64`, `2 ^ 128`, even `2 ^ 200` (a 61-digit number) come back exact. Available memory and runtime resource limits still bound computations; arbitrary precision means that arithmetic is not restricted to one fixed machine-word width.

Where exactness *does* end is floats: `2.0 ^ 64` prints `1.8446744073709552e19`, a Float in scientific notation. This particular power of two is exactly representable, but nearby integers need not be: binary64 has 53 bits of significand precision, roughly 15-17 significant decimal digits. So the boundary to watch is not "is my number too big?" but "did a float sneak into my integer arithmetic?" — the §1.5 rule again: floats are contagious; integers and rationals are exact.

Try. Compute `2 ^ 200`, then `(2 ^ 64) * (2 ^ 64) == 2 ^ 128`, then `2.0 ^ 64`. Which results are exact, and why?

Exercise 1.3 — precedence

Expression	Answer	Why
`(2+3) * (4−1)`	`15`	parens first, then multiplication
`10 − 2 * 3`	`4`	`*` before −, so `10 − 6`

Expression	Answer	Why
`2 ^ 3 ^ 2`	`512`	Axioma's `^` is **right-associative**, so `2^(3^2) = 2^9 = 512`
`(2 ^ 3) ^ 2`	`64`	parens force left-grouping: `8^2 = 64`
`100 / 7`	`100/7`	division is **exact** — an uneven integer quotient is a Rational fraction, not a rounded integer
`100 % 7`	`2`	the remainder of whole-number division (`100 div 7` gives the `14`)
`100.0 / 7`	`14.285714285714286`	a float operand makes it float division — the decimal approximation of `100/7`

Commentary. Mathematicians and most calculators treat `^` as **right-associative**: `2^3^2` means `2^(3^2) = 2^9 = 512`. Axioma follows that convention — `2 ^ 3 ^ 2` is `512`. Python uses `**` and is right-associative for the same reason (`2**3**2 = 512`); Axioma's `**` agrees too.

Note this makes `^` the *exception* among the arithmetic operators: `+`, `-`, `*`, and `/` are all left-associative. If you want the left grouping for exponentiation, parenthesize: `(2 ^ 3) ^ 2` gives `64`.

Notice that `100 div 7 = 14` and `100 % 7 = 2` together tell you the full story: `7 × 14 + 2 = 100`. This is the *division algorithm*, and it's the reason modulo exists. (Plain `/` stays out of it: `100 / 7` is the exact fraction `100/7`.)

Exercise 1.4 — modulo and divisibility

Expression	Answer
`12 % 3`	`0`
`12 % 4`	`0`
`12 % 5`	`2`
`(2^10) % 7`	`2`

Commentary. `12 % n` is `0` exactly when `n` divides 12 evenly. 3 and 4 are *divisors* of 12 ($12 = 3 \times 4 = 4 \times 3 = 6 \times 2 = 12 \times 1$); 5 is not.

The last one — `(2^10) % 7` — is `1024 % 7 = 2`. It's worth checking by hand: `7 × 146 = 1022`, so `1024 - 1022 = 2`. ✓

Exercise 1.5 — when does Axioma error?

Expression	Result	What happened
`"5" + 3`	`"53"`	int → string, then concatenated
`5 + "3"`	`"53"`	int → string on the left, then concatenated
`5 - "3"`	**ERROR**	- has no string interpretation
`5 * "3"`	`"33333"`	repetition — Python-style
`5 / "3"`	**ERROR**	/ has no string interpretation
`"hi" * 4`	`"hihihihi"`	repetition
`"hi" / 4`	**ERROR**	division on a string is meaningless
`"hi" - "bye"`	**ERROR**	subtraction between strings has no meaning

Rule (one possible reading). Axioma "tries" to interpret an operator when it has a sensible string interpretation:

- + becomes concatenation
- * becomes repetition (one side must be an integer)

Anything else — -, /, %, ^ — errors when either side is a string. The rule isn't perfectly clean (you could argue `"abc" * 0` should be `""`, which it actually is), but the take-home is to check the documented operand types. An unsupported combination raises an error.

Exercise 1.6 — open

One answer (of infinitely many):

```
println((100 - 50 + 30 * 2) / 5 % 7 ^ 2)
```

Evaluation: `100 - 50 + 60 = 110`, then `110 / 5 = 22`, then `22 % 49 = 22`. Result: `22`. Between 0 and 100. ✓

A structurally different second answer:

```
println(2 ^ 5 * 3 - (10 / 2) % 4 + 1)
```

Evaluation: `32 * 3 - 5 % 4 + 1 = 96 - 1 + 1 = 96`. Result: `96`.

Commentary. The point of this exercise isn't to find *the right answer* — it's to gain comfort building larger expressions with confidence about what they'll evaluate to. If yours evaluated to something between 0 and 100 the first time, you understood the material.

Exercise 1.7 — grouping is not an algebraic law

Expression	Default grouping	Result
`20 - 6 - 3`	`(20 - 6) - 3`	`11`
`24 / 4 / 2`	`(24 / 4) / 2`	`3`
`2 ** 3 ** 2`	`2 ** (3 ** 2)`	`512`
`-2 ^ 2`	`-(2 ^ 2)`	`-4`
`(-2) ^ 2`	Negative base explicitly grouped	`4`

```
println(20 - 6 - 3)   # 11
println(20 - (6 - 3)) # 17
println(24 / 4 / 2)   # 3
println(24 / (4 / 2)) # 12
println(2 ** 3 ** 2)  # 512
println((2 ** 3) ** 2) # 64
println(-2 ^ 2)       # -4
println((-2) ^ 2)     # 4
```

The power results 512 and 64 disprove mathematical associativity for these operands. Right-associative syntax merely selects the first grouping when parentheses are absent. Precedence does not determine when a side-effectful operand prints; evaluation rules answer that separate question.

Exercise 1.8 — optional function and Boolean forms

```
add(a, b) = a + b
double(x) = 2 * x
inc(x) = x + 1
println(3 `add` 4 * 2)     # 14
println(add(3, 4 * 2))     # 11
println((inc ∘ double)(3)) # 7
println(3 |> double |> inc) # 7
println(false and (1 / 0)) # false
```

The backtick form is `(add(3, 4)) * 2`; the ordinary call places `4 * 2` inside its second argument. `inc ∘ double(3)` groups as `inc ∘ (double(3))`, so its right operand is the number 6, not the function `double`. The valid composition calls `double` and then `inc`, as does the pipe. The Boolean example skips division because `and` short-circuits on Boolean false, not because `/` has low precedence (it actually binds more tightly than `and`). See Appendix A for the broader rules.

Exercise 1.9 - optional files and project settings

The source remains an ordinary file named `tally.ax`:

```
println(6 * 7)
```

From its directory:

```
axioma tally.ax
axioma --typecheck tally.ax
axioma --typecheck --run tally.ax
```

The first and third commands print 42. The second reports `type check clean` and does not print the calculation. None needs a configuration. If you choose to make a project, this is sufficient `axioma.json` content:

```
{
  "version": 1,
  "entryPoints": ["tally.ax"]
}
```

Then `axioma --project .` prints 42. With two entries, the no-source command reports that it requires exactly one entry; `axioma --project . tally.ax` selects this file explicitly. An explicit source must belong to the selected project.

`main.ax` is not a required or special name. Installation does not require a project configuration. Creating one is a separate choice made in your own project, and subsequent source runs discover the nearest configuration automatically. A stricter checker setting changes diagnostics; it does not replace the ordinary host's advisory execution policy. `--typecheck --run` makes type errors stop the run. The closed HM language has separate mandatory checks, taught in Chapter 54.

How to use these solutions

Don't read them before attempting the exercises. The point of the exercise is the **predict** step before you check — and you only get the benefit of the predict step if you do it yourself first.

After you've answered an exercise, *then* compare. If yours is correct, ask: was my reasoning the same as the solution's reasoning? If yours is different but still correct, ask: which is clearer to a future reader?

Chapter 2 · Solutions

Exercise 2.1 — circle quantities for radius 7

```
#language axioma/beginner
pi: 3.14159
radius: 7
diameter: 2 * radius
circumference: 2 * pi * radius
area: pi * radius * radius
area_bigger: pi * 8 * 8
println("radius =", radius)
println("diameter =", diameter)
println("circumference =", circumference)
println("area =", area)
println("area_bigger =", area_bigger)
```

Output:

```
radius = 7
diameter = 14
circumference = 43.98226
area = 153.93791
area_bigger = 201.06176
```

Commentary. Notice how each binding reads as a short declarative sentence. The `area_bigger` line uses literal `8`s; a cleaner version would bind `radius_bigger = radius + 1` and reuse it. Both are correct.

Exercise 2.2 — distance

```
#language axioma/beginner
x1: 0
y1: 0
x2: 3
y2: 4
dx: x2 - x1
dy: y2 - y1
println(sqrt(dx ^ 2 + dy ^ 2))
```

Output: `5.0`.

Commentary. The 3-4-5 right triangle is one of the famous integer solutions to the Pythagorean theorem.

If you predicted `5`, you've internalized what the formula means.

You can write the whole thing without intermediate bindings:

```
println(sqrt((x2 - x1) ^ 2 + (y2 - y1) ^ 2))
```

That works too. Which version is clearer to you? There's no single right answer — it depends on whether `dx` and `dy` mean something on their own. If you were also going to use `dx` somewhere else, naming it would pay off. Here, naming it is just clarity.

Exercise 2.3 — BMI

```
#language axioma/beginner
weight: 70.0
height: 1.75
height_squared: height * height
bmi: weight / height_squared
println(bmi)
```

Output: `22.857142857142858`.

Commentary. The arithmetic is mundane. The one-line form

```
bmi: weight / (height * height)
```

works too; naming the denominator is a readability choice — `height_squared` says *why* the multiplication is there.

Exercise 2.4 — types

```
#language axioma/beginner
a: 7
b: 2
println(a / b)              # 7/2     — exact rational division
println(a + 0.0)            # 7.0     — promoted to Float
a_float: a + 0.0
println(a_float / b)        # 3.5     — float / int gives float
```

Output:

```
7/2
7.0
3.5
```

Commentary. Three numeric types in three lines.

First, `a / b = 7/2` — both operands are integers, so the division is *exact* and the uneven quotient is a

Rational fraction. Same as in Chapter 1.

Second, `a + 0.0` prints `7.0`: adding a Float produces a Float. The third result, `3.5`, is also a Float. You can check either directly with `type(a_float)` or `type(a_float / b)`.

Exercise 2.5 — silent rebinding

```
axioma> n: 1
axioma> n: n + 1
axioma> n
2
```

Commentary. `n` becomes 2. Axioma evaluated the right-hand side first (`n + 1 = 1 + 1 = 2`), then rebound `n` to that — with no warning of any kind. (`n = n + 1` would have done exactly the same thing; the two spellings are synonyms.)

The silence is precisely what makes this dangerous in a long program: the *previous* value of `n` is gone, and nothing told you. If five lines down you wrote code that assumed `n` was still `1`, it now sees `2` and you'll have a hard time figuring out why — the interpreter won't point at the rebinding line for you. The protection has to be a habit, not a diagnostic: give new values new names (`n2: n + 1`) unless you *mean* to update, and save deliberate updates for Chapter 16. Naming hygiene pays compounding interest.

Exercise 2.6 — open

There's no single right answer here. One version:

```
#language axioma/beginner
bill_total: 42.50
tip_percent: 20
tip: bill_total * tip_percent / 100
total: bill_total + tip
println("tip:    $", tip)
println("total:  $", total)
```

Output:

```
tip:    $ 8.5
total:  $ 51
```

Commentary. The point of the exercise is the *naming* — would you call it `bill_total` and `tip_percent` and `tip` and `total`? Or something else? Try reading your version aloud. If it sounds like English, you've named well.

Notice that `total` came out as `51` (no `.0`), but it *is* a float because `tip` is a float. Same display rule from §2.6.

How to use these solutions

Don't read them before attempting the exercises. The predict step is the whole point. After predicting and checking, ask: was my reasoning the same as the solution's? If yours is different but also right, you may have found a cleaner version. That's the goal.

Chapter 2a · Solutions

Exercise 2a.1 — four ticks

```
#language axioma/beginner
repeat 4 [
  println("tick")
]
```

Output:

```
tick
tick
tick
tick
```

Commentary. `loop 4` with an `end` body is the same program. The count is not a variable; there is nothing to print besides the word.

Exercise 2a.2 — walk a list

```
#language axioma/beginner
for v in [2, 4, 6] [
  println(v)
]
for v in [2, 4, 6] [
  println(v * 2)
]
```

Output:

```
2
4
6
4
8
12
```

Commentary. Two loops, two jobs. The second `v` is a new binder; it does not remember the first loop. You could bind `xs: [2, 4, 6]` once and write `for v in xs` twice — that name is yours, not the loop's.

Exercise 2a.3 — a running total

```
#language axioma/beginner
total: 0
for n in 1..5 [
  total = total + n
]
println(total)
```

Output:

```
15
```

Commentary. `n` is the loop's. `total` is yours, so the body assignment writes through. `1 + 2 + 3 + 4 + 5 = 15`.

Exercise 2a.4 — countdown

```
#language axioma/beginner
n: 5
while (n >= 1) [
  println(n)
  n = n - 1
]
```

Output:

```
5
4
3
2
1
```

Commentary. The test is `n >= 1`, so `0` is never printed. After the loop, `n` is `0` — `while` binds nothing, so the counter survives. `for n in 5..1` would also count down (a range walks its own order); the exercise asked for `while` so the update is visible.

Exercise 2a.5 (open) — a table of squares

One solution:

```
#language axioma/beginner
for n in 1..8 [
  println(n, n * n)
]
```

Output:

```
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
```

Commentary. `n` is the loop's binder. A `while` version would need a name of yours (`i: 1` … `i = i + 1`) because `while` binds nothing. Either construct is a correct answer; the open question is which name is whose.

Exercise 2a.6 — worded steps

```
#language axioma/beginner
for i from 20 to 2 by 7 [println(i)]
for i from 20 to 2 by 7
  println(i)
end
for i from 20 to 2 by 6 [println(i)]
```

The first two loops each print 20, 13, 6. The next step would be -1, past 2. Magnitude 6 reaches 20, 14, 8, 2. The direction comes from the bounds.

Chapter 3 · Solutions

Exercise 3.1 — circle_area

```
#language axioma/beginner
pi: 3.14159
circle_area: func(radius) [pi * radius * radius]
println(circle_area(1))
println(circle_area(5))
println(circle_area(100))
```

Output:

```
3.14159
78.53975
31415.9
```

Commentary. Each call binds `radius` to a new value, and the body runs with that binding. The function does not "remember" anything between calls — `pi` is shared because it lives in the outer scope, but each call gets a fresh `radius`.

Exercise 3.2 — f_to_c

```
#language axioma/beginner
f_to_c: func(f) [(f - 32) * 5.0 / 9.0]
println(f_to_c(32))
println(f_to_c(212))
println(f_to_c(98.6))
```

Output:

```
0
100
37
```

Commentary. The third line is the famous body-temperature conversion. The output is `37` (not `37.0`) because of Axioma's trailing-`.0` stripping — the value internally is `37.0`.

Why `5.0 / 9.0` and not `5 / 9`? Not for correctness — `5 / 9` is the *exact rational* `5/9`, so the integer spelling gives the same three answers here (`0`, `100`, `37`; try it). The float literals are a *signal*: temperatures are measurements, and writing `5.0 / 9.0` says "this function lives in decimal-land" — any Fahrenheit input, integer or not, comes back a float instead of a surprising exact fraction like `170/9` for `f_to_c(66)`. Chapter 1's rule: floats are contagious, rationals are exact.

Exercise 3.3 — bmi function

```
#language axioma/beginner
bmi: func(weight, height) [
  h2: height * height
  weight / h2
]
println(bmi(70.0, 1.75))    # 22.857... — normal weight
println(bmi(85.0, 1.80))    # 26.234... — overweight
println(bmi(55.0, 1.65))    # 20.202... — normal weight
```

Commentary. The same style as Exercise 2.3 — we name `height * height` as `h2` for readability; the one-line `weight / (height * height)` works too. The function body has two statements; only the *last* one (`weight / h2`) is the return value.

Exercise 3.4 — tip

```
#language axioma/beginner
tip: func(bill, percent) [bill * percent / 100]
println(tip(50.0, 20))      # 10.0
println(tip(100.0, 15))     # 15.0
println(tip(42.50, 18))     # 7.65
```

Commentary. Note that `percent` here is an integer like `20`, not a fraction like `0.20`. The function divides by `100` to compensate. Either design works; whichever you pick, *be consistent* and write that choice into the parameter name. `percent` already suggests "out of 100"; a parameter named `tip_rate` would suggest a fraction.

Exercise 3.5 — is_even

```
#language axioma/beginner
is_even: func(n) [n % 2 == 0]
println(is_even(4))     # true
println(is_even(7))     # false
println(is_even(0))     # true
```

Commentary. This is your first function that *returns a Boolean* — `true` or `false`, not a number. Boolean-returning functions are the workhorse of Chapter 5 (conditionals): `is_even`, `is_positive`, `is_leap_year`, `is_palindrome`, and so on.

A common beginner mistake here is to write `if n % 2 == 0 then true else false`. That works, but it's redundant — the expression `n % 2 == 0` is *already* either `true` or `false`. Use the simpler form.

Exercise 3.6 — open

```
#language axioma/beginner
monthly_pay: func(hourly_rate, hours_per_week) [
  weeks_per_month: 4.33
  hourly_rate * hours_per_week * weeks_per_month
]
println("$", monthly_pay(25.0, 40))   # $ 4330.0
println("$", monthly_pay(40.0, 20))   # $ 3464.0
```

Commentary. Many right answers. The point is the *naming*. If a future reader (including future-you) can guess what `monthly_pay` does without reading the body, you've named well.

`weeks_per_month = 4.33` is an approximation (there are about 4.33 weeks in an average month). The choice to name it — instead of inlining `4.33` — is the same instinct as naming `pi`: a constant deserves a name even if it only appears once.

Exercise 3.7 — receiver calls and explicit migration

This optional solution uses the full language. Save it separately from the beginner-mode exercises above.

```
decorate(value, suffix, ending) = str(value) + suffix + ending
println("Joan".decorate("!", ";"))
println(5.decorate("!", ";"))

choose(x) = (n -> x + n)
choose(x, n) = x * n
println(5.choose(2))
println(choose(5)(2))

maker(x) = (n -> x + n)
println(maker(5)(2))
add_five: 5.maker()
println(add_five(2))
thunk(x) = func() [x + 1]
println(thunk(5)())
```

Output:

```
Joan!;
5!;
10
7
7
7
6
```

Commentary. Both `decorate` calls pass their receiver first. Its explicit `str` conversion makes the string construction clear for either input type. `5.choose(2)` selects the two-argument definition and multiplies; `choose(5)(2)` selects the unary definition, receives a function and calls that function, which adds. To retain an earlier scalar result-call meaning, write `maker(5)(2)` or bind the returned function. `5.maker()` returns that function; the parentheses do not invoke it a second time. Likewise, `thunk(5)()` deliberately invokes the returned zero-argument function.

How to use these solutions

Don’t peek. Predict each output before checking, even when you’re sure. The compounding payoff of the predict-then-check habit is the single biggest thing you can do for your programming skills this term.

Chapter 4 · Solutions

Each solution shows **all six stages**. The commentary calls out the specific stage where most students get stuck.

Exercise 4.1 — triple

```
#language axioma/beginner
# Data: any number (Integer or Float)
# Signature: triple: Integer -> Integer
# Examples:
#   triple(0)   = 0
#   triple(5)   = 15
#   triple(-7)  = -21
# Template:
# triple: func(n) [
#   ... n ...
# ]
# Body:
triple: func(n) [n * 3]
# Tests:
println(triple(0))      # 0
println(triple(5))      # 15
println(triple(-7))     # -21
```

Commentary. Stage 3 (Examples) is trivial here, but write it anyway. The habit matters more than the difficulty.

Exercise 4.2 — discount

```
#language axioma/beginner
# Data: price (Float, the original price); percent_off (Float, like 25 for 25%)
# Signature: discount: Float Float -> Float
# Examples:
#   discount(100.0, 25)  = 75
#   discount(50.0, 50)   = 25
#   discount(99.99, 10)  = 89.991
# Template:
# discount: func(price, percent_off) [
#   ... price ... percent_off ...
```

```
# ]
# Body:
discount: func(price, percent_off) [
  savings: price * percent_off / 100
  price - savings
]
# Tests:
println(discount(100.0, 25))    # 75.0
println(discount(50.0, 50))     # 25.0
println(discount(99.99, 10))    # 89.991
```

Commentary. Note the intermediate `savings: ...` inside the body. We *could* inline it (`price - price * percent_off / 100`), and that would work — but naming the intermediate value makes the function read aloud as "the discounted price *is* the price minus the savings", which is what we mean. Naming intermediates is the single biggest readability lever in Part I.

Exercise 4.3 — dollars_to_cents

```
#language axioma/beginner
# Data: dollars (Float)
# Signature: dollars_to_cents: Float -> Integer
# Examples:
#   dollars_to_cents(1.50)   = 150
#   dollars_to_cents(0.99)   = 99
#   dollars_to_cents(42.42)  = 4242
# Template:
# dollars_to_cents: func(dollars) [
#   ... dollars ...
# ]
# Body:
dollars_to_cents: func(dollars) [int(dollars * 100)]
# Tests:
println(dollars_to_cents(1.50))     # 150
println(dollars_to_cents(0.99))     # 99
println(dollars_to_cents(42.42))    # 4242
```

Commentary. The signature explicitly says the output is `Integer`, even though the input is `Float` — that asymmetry is the whole point of the exercise. The body uses `int(...)` to convert Float→Integer.

You could also use `round(...)` instead of `int(...)`. They behave differently for non-integer-cents inputs (e.g., what should `dollars_to_cents(0.005)` be — 0 or 1?). The Design Recipe's **examples stage** is where this question would have surfaced. If you listed `dollars_to_cents(0.005)` as an example and committed to an answer, you'd know whether `int` or `round` was right.

Exercise 4.4 — rectangle_perimeter

```
#language axioma/beginner
# Data: width and height (both Floats; could be Integer too)
# Signature: rectangle_perimeter: Float Float -> Float
# Examples:
#   rectangle_perimeter(3, 4)       = 14
#   rectangle_perimeter(10.0, 5.5)  = 31
#   rectangle_perimeter(0, 7)       = 14
# Template:
# rectangle_perimeter: func(width, height) [
#   ... width ... height ...
# ]
# Body:
rectangle_perimeter: func(width, height) [2 * (width + height)]
# Tests:
println(rectangle_perimeter(3, 4))         # 14
println(rectangle_perimeter(10.0, 5.5))    # 31.0
println(rectangle_perimeter(0, 7))         # 14
```

Commentary. Why is the third test 14 and not 7? Because the perimeter is `2 * (0 + 7) = 14` — a degenerate rectangle whose width is zero is really a line segment of length 7, traversed twice (there-and-back). Edge cases like this are exactly why we write examples *before* the body.

Exercise 4.5 — is_positive

```
#language axioma/beginner
# Data: an integer or float
# Signature: is_positive: Float -> Boolean
# Examples:
#   is_positive(5)   = true
#   is_positive(0)   = false       <-- decision: is zero "positive"? No.
#   is_positive(-3)  = false
# Template:
# is_positive: func(n) [
#   ... n ...
# ]
# Body:
is_positive: func(n) [n > 0]
# Tests:
println(is_positive(5))     # true
println(is_positive(0))     # false
println(is_positive(-3))    # false
```

Commentary. Stage 3 is *not* trivial here. Is zero positive? Different mathematicians have different conventions; you have to pick one. The examples force you to commit. In our solution we said "strictly positive", so `is_positive(0) = false`. If you'd said non-negative, your body would be `n >= 0` and

Stage 3 would have shown `is_positive(0) = true`. Either is right; consistency matters.

Exercise 4.6 — open (`split_bill`)

```
#language axioma/beginner
# Data: total (Float) and number of diners (Integer >= 1)
# Signature: split_bill: Float Integer -> Float
# Examples:
#   split_bill(60.0, 3)   = 20
#   split_bill(100.0, 4)  = 25
#   split_bill(99.99, 3)  = 33.33
# Template:
# split_bill: func(total, n) [
#   ... total ... n ...
# ]
# Body:
split_bill: func(total, n) [total / n]
# Tests:
println(split_bill(60.0, 3))      # 20.0
println(split_bill(100.0, 4))     # 25.0
println(split_bill(99.99, 3))     # 33.33
```

Commentary. Chapter 1's number rules matter here. If the caller passes `split_bill(60, 3)` (with an **integer** total `60`, not float `60.0`), the answer is the integer `20`; pass `split_bill(60, 7)` and it's the exact rational `60/7` — correct, but not what a restaurant receipt prints. With a float total the answer is a decimal every time. Try all three.

The Design Recipe should have surfaced this: stage 1 said "total is a Float", so all your tests passed `60.0`, not `60`. That's the recipe doing its quiet work: by forcing you to commit to *types* at stage 1, it kept you out of trouble.

How to use these solutions

Don't peek. Try each exercise yourself, all six stages, before comparing. The point isn't to write *the same* code as the solution — it's to walk through *the same process*. If your data definition, signature, and examples lined up cleanly and your body fell out, you got the lesson.

Chapter 5 · Solutions

Exercise 5.1 — max_of_two

```
#language axioma/beginner
# Data: two numbers
# Signature: max_of_two: Float Float -> Float
# Examples:
#   max_of_two(3, 7)    = 7
#   max_of_two(5, 5)    = 5         <-- tie case
#   max_of_two(-2, -8)  = -2        <-- negatives
max_of_two: func(a, b) [
  if a > b then a else b
]
println(max_of_two(3, 7))     # 7
println(max_of_two(5, 5))     # 5
println(max_of_two(-2, -8))   # -2
```

Commentary. The tie case `max_of_two(5, 5) = 5` is satisfied because `a > b` is `false` when they're equal, so the `else` branch returns `b` — which happens to equal `a`. The function would also work with `>=` instead of `>` — same answer in every case.

There's already a built-in `max` in Axioma (it does the same thing). The exercise is about understanding `if/else`, not reinventing `max`.

Exercise 5.2 — temperature_category

```
#language axioma/beginner
# Data: temperature in Celsius (Float)
# Signature: temperature_category: Float -> String
# Examples:
#   temperature_category(-5)  = "freezing"
#   temperature_category(10)  = "cold"
#   temperature_category(20)  = "comfortable"
#   temperature_category(30)  = "hot"
temperature_category: func(c) [
  if c <= 0 then "freezing"
  else if c <= 15 then "cold"
  else if c <= 25 then "comfortable"
  else "hot"
]
```

```
println(temperature_category(-5))    # freezing
println(temperature_category(10))    # cold
println(temperature_category(20))    # comfortable
println(temperature_category(30))    # hot
```

Commentary. Notice the order. We test the *coldest* boundary first (`c <= 0`), then each subsequent boundary is *higher*. The cascade only reaches `else if c <= 15` after the `c <= 0` branch *failed*, so we know at that point that `c > 0`. This implicit narrowing is what makes cascades read naturally.

Border behavior: what does `temperature_category(0)` return? It's `<= 0`, so "freezing". And `temperature_category(15)`? "cold". *Pick which side of each boundary you mean, then encode it consistently with* `<` *or* `<=`. This is the kind of detail the Design Recipe's example stage forces you to confront before you write code.

Exercise 5.3 — is_in_range

```
#language axioma/beginner
# Data: n, low, high (all Floats, with low <= high)
# Signature: is_in_range: Float Float Float -> Boolean
# Examples:
#   is_in_range(5, 1, 10)   = true
#   is_in_range(15, 1, 10)  = false
#   is_in_range(1, 1, 10)   = true       <-- inclusive at low boundary
#   is_in_range(10, 1, 10)  = true       <-- inclusive at high boundary
#   is_in_range(0, 1, 10)   = false
is_in_range: func(n, low, high) [
  n >= low and n <= high
]
println(is_in_range(5, 1, 10))     # true
println(is_in_range(15, 1, 10))    # false
println(is_in_range(1, 1, 10))     # true
println(is_in_range(10, 1, 10))    # true
println(is_in_range(0, 1, 10))     # false
```

Commentary. No `if/else` needed — the `and` of two comparisons *is* a Boolean. This is the most important lesson of the chapter: a Boolean expression doesn't need to be wrapped in `if true then true else false`. The Boolean expression is *already* the answer.

The choice between inclusive and exclusive at the boundaries — `>=` vs `>`, `<=` vs `<` — is a design choice. The examples committed us to "inclusive at both ends," and the body honored that.

The optional alias spelling is `low <= n && n <= high`. It has the same result and boundary behavior as the word form above.

Exercise 5.4 — tip_smart

```
#language axioma/beginner
# Data: bill (Float), percent (Float)
# Signature: tip_smart: Float Float -> Float
# Examples:
#   tip_smart(8.0, 20)    = 2       (small bill — flat $2)
#   tip_smart(50.0, 20)   = 10      (regular)
#   tip_smart(100.0, 15)  = 15      (regular)
tip_smart: func(bill, percent) [
  if bill < 10 then 2.0
  else bill * percent / 100
]
println(tip_smart(8.0, 20))      # 2.0
println(tip_smart(50.0, 20))     # 10.0
println(tip_smart(100.0, 15))    # 15.0
```

Commentary. The two-branch `if/else` switches between *two different formulas* depending on a condition on the inputs. This is how policy rules get encoded in software — "if X, do this; otherwise, do that." Familiar to anyone who's set up an automatic email rule.

Exercise 5.5 — quadrant

```
#language axioma/beginner
# Data: x, y (Floats)
# Signature: quadrant: Float Float -> String
# Decision (committed up front!): if either coordinate is 0, the point
# is "on axis" — not in any quadrant.
# Examples:
#   quadrant(3, 4)    = "I"
#   quadrant(-3, 4)   = "II"
#   quadrant(-3, -4)  = "III"
#   quadrant(3, -4)   = "IV"
#   quadrant(0, 5)    = "on axis"
#   quadrant(0, 0)    = "on axis"
quadrant: func(x, y) [
  if x == 0 or y == 0 then "on axis"
  else if x > 0 and y > 0 then "I"
  else if x < 0 and y > 0 then "II"
  else if x < 0 and y < 0 then "III"
  else "IV"
]
println(quadrant(3, 4))      # I
println(quadrant(-3, 4))     # II
println(quadrant(-3, -4))    # III
println(quadrant(3, -4))     # IV
```

```
println(quadrant(0, 5))      # on axis
println(quadrant(0, 0))      # on axis
```

Commentary. Putting the `on axis` case *first* means the rest of the cascade can assume `x != 0` and `y != 0`. The `else "IV"` at the bottom is reached only when none of the preceding conditions hit — which (given we've ruled out the axes) means `x > 0 and y < 0`.

This is the most common bug source in cascades: the *order matters*. If you put the `else "IV"` case as `else if x > 0 and y < 0 then "IV"` and there's a *new* combination you haven't thought of, the `else` catches it. With an open `else`, you'd want to write a *defensive* last-line check — but here we've enumerated all cases the recipe asked for.

Exercise 5.6 — open (umbrella decision)

```
#language axioma/beginner
umbrella_decision: func(rain_chance_percent, has_jacket) [
  if rain_chance_percent >= 50 then "bring umbrella"
  else if rain_chance_percent >= 25 and not has_jacket then "bring umbrella"
  else "leave umbrella at home"
]
println(umbrella_decision(80, true))     # bring umbrella
println(umbrella_decision(30, true))     # leave umbrella at home
println(umbrella_decision(30, false))    # bring umbrella
println(umbrella_decision(10, false))    # leave umbrella at home
```

Commentary. Many right answers. Note the use of `and` and `not` in the same line: "rain chance is at least 25% **and** no jacket." This is where Booleans earn their keep — encoding policy with the same connectives you'd use to *describe* the policy in English.

Exercise 5.7 — guarded temperature categories

```
#language axioma/beginner
temperature_guard(c) when c <= 0 = "freezing"
temperature_guard(c) when c <= 15 = "cold"
temperature_guard(c) when c <= 25 = "comfortable"
temperature_guard(c) = "hot"
println(temperature_guard(-5)) # freezing
println(temperature_guard(0))  # freezing
println(temperature_guard(15)) # cold
println(temperature_guard(25)) # comfortable
println(temperature_guard(30)) # hot
```

The first applicable guard wins. Putting the c<=25 clause first would classify freezing inputs as comfort-

able. Removing the final clause makes 30 return none.

Exercise 5.8 — cond umbrella policy

```
umbrella_cond: func(chance, jacket) [
    cond
    | chance >= 50 => "bring umbrella"
    | chance >= 25 and not jacket => "bring umbrella"
    | else => "leave umbrella at home"
]
println(umbrella_cond(80, true))  # bring umbrella
println(umbrella_cond(30, true))  # leave umbrella at home
println(umbrella_cond(30, false)) # bring umbrella
println(umbrella_cond(10, false)) # leave umbrella at home
println(cond | false => 1) # none
```

The conditions and order are unchanged from the cascade. Absence of a fallback is observable; it is not an implicit policy result.

Exercise 5.9 — pair patterns and guards

```
pair_order: func(pair) [
    case pair
    | (x, y) when x < y => "ascending"
    | (x, y) when x == y => "equal"
    | (x, y) => "descending"
    | _ => "not a pair"
]
println(pair_order((1, 2))) # ascending
println(pair_order((2, 2))) # equal
println(pair_order((3, 2))) # descending
println(pair_order("pair")) # not a pair
println(match (1, 2) with | (x, y) when x < y => "ascending" | _ => "other") #
 ↪  ascending
println(switch (1, 2) | (x, y) when x < y => "ascending" | _ => "other") # ascending
```

This function's pair inputs are specified to contain numbers; the tuple pattern alone does not validate numeric fields. Non-comparable fields could make a guard error. Pattern failure and guard errors are different outcomes.

Exercise 5.10 — result-first equations

```
#language axioma/beginner
temperature_equation(c) = "freezing", if c <= 0
                        = "cold", if c <= 15
                        = "comfortable", if c <= 25
                        = "hot", otherwise
println(temperature_equation(0))  # freezing
println(temperature_equation(15)) # cold
println(temperature_equation(25)) # comfortable
println(temperature_equation(30)) # hot
```

False guards try the next clause. No eligible clause at a supported arity returns none. An error evaluating a guard propagates, even if a later clause could have handled the input. The optional full-language check below shows the difference without changing the dispatch rules:

```
risky(n) when 10 / n > 1 = "selected"
risky(n) = "fallback"
println(error?(try(risky(0)))) # true
println(risky(20)) # fallback
println(cond | true => "chosen" | else => 10 / 0) # chosen
```

A false guard skipped a body; it did not catch an error. Likewise, the unselected cond body is never evaluated.

How to use these solutions

Don't read them before attempting the exercises. The recipe is the point. After you have your six stages on paper, *then* compare.

Exercise 5.11 — optional guard bindings

```
small_order_limit(value)
  | (price, quantity) <- value,
    price > 0, quantity > 0,
    let subtotal = price * quantity,
    subtotal <= 100 = subtotal
  | let subtotal = 999, false = subtotal
  | otherwise = 0
println(small_order_limit((25, 4))) # 100
println(small_order_limit((25, 5))) # 0
println(small_order_limit("bad"))   # 0
```

The later branch runs only when the first fails. Its `subtotal` belongs to that attempt; the false condition discards that choice, so the final branch returns 0. It cannot alter the first branch's binding or an outer

name. If a branch had performed an explicit effect before failing, that effect would remain.

Exercise 5.12 — capture equality

```
func equal_pair((a, b)) when a == b [true]
func equal_pair(_) [false]
println(equal_pair((4, 4))) # true
println(equal_pair((4, 5))) # false
```

`(x, x)` duplicates a capture. `(a, b) as a` reuses the field name for the whole tuple. `(a, b) as pair` gives the fields and the whole distinct names. Equality belongs in the guard. This optional example requires the evaluator.

Exercise 5.13 — bounded sequence search

```
let p = sequence_pattern([{segment: "left", min: 1}, {segment: "right", min: 1}])
let r = sequence_match_all([1, 2, 3], p, {
  extract: func(b) [(len(b.left), len(b.right))]
})
println(r.values) # [(1, 2), (2, 1)]
println(r.exhausted) # true
let capped = sequence_match_all([1, 2, 3], p, {matches: 1})
println(capped.status) # incomplete
println(capped.reason) # match_limit
println(sequence_match_all([1], p).status) # no_match
let first = sequence_match_first([1, 2, 3], p)
println(first.complete)  # true
println(first.exhausted) # false
```

Each segment must contain at least one element. Shorter left segments come first. A second accepted decomposition proves that the result cap truncated the all-mode search. A one-element subject cannot supply two nonempty segments, so its exhausted search proves no match. First success satisfies the first request, without proving uniqueness or exhausting alternatives. This optional solution requires full Axioma and the evaluator.

Exercise 5.14 — nested event search

```
let login = seqpat ['login, person]
let history = seqpat ['events, [..earlier, nested(login), ..later]]
let r = sequence_match_all(
  ['events, [['ping], ['login, 'ada], ['ping], ['login, 'bob]]], history)
println(len(r.matches)) # 2
println(len(r.matches[1].earlier)) # 1
```

```
println(len(r.matches[2].earlier)) # 3
let limited = sequence_match_all(
  ['events, [['ping], ['login, 'ada], ['ping], ['login, 'bob]]], history,
  {matches: 1})
println(limited.status) # incomplete
println(limited.reason) # match_limit
```

The nested event header consumes one event while the two segments divide the surrounding event list. Shorter prefixes come first, so Ada precedes Bob. One additional accepted candidate proves truncation. All components share one work budget, and captures remain explicit result data. This solution is intentionally evaluator-only.

Chapter 6 · Solutions

Exercise 6.1 — count_zeros (no-recursion version)

```
#language axioma/beginner
# Data: array of integers, length 0..3 only
# Signature: count_zeros: Array -> Integer
# Examples:
#   count_zeros([])          = 0
#   count_zeros([0])         = 1
#   count_zeros([5])         = 0
#   count_zeros([0, 0, 5])   = 2
count_zeros: func(arr) [
  if len(arr) == 0 then 0
  else if len(arr) == 1 then (if arr[1] == 0 then 1 else 0)
  else if len(arr) == 2 then
    (if arr[1] == 0 then 1 else 0) + (if arr[2] == 0 then 1 else 0)
  else (
    (if arr[1] == 0 then 1 else 0) +
    (if arr[2] == 0 then 1 else 0) +
    (if arr[3] == 0 then 1 else 0)
  )
]
println(count_zeros([]))          # 0
println(count_zeros([0]))         # 1
println(count_zeros([5]))         # 0
println(count_zeros([0, 0, 5]))   # 2
```

Commentary. This version manually expands one branch per possible length. It's painful at length 4, impossible at "any length." Chapter 7 will collapse all four branches into a two-line recursion: *empty array → 0; non-empty → (1 if first is 0 else 0) + count_zeros(rest).* That's why we need recursion.

Exercise 6.2 — is_short

```
#language axioma/beginner
is_short: func(arr) [ len(arr) < 3 ]
println(is_short([]))         # true
println(is_short([1, 2]))     # true
println(is_short([1, 2, 3]))  # false
```

Commentary. No `if/else` needed — the comparison is already a Boolean. (Same lesson as Exercise

5.3 — Boolean expressions don't need to be wrapped in `if true then true else false`.)

Exercise 6.3 — swap_first_last (3-element)

```
#language axioma/beginner
# Signature: swap_first_last: Array -> Array
swap_first_last: func(arr) [
  [arr[3], arr[2], arr[1]]
]
println(swap_first_last([1, 2, 3]))           # [3, 2, 1]
println(swap_first_last(["a", "b", "c"]))     # ["c", "b", "a"]
```

Commentary. The body returns a three-element literal containing three index expressions in reverse order. Commas distinguish this array from a sequence of statements.

The general "reverse an array of any length" function will be a Chapter 7 exercise.

Exercise 6.4 — second_or_zero

```
#language axioma/beginner
second_or_zero: func(arr) [
  if len(arr) < 2 then 0
  else arr[2]
]
println(second_or_zero([]))             # 0
println(second_or_zero([10]))           # 0
println(second_or_zero([10, 20]))       # 20
println(second_or_zero([10, 20, 30]))   # 20
```

Commentary. The defensive check comes *first*, before the indexing. If we wrote `if arr[2] != none then arr[2] else 0`, the out-of-bounds access on `[]` would crash before the `if` ever ran. Order matters: check that the operation is *legal* before you perform it.

Exercise 6.5 — glue_with_separator

```
#language axioma/beginner
# Signature: glue: Any Array Array -> Array
glue: func(sep, a, b) [
  a + push([], sep) + b
]
println(glue("-", [1, 2], [3, 4]))        # [1, 2, "-", 3, 4]
println(glue(0, [1], [2]))                # [1, 0, 2]
println(glue("|", [], ["x"]))             # ["|", "x"]
```

Commentary. `push([], sep)` builds a one-element Array containing `sep`. In this expression, `a + [sep] + b` also works. The explicit trailing-comma form `[sep,]` is useful when square brackets could otherwise be read as a branch or function body (§6.6).

Exercise 6.6 — three things to know about an array

```
#language axioma/beginner
xs: [14, 7, 22, 7, 100, 7]

firstof: func(arr) [ first(arr) ]
lastof: func(arr) [ arr[len(arr)] ]
lenof: func(arr) [ len(arr) ]

println(firstof(xs))   # 14
println(lastof(xs))    # 7
println(lenof(xs))     # 6

describe: func(arr) [
  "first=" + firstof(arr) + " last=" + lastof(arr) + " len=" + lenof(arr)
]
println(describe(xs))  # first=14 last=7 len=6
```

Commentary. Three trivial functions plus one that composes them. The point isn't the math — it's that *one* array can be the input to several different functions. We'll see this constantly in Part III when we get to `map`, `filter`, `reduce`.

There's an Axioma `last(arr)` builtin too, so `lastof = func(arr) [last(arr)]` is the more idiomatic version of that helper.

Exercise 6.7 — aliases and rejected growth

The valid extension produces `[3, 2, 4, 5, 6]`; both names denote the same Array. The rejected batch must validate all candidates first, so it cannot append `5` before discovering the String. Nested candidates likewise receive no new contracts on rejection. An unannotated Array can accept a String at runtime; static checking can still report a different conclusion.

```
scores :: Array of Integer: [1, 2]
other_scores: scores
other_scores[1] = 3
push(other_scores, 4)
extend(other_scores, [5, 6])
println(scores)                    # [3, 2, 4, 5, 6]
```

How to use these solutions

Look only after you have your own attempt on paper. The recipe is the point — typing six versions of `count_zeros` is much more educational than reading one.

Chapter 7 · Solutions

Exercise 7.1 — count_zeros (recursive)

```
#language axioma/beginner
# Data: array of any length
# Signature: count_zeros: Array -> Integer
# Examples:
#   count_zeros([])             = 0
#   count_zeros([0])            = 1
#   count_zeros([1, 0, 2, 0, 0]) = 3
#   count_zeros([1, 2, 3])      = 0
count_zeros: func(arr) [
  if len(arr) == 0 then 0
  else if first(arr) == 0 then 1 + count_zeros(rest(arr))
  else count_zeros(rest(arr))
]
println(count_zeros([]))             # 0
println(count_zeros([0]))            # 1
println(count_zeros([1, 0, 2, 0, 0])) # 3
println(count_zeros([1, 2, 3]))      # 0
```

Commentary. Compare this with Exercise 6.1. The length-explicit version was 9 lines and only handled arrays of length 0..3. The recursive version is 4 lines and handles arrays of any length. *That's* the payoff of structural recursion.

The body splits into three cases: empty, first-is-zero, otherwise. It's the same shape as `has_value` from §7.7 — boolean test on the front, recur on the rest, combine.

Exercise 7.2 — smallest

```
#language axioma/beginner
smallest: func(arr) [
  if len(arr) == 0 then none
  else if len(arr) == 1 then first(arr)
  else [
    rest_smallest: smallest(rest(arr))
    if first(arr) < rest_smallest then first(arr) else rest_smallest
  ]
]
println(smallest([]))             # none
```

```
println(smallest([7]))             # 7
println(smallest([3, 1, 4, 1, 5, 9])) # 1
```

Commentary. Identical structure to `largest` from §7.6, with `>` flipped to `<`. This is the moment when you start to *see* the pattern: "recursive aggregate over a list" is one shape, and the combining rule is the only thing that changes.

Exercise 7.3 — sum_of_squares

```
#language axioma/beginner
sum_of_squares: func(arr) [
  if len(arr) == 0 then 0
  else first(arr) * first(arr) + sum_of_squares(rest(arr))
]
println(sum_of_squares([]))        # 0
println(sum_of_squares([3]))       # 9
println(sum_of_squares([1, 2, 3])) # 14
```

Commentary. Compare with `sum_list` from §7.2 — only the combining rule changed from `first(arr) + recur` to `first(arr) * first(arr) + recur`. The skeleton is identical.

(For very long arrays we'd evaluate `first(arr)` twice; a local binding `v: first(arr); v * v + recur` would compute it once. Premature optimization at this point — but a Chapter-12 refactor.)

Exercise 7.4 — count_above

```
#language axioma/beginner
count_above: func(arr, threshold) [
  if len(arr) == 0 then 0
  else if first(arr) > threshold then 1 + count_above(rest(arr), threshold)
  else count_above(rest(arr), threshold)
]
println(count_above([], 10))               # 0
println(count_above([5, 12, 8, 99], 10))   # 2
println(count_above([1, 2, 3], 100))       # 0
```

Commentary. Two-argument recursion. The `threshold` parameter doesn't change as we descend — we pass it through unchanged in both branches. The only "real" recursion is on the array.

In Part III we'll meet `filter` and `count_if`, which generalize this pattern: "test the front; if it passes, count it; either way, recur on the rest."

Exercise 7.5 — double_each

```
#language axioma/beginner
double_each: func(arr) [
  if len(arr) == 0 then []
  else push([], 2 * first(arr)) + double_each(rest(arr))
]
println(double_each([]))         # []
println(double_each([1]))        # [2]
println(double_each([1, 2, 3]))  # [2, 4, 6]
```

Commentary. This is the first exercise where the *recursive return value is an array*, not a number. The body says: take the first element, double it, package it in a singleton (using `push([], ...)` per §6.6's workaround), then glue that onto the front of the recursively-doubled rest.

In Part III this becomes `map(func(x) [2 * x], arr)` — the combining rule is hidden inside a higher-order function.

Exercise 7.6 — reverse

```
#language axioma/beginner
reverse_arr: func(arr) [
  if len(arr) == 0 then []
  else reverse_arr(rest(arr)) + push([], first(arr))
]
println(reverse_arr([]))        # []
println(reverse_arr([42]))      # [42]
println(reverse_arr([1, 2, 3])) # [3, 2, 1]
```

Commentary. The crucial design choice was *where* to put `first(arr)` in the output. We chose: at the *end*. The reverse of `[1, 2, 3]` is the reverse of `[2, 3]` followed by `[1]`. That gives `[3, 2] + [1] = [3, 2, 1]`. ✓

What happens at each step:

```
reverse_arr([1, 2, 3])
  = reverse_arr([2, 3]) + [1]
  = (reverse_arr([3]) + [2]) + [1]
  = ((reverse_arr([]) + [3]) + [2]) + [1]
  = (([] + [3]) + [2]) + [1]
  = ([3] + [2]) + [1]
  = [3, 2] + [1]
  = [3, 2, 1]
```

There's a sneaky cost here: each `+` on arrays copies. So `reverse_arr` on an `N`-element array does work proportional to N^2, not `N`. Chapter 15's accumulator-style rewrite makes it `N`.

This is also why Axioma has a built-in `reverse` — it does the work efficiently. The exercise is in seeing how the recursion works, not in deploying the fastest possible primitive.

Exercise 7.7 — is_palindrome

```
#language axioma/beginner
is_palindrome: func(s) [
  n: length(s)
  if n <= 1 then true
  else if s[1] != s[n] then false
  else is_palindrome(s[2: n - 1])
]

is_palindrome2: func(s) [
  go: func(start, finish) [
    if start >= finish then true
    else if s[start] != s[finish] then false
    else go(start + 1, finish - 1)
  ]
  go(1, length(s))
]

println(is_palindrome(""))            # true
println(is_palindrome("a"))           # true
println(is_palindrome("civic"))       # true
println(is_palindrome("cynic"))       # false
println(is_palindrome("redivider"))   # true
println(is_palindrome("runner"))      # false
println(is_palindrome2("civic"))      # true
println(is_palindrome2("cynic"))      # false
println(is_palindrome2("redivider"))  # true
```

Commentary. The slice form and the index form are the same three questions. What changes is the *smaller problem*: a shorter string versus a narrower window. The index form does not allocate, so it is the one you want if the string is long; the slice form is the one that matches the paper algorithm one-for-one.

Axioma strings are 1-indexed and slices are inclusive. The middle of a length-`n` string is `s[2: n - 1]`, not `s[1: n - 1]`. Python's `s[1:-1]` and JavaScript's `s.slice(1, s.length - 1)` name the same characters with different numbers (0-based, exclusive end). Copy those numbers into Axioma and you drop the wrong characters. `cynic` is the example that catches the fencepost: a slice that overshoots becomes empty, empty is a palindrome, and you have a wrong `true` at exit 0.

Exercise 7.8 — take_while and drop_while

```
#language axioma/beginner
take_while: func(arr, pred) [
  if len(arr) == 0 then []
  else if pred(first(arr)) then [first(arr) | take_while(rest(arr), pred)]
  else []
```

```
]

drop_while: func(arr, pred) [
  if len(arr) == 0 then []
  else if pred(first(arr)) then drop_while(rest(arr), pred)
  else arr
]

data: [2, 6, 42, 5, 7, 20, 3]
is_even: func(n) [n % 2 == 0]
println(take_while(data, is_even))   # [2, 6, 42]
println(drop_while(data, is_even))   # [5, 7, 20, 3]
println(take_while([], is_even))     # []
println(drop_while([], is_even))     # []
println(take_while([1, 2, 3], is_even))  # []
println(drop_while([1, 2, 3], is_even))  # [1, 2, 3]
println(take_while([2, 4], is_even))     # [2, 4]
println(drop_while([2, 4], is_even))     # []
```

Commentary. Both stop at the first failure. `take_while` *builds* a prefix (cons the head onto the recursive prefix, or return `[]` when the head fails). `drop_while` *skips* (recur with no combining, or return the array as-is when the head fails). `20` is even and it is in the `drop_while` result because the walk already stopped.

`drop_while` is in tail position. `take_while` is not — the cons waits for the recursive prefix. For this exercise both are the natural structural recursions. An accumulator `take_while` that conses onto the front of `acc` would build the prefix backwards and need a `reverse` at the end; that is a Chapter 15 rewrite, not a requirement here.

Together they partition `arr` into `(prefix, rest)` at the first failure. `take_while(arr, pred) + drop_while(arr, pred)` equals `arr` for every `arr` and `pred`.

How to use these solutions

Look only after you have your own attempt on paper. Type the recursion out yourself; the muscle memory of writing the two-case template is more useful than reading mine.

Chapter 8 · Solutions

The shared preamble (all six exercises start with this):

```
#language axioma/beginner
concept Node
Node has value: 0
Node has left: none
Node has right: none

# Standard test tree:
#       3
#      / \
#     1   5
#        / \
#       4   7
leaf4: a Node {value: 4}
leaf7: a Node {value: 7}
n5: a Node {value: 5, left: leaf4, right: leaf7}
leaf1: a Node {value: 1}
root: a Node {value: 3, left: leaf1, right: n5}
```

Exercise 8.1 — tree_sum

```
tree_sum: func(t) [
  if t == none then 0
  else t.value + tree_sum(t.left) + tree_sum(t.right)
]
println(tree_sum(none))    # 0
println(tree_sum(leaf4))   # 4
println(tree_sum(root))    # 20
```

Commentary. Compare with `sum_list` from §7.2 — same shape, but with *two* recursive calls (one per child) because trees branch. The combiner is `t.value + left_sum + right_sum`.

Exercise 8.2 — count_leaves

```
count_leaves: func(t) [
  if t == none then 0
  else if t.left == none and t.right == none then 1
  else count_leaves(t.left) + count_leaves(t.right)
]
println(count_leaves(none))    # 0
println(count_leaves(leaf4))   # 1
println(count_leaves(root))    # 3
```

Commentary. Three branches: empty (0), leaf (1), internal (recur). The "is leaf" test is `t.left == none and t.right == none`. An internal node *itself* doesn't count — we just add up the leaves in the two subtrees.

This is the first function where the *structure* of a node matters as much as its *children*. Recipes scale: data with three flavors → function with three branches.

Exercise 8.3 — tree_max

```
tree_max: func(t) [
  if t == none then none
  else if t.left == none and t.right == none then t.value
  else [
    lm: tree_max(t.left)
    rm: tree_max(t.right)
    best_child:       if lm == none then rm
      else if rm == none then lm
      else if lm > rm then lm else rm
    if t.value > best_child then t.value else best_child
  ]
]
println(tree_max(none))    # none
println(tree_max(root))    # 7
```

Commentary. Three branches: empty (`none`), leaf (just the value), internal (a four-way comparison). Inside the internal branch we have to be defensive: a non-empty internal node can still have one empty subtree, in which case the recursive call on the empty side returns `none`, which is unusable for `>`. The `if lm == none then rm` cascade handles that.

An empty result still needs an explicit case. Chapter 18 · Types and Data Modeling introduces `Option` and tagged variants, so a result can express absence or a numeric payload directly. Typed constructor matches can check coverage of those cases. A Concept partition does not remove the need to handle an empty result.

Exercise 8.4 — tree_min

```
tree_min: func(t) [
  if t == none then none
  else if t.left == none and t.right == none then t.value
  else [
    lm: tree_min(t.left)
    rm: tree_min(t.right)
    best_child:       if lm == none then rm
      else if rm == none then lm
      else if lm < rm then lm else rm
    if t.value < best_child then t.value else best_child
  ]
]
println(tree_min(none))    # none
println(tree_min(root))    # 1
```

Commentary. Mirror of 8.3 with > flipped to <. The structure is identical — the only thing that changed is which direction we're optimizing.

Exercise 8.5 — inorder

```
concat3: func(a, b, c) [a + b + c]
inorder: func(t) [
  if t == none then []
  else concat3(inorder(t.left), push([], t.value), inorder(t.right))
]
println(inorder(none))    # []
println(inorder(root))    # [1, 3, 4, 5, 7]
```

Commentary. The classic in-order tree traversal: left subtree, this node, right subtree, in that order. We need a 3-way concatenation in the middle. Two reasonable approaches:

- Define a helper `concat3(a, b, c)` (what we did).
- Inline: `inorder(t.left) + push([], t.value) + inorder(t.right)`.

The second works too. The helper makes the pattern explicit.

The singleton `push([], t.value)` packages the node value in an Array. The explicit one-element literal `[t.value,]` is an alternative (§6.6).

If we'd used `preorder` (this node first), the code would be `push([], t.value) + preorder(t.left) + preorder(t.right)`. If we'd used `postorder`, the value goes last. Same recursion shape, different ordering of the three pieces.

Exercise 8.6 — tree_height_balanced

```
tree_depth: func(t) [
  if t == none then 0
  else [
    dl: tree_depth(t.left)
    dr: tree_depth(t.right)
    1 + (if dl > dr then dl else dr)
  ]
]
abs_diff: func(a, b) [if a > b then a - b else b - a]

tree_balanced: func(t) [
  if t == none then true
  else [
    dl: tree_depth(t.left)
    dr: tree_depth(t.right)
    if abs_diff(dl, dr) > 1 then false
    else tree_balanced(t.left) and tree_balanced(t.right)
  ]
]
println(tree_balanced(none))    # true
println(tree_balanced(root))    # true

# A deliberately unbalanced (lopsided) tree:
lop:   a Node {value: 1, left: a Node {value: 2,
                                       left: a Node {value: 3}}}
println(tree_balanced(lop))     # false  (depth 3 on left, 0 on right)
```

Commentary. Two layers of recursion, which is why this one is hard. `tree_balanced(t)` says:

- Empty trees are balanced.
- A non-empty node is balanced if (a) its own subtrees differ in depth by ≤ 1, *and* (b) both subtrees are *themselves* balanced.

That second clause is the subtle bit. Just checking the root's own subtree-depth difference isn't enough — a deeply lopsided left subtree could itself be unbalanced even if its overall depth matches the right.

There's an efficiency issue: `tree_depth` walks every node, and we call it *at every recursive level*. That's quadratic time in the tree's size. A smarter version returns *both* a depth and a balance-bit in one pass, so it stays linear. We'll learn the technique in Chapter 15. For now: simple is fine.

How to use these solutions

Look only after you have your own attempt on paper. The recipe is the point — and trees give you twice as many chances to apply it as lists do.

Chapter 9 · Solutions

Common preamble for the household exercises:

```
#language axioma/beginner
concept Person
Person has name: ""
Person has household: none

concept Household
Household has address: ""
Household has members: []

h1: a Household {address: "123 Main", members: [a Person {name: "Alice"}, a Person
 ↪  {name: "Bob"}]}
alice: a Person {name: "Alice", household: h1}
bob: a Person {name: "Bob", household: h1}

h2: a Household {address: "456 Oak", members: [a Person {name: "Carl"}]}
carl: a Person {name: "Carl", household: h2}
```

Common preamble for the expression exercises:

```
concept Lit
Lit has value: 0

concept Op
Op has kind: "+"
Op has lhs: none
Op has rhs: none

e1: a Lit {value: 1}
e2: a Lit {value: 2}
plus: an Op {kind: "+", lhs: e1, rhs: e2}
e3: a Lit {value: 3}
e4: a Lit {value: 4}
minus: an Op {kind: "-", lhs: e3, rhs: e4}
expr: an Op {kind: "*", lhs: plus, rhs: minus}
```

Exercise 9.1 — name_in_household

```
names_in_members: func(members) [
  if len(members) == 0 then []
  else push([], first(members).name) + names_in_members(rest(members))
]
names_in_household: func(p) [
  if p == none then []
  else if p.household == none then []
  else names_in_members(p.household.members)
]
println(names_in_household(alice))   # ["Alice", "Bob"]
println(names_in_household(a Person {name: "Lonely", household: none}))   # []
```

Commentary. Two layers of traversal: `names_in_household` hops one step out (Person → Household), then `names_in_members` does the list-recursion (Array of Persons → Array of String). Only `names_in_members` calls itself. This is a helper followed by list recursion, not mutual recursion between the two functions.

The `push([], first(members).name)` constructs a singleton Array, as shown from §6.6. The `+` then glues this one-name array onto the names collected from the rest of the members.

Exercise 9.2 — count_total_people

```
count_total_people: func(hs) [
  if len(hs) == 0 then 0
  else len(first(hs).members) + count_total_people(rest(hs))
]
println(count_total_people([]))         # 0
println(count_total_people([h1, h2]))   # 3
```

Commentary. Standard list recursion (Chapter 7), but the "combine" rule reaches *into* each element to get `len(first(hs).members)`. The recursion is on the array of households, not on the members within each household — those are handled all at once by `len(...)`. One walk through the array, one `len()` per household.

Exercise 9.3 — expr_depth

```
expr_depth: func(e) [
  if e is Lit then 1
  else if e is Op then [
    dl: expr_depth(e.lhs)
    dr: expr_depth(e.rhs)
    1 + (if dl > dr then dl else dr)
```

```
  ]
  else 0
]
println(expr_depth(e1))    # 1
println(expr_depth(plus))  # 2
println(expr_depth(expr))  # 3
```

Commentary. Two branches — one per variant. The structure mirrors `tree_depth` from Chapter 8, but with `is Lit` and `is Op` instead of `t == none` and "internal node." The math is the same: leaf returns 1, internal returns 1 + max of children's depths.

Exercise 9.4 — count_ops

```
count_ops: func(e) [
  if e is Lit then 0
  else if e is Op then 1 + count_ops(e.lhs) + count_ops(e.rhs)
  else 0
]
println(count_ops(e1))     # 0
println(count_ops(plus))   # 1
println(count_ops(expr))   # 3
```

Commentary. A `Lit` is not an operator, so it contributes `0`. An `Op` is, so it contributes `1` plus whatever the subexpressions contribute. Same recursion shape as `tree_size` from §8.5; only the combining rule changed (`0` and `1 + recur + recur` vs `0` and `1 + recur + recur` — actually it *is* the same combining rule, because every `Op` counts as exactly one node).

Exercise 9.5 — add_one_to_lits

```
add_one_to_lits: func(e) [
  if e is Lit then a Lit {value: e.value + 1}
  else if e is Op then an Op {
    kind: e.kind,
    lhs: add_one_to_lits(e.lhs),
    rhs: add_one_to_lits(e.rhs)
  }
  else none
]
new_e1: add_one_to_lits(e1)
println(new_e1.value)        # 2
new_plus: add_one_to_lits(plus)
println(new_plus.lhs.value)  # 2
println(new_plus.rhs.value)  # 3
```

Commentary. First time we *construct* in both branches. For a `Lit`, we build a new `Lit` with `value`

`+ 1`. For an `Op`, we build a new `Op` with the same `kind` but with each sub-expression transformed by a recursive call.

Important: this returns a *new* expression. The original is unchanged. The output expression has the same shape and the same operator kinds — only the literal values are different.

This is a *transformation pass* — the kind of thing a compiler does. Real compilers do dozens of these passes, each a structurally-recursive walk of an AST that produces a new AST.

Exercise 9.6 — extend the evaluator

```
eval_expr: func(e) [
  if e is Lit then e.value
  else if e is Op then [
    l: eval_expr(e.lhs)
    r: eval_expr(e.rhs)
    if e.kind == "+" then l + r
    else if e.kind == "-" then l - r
    else if e.kind == "*" then l * r
    else if e.kind == "/" then l / r
    else if e.kind == "^" then l ^ r    # NEW
    else none
  ]
  else none
]
pow_expr: an Op {
  kind: "^",
  lhs: a Lit {value: 2},
  rhs: a Lit {value: 8}
}
println(eval_expr(pow_expr))   # 256
```

Commentary. Adding a binary operator is a one-line change: extend the `else if e.kind == ...` cascade with the new symbol. The data definition (`Op has kind: "+"`) didn't need to change — `kind` is just a string, and the evaluator decides which strings mean what.

The *harder* variation — a unary `"neg"` via a third Concept — exposes the trade-off. With three Concepts the evaluator grows a third `else if e is UnaryOp then ...` branch, and every other pass over the expression tree also has to add a branch. Real-world interpreters trade this off: more variants means clearer type-level distinctions but more boilerplate per pass.

(Chapter 22 — the meta-circular evaluator for Axioma itself — will use *dozens* of variants. The pattern scales; the boilerplate is real.)

How to use these solutions

Try first; compare second. Every exercise here is small but the *pattern* — branch on shape, recur on children, combine — generalizes to interpreters, compilers, structural editors, spreadsheet engines, query planners, and everything else built on recursive data.

Chapter 10 · Solutions

Exercise 10.1 — triple_each

```
#language axioma/beginner
triple_each: func(arr) [ map(func(x) [x * 3], arr) ]
println(triple_each([]))            # []
println(triple_each([1, 2, 3]))     # [3, 6, 9]
```

Commentary. The whole point of `map` is that this is the *entire* solution. The recursion is hidden inside the builtin. Compare with Exercise 7.5 (`double_each`), which was the same shape but you had to write the recursion by hand.

Exercise 10.2 — keep_positive

```
#language axioma/beginner
keep_positive: func(arr) [ filter(func(x) [x > 0], arr) ]
println(keep_positive([]))            # []
println(keep_positive([1, -2, 3, 0]))  # [1, 3]
```

Commentary. Predicate names by convention read like yes/no questions: `is_positive` would also be fine. Note the predicate `x > 0` excludes zero — `>= 0` would include it. Asking the user *which* they want is the kind of "design choice" HtDP calls out as part of the recipe (Chapter 4 step 1: nail down the data).

Exercise 10.3 — product

```
#language axioma/beginner
product: func(arr) [ reduce(func(a, b) [a * b], 1, arr) ]
println(product([]))          # 1
println(product([4]))         # 4
println(product([2, 3, 5]))   # 30
```

Commentary. The seed value `1` is the *identity* for multiplication — multiplying by 1 doesn't change the result. For `+` the identity is `0`, for `*` it's `1`, for string concatenation it's `""`. Knowing this for any "fold" operation is one of the genuinely *mathematical* habits you pick up from functional programming.

`product([]) == 1` may feel surprising — there are no factors! But it has to be: any `product(extra :: rest)` should equal `extra * product(rest)`, which only works if `product([])` is the identity. This is

the same logic as "0! = 1" in mathematics.

Exercise 10.4 — sum_of_squares (HOF style)

```
#language axioma/beginner
sum_of_squares: func(arr) [
  reduce(func(a, b) [a + b], 0,
         map(func(x) [x * x], arr))
]
println(sum_of_squares([]))         # 0
println(sum_of_squares([1, 2, 3]))  # 14
```

Commentary. Compare with Exercise 7.3's recursive version (`if len(arr) == 0 then 0 else first(arr) * first(arr) + ...`). Same answer, no `if`, no `first`, no recursion in your code. Two operations: `map` to transform each element, `reduce` to combine them.

You could fuse the two into one `reduce` (`reduce(func(a, x) [a + x * x], 0, arr)`), avoiding the intermediate squared array. The fused version is more efficient for huge arrays; the separated version is easier to read and to *modify* later ("square and *also* round to nearest int" is one extra `map`). Pick clarity first, optimize when measured.

Exercise 10.5 — all_of

```
#language axioma/beginner
all_of: func(pred, arr) [
  if len(arr) == 0 then true
  else if pred(first(arr)) then all_of(pred, rest(arr))
  else false
]
println(all_of(func(x) [x > 0], []))         # true
println(all_of(func(x) [x > 0], [1, 2, 3]))  # true
println(all_of(func(x) [x > 0], [1, -1, 3])) # false
```

Commentary. This is the **dual** of `any_of` from §10.7: the base case flips (`true` instead of `false`), and so do the two branches of the conditional. Read the two side by side — the symmetry is striking.

The vacuous-truth answer for `all_of(_, [])` matches the mathematical universal quantifier: "for all x in the empty set, P(x)" is true because there are zero violations. This is the same logic that makes `0! = 1` and `product([]) = 1` (Ex.10.3). Identity elements, vacuous quantification, and recursive base cases are all the same idea wearing different hats.

Exercise 10.6 (open) — count_if, two ways

```
#language axioma/beginner
# Style A — direct recursion (mirror count_above from Ex.7.4)
count_if_rec: func(pred, arr) [
  if len(arr) == 0 then 0
  else if pred(first(arr)) then 1 + count_if_rec(pred, rest(arr))
  else count_if_rec(pred, rest(arr))
]

# Style B — one-liner using filter and len
count_if_hof: func(pred, arr) [ len(filter(pred, arr)) ]

println(count_if_rec(func(x) [x > 10], []))             # 0
println(count_if_rec(func(x) [x > 10], [3, 12, 8, 99])) # 2
println(count_if_hof(func(x) [x > 10], []))             # 0
println(count_if_hof(func(x) [x > 10], [3, 12, 8, 99])) # 2
```

Discussion.

- **Readability.** Style B is *much* shorter and reads almost like English: "count the items that pass the filter." Style A spells out the algorithm; Style B describes the result.
- **Memory.** Style B builds an intermediate array (`filter` returns the full filtered list), then discards it. Style A walks the input once and accumulates a number. For a billion-element array, Style A is more efficient.
- **Changeability.** Suppose the spec changes: "count items that pass, *until* you see a none element, then stop." Style A is trivial to modify (add a third branch to the `if`). Style B requires reshuffling — `filter` doesn't know how to stop early.

Neither style is universally better. HtDP's habit is to *first* write the version that's easiest to read, *then* optimize if a measurement says you have to. Style B first; switch to Style A only when you actually need to.

Exercise 10.13 - broadcast calibration

```
#language axioma/beginner
calibrate: func(reading, gain, offset) [reading * gain + offset]
expect("one reading", calibrate(2, 3, 1), 7)
expect("scalar options", calibrate.([0, 1, 2], 3, 1), [1, 4, 7])
expect("per-reading gains", calibrate.([0, 1, 2], [2, 3, 4], 1), [1, 4, 9])
expect("singleton gain", calibrate.([0, 1, 2], [3], 1), [1, 4, 7])
expect("empty readings", calibrate.([], 3, 1), [])
```

Commentary. The scalar function needs no loop and knows nothing about arrays. Broadcasting chooses the arguments for each position. Lengths three and two are incompatible; neither axis is a singleton. Shape validation fails before scalar callbacks, so no readings are calibrated in that invalid call.

Exercise 10.14 - fusion and a reduction boundary

```
#language axioma/beginner
square: func(x) [x * x]
increment: func(x) [x + 1]
nums: [1, 2, 3]
fused: increment.(square.(nums))
squares: square.(nums)
staged: increment.(squares)
expect("connected dots", fused, [2, 5, 10])
expect("separate stages", staged, [2, 5, 10])
expect("then total", reduce(func(a, b) [a + b], 0, fused), 17)
expect("empty fused walk", increment.(square.([])), [])
```

Commentary. `fused` creates only the final array. The other version also creates `squares`. The reduction consumes the finished array; it is an ordinary call, so it does not join the broadcast loop. Compare with Exercise 10.4: explicitly doing the whole calculation inside one reduction is another design, with its own scalar combining function.

Exercise 10.15 (open) - observable order

The fused form prints `inside 1`, `outside 11`, `inside 2`, `outside 12`. The ordinary `broadcast` call and the pipe form both print `inside 1`, `inside 2`, `outside 11`, `outside 12`. All three return `[22, 24]`.

Discussion. Choose an ordinary call boundary or an explicit intermediate binding when stage order matters. For a singleton computation that must run once, bind its result before including it in a larger broadcast. Leaving that dotted computation inside the fused expression lets it run at each final output position. The decision is about effects and intended reuse, not just the numerical answer.

Exercise 10.16 - symbolic function values

```
smaller(x, y) = if x < y then x else y
operator (&&&) = smaller
expect("infix minimum", 8 &&& 3, 3)
expect("minimum reduction", reduce((&&&), 100, [8, 3, 9]), 3)
```

Both spellings call `smaller`. Its ordinary eager parameters evaluate their arguments before running the body, so an error in an argument remains an error. Subtraction grouped left yields `13`; grouped right yields `17`. A non-associative declaration requires `(20 ⊛ 5) ⊛ 2` or `20 ⊛ (5 ⊛ 2)`. Its metadata never changes the target's body or parameter evaluation rules.

Exercise 10.18 — independent type parameters

```
keep_right: func[A, B](left :: A, right :: B) :: B [right]
let after_integer = keep_right(10)
println(after_integer("blue"))
println(after_integer(20))
```

The output is `blue` and `20`. The partial application fixes A to Integer, while each later call chooses B independently. With one shared T for both arguments and the result, the initial Integer would constrain the later input too, and the String call would fail. The return contract preserves the type chosen for B, not the actual value passed to the function.

Exercise 10.19 — demand and capture

The first program captures the original immutable seed cell; a new `let seed` shadows it. Mutation of that original cell is different:

```
var seed = 10
declare answer :: Float = seed + 1
seed = 20
println(answer)                    # 21.0
```

If the first read precedes the mutation, it caches 11.0 and both reads return 11.0. A typed hole waits for an explicit write; it has no expression to force.

```
var seed = 10
declare answer :: Float = seed + 1
println(answer)                    # 11.0
seed = 20
println(answer)                    # 11.0
```

Exercise 10.20 — typed concatenation and sections

Two empty Lists leave T unresolved and return an empty List. Integer and String inputs disagree. An empty first List followed by a String List chooses String. Concatenation consumes another List; appending one Array element nests it:

```
join_lists[T] :: List of T -> List of T -> List of T
join_lists(xs, ys) = xs + ys
operator (++) = join_lists with
    precedence: sum
    associativity: right
    doc: "Concatenate two Lists with the same element type."
end
```

```
println(join_lists(`[], `[]))                 # `[]
println(error?(try(join_lists(`[1], `["s"])))) # true
println(join_lists(`[], `["s"]))              # `["s"]
let concatenate_three = (++ `[3])
println(concatenate_three(`[1, 2]))           # `[1, 2, 3]
append_array(xs, item) = xs :+ item
operator (⊛) = append_array
let append_three_array = (⊛ [3])
println(append_three_array(`[1, 2]))          # `[1, 2, [3]]
```

The doc slot describes the shared element type. Removing it makes doc("++") fall back to the target function documentation; this example has no function documentation, so its card reports that none is attached. The call prints the card and returns none.

Exercise 10.21 — which operand is missing?

The first pair subtracts from different operands; the second pair changes which operand is the base. A right String section appends the fixed suffix:

```
println((- 2)(10))                    # 8
println((2 -)(10))                    # -8
println((^ 2)(3))                     # 9
println((2 ^)(3))                     # 8
println((+1))                        # 1
println(type((+ 1)))                  # Function
let emphasize = (+ "!")
println(map(emphasize, ["yes", "go"])) # ["yes!", "go!"]
println(("!" +)("yes"))               # !yes
```

The gap after the leading plus selects a section. Without that gap, (+1) is unary plus. The missing left operand in (+ "!") receives each String; putting the fixed mark on the left instead prepends it.

Exercise 10.22 — composing two relationships

```
compose_value[A, B, C] :: (B -> C), (A -> B), A -> C
compose_value(f, g, x) = f(g(x))
println(compose_value(str, n => n + 1, 2))
println(type(compose_value(str, n => n + 1, 2)))
```

The output is 3 and String. A and B are Integer in this call; C is String. A shared T would demand that the final String preserve the Integer input type, so the call would fail its declared relationship. The signature constrains types; it does not prove any algebraic law about the two functions.

Solution 10.23 — preserving a checked mutation

```
add_callback[T] :: Array of (T -> T), T -> T
add_callback(fs, x) = [append(fs, n => n); fs[len(fs)](x)]
callbacks: []
println(add_callback(callbacks, 9))
println(len(callbacks))
println(callbacks[1]("free"))
```

The output is `9`, `1`, and `free`. The call fixes T to Integer, so the inserted callback must accept and return Integer while called in that activation. A String result is rejected. Cleanup preserves the appended slot and removes its temporary stored view; the unannotated caller can then use the underlying identity callback with a String. A callback extracted as a checked value during the call retains its own contract.

Exercise 10.24 — a missing value without a type choice

```
keep_optional[T] :: Array of (T | None), T -> T
keep_optional(values, fallback) = fallback
println(keep_optional([none, 1], 2))
residual: keep_optional([none])
println(residual("ready"))
println(residual(3))
```

The output is `2`, `ready`, and `3`, in both backends. The concrete None arm accepts `none` without choosing T. The partial's independent calls can then choose String and Integer. A partial built with `[1]` has already chosen Integer, so passing a String later fails. The Array's identity stays shared; mutating it can supply new evidence when a partial is invoked again.

Solution 10.25 — a spread snapshots the outer collection

```
items: [1]
extend_items: func() [push(items, 2); 0]
pack: func(...values) [values]
println(pack(...items, extend_items(), ...items))
println(items)
```

The output is `[1, 0, 1, 2]` followed by `[1, 2]`. The first spread copies the one outer element before `extend_items` runs. That call appends 2 and supplies 0 as its own argument. The second spread sees both elements. The function's variadic binding collects all four supplied values. Shared nested objects would retain identity; this is not a deep copy. This example uses the evaluator.

Solution 10.26 — an alias and a data boundary

```
type Endo[T] = T -> T
type StringEndo = Endo of String
emphasize :: StringEndo: s => s + "!"
println(emphasize("ready"))
type Batch = {count :: Integer}
batch: checked({count: 2, label: "morning"}, Batch)
println(batch.label)
println(type(checked(3, Float)))
```

The output is `ready!`, `morning`, and `Integer`. Transparent substitution preserves the function contract. Validation accepts extra fields and returns the original object without numeric conversion or schema stamping. It checks a snapshot; use `::` when future writes must obey a schema.

Solution 10.27 — named spread snapshots and collisions

```
options: {"a": 1}
change: func() [options["a"] = 9; 2]
pair: func(a, b) [(a, b)]
println(pair(...: options, b: change()))
println(options["a"])
println(try(pair(...: options, a: 3, b: 2)) is Error)
```

The output is `(1, 2)`, `9`, and `true`. The named spread snapshots $a = 1$ before change runs. The final call supplies a twice and errors before pair's body runs. Reversing the explicit label and spread still errors. This example uses the evaluator.

Solution 10.28 — a promise per input

```
double_up :: (Integer -> Integer) & (String -> String)
double_up(x) = if x is Integer then x * 2 else x + x
println(double_up(21))
println(double_up("ab"))
println(try(double_up(1.5)) is Error)
```

The output is `42`, `abab`, and `true`. `1.5` satisfies neither arm's parameter, so the call is refused before the body runs; the caller is at fault. The union signature promises only that some Integer or String comes back for some Integer or String argument. `str(3)` is a String, so that promise holds for every call, and no call could detect the lost connection between input and output. The intersection keeps one promise per arm, so a body that returns `str(x)` fails on its first Integer call. This example uses the evaluator, because `--vm` does not compile `try`.

Chapter 11 · Solutions

Exercise 11.1 — rewrites of 10.1, 10.2, 10.3 with lambda

```
# Lambda needs the unrestricted mode — no `#language axioma/beginner`.
triple_each: func(arr) [ map(lambda x => x * 3, arr) ]
keep_positive: func(arr) [ filter(lambda x => x > 0, arr) ]
product: func(arr) [ reduce(lambda (a, b) => a * b, 1, arr) ]

println(triple_each([1, 2, 3]))    # [3, 6, 9]
println(keep_positive([1, -2, 3])) # [1, 3]
println(product([2, 3, 5]))        # 30
```

Commentary. Side-by-side comparison with the Ch.10 solutions: only the inline function form has changed. The outer `func(arr) [...]` wrappers are still there because they take a *named* parameter (`arr`) and define a *reusable* function. That's the pattern §11.4 calls out: lambda for the inline one-off, `func` for what gets a name.

Exercise 11.2 — negate_all

```
negate_all: func(arr) [ map(lambda x => 0 - x, arr) ]
println(negate_all([]))            # []
println(negate_all([1, -2, 3]))    # [-1, 2, -3]
```

Commentary. `0 - x` flips the sign. A unary minus operator `-x` also works in Axioma (`lambda x => -x`) but the `0 - x` form is unambiguous if you ever mis-read.

Exercise 11.3 — split_by (one-pass partition)

```
split_by: func(pred, arr) [
  if len(arr) == 0 then ([], [])
  else [
    r: split_by(pred, rest(arr));
    yes_arr: r[1];
    no_arr: r[2];
    h: first(arr);
    if pred(h) then (push([], h) + yes_arr, no_arr)
    else (yes_arr, push([], h) + no_arr)
```

```
  ]
]
println(split_by(lambda x => x > 0, []))             # ([], [])
println(split_by(lambda x => x > 0, [1, -2, 3, -4])) # ([1, 3], [-2, -4])
```

Commentary. The pair is built once at the recursive call, unpacked into `yes_arr` and `no_arr`, and then *one* of them gets `h` prepended depending on `pred(h)`. This is the first exercise where the recursive *value* isn't a list — it's a tuple. The shape of the recursion still mirrors the data ("empty or first-and-rest"); only the return type changes.

Why is this single-pass important? The two-`filter` version walks the array twice — once for the yes side, once for the no side. For a large array that's twice the work. For a *stream* (Chapter 17 preview) that you can only walk once, two-`filter` isn't even possible.

(Reminder on the name: `partition` is a reserved Axioma keyword — see §11.3's `insert` / `insert_sorted` story.)

Exercise 11.4 — compose

```
compose: lambda (f, g) => (lambda x => g(f(x)))
println(compose(lambda x => x + 1, lambda x => x * 2)(5))   # 12
```

Commentary. Read the type aloud: `compose` takes two functions, returns a new function. The new function takes one argument, runs `f` on it, then runs `g` on the result. Read in math notation: `compose(f, g) = g ∘ f`.

(Mathematicians sometimes define `compose(f, g) = f ∘ g` instead — apply `g` first then `f`. There's no winning here; pick a convention and document it. The version above matches Python's `pipe` style and the Unix `|` pipeline.)

Exercise 11.5 — sort_descending

```
insert_sorted: func(less, x, sorted) [
  if len(sorted) == 0 then push([], x)
  else if less(x, first(sorted)) then push([], x) + sorted
  else push([], first(sorted)) + insert_sorted(less, x, rest(sorted))
]
sort_by: func(less, arr) [
  if len(arr) == 0 then []
  else insert_sorted(less, first(arr), sort_by(less, rest(arr)))
]
sort_descending: lambda arr => sort_by(lambda (a, b) => a > b, arr)
println(sort_descending([3, 1, 4, 1, 5]))   # [5, 4, 3, 1, 1]
```

Commentary. The outer lambda *partially applies* `sort_by`: it pre-fills the `less` argument with `lambd`

`a (a, b) => a > b` and leaves `arr` to be supplied at call time. This is called **partial application** or **currying**, and it's the entire trick: build small specific functions out of bigger general ones, by fixing some of the arguments.

Languages with built-in currying (Haskell, OCaml) would let you write `sort_descending: sort_by(lambda (a, b) => a > b)` — no outer lambda needed. Axioma's call syntax doesn't curry automatically; you wrap the partial application in a lambda yourself. That's *almost* as good and considerably more explicit.

Exercise 11.6 (open) — pipe

```
pipe: func(x, fs) [
  reduce(lambda (acc, f) => f(acc), x, fs)
]
println(pipe(5, [lambda x => x + 1,
                 lambda x => x * 2,
                 lambda x => x - 3]))           # 9

# A chained example: take a string, normalize it, count words.
normalize_and_count:   lambda s => pipe(s, [lambda s => trim(s),
                       lambda s => lower(s),
                       lambda s => split(s, " "),
                       lambda parts => len(filter(lambda w => len(w) > 0, parts))])
println(normalize_and_count("   Hello   World  "))    # 2
```

Discussion.

The key insight is that `reduce` is the right tool because the job is "fold an array of functions over a starting value." The accumulator (`acc`) is the running value being transformed; the reducer combines `acc` with the next function by calling it. The pipeline `pipe(x, [f, g, h])` evaluates as `h(g(f(x)))`.

Pipeline style reads **left-to-right** in the order the transformations apply. Nested-call style (`h(g(f(x)))`) reads **right-to-left**, which is the reverse of the order things happen. Both are correct; pipeline often documents intent better, especially when the chain has four or five steps.

This is why Elixir's `|>` and F#'s `|>` operators exist as *language syntax* — they're so useful that the languages elevate them above being just a function. Axioma has the operator too (`[3, 1, 2] |> sort |> len` → `3`); what you've just built is the same idea in function space.

Chapter 12 · Solutions

Exercise 12.1 — bmi_category

```
#language axioma/beginner
bmi_category: func(weight_kg, height_m) [
  h_squared: height_m * height_m;
  b: weight_kg / h_squared;
  if b < 18.5 then "underweight"
  else if b < 25 then "normal"
  else if b < 30 then "overweight"
  else "obese"
]
println(bmi_category(50, 1.75))    # underweight
println(bmi_category(70, 1.75))    # normal
println(bmi_category(85, 1.75))    # overweight
println(bmi_category(100, 1.75))   # obese
```

Commentary. Two locals: `h_squared` (avoids recomputing `height_m * height_m`) and `b` (gives the BMI a meaningful name we can refer to four times in the cascade). Neither escapes the function. The shape "compute some intermediate quantity, then dispatch on it" is one of the most common in all of programming.

(Quiz: why bother with `h_squared` when `b: weight_kg / (height_m * height_m)` works inline? Because the local documents the intermediate quantity — and computes it once.)

Exercise 12.2 — triangle_area (Heron's formula)

```
#language axioma/beginner
triangle_area: func(a, b, c) [
  s: (a + b + c) / 2.0;
  sqrt(s * (s - a) * (s - b) * (s - c))
]
println(triangle_area(3, 4, 5))    # 6.0
println(triangle_area(5, 5, 6))    # 12.0
```

Commentary. `s` is computed once and used four times. Without the local, you'd write `(a + b + c) / 2.0` four times — same result, four times the typing, four chances to make a typo.

The `2.0` is a style choice, not a correctness fix: `(3 + 4 + 5) / 2` is exactly `6`, and for an odd perimeter like `2, 3, 4` the semi-perimeter is the exact rational `9/2` — which `sqrt` accepts happily (the area comes

out `2.9047…` either way). The float spelling just keeps the intermediates in decimal-land, which suits a geometry measurement. (Chapter 1's `7 / 2` story: division is exact; floats are a choice you opt into.)

Exercise 12.3 — running_average

```
#language axioma/beginner
running_average: func(arr) [
  n: len(arr);
  take_i: func(k, src) [
    if k == 0 then []
    else push([], first(src)) + take_i(k - 1, rest(src))
  ];
  avg_first_i: func(i) [
    chunk: take_i(i, arr);
    reduce(func(a, b) [a + b], 0.0, chunk) / i
  ];
  make_all: func(j) [
    if j > n then []
    else push([], avg_first_i(j)) + make_all(j + 1)
  ];
  if n == 0 then [] else make_all(1)
]
println(running_average([]))          # []
println(running_average([4]))         # [4.0]
println(running_average([2, 4, 6]))   # [2.0, 3.0, 4.0]
```

Commentary. Three local helpers — `take_i` (prefix of length `k`), `avg_first_i` (average of first `i` elements), `make_all` (build the result by walking `j` from 1 to `n`). None of them leak. The function exports exactly one name (`running_average`) and hides three.

This is the densest example so far of the "private workshop" metaphor: a function that knows three helper concepts but only the result function appears in the namespace. A real production routine for, say, sliding-window statistics on a sensor stream might hide *ten* helpers like this.

The `0.0` accumulator seed again forces float math; the running averages come out as decimals like `2.5` rather than exact fractions like `5/2`.

Exercise 12.4 — pipeline refactor

```
#language axioma/beginner
mean_of_big_evens: func(nums) [
  above_10: filter(func(x) [x > 10], nums);
  evens: filter(func(x) [x % 2 == 0], above_10);
  total: reduce(func(a, b) [a + b], 0, evens);
  total / len(evens)
]
```

```
println(mean_of_big_evens([3, 11, 14, 17, 22, 8, 25, 30]))   # 22
```

Commentary. The exact same code as §10.6 — but the top level now exports one name (`mean_of_big_evens`) instead of four (`above_10`, `evens`, `total`, `answer`). The intermediate variables existed for *clarity inside the computation*, not for use by anyone else. Locals are exactly the right place for them.

When you load this file into a REPL, you can call `mean_of_big_evens([5, 12, 14, 22])` immediately. You can *not* ask for `above_10` — it doesn't exist outside the function call. That's the *encapsulation* §12.8 named.

Exercise 12.5 — count_words_per_line

```
#language axioma/beginner
count_words_per_line: func(lines) [
  word_count: func(s) [
    raw: split(s, " ");
    non_empty: filter(func(w) [len(w) > 0], raw);
    len(non_empty)
  ];
  map(word_count, lines)
]
println(count_words_per_line([]))                                  # []
println(count_words_per_line(["hello world", "the", "a b c d"])) # [2, 1, 4]
println(count_words_per_line(["a  b c"]))                          # [3]
```

Commentary. `word_count` is a *private* helper that handles the "multiple-space" corner case (`split("a b", " ")` returns `["a", "", "b"]` — the empty string between the two spaces). The `filter(... len(w) > 0)` drops those empties.

`map(word_count, lines)` is the outer shape: "do the word-count thing to each line, return the array of counts." The whole function is six lines. Without the local helper, the outer would have to inline the split/filter/len logic inside a lambda — readable, but harder to test (you can't unit-test `word_count` in isolation when it has no name).

Exercise 12.6 (open) — make_counter

```
# Captured state: rebind updates the enclosing counter cell.
make_counter: func() [
  n: 0;
  func() [
    rebind n = n + 1;
    n
  ]
]
c: make_counter()
```

```
v1: c()
v2: c()
v3: c()
println(v1)    # 1
println(v2)    # 2
println(v3)    # 3
d: make_counter()
w1: d()
v4: c()
println(w1)    # 1   (d has its own n)
println(v4)    # 4   (c's count continued)
```

Answers to the questions.

1. **Is your counter *pure*?** No. A function is "pure" if it returns the same answer for the same inputs and has no visible side effects. `c()` is called with no arguments each time and returns a different answer (1, then 2, then 3). It also mutates internal state. So `c` is impure — and that's the point of this exercise.
2. **What about the beginner discipline?** The line `rebind n = n + 1` is exactly what the "one name, one value" habit forbids — and the counter cannot work without it: remembering how many times you've been called *is* a change of state. Note that the `#language axioma/beginner` pragma still runs this program — the discipline is a habit the book teaches, not a parser restriction — but conceptually this exercise is where you step across the line between "values are fixed once" (functional) and "values change over time" (imperative). Chapter 16 crosses it deliberately.

   You've just crossed that line. Chapter 16 (Mutation) gives it a proper introduction, but you've already met it here in its sneakiest form — *closure of a mutable local*. Functions that carry mutable state are how you build counters, caches, state machines, and (in larger form) objects with private fields.

Reflection. Lambdas + closures + mutable locals add up to *objects* in everything-but-name. A counter is an "object" with one method (`c()`) and one private field (`n`). Smalltalk and Self based their entire object systems on exactly this substrate; JavaScript still implements closures as objects. The boundary between "functional programming" and "object-oriented programming" is much fuzzier than the camps let on.

Chapter 13 · Solutions

Common preamble for the friendship exercises:

```
#language axioma/beginner
concept Person
Person has name: ""
Person has age: 0

concept Friendship
Friendship has source: none
Friendship has target: none
Friendship has years: 0
Friendship has closeness: 0

alice: a Person {name: "Alice", age: 30}
bob:   a Person {name: "Bob",   age: 25}
carol: a Person {name: "Carol", age: 35}
dave:  a Person {name: "Dave",  age: 28}

f1: a Friendship {source: alice, target: bob,   years: 5, closeness: 4}
f2: a Friendship {source: alice, target: carol, years: 2, closeness: 3}
f3: a Friendship {source: bob,   target: carol, years: 3, closeness: 5}
f4: a Friendship {source: carol, target: dave,  years: 1, closeness: 2}
friendships: [f1, f2, f3, f4]

friendships_of: func(p) [
  filter(func(f) [f.source == p or f.target == p], friendships)
]
other_party: func(p, f) [
  if f.source == p then f.target else f.source
]
direct_friends: func(p) [
  map(func(f) [other_party(p, f).name], friendships_of(p))
]
```

Common preamble for the road-map exercises:

```
concept City
City has name: ""
concept Road
Road has src: none
Road has dst: none
```

```
Road has miles: 0

nyc: a City {name: "NYC"}
phl: a City {name: "Philadelphia"}
dc:  a City {name: "DC"}
bos: a City {name: "Boston"}

roads: [
  a Road {src: nyc, dst: phl, miles: 95},
  a Road {src: phl, dst: dc,  miles: 140},
  a Road {src: nyc, dst: bos, miles: 215},
  a Road {src: dc,  dst: nyc, miles: 225}
]
```

Common preamble for the part/assembly exercises:

```
concept BasicPart
BasicPart has name: ""
BasicPart has weight: 0

concept Assembly
Assembly has name: ""
Assembly has parts: []

wheel: an Assembly {
  name: "wheel",
  parts: [
    a BasicPart {name: "rim",    weight: 500},
    a BasicPart {name: "tire",   weight: 700},
    a BasicPart {name: "spokes", weight: 200}
  ]
}
frame: a BasicPart {name: "frame", weight: 3000}
bike: an Assembly {
  name: "bicycle",
  parts: [frame, wheel, wheel]
}
```

Exercise 13.1 — mutual_friends

```
in_arr: func(arr, x) [
  if len(arr) == 0 then false
  else if first(arr) == x then true
  else in_arr(rest(arr), x)
]
mutual_friends: func(p, q) [
  fp: direct_friends(p);
```

```
  fq: direct_friends(q);
  filter(func(n) [in_arr(fq, n)], fp)
]
println(mutual_friends(alice, bob))    # ["Carol"]
println(mutual_friends(alice, dave))   # ["Carol"]
println(mutual_friends(carol, alice))  # ["Bob"]
```

Commentary. Two stages: get `direct_friends(p)` and `direct_friends(q)`, then keep the names that appear in both. The set-intersection logic is hand-rolled via `in_arr` because Axioma's `intersection` operator works on the Set type (`{...}`), not on arrays of names — and the comprehension we already use to build `direct_friends` returns an array (so the names stay ordered, which is the more useful shape for downstream code).

`in_arr` is a one-liner you'll reach for again — many of the remaining graph algorithms come down to "is this element in the visited list?" or similar membership tests. (Ch.20's MVL chapter introduces Belnap's four-valued logic, which gives a more nuanced version of "is X in Y": *both*, *neither*, *yes*, *no*. For now, plain Boolean does the job.)

> **Why not `contains`?** Because `contains` is reserved in Axioma (`keywords()` lists every reserved word). `in_arr` is unambiguous; `member` would also work.

Exercise 13.2 — friendship_years_total

```
friendship_years_total: func(p) [
  fs: friendships_of(p);
  reduce(func(acc, f) [acc + f.years], 0, fs)
]
println(friendship_years_total(alice))   # 7
println(friendship_years_total(carol))   # 6
println(friendship_years_total(dave))    # 1
```

Commentary. Two-step: `friendships_of` (filter) returns the array of friendships involving `p`; `reduce` walks that array adding up the `years` fields. The `0` seed is the additive identity. Notice we don't have to handle the empty case specially — `reduce` returns the seed when the array is empty.

This is a *very* common shape: "fetch the related records, then aggregate one field." Database SQL spells it `SELECT SUM(years) FROM friendships WHERE source = ? OR target = ?`; we just spelled it more loudly.

Exercise 13.3 — total_route_miles

```
leg_miles: func(a, b) [
  candidates: filter(func(r) [r.src == a and r.dst == b], roads);
  if len(candidates) == 0 then -1
  else first(candidates).miles
```

```
]
total_route_miles: func(route) [
  n: len(route);
  if n <= 1 then 0
  else [
    leg: leg_miles(first(route), first(rest(route)));
    if leg == -1 then -1
    else [
      rest_total: total_route_miles(rest(route));
      if rest_total == -1 then -1
      else leg + rest_total
    ]
  ]
]
println(total_route_miles([nyc, phl]))           # 95
println(total_route_miles([nyc, phl, dc]))       # 235
println(total_route_miles([nyc, phl, dc, nyc]))  # 460
println(total_route_miles([bos, nyc]))           # -1
```

Commentary. The bind `n: len(route)` names the length once — the inline `if len(route) <= 1 then 0 else [...]` works too, but the local documents the intent and reads better in the comparisons that follow.

The propagating-`-1` pattern is what HtDP would call a *tagged-failure* return — borrowed from C-style "return -1 on error." Idiomatic Axioma would return `none` or an `om` (SETL-unknown) instead; we use `-1` because it composes with arithmetic without needing an `Option`-style monad. The chapter intentionally hasn't introduced sum types or option types yet — that's Ch.18's territory.

The four assertions exercise the three exits: success (95, 235, 460) and propagated failure (-1). For a real route-planner, you'd also want to surface *which* leg failed; one more local would do it.

Exercise 13.4 — count_parts

```
count_parts: func(p) [
  if p is BasicPart then 1
  else if p is Assembly then count_parts_arr(p.parts)
  else 0
]
count_parts_arr: func(arr) [
  if len(arr) == 0 then 0
  else count_parts(first(arr)) + count_parts_arr(rest(arr))
]
println(count_parts(frame))   # 1
println(count_parts(wheel))   # 3
println(count_parts(bike))    # 7
```

Commentary. Textbook mutual recursion: `count_parts` is the "variant dispatcher" (one branch per Part flavor); `count_parts_arr` is the "list walker" (recurse on the tail). They call each other across the

Part / Array boundary.

A BasicPart contributes 1. An Assembly contributes the sum over its parts (which may themselves be Assemblies, recursively). The arithmetic is `1 + 3 + 3 = 7` for the bike (one frame, three parts per wheel, two wheels).

This is the *count-leaves-in-a-tree* pattern (Ch.8 Ex.8.5 was the binary-tree version). The shape generalizes: replace `len 1` nodes with arbitrary-arity "internal" nodes and you get arbitrary trees. With reified edges (Ch.13 §13.2) you get arbitrary graphs.

Exercise 13.5 — flatten_parts

```
flatten_parts: func(p) [
  if p is BasicPart then push([], p)
  else if p is Assembly then flatten_arr(p.parts)
  else []
]
flatten_arr: func(arr) [
  if len(arr) == 0 then []
  else flatten_parts(first(arr)) + flatten_arr(rest(arr))
]
println(len(flatten_parts(frame)))   # 1
println(len(flatten_parts(wheel)))   # 3
println(len(flatten_parts(bike)))    # 7

sum_weights_of: func(arr) [
  if len(arr) == 0 then 0
  else first(arr).weight + sum_weights_of(rest(arr))
]
println(sum_weights_of(flatten_parts(bike)))   # 5800
```

Commentary. Same shape as `count_parts`, but the per-leaf contribution is a one-element array of *the part itself* and the combiner is array concatenation (+) instead of numeric addition. (`push([], p)` builds the singleton; the trailing-comma spelling `[p,]` from §6.6 does the same job.)

The verification `sum_weights_of(flatten_parts(bike)) == 5800` re-derives `weight_of(bike)` from §13.4 a different way. That's not redundant — it's *triangulation*. Two independent code paths should agree on the same answer; if they don't, you've found a bug. Real production code uses this technique constantly (a *round-trip property test*: encode then decode and check the result equals the input).

Exercise 13.6 (open) — route_exists

```
outgoing: func(c) [
  filter(func(r) [r.src == c], roads)
]
neighbors: func(c) [
  map(func(r) [r.dst], outgoing(c))
]
reach_via: func(s, d, fuel) [
  if s == d then true
  else if fuel == 0 then false
  else [
    nbrs: neighbors(s);
    any_reach(nbrs, d, fuel - 1)
  ]
]
any_reach: func(arr, d, fuel) [
  if len(arr) == 0 then false
  else if reach_via(first(arr), d, fuel) then true
  else any_reach(rest(arr), d, fuel)
]
route_exists: func(s, d) [
  reach_via(s, d, len(roads) + 1)
]
println(route_exists(nyc, dc))   # true
println(route_exists(bos, dc))   # false
println(route_exists(dc, bos))   # true
```

Commentary. This is the §13.3 algorithm packaged up. The fuel budget is `len(roads) + 1` — any simple path through the graph uses at most `len(roads)` edges, plus one for headroom. For a *much* larger graph this would be inefficient (we re-traverse cycles repeatedly within the fuel budget); a real implementation uses an explicit *visited set* (Ch.15 territory: accumulators). For a four-node graph the fuel approach is fine and is genuinely simpler to reason about.

Answers to the chapter questions.

1. **What happens without the fuel parameter?** A cycle like `dc → nyc → bos` and back to `nyc` would loop forever. The reach_via call would dispatch to a neighbor, which dispatches back, infinitely.
2. **What happens with fuel too small?** False negatives. A genuine route requiring more hops than `fuel` would report `false` when it should be `true`. The correctness of `route_exists` depends on choosing fuel ≥ the diameter of the graph.

Chapter 14 develops this discipline formally. Generative recursion *requires* a termination argument — the fuel parameter is one concrete realization of "argument that the search must end." The argument here: every recursive call decrements `fuel`, which is a non-negative integer, which must reach zero. Recursion bounded ⇒ no infinite loop ⇒ algorithm terminates.

How to use these solutions

Try first; compare second. The four-quadrant view from §13.5 is the map you'll re-use whenever a new domain shows up: do edges have data? do cycles exist? Those two questions sort 95% of the modeling decisions you'll make. The recipe — find the cross- references, write one function per concept, recurse along the cross-references, recurse along the edge arrays — is what's left.

Chapter 14 · Solutions

Exercise 14.1 — power (fast exponentiation)

```
#language axioma/beginner
power: func(b, n) [
  if n == 0 then 1
  else if n % 2 == 0 then [
    bb: b * b;
    half: n / 2;
    power(bb, half)
  ]
  else b * power(b, n - 1)
]
println(power(2, 10))   # 1024
println(power(3, 0))    # 1
println(power(2, 16))   # 65536
println(power(5, 3))    # 125
```

Commentary. Three cases. The trick is recognizing the algebraic identity `b^(2k) = (b * b)^k`: doubling the base and halving the exponent leaves the answer unchanged. Combined with the "if odd, peel one off and recurse" case, this gives `O(log n)` multiplications instead of the naive `O(n)`.

Termination argument (the answer to the chapter question): the exponent `n` strictly decreases on every recursive call.

- Even-case: `n → n / 2`, which is strictly less when `n > 0`.
- Odd-case: `n → n - 1`, which is strictly less.

`n` is a non-negative integer, so a strictly decreasing sequence must hit `0` (the base case) in finitely many steps. In fact, no more than `2 * log2(n)` steps — each level either halves or decrements-then-halves.

> **A note on big exponents.** Axioma integers are arbitrary precision, so `power(2, 64)` is the exact `18446744073709551616` and `power(2, 200)` agrees with `2 ^ 200` digit for digit. The function is correct for any non-negative integer exponent.

Exercise 14.2 — nat_log2

```
#language axioma/beginner
nat_log2: func(n) [
  if n <= 1 then 0
  else [
```

```
    half: n div 2;
    1 + nat_log2(half)
  ]
]
println(nat_log2(1))     # 0
println(nat_log2(2))     # 1
println(nat_log2(7))     # 2
println(nat_log2(1024))  # 10
```

Commentary. The simplest generative recursion in the chapter. Halve until you're at `1` (or below), count the halvings.

Termination argument: `n` is a positive integer. Each recursive call replaces `n` with `n div 2` (the integer part — plain `/` would give exact fractions like `7/2` and both the count and the termination bound would go wrong), which is strictly smaller when `n >= 2`. (When `n == 1` we hit the base case without recursing.) So the recursion depth is bounded by the number of times `n` can be halved before reaching `1` — which is, by definition, `log2(n)`.

`nat_log2(1024) = 10` is correct because `2^10 = 1024`. The function works correctly for any `n >= 1`; for `n == 0` it returns `0` (a defensible-but-arbitrary choice — strictly speaking `log2(0)` is undefined).

Exercise 14.3 — hanoi

```
#language axioma/beginner
hanoi: func(n, src, dst, via) [
  if n == 0 then []
  else hanoi(n - 1, src, via, dst)
       + push([], (src, dst))
       + hanoi(n - 1, via, dst, src)
]
println(hanoi(1, "A", "C", "B"))             # [("A", "C")]
println(hanoi(2, "A", "C", "B"))
println(len(hanoi(3, "A", "C", "B")))        # 7
println(len(hanoi(5, "A", "C", "B")))        # 31
```

Commentary. One of the cleanest pieces of recursion in computer science.

The algorithmic insight: to move `n` disks from `src` to `dst`,

- first move the top `n−1` disks out of the way (to `via`),
- then move the now-exposed largest disk (from `src` to `dst`),
- then move the `n−1` disks from `via` onto `dst`.

The recursion writes itself once you've seen the trick. Each call either solves the trivial base case (zero disks → no moves) or delegates to two smaller calls of size `n - 1` each.

Why generative? The subproblem of size `n - 1` is *computed* — `n - 1` isn't a structural piece of any input data, it's a new integer we derived. (Compare: in `sum_list(rest(arr))`, `rest(arr)` is a structural piece of `arr`.) The peg arguments also get *rearranged* — `src/via/dst` becomes `src/dst/via` in the first

call, `via/src/dst` in the second. That permutation is generative too.

Termination argument: `n` is a non-negative integer. Each recursive call passes `n - 1`. Strict decrease, bounded below by `0`, so we hit the base case in exactly `n` levels of nested recursion (with `2^n - 1` total calls at the leaves).

The output array length is `2^n - 1`: `1, 3, 7, 15, 31, ...`. That exponential cost is *fundamental* — any algorithm that solves Hanoi must make at least that many moves, because every disk must be moved at least once, and the discipline of "never put larger on smaller" forces re-shuffling.

Exercise 14.4 — bsearch_count

```
#language axioma/beginner
bsearch_count_in: func(arr, target, lo, hi, cmps) [
  if lo > hi then (0, cmps)
  else [
    mid: (lo + hi) div 2;
    v: arr[mid];
    if v == target then (mid, cmps + 1)
    else if target < v then bsearch_count_in(arr, target, lo, mid - 1, cmps + 1)
    else                    bsearch_count_in(arr, target, mid + 1, hi, cmps + 1)
  ]
]
bsearch_count: func(arr, target) [
  bsearch_count_in(arr, target, 1, len(arr), 0)
]
println(bsearch_count([1, 3, 5, 7, 9], 5))    # (3, 1) — mid hit first try
println(bsearch_count([1, 3, 5, 7, 9], 1))    # (1, 2)
println(bsearch_count([1, 3, 5, 7, 9], 11))   # (0, 3)
println(bsearch_count([1, 3, 5, 7, 9, 11, 13, 15, 17, 19], 19))  # (10, 4)
```

Commentary. The same binary search as §14.5, threaded with an accumulator `cmps` that increments on every comparison. The helper carries the counter; the outer wrapper provides the initial state `cmps = 0`.

This is **accumulator-style recursion** (Chapter 15's main topic) snuck in early. Every recursive call passes the *current* count along; nothing's lost between levels. Without an accumulator, you'd have to fold the count up after the fact — doable, but less direct than just carrying it down.

Why the counts? Algorithm-design tradition: counting comparisons gives you a model-independent way to measure work that doesn't depend on CPU speed or compiler optimizations. For binary search the count is bounded by `log2(n) + 1`; for linear search it's at worst `n`. The difference is dramatic for big arrays — `log2(1_000_000) ≈ 20` vs. `1_000_000`.

Tuples such as `(3, 1)` are values you can index and destructure (Chapter 6 §6.7), not just printable strings.

Exercise 14.5 — merge_three

```
#language axioma/beginner
merge2: func(a, b) [
  if len(a) == 0 then b
  else if len(b) == 0 then a
  else [
    fa: first(a);
    fb: first(b);
    if fa <= fb then push([], fa) + merge2(rest(a), b)
    else push([], fb) + merge2(a, rest(b))
  ]
]
merge_three: func(a, b, c) [
  la: len(a);
  lb: len(b);
  lc: len(c);
  if la == 0 and lb == 0 and lc == 0 then []
  else if la == 0 then merge2(b, c)
  else if lb == 0 then merge2(a, c)
  else if lc == 0 then merge2(a, b)
  else [
    fa: first(a);
    fb: first(b);
    fc: first(c);
    if fa <= fb and fa <= fc then push([], fa) + merge_three(rest(a), b, c)
    else if fb <= fa and fb <= fc then push([], fb) + merge_three(a, rest(b), c)
    else push([], fc) + merge_three(a, b, rest(c))
  ]
]
println(merge_three([1, 4, 7], [2, 5, 8], [3, 6, 9]))
println(merge_three([], [1, 2], [3, 4]))
println(merge_three([], [], []))
println(merge_three([1, 2], [], []))
```

Commentary. Native three-way merge is a case-explosion: four "exit" cases (one of the three arrays is empty, or all three are), plus the three "interior" cases (each branch picks one of the three first-elements based on `min`-of-three).

The local lengths `la, lb, lc` aren't *necessary* — we could inline `len(a)` everywhere — but they cost almost nothing and make the code dramatically more readable.

The alternative: `merge2(a, merge2(b, c))`. Simpler. Two-pass (builds an intermediate sorted array of length `len(b) + len(c)`, then merges that with `a`). Native three-way avoids the intermediate but isn't dramatically faster in practice. Both are linear in `len(a) + len(b) + len(c)`.

Termination argument: the total work is bounded by `len(a) + len(b) + len(c)`, which strictly decreases by 1 at each recursive step (one element is consumed). Reaches zero in at most that many calls.

Exercise 14.6 (open) — is_perfect_square

```
#language axioma/beginner
is_psq_search: func(n, lo, hi) [
  if lo > hi then false
  else [
    mid: (lo + hi) div 2;
    sq: mid * mid;
    if sq == n then true
    else if sq < n then is_psq_search(n, mid + 1, hi)
    else is_psq_search(n, lo, mid - 1)
  ]
]
is_perfect_square: func(n) [
  if n < 0 then false
  else is_psq_search(n, 0, n)
]
println(is_perfect_square(0))     # true
println(is_perfect_square(1))     # true
println(is_perfect_square(16))    # true
println(is_perfect_square(17))    # false
println(is_perfect_square(144))   # true
println(is_perfect_square(143))   # false
```

Commentary. The same binary-search-on-an-integer-range structure as `bsearch` from §14.5, but the *array* is replaced by a *predicate*. At each step we don't compare to a stored value — we *compute* `mid * mid` and compare to `n`.

This is binary search applied to *implicit data* — the sorted sequence `0², 1², 2², 3², …, n²`. We never construct that array; we just evaluate elements on demand. Same termination argument, same `O(log n)` step count.

Answers to the chapter questions.

1. **What's the search range, and why does halving it terminate?** The range is `[lo, hi]`, initially `[0, n]`. Each recursive call shrinks the range by either jumping to `[mid + 1, hi]` (which has size `hi - mid`, strictly less than `hi - lo + 1`) or to `[lo, mid - 1]` (size `mid - lo`, strictly less). The range size is a non-negative integer that strictly decreases, so it must reach `0` (when `lo > hi`).
2. **Why is `sqrt(n) * sqrt(n) == n` unreliable for large `n`?** `sqrt` returns a floating-point number. Floats have finite precision (about 15 significant decimal digits for 64-bit double precision). For `n` large enough — roughly above `2^52` — `sqrt(n) * sqrt(n)` doesn't exactly equal `n` even when `n` is a true perfect square. You'd get spurious `false`s for very large perfect squares. The integer binary search avoids floating-point entirely; it's *exact* by construction.

   This is one of the recurring reasons to prefer integer-only algorithms when correctness matters: floats are a *model* of real numbers, not the same thing. (Axioma's exact rationals dodge this for division — `7 / 2` really is `7/2` — but `sqrt`, trig, and friends still land you in float territory, where the model leaks.)

How to use these solutions

The four-step recipe for generative recursion — *trivial case* + *subproblem* + *combiner* + *termination* — is the take-away of the chapter. When you sit down to design a new algorithm, explicitly write each of the four. If you can't write the termination argument, you don't yet have an algorithm.

Pop quiz for after you've worked all six exercises:

- Which uses *structural* recursion in a helper but *generative* recursion at the top level? (Answer: 14.4 — the binary search is generative; the accumulator passing is structural through the call chain.)
- Which uses *zero* structural recursion? (Answer: 14.1 and 14.2 — pure number-shrinking.)
- Which would benefit most from converting to an accumulator style in Chapter 15? (Answer: 14.4 already has the accumulator; 14.5 and the §14.4 mergesort would benefit, since the recursion produces lots of intermediate arrays.)

Chapter 15 · Solutions

Exercise 15.1 — sum_acc

```
#language axioma/beginner
sum_acc: func(arr, acc) [
  if len(arr) == 0 then acc
  else sum_acc(rest(arr), acc + first(arr))
]
sum: func(arr) [ sum_acc(arr, 0) ]
println(sum([]))                # 0
println(sum([7]))               # 7
println(sum([1, 2, 3, 4, 5]))   # 15
```

Commentary. The textbook conversion of Ch.7's natural `sum_list` into accumulator form. The wrapper `sum` supplies the seed (`0`, the additive identity). The worker `sum_acc` carries the running total in its second parameter.

The recursive call is `sum_acc(rest(arr), acc + first(arr))` — nothing pending after the call. This is *tail-recursive*, and Axioma's self-tail-call optimization runs it in constant stack space at any depth (verified at 100,000 levels — the R5RS-Scheme behavior). The optimization is per-function: mutual tail recursion between two functions still hits the ~50,000-frame ceiling.

The agreement with §15.1's natural `sum_list` on every input is guaranteed because addition is associative *and* commutative, which makes the left-fold `((0 + 1) + 2) + 3` equal to the right-fold `1 + (2 + (3 + 0))`.

Exercise 15.2 — prod_acc

```
#language axioma/beginner
prod_acc: func(arr, acc) [
  if len(arr) == 0 then acc
  else prod_acc(rest(arr), acc * first(arr))
]
prod: func(arr) [ prod_acc(arr, 1) ]
println(prod([]))               # 1
println(prod([7]))              # 7
println(prod([2, 3, 4]))        # 24
println(prod([1, 2, 3, 4, 5]))  # 120
```

Why not seed with `0`? Because `0 * x = 0` for any `x`. A zero seed would make `prod` return `0` for *every*

input. The correct seed is the *identity* element of the operation — `1` for multiplication, `0` for addition, `[]` for concatenation, the universe set for intersection.

This is one of the recurring patterns of accumulator design: **the seed is the operation's identity element**. Every mathematical operation that's associative has a unique identity (or doesn't have one at all, in which case the accumulator needs special handling — see Ex.15.4 of `max_of` and the "seed with first element" trick).

Exercise 15.3 — len_acc

```
#language axioma/beginner
len_acc: func(arr, acc) [
  if len(arr) == 0 then acc
  else len_acc(rest(arr), acc + 1)
]
my_len: func(arr) [ len_acc(arr, 0) ]
println(my_len([]))          # 0
println(my_len([1, 2, 3]))   # 3
```

Commentary. "Count things" doesn't care about the things themselves — only their *number*. The accumulator counts; the head of the array gets dropped each step (we never use `first(arr)`).

This is exactly how Axioma's builtin `len` is implemented for arrays under the hood. Re-deriving it in three lines of beginner code is a small flex: you've just written *one of the most-used list operations in the language* from primitives.

Exercise 15.4 — count_evens_acc

```
#language axioma/beginner
count_evens_acc: func(arr, acc) [
  if len(arr) == 0 then acc
  else [
    bump: if first(arr) % 2 == 0 then 1 else 0;
    count_evens_acc(rest(arr), acc + bump)
  ]
]
count_evens: func(arr) [ count_evens_acc(arr, 0) ]
println(count_evens([]))                  # 0
println(count_evens([1, 3, 5]))           # 0
println(count_evens([2, 4, 6]))           # 3
println(count_evens([1, 2, 3, 4, 5, 6]))  # 3
```

Commentary. Same shape as `len_acc`, but the per-element contribution depends on the element. We compute `bump` (either `0` or `1`) as a local and add it to the accumulator. The local also serves as documentation — it would be perfectly correct (and shorter) to inline the conditional, but readability wins.

Equivalently, you could use `filter` + `len`: `len(filter(func(x) [x % 2 == 0], arr))` — two passes (the filter builds an intermediate array, then `len` walks it). The accumulator version is one pass and never allocates. For small arrays the difference is invisible; for large arrays it matters.

Exercise 15.5 — sum_and_count

```
#language axioma/beginner
sc_acc: func(arr, s, c) [
  if len(arr) == 0 then (s, c)
  else sc_acc(rest(arr), s + first(arr), c + 1)
]
sum_and_count: func(arr) [ sc_acc(arr, 0, 0) ]
println(sum_and_count([]))                    # (0, 0)
println(sum_and_count([10, 20, 30]))          # (60, 3)
println(sum_and_count([1.0, 2.0, 3.0, 4.0])) # (10.0, 4)
```

Commentary. Two accumulators carried in parallel. The base case returns *both* as a tuple. The recursive step updates *both* in the same call.

This pattern generalizes immediately to as many parallel accumulators as you need. The classical "online mean and variance" algorithm carries *three* accumulators (sum, sum-of-squares, count). The Boyer-Moore majority-vote algorithm carries *two* (current candidate, current count). Anything that can be expressed as a single-pass streaming statistic has this shape.

Mean from this is one line: `sum_and_count(arr)[1] / sum_and_count(arr)[2]` (which calls the worker twice — for a purer one-shot version, decompose the tuple in a local).

Exercise 15.6 (open) — running_sums

```
#language axioma/beginner
rs_acc: func(arr, total, out) [
  if len(arr) == 0 then out
  else [
    nt: total + first(arr);
    rs_acc(rest(arr), nt, out + push([], nt))
  ]
]
running_sums: func(arr) [ rs_acc(arr, 0, []) ]
println(running_sums([]))             # []
println(running_sums([7]))            # [7]
println(running_sums([1, 2, 3, 4]))   # [1, 3, 6, 10]
```

Commentary. Three values in flight per step:

- `arr` — the remaining input
- `total` — the running sum so far
- `out` — the output array being built incrementally

Each step computes the new total (`total + first(arr)`), appends it to `out`, and recurses with all three updated.

The output array grows *at the end*, which means we're doing `O(n)` concatenations of growing arrays — total cost `O(n²)`. A more careful implementation prepends to the front and reverses at the end, getting it down to `O(n)`. For a textbook example, clarity wins; the cost optimization is a fine exercise for the reader.

This *exact* pattern shows up everywhere: "carry the partial answer, carry the in-progress output, return the output at base case." It's the imperative-loop pattern made functional — and exactly what Chapter 16's `while` loop will do with mutation instead of recursion.

Chapter 16 · Solutions

Exercise 16.1 — sum_with_loop

```
#language axioma/beginner
sum_with_loop: func(arr) [
  i: 1;
  total: 0;
  n: len(arr);
  while (i <= n) [
    total = total + arr[i];
    i = i + 1
  ];
  total
]
println(sum_with_loop([]))                # 0
println(sum_with_loop([7]))               # 7
println(sum_with_loop([1, 2, 3, 4, 5]))   # 15
```

Commentary. The accumulator-recursion of §15.1 *literally* becomes a `while` loop: `i` plays the role of the index "counter," `total` plays the role of the accumulator, and the loop body advances both. The function-call cost of the recursive version is replaced by raw mutation in place.

The `n: len(arr)` bind names the length up front — `while (i <= len(arr)) [...]` works too, but the local reads better and avoids re-computing `len(arr)` on every iteration (though Axioma's `len` is `O(1)` for arrays, so this is a clarity improvement, not a perf one).

Compare and contrast: this code reads almost identically to the Python:

```
def sum_with_loop(arr):
    i, total, n = 0, 0, len(arr)
    while i < n:
        total += arr[i]
        i += 1
    return total
```

With the obvious differences in indexing (Axioma is 1-based; Python 0-based) and syntax. Mutation + `while` is the *universal* programming idiom; it just took us 15 chapters to get here.

Exercise 16.2 — make_bank_account

```
#language axioma/beginner
make_bank_account: func(start_balance) [
  balance: start_balance;
  func(amount) [
    new_b: balance + amount;
    if new_b < 0 then balance
    else [
      rebind balance = new_b;
      balance
    ]
  ]
]
acc: make_bank_account(100)
v1: acc(50)     # 150
v2: acc(-30)    # 120
v3: acc(-200)   # 120 (refused)
v4: acc(0)      # 120 (query)
```

Commentary. The closure captures `balance`. Each call:

- Compute the would-be new balance (`balance + amount`).
- If it would overdraw, return the *unchanged* balance — no mutation, no side effect.
- Otherwise, *commit*: update `balance` to the new value, return the new value.

This is the canonical "private-field" pattern. `balance` is *invisible* outside the closure — no top-level code can read or write it directly; the only access is through `acc(amount)`. That's encapsulation — the cornerstone of object-oriented design.

A `acc(0)` "query" returns the current balance without changing it (because `balance + 0 == balanc e`, the "commit" branch writes the same value back). For a cleaner query, you'd refactor into a tuple-of-functions like `(deposit, withdraw, query)` — exactly the §16.4 turnstile pattern, with three methods on one piece of private state.

Exercise 16.3 — traffic_light

```
#language axioma/beginner
make_traffic_light: func() [
  s: "red";
  func() [
    if s == "red" then [rebind s = "green"]
    else if s == "green" then [rebind s = "yellow"]
    else [rebind s = "red"];
    s
  ]
]
```

```
light: make_traffic_light()
l1: light()   # green
l2: light()   # yellow
l3: light()   # red
l4: light()   # green
```

Commentary. A 3-state cycle. The cascade `if … else if … else …` reads the current state and chooses the next one. After the `if`, we read the (now-updated) state and return it.

Notice the brackets around each assignment branch: `[s = "green"]`. The Axioma parser requires explicit blocks for assignment statements inside `if` arms (single-expression arms without brackets are treated as expressions, and assignment isn't an expression in this context). The brackets are noise, but they're *consistent* noise — once you see the pattern in a few state machines, it's automatic.

Three-state cycles are everywhere: rock-paper-scissors, traffic lights, day-shift/swing-shift/night-shift schedules, RGB color rotators. The same code shape solves all of them; just rename the states.

Exercise 16.4 — count_passing

```
#language axioma/beginner
make_turnstile: func() [
  state: "locked";
  people: 0;
  coin: func() [
    if state == "locked" then [rebind state = "unlocked"]
    else [rebind state = state];
    state
  ];
  push_t: func() [
    if state == "unlocked" then [
      rebind state = "locked";
      rebind people = people + 1
    ]
    else [rebind state = state];
    state
  ];
  passing_count: func() [ people ];
  (coin, push_t, passing_count)
]
t: make_turnstile()
coin: t[1]
push_t: t[2]
passing_count: t[3]
r1: coin()
r2: push_t()
println(passing_count())   # 1
```

Commentary. Three functions share the same private state. The trick: `make_turnstile` defines `stat`

`e` and `people` as locals (so they're captured by the closures), then returns the *three* closures as a tuple. Callers destructure the tuple and use each function independently — but the three functions all *see the same `state` and `people` cells*.

This is **the multi-method object pattern**. Smalltalk, Self, JavaScript (without `class` syntax), Lua, Scheme — all do this. A "class" is really *a function that returns a tuple-of-functions sharing private state*.

You've just hand-rolled an object system in two screens of Axioma. The only thing missing for full OOP is *inheritance* — and that's what Chapter 18 will revisit when Concepts and sum types come back into the picture.

Exercise 16.5 — find_first_with_loop

```
#language axioma/beginner
find_first_with_loop: func(arr, pred) [
  i: 1;
  n: len(arr);
  found: none;
  while (i <= n and found == none) [
    v: arr[i];
    if pred(v) then [found = v]
    else [found = found];
    i = i + 1
  ];
  found
]
println(find_first_with_loop([1, 3, 5, 7], func(x) [x > 4]))   # 5
println(find_first_with_loop([1, 3, 5], func(x) [x > 100]))    # none
println(find_first_with_loop([], func(x) [true]))              # none
```

Commentary. Two loop conditions: `i <= n` (haven't run past the end) *and* `found == none` (haven't found anything yet). Once either is false, the loop exits.

`found` starts as `none` and becomes the first matching element. The `else [found = found]` no-op satisfies the parser's expression-statement requirement (assignments in `if` arms need brackets).

The recursive version is shorter:

```
find_first_rec: func(arr, pred) [
  if len(arr) == 0 then none
  else if pred(first(arr)) then first(arr)
  else find_first_rec(rest(arr), pred)
]
```

Both are correct. The loop version uses constant stack space (no recursive frames), reads imperatively, and stops *exactly* when it finds a match — no wasted work, no pending pending computation.

The loop version returns and prints `none` when no element matches. Distinct from `om` (the SETL "unknown" value), which renders as `Ω`.

Exercise 16.6 (open) — make_cell

```
#language axioma/beginner
make_cell: func(initial_alive) [
  alive: initial_alive;
  func(neighbors) [
    newstate: alive;
    if alive then [
      if neighbors == 2 or neighbors == 3 then [newstate = true]
      else [newstate = false]
    ]
    else [
      if neighbors == 3 then [newstate = true]
      else [newstate = false]
    ];
    rebind alive = newstate;
    alive
  ]
]
c: make_cell(true)
s1: c(2)   # true  (lives with 2 neighbors)
s2: c(1)   # false (dies — under-population)
s3: c(3)   # true  (born again — exactly 3 neighbors)
s4: c(0)   # false (dies — under-population)
```

Commentary. Conway's rules collapse into one nested `if`. The cell's `alive` field is captured by the closure; each `c(neighbors)` call reads the current state, computes the new state per Conway's table, mutates, and returns.

(Note the rename: `next` is reserved in Axioma — we used `newstate` instead.)

Orchestrating many cells. The minimum each cell needs: its own `alive` state and an `update(neighbors)` method. That's exactly what `make_cell` provides. A grid would be:

- A 2D array `grid[row][col]` of cells.
- A function that counts living neighbors for each cell.
- A "step the grid" function that *first* computes all the neighbor counts from the *current* state, *then* calls every `update` — never read-and-update in one pass, or the cells you've already updated will pollute the counts of the cells you haven't yet.

The two-phase split (count, then update) is the canonical parallel-step semantics. Real Game of Life implementations use double-buffering: maintain two grids, update one from the other, then swap. The cell-as-closure version demonstrates the *unit*; scaling to a grid is engineering.

Exercise 16.7 — an alias and a functional copy

```
type PointCursor = struct mutable {x:: Float, y:: Float}
let cursor = PointCursor(1.0, 2.0)
let alias_cursor = cursor
let moved = {cursor with x: 3.0}
alias_cursor.y = 8.0
println(cursor.y)
println(moved.y)
println(alias_cursor is/same cursor)
println(moved is/same cursor)
```

The output is `8.0`, `2.0`, `true`, `false`. The alias shares the original; the functional update constructs another mutable record. Identity is independent of field contents. Untouched nested mutable values, if present, would remain shared.

Exercise 16.8 — local scope and mutable cells

The local version prints `21`, then `10`; bare assignment prints `21`, then `21`. Both cells are mutable. The difference is which cell the branch uses: a fresh shadow, or the existing function binding. With `let`, `score += 1` fails because the fresh cell is immutable.

```
function local_solution()
  score = 10
  if true
    local score = 20
    score += 1
    println(score)
  end
  score
end
function assignment_solution()
  score = 10
  if true
    score = 20
    score += 1
    println(score)
  end
  score
end
println(local_solution())
println(assignment_solution())
```

Output: `21`, `10`, `21`, `21`, in that order.

Exercise 16.9 — choose the destination

The four rows print `(30, 20)` / `10`, `(30, 20)` / `10`, `(30, 30)` / `10`, and `(30, 20)` / `30`, respectively. Ordinary assignment and `local` each introduce an inner-function binding in this example. `rebind` reaches the outer function's cell; `global` reaches the module cell.

This combined witness uses four inner functions and resets the outer `x` after the nearest-cell case, so each case starts with outer `x = 20`:

```
x = 10
module_value() = x
function compare_destinations()
  x = 20
  function ordinary()
    x = 30
    x
  end
  function fresh()
    local x = 30
    x
  end
  function nearest()
    rebind x = 30
    x
  end
  function module_write()
    global x = 30
    x
  end
  result = ordinary()
  println((result, x))         # (30, 20)
  println(module_value())
  result = fresh()
  println((result, x))         # (30, 20)
  println(module_value())
  result = nearest()
  println((result, x))         # (30, 30)
  println(module_value())
  x = 20
  result = module_write()
  println((result, x))         # (30, 20)
  println(module_value())
end
compare_destinations()
```

If `inner` first declares `local x = 25`, its subsequent `rebind x = 30` updates that same inner cell. "Nearest existing" includes the current scope:

```
function nearest_local()
  x = 20
  function inner()
    local x = 25
    rebind x = 30
    x
  end
  result = inner()
  (result, x)
end
println(nearest_local())       # (30, 20)
```

Exercise 16.10 — follow the captured cell

With ordinary assignment, both reads print 12. The closure sees later changes to its captured cell; the original repeated-`local` program instead introduces a different cell, leaving the captured one at 1.

```
local x = 1
peek = func() [x]
x = x + 1
x += 10
println(peek())                # 12
println(x)                     # 12
```

With `let`, the new cell is immutable, so the increment fails. The initializer still reads the previous `x`, and the earlier closure still retains it:

```
local x = 1
peek = func() [x]
let x = x + 1
x += 10                       # ERROR: Cannot reassign 'x'
```

Exercise 16.11 — a hole has a type, not a default

The cell is uninitialized; no Function value exists yet. Reading it before the fill raises an uninitialized-binding error. With `let`, the first fill seals the cell and a second assignment fails because the binding is immutable. Use `var` to allow later replacements; the `Function` annotation still rejects an integer. These refusals are runtime guarantees, even where the lightweight checker cannot prove the function's type.

```
var callback :: Function = _
callback = func(n) [n + 1]
println(callback(2))          # 3
callback = func(n) [n * 2]
println(callback(2))          # 4
```

Exercise 16.12 — fill or shadow?

```
local result :: Integer
peek = func() [result]
if true then [result = 10] else [result = 20]
println(result)               # 10
println(peek())               # 10
```

Both assignments target the existing outer cell. Replacing them with fresh `local result = ...` declarations initializes separate branch-local cells; the outer hole remains unreadable. Closure definition captures the cell without reading it. A call before the branch raises an uninitialized-binding error; a call after the fill sees its value. The checker diagnoses the early read through this known closure and accepts the later call. Unknown callees remain a runtime boundary.

```
let result :: Integer
if true then [result = 10] else [result = 20]
println(result)               # 10
```

Only one branch executes and supplies the immutable cell's one successful fill. A subsequent assignment to that cell fails. `= _` may be written explicitly in either declaration without changing these results. Neither form creates a default value or makes the name visible before its declaration.

Exercise 16.13 — which paths need a fill?

The value `0` completes the branch normally, so that branch now reaches the later read. Both branches must fill the outer cell:

```
function choose_result(cancel)
    local result :: Integer
    if cancel then [result = 0] else [result = 10]
    result
end
println(choose_result(true))    # 0
println(choose_result(false))   # 10
```

A `continue` before the post-test assignment reaches `result == 1` with an unfilled cell. The condition's read fails; executing the body once does not mean every statement in it executes. Taking `&result` obtains its cell; `:result` fetches the stored value and therefore fails until the cell is filled.

Exercise 16.14 — justify the evidence

A positive count and a nonempty array guarantee the first iteration, but the fill must precede any exit reaching the read. A zero count leaves `repeated` uninitialized. Moving `*address = 10` before the fresh shadow still fills the same earlier cell and still prints `(10, 20)`. Omitting `fill()` leaves the outer

cell uninitialized: describing a computation does not execute it. Moving the division before the first assignment leaves `completed` uninitialized after `try` catches the error. All three unsafe reads receive checker diagnostics.

Each original example needs only one successful fill per cell, so each works with `let`. The shadow creates another cell; it is not a second write to the first one. For example:

```
let answer :: Integer
let address = &answer
let answer = 20
*address = 10
println((*address, answer))    # (10, 20)
```

An arbitrary unknown count or callee may still work at runtime without giving the checker enough evidence. That is an analysis limit, not an invented default value or a different rule for immutable bindings.

Exercise 16.15 — four independent questions

Scope selects the cell, initialization determines whether it has a readable value, the annotation constrains assigned values, and mutability determines whether the successful fill can be replaced. Without an annotation, a mutable cell may change value type. `:: Integer` rejects the string; `let` rejects the second assignment even if it is another Integer. An early read fails, and a failed first assignment leaves the cell unreadable. Assigning `none`, `om`, or an empty Array initializes a real value.

```
let items
items = [1]
push(items, 2)
println(items)                # [1, 2]
```

The binding still denotes the same Array. Mutating that Array does not replace the binding; `items = [3]` would attempt replacement and fail.

Exercise 16.16 — evidence and uncertainty

Supplying `n: 7` skips the default expression completely. The body fills `answer`, so its later read succeeds. A completed write survives an error that occurs afterwards:

```
local answer
fill = func(n = [rebind answer = 7; 1 / 0]) [rebind answer = n]
try [fill()]
println(answer)               # 7; passes --typecheck
```

The body never starts; the default's completed fill survives. A definite read identifies an unfilled cell. An opaque call leaves uncertainty. Both read diagnostics stop `--typecheck`; runtime still requires a value.

Exercise 16.17 — decide what a transaction keeps

Early return is a successful exit: commit occurs before the function returns. An escaping error rolls back the second fact, while the outer Array retains its append. The catch runs after rollback:

```
relation enrolled(x)
enrol: func(name) [
    with transaction [
        insert("enrolled", name)
        return "saved"
    ]
    "unreachable"
]
println(enrol("Ada"))                       # saved
println(len({ X | X <- enrolled(X) }))      # 1

events: []
failed: try(with transaction [
    insert("enrolled", "Grace")
    push(events, "attempt")
    1 / 0
])
println(error?(failed))                     # true
println(len({ X | X <- enrolled(X) }))      # 1
println(events)                             # ["attempt"]

captured: with transaction [
    insert("enrolled", "Grace")
    try(1 / 0)
]
println(error?(captured))                   # true
println(len({ X | X <- enrolled(X) }))      # 2
```

The inner `try` produces a caught error value, so the final scope commits. An explicit commit in a helper is refused before it can close the scope's journal. If that refusal escapes, the scope rolls back; if caught inside, its journal remains usable. A suspended producer could leave the scope's lifetime unfinished, so producer construction, suspension, and demand during the scope, and scopes executing inside producers, are refused. Eager `generate` completes synchronously and is supported.

Chapter 17 · Solutions

Common preamble — the four basic stream operators that every exercise builds on:

```
#language axioma/beginner
nats_from: func(k) [ (k, func() [nats_from(k + 1)]) ]
stream_head: func(s) [ s[1] ]
stream_tail: func(s) [ s[2]() ]
take_stream: func(n, s) [
  if n == 0 then []
  else [
    h: stream_head(s);
    t: stream_tail(s);
    push([], h) + take_stream(n - 1, t)
  ]
]
map_stream: func(f, s) [
  (f(stream_head(s)), func() [map_stream(f, stream_tail(s))])
]
filter_stream: func(pred, s) [
  h: stream_head(s);
  if pred(h) then (h, func() [filter_stream(pred, stream_tail(s))])
  else filter_stream(pred, stream_tail(s))
]
```

Exercise 17.1 — integers_from

```
integers_from: func(start, step) [
  (start, func() [integers_from(start + step, step)])
]
println(take_stream(5, integers_from(0, 1)))    # [0, 1, 2, 3, 4]
println(take_stream(5, integers_from(10, 5)))   # [10, 15, 20, 25, 30]
println(take_stream(5, integers_from(0, -1)))   # [0, -1, -2, -3, -4]
```

Commentary. The classic generalization of `nats_from`. Two parameters threaded through the recursion: `start` advances by `step` each call. The thunk captures both.

Negative `step` works because `start + step` is just integer addition — no implicit "must be positive." The stream descends as easily as it ascends.

This single function subsumes `nats_from` (`integers_from(k, 1)`) and could be the basis of arithmetic progressions in any base. Combined with `take_while`, it expresses *bounded* ranges without needing a separate `range(a, b)` builtin: `take_while(func(x) [x <= 100], integers_from(1, 1))`.

Exercise 17.2 — take_while

```
take_while: func(pred, s) [
  h: stream_head(s);
  if pred(h) then push([], h) + take_while(pred, stream_tail(s))
  else []
]
println(take_while(func(x) [x < 5], nats_from(1)))         # [1, 2, 3, 4]
println(take_while(func(x) [x > 0], integers_from(10, -1))) # [10, 9, ..., 1]
```

Commentary. Eager unfolding — `take_while` *materializes* a prefix of the stream into an array. The recursion stops at the first element that fails the predicate.

Important caveat: if no element ever fails the predicate, `take_while` loops forever. `take_while(func(x) [true], nats_from(1))` would try to materialize all the naturals, which is impossible. The exercise prompt explicitly says "the exercise assumes a finite false-prefix."

A safer version takes a *maximum count* as a third argument: `take_while_max(n, pred, s)` — but that's a different function, useful for different purposes.

Exercise 17.3 — zip_streams

```
zip_streams: func(s1, s2) [
  ((stream_head(s1), stream_head(s2)),
   func() [zip_streams(stream_tail(s1), stream_tail(s2))])
]
println(take_stream(5, zip_streams(nats_from(1), nats_from(100))))
# [(1, 100), (2, 101), (3, 102), (4, 103), (5, 104)]
```

Commentary. Walk two streams in lockstep, pairing their heads. The head of the zipped stream is a *tuple* of the two input heads; the tail is a thunk that recursively zips the tails.

If either stream is shorter than the other, this version loops forever trying to advance the shorter one past its end — but in this chapter we only work with infinite streams, so the question doesn't come up. A "stop at the shorter" version would check both heads and emit an end-marker when one fails to advance.

`zip_streams` plus `take_stream` plus `map_stream` lets you do *correlated computations* on parallel infinite sequences: "add element-wise the squares and the cubes of the naturals," "compare the Fibonacci numbers to the Lucas numbers at each index," etc.

Exercise 17.4 — stream_of_squares

```
stream_of_squares: map_stream(func(x) [x * x], nats_from(1))
println(take_stream(5, stream_of_squares))   # [1, 4, 9, 16, 25]
```

Commentary. One line. `map_stream` does all the work; we just feed it the right ingredients. This is the *whole point* of composing stream operations: simple primitives compose into rich derived streams without ceremony.

Notice we *don't* iterate explicitly anywhere. There's no `for`, no `while`, no manual `i++` — just "apply this function to every element of this stream." The stream's laziness means `stream_of_squares` is *defined* without any work actually happening; the work only happens when `take_stream` peels off the first five values.

Exercise 17.5 — every_kth

```
drop_k: func(s, m) [
  if m == 0 then s
  else drop_k(stream_tail(s), m - 1)
]
every_kth: func(s, k) [
  (stream_head(s), func() [every_kth(drop_k(s, k), k)])
]
println(take_stream(5, every_kth(nats_from(1), 2)))   # [1, 3, 5, 7, 9]
println(take_stream(5, every_kth(nats_from(1), 5)))   # [1, 6, 11, 16, 21]
```

Commentary. Two functions:

- `drop_k(s, m)` advances a stream forward `m` steps, returning the stream from there on. Pure recursion on the count `m`; no work on the stream's contents.
- `every_kth(s, k)` yields the current head, then in the thunk it `drop_k`s ahead by `k` and recurses.

Why `drop_k(s, k)` instead of `drop_k(s, k - 1)`? Because between successive "k-th" elements you need to advance `k` steps total: one to consume the current head, plus `k - 1` more to land on the next k-th position. So you advance by `k`, *not* `k - 1`.

Verify: with `k = 2`, after yielding 1 you advance 2 → land on 3; yield 3, advance 2 → land on 5. Matches.

This sampling pattern is useful for "downsample a sensor stream" or "every 60th frame of a video stream" — anything where the source produces faster than you want to consume.

Exercise 17.6 (open) — sieve of Eratosthenes

```
sieve: func(s) [
  p: stream_head(s);
  (p, func() [
    sieve(filter_stream(func(x) [x % p != 0], stream_tail(s)))
  ])
]
primes: sieve(integers_from(2, 1))
println(take_stream(10, primes))   # [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Commentary. Five lines for the prime sieve. Every recursive call peels off the *next* prime `p`, then sieves the rest of the stream to remove every multiple of `p` before recursing.

Answers to the chapter questions.

1. **Why does each prime sieve the *rest*?** Because once you've identified `p` as prime, every multiple of `p` greater than `p` cannot possibly be prime. Removing them *now* means `sieve` doesn't have to re-test them later. The very first call to `sieve` (on `integers_from(2, 1)`) peels off 2, then sieves out every even number from the rest. The next call works on `[3, 5, 7, 9, 11, 13, ...]` — but 9 is `3 * 3` and gets sieved out at the next layer, so the *next* prime emitted is 5. And so on.
2. **What's the asymptotic cost?** Painful, actually. Each prime sets up a new `filter_stream` layer. By the time we're producing the `k`-th prime, every value being considered has to pass through `k - 1` layers of nested filter. For the first 10 primes that's fine; for the first 10,000 primes the cost grows roughly as $O(n^2)$ per prime extracted. Plenty fast for textbook examples, useless for "list me the first million primes."
3. **The efficient sieve.** The classical Sieve of Eratosthenes is *not* this. It allocates an array of size `n`, marks composites by walking each prime's multiples, and reads off the surviving primes — `O(n log log n)` total. The stream version *looks like* the sieve but implements a *trial-division-style* algorithm. The stream version wins on *aesthetics* (a five-line recursive function vs. a 20-line array-mutation loop) and loses on *performance* ($O(n^2)$ vs. `O(n log log n)`). This trade-off — *expressiveness vs. efficiency* — is the recurring tension at the heart of high-level languages. Real implementations of `primes` in production libraries pick efficiency.

How to use these solutions

The stream pattern is *defined* by the `(head, thunk)` pair. Every stream function is one of:

- A producer: returns a `(value, thunk)` pair from scratch (`nats_from`, `integers_from`).
- A transformer: returns a `(transformed_head, thunk-that-transforms-the-tail)` pair (`map_stream`, `filter_stream`, `zip_streams`, `every_kth`).
- A consumer: peels off some prefix into an array (`take_stream`, `take_while`).
- A recursive sieve-style operation that combines all three (`sieve`).

Once the pair-and-thunk *shape* feels natural, you can write any stream operation in a few lines. The

verbosity is front-loaded into the four primitives; everything else composes cleanly.

Solution 17.7 — replay and saved positions

```
let g = (x | x <- [10, 20, 30])
let replay = sequence(g)
println(first(replay, 2))                    # [10, 20]
println(first(replay, 2))                    # [10, 20]
println(collect(replay))                     # [10, 20, 30]
let c = cursor([10, none, 30])
let saved = cursor_copy(c)
cursor_next(c)
println(c.value)                            # none
println(c.done)                             # false
cursor_seek(c, 99)
println(c.position)                         # 4
println(c.done)                             # true
println(saved.position)                     # 1
println(saved.value)                        # 10
```

The first read consumes two Generator values. The second read replays them. Collection pulls the remaining value and observes exhaustion. The saved cursor shares storage but keeps position 1. At position 2, `none` is a real value; at position 4 it means exhaustion, distinguished by `.done`. A second binding to `c` would alias its mutable position; `cursor_copy` is the bookmark.

Exercise 17.8 — a bounded parallel walk

```
let rows = sequence_zip(1.., 10.., "abc")
println(first(rows, 2))
println(first(rows, 2))
println(force(rows))
```

The first two calls print `[(1, 10, "a"), (2, 11, "b")]`. The final call prints `[(1, 10, "a"), (2, 11, "b"), (3, 12, "c")]`. Cached cells make replay stable, and the finite String bounds the walk. A combined `for` header walks a Cartesian product and re-enters its inner generators for each outer value.

Chapter 18 · Solutions

Common preamble for the Shape exercises:

```
concept Shape
Circle extends Shape
Circle has radius: 0.0
Rectangle extends Shape
Rectangle has width:  0.0
Rectangle has height: 0.0
Triangle extends Shape
Triangle has base:   0.0
Triangle has height: 0.0
Shape partition Circle, Rectangle, Triangle
```

Exercise 18.1 — perimeter

```
perimeter: func(s) [
  if s is Circle    then 2.0 * 3.14159 * s.radius
  else if s is Rectangle then 2.0 * (s.width + s.height)
  else if s is Triangle  then 3.0 * s.base
  else 0.0
]
println(perimeter(a Circle {radius: 5.0}))                  # ~31.42
println(perimeter(a Rectangle {width: 3.0, height: 4.0}))    # 14.0
println(perimeter(a Triangle {base: 5.0, height: 4.33}))     # 15.0
```

Commentary. Exactly the same `is`-cascade shape as `area`, but with different per-variant formulas. The *structure* of the function is "variant-dispatch then arithmetic" — change the arithmetic, keep the structure. That factor-out-the-shape principle is the same instinct that drove `map` and `filter` in Part III.

The Triangle case assumes an equilateral triangle so the "perimeter" formula needs only `base`. If we wanted general triangles we'd add a `side2` and `side3` field — but at that point we should think about whether `Triangle` is really *one shape* or *several* (right triangles, equilateral, isoceles, scalene). The HtDP design recipe answer: refactor when the data definition outgrows its current shape.

Exercise 18.2 — LightState

```
concept LightState
RedLight extends LightState
GreenLight extends LightState
YellowLight extends LightState
LightState partition RedLight, GreenLight, YellowLight

next_light: func(s) [
  if s is RedLight    then a GreenLight  {}
  else if s is GreenLight  then a YellowLight {}
  else if s is YellowLight then a RedLight    {}
  else none
]
r: a RedLight {}
g: next_light(r)
y: next_light(g)
r2: next_light(y)
println(g is GreenLight)    # true
println(y is YellowLight)   # true
println(r2 is RedLight)     # true
```

Commentary. A 3-cycle via fresh instances. Each variant is *type-tagged* but carries no data (the braces in `a GreenLight {}` are empty); the entity has a classification even when it has no additional fields. Fresh entities remain distinct; this is not an enum singleton.

Compare to Ch.16's traffic light, which used a mutable string variable. Two completely different mechanisms for the same logical effect:

- Ch.16: one mutable cell, three legal *string* values
- Ch.18: three distinct *concept types*, each instantiable

The Concept version is *type-safer*: writing `a BlueLight {}` would be obviously wrong (no such Concept exists). The string version's misspell (`"yelow"`) would silently fall through to `else`. Trade-offs again: the Concept version is more verbose to set up, more robust once running.

Exercise 18.3 — biological taxonomy

```
concept Animal
Animal has name: ""
Vertebrate extends Animal
Invertebrate extends Animal
Animal partition Vertebrate, Invertebrate

Mammal extends Vertebrate
Bird extends Vertebrate
Fish extends Vertebrate
```

```
Reptile extends Vertebrate
Amphibian extends Vertebrate
Vertebrate partition Mammal, Bird, Fish, Reptile, Amphibian

dog:    a Mammal     {name: "Rex"}
robin:  a Bird       {name: "Robin"}
salmon: a Fish       {name: "Sal"}
gecko:  a Reptile    {name: "Geck"}
frog:   an Amphibian {name: "Hop"}

is_warm_blooded: func(a) [
  if a is Mammal then true
  else if a is Bird then true
  else false
]
println(is_warm_blooded(dog))    # true
println(is_warm_blooded(robin))  # true
println(is_warm_blooded(salmon)) # false
println(is_warm_blooded(gecko))  # false
println(is_warm_blooded(frog))   # false
```

Commentary. A *two-level* taxonomy. The first partition splits Animals into Vertebrates and Invertebrates; the second splits Vertebrates into the five vertebrate classes biologists recognize.

`is_warm_blooded` only cares about the vertebrate-class level — so we dispatch on that level directly via `is Mammal / Bird`. Notice we don't need to test `is Vertebrate` first; `is Mammal` *implies* `is Vertebrate` *implies* `is Animal`. The transitivity of the `is` relation does the checking for us.

Real biological taxonomy goes much deeper — Kingdom → Phylum → Class → Order → Family → Genus → Species, seven levels for a single species. The same `is`/`partition` machinery scales.

Exercise 18.4 — `expr_eval` revisited

```
concept Expression
Lit extends Expression
Lit has value: 0
Op extends Expression
Op has kind: "+"
Op has lhs: none
Op has rhs: none
Expression partition Lit, Op

eval_expr: func(e) [
  if e is Lit then e.value
  else if e is Op then [
    l: eval_expr(e.lhs);
    r: eval_expr(e.rhs);
    if e.kind == "+" then l + r
```

```
    else if e.kind == "-" then l - r
    else if e.kind == "*" then l * r
    else if e.kind == "/" then l / r
    else 0
  ]
  else 0
]
e1: a Lit {value: 1}
e2: a Lit {value: 2}
plus: an Op {kind: "+", lhs: e1, rhs: e2}
println(eval_expr(plus))   # 3
```

Commentary. The Ch.9 code reorganized to have *both* `Lit extends Expression` and `Op extends Expression` *and* an explicit `Expression partition Lit, Op`. The function body is unchanged from Ch.9.

The partition records an intended classification contract that can be queried over the live extent. Constructor-exhaustiveness checking is available separately with typed `variant`/`data` matches. For the reader, "Expression is *exactly* Lit or Op" is useful information. The partition *says* it.

Exercise 18.5 — count_shapes_by_type

```
csbt_acc: func(arr, nc, nr, nt) [
  if len(arr) == 0 then (nc, nr, nt)
  else [
    s: first(arr);
    if s is Circle    then csbt_acc(rest(arr), nc + 1, nr, nt)
    else if s is Rectangle then csbt_acc(rest(arr), nc, nr + 1, nt)
    else if s is Triangle  then csbt_acc(rest(arr), nc, nr, nt + 1)
    else csbt_acc(rest(arr), nc, nr, nt)
  ]
]
count_shapes_by_type: func(arr) [ csbt_acc(arr, 0, 0, 0) ]
c1: a Circle {radius: 1.0}
c2: a Circle {radius: 2.0}
c3: a Circle {radius: 3.0}
r1: a Rectangle {width: 1.0, height: 2.0}
t1: a Triangle {base: 1.0, height: 1.0}
println(count_shapes_by_type([c1, r1, t1, c2, c3]))   # (3, 1, 1)
println(count_shapes_by_type([]))                      # (0, 0, 0)
```

Commentary. Ch.15's accumulator-recursion pattern with three carriers (`nc, nr, nt`) instead of one. The dispatch chooses *which* accumulator gets incremented at each step.

This shape generalizes to *N* variants. The pattern: one accumulator per variant. The base case returns the tuple of all accumulators. The recursive step uses `is`-dispatch to choose which accumulator to bump.

A more abstract version would use a *dictionary* mapping variant-name → count, growing the dictionary

as new variants are encountered. Axioma's hash-literal syntax (`{kind: 0}`) makes that practical too; the tuple version is simpler and fine for small known variant counts.

Exercise 18.6 (open) — your own variant data

Sample answer: musical notes

```
concept Sound
Sound has duration: 0.0   # seconds

Note extends Sound
Note has freq_hz: 440.0   # default A4

Chord extends Sound
Chord has notes: []       # array of Sound (recursive!)

Rest extends Sound
# Rest has no extra fields — duration alone is enough

Sound partition Note, Chord, Rest

describe: func(s) [
  if s is Note  then "single tone at " + s.freq_hz + " Hz"
  else if s is Chord then "chord of " + len(s.notes) + " notes"
  else if s is Rest  then "silence for " + s.duration + " sec"
  else "unknown"
]

middle_c:    a Note {freq_hz: 261.63, duration: 0.5}
chord:       a Chord {notes: [middle_c], duration: 1.0}
rest_a_beat: a Rest {duration: 0.25}

println(describe(middle_c))
println(describe(chord))
println(describe(rest_a_beat))
```

Commentary on the design.

1. **Root** — every Sound has a duration. Useful for sequencing regardless of pitch/silence.
2. **Variants** — Note (one tone), Chord (multiple tones at once), Rest (silence). The three classical primitives of western music notation.
3. **Recursion** — Chord contains an array of Sounds, which can itself include Chords. A jazz chord might literally be modeled this way.
4. **Dispatch** — `describe` switches on the variant tag and uses the variant-specific field.

Compared to other languages.

- **Java**: each variant would be a class, possibly an abstract `Sound` superclass. Axioma's `extends` mirrors Java's `extends` closely.

- **ML/Haskell**: `data Sound = Note Float Float | Chord [Sound] | Rest Float`. Sum types make the variant set obviously closed.
- **Python**: classes with shared `duration` attribute plus isinstance dispatch. Looks almost identical to Axioma.

Each language has its idiom; Axioma's `is` + `extends` + `partition` reads similarly to the inheritance approaches and adds queryable classification laws. Check the partition over the live extent; this does not add constructor-exhaustiveness checking to an `is` cascade.

Exercise 18.7 — a closed LightState

```
type LightState = variant RedLight | GreenLight | YellowLight
next_light: func(s :: LightState) [
    match s with
    | RedLight => GreenLight
    | GreenLight => YellowLight
    | YellowLight => RedLight
]
println(next_light(RedLight))   # GreenLight
println(next_light(GreenLight)) # YellowLight
println(next_light(YellowLight)) # RedLight
```

The equivalent legacy header is `data LightState = RedLight | GreenLight | YellowLight`. With the annotated parameter, removing YellowLight produces the missing-constructor diagnostic. Without an identifiable input type, that static coverage check can be skipped. A partitioned Concept hierarchy does not provide the same check.

Exercise 18.8 — structural input, nominal value

```
type PointRow = {x:: Float, y:: Float}
type Point = struct {x:: Float, y:: Float}
save_point: func(row :: PointRow) [Point(row.x, row.y)]
let row :: PointRow = {x: 3.0, y: 4.0}
let point = save_point(row)
let moved = {point with x: 6.0}
println(row is PointRow) # true
println(row is Point)    # false
println(point is Point)  # true
println(point.x)         # 3.0
println(moved.x)         # 6.0
```

A Dictionary alias supplies no required x/y field contract. The schema describes input shape; the constructor marks a validated nominal Point.

Exercise 18.9 — aliases and shallow copies

```
type PointCursor = struct mutable {x:: Float, y:: Float, trail:: Array}
let cursor = PointCursor(1.0, 2.0, [0])
let other_cursor = cursor
cursor.x = 3.0
let copy = {cursor with y: 8.0}
copy.trail[1] = 9
println(other_cursor.x) # 3.0
println(copy is/same cursor) # false
println(cursor.trail[1]) # 9
println(error?(try([cursor.x = "bad"]))) # true
println(cursor.x) # 3.0
```

The alias is the same mutable record. The functional copy is another outer record but shares the trail Array. `let` protects the binding, not field contents. The failed write leaves x unchanged.

Exercise 18.10 — vocabulary and distinctions

```
type DrawingId = opaque Integer
type Distance = Float
type Coordinate = Integer | Float
type Position = (Float, Float)
type Direction = enum North, East, South, West
let id = DrawingId(7)
let distance :: Distance = 2.5
let coordinate :: Coordinate = 3
let position :: Position = (3.0, 4.0)
let direction :: Direction = East
println(7 is DrawingId) # false
println(match id with | DrawingId(n) => n) # 7
println(distance is Float) # true
println(coordinate is Coordinate) # true
println(position[2]) # 4.0
println(direction.ord) # 1
Digit ranges 0..9
println(7 is Digit) # true
println(12 is Digit) # false
```

DrawingId adds a wrapper distinction and Direction adds nominal enum members. The alias, union and tuple product describe existing values. Digit restricts membership without wrapping each Integer; Chapter 19 develops these restrictions.

Exercise 18.11 — a drawing application

Use a schema for imported rows, immutable structs for saved points, a variant for geometric alternatives, a mutable struct for the cursor, and Concepts for artists and their domain relationships. A Dictionary alias would lose the row field contract; mutable points would make snapshots vulnerable to alias writes; Concepts would add unnecessary entity semantics to a closed geometry calculation. An opaque ID distinguishes identifiers from ordinary numbers. A typed variant match can be checked for missing constructors, whereas a Concept partition is queried over entities and does not make an `is` cascade statically exhaustive.

Exercise 18.12 — guarded geometry

```
type Shape = variant Circle(Float) | Rect(Float, Float) | Dot
geometry_label: func(s :: Shape) [
    match s with
    | Circle(r) when r == 1 => "unit circle"
    | Circle(r) when r > 0 => "circle"
    | Circle(_) => "invalid circle"
    | Rect(w, h) when w > 0 and h > 0 and w == h => "square"
    | Rect(w, h) when w > 0 and h > 0 => "rectangle"
    | Rect(_, _) => "invalid rectangle"
    | Dot => "point"
]
println(geometry_label(Circle(1.0))) # unit circle
println(geometry_label(Circle(2.0))) # circle
println(geometry_label(Circle(0.0))) # invalid circle
println(geometry_label(Rect(2.0, 2.0))) # square
println(geometry_label(Rect(2.0, 3.0))) # rectangle
println(geometry_label(Rect(0.0, 3.0))) # invalid rectangle
println(geometry_label(Dot)) # point
```

A unit circle also passes r>0, so its more specific arm must be first. The unguarded constructor arms cover rejected payloads explicitly. Removing Circle(_) makes the typed match non-exhaustive; two numeric predicates are not an unguarded coverage declaration.

Exercise 18.13 — nested scene

```
type Shape = variant Circle(Float) | Dot
type Scene = variant OnCanvas(Shape) | Hidden
scene_radius: func(scene :: Scene) [
    case scene
    | OnCanvas(Circle(r)) when r > 0 => r
    | OnCanvas(_) => 0.0
    | Hidden => 0.0
]
```

```
scene_radius_switch: func(scene :: Scene) [
    switch scene
    | OnCanvas(Circle(r)) when r > 0 => r
    | OnCanvas(_) => 0.0
    | Hidden => 0.0
]
println(scene_radius(OnCanvas(Circle(2.0)))) # 2.0
println(scene_radius(OnCanvas(Dot))) # 0.0
println(scene_radius(Hidden)) # 0.0
println(scene_radius_switch(OnCanvas(Circle(2.0)))) # 2.0
println(case "later" | "now" => 1) # none
println(error?(try(switch/strict "later" | "now" => 1))) # true
```

A nested mismatch or false guard proceeds to the outer fallback. A guard error propagates; the fallback is not an exception handler. Case and switch share the matcher's extraction and selection rules.

Exercise 18.14 (optional) — spelling equivalence

```
type Point = struct {x:: Float, y:: Float}
type RecordPoint = record {x:: Float, y:: Float}
type PointCursor = struct mutable {x:: Float, y:: Float}
type ShortCursor = struct mut {x:: Float, y:: Float}
type RecordCursor = record mut
    x :: Float
    y :: Float
end
type LongCursor = record mutable {x:: Float, y:: Float}
point_trace: func(make_point) [
    let point = make_point(3.0, 4.0)
    let moved = {point with x: 6.0};
    (point.x, moved.x, point == make_point(3.0, 4.0),
     error?(try([point.x = 6.0])))
]
cursor_trace: func(make_cursor) [
    let cursor = make_cursor(3.0, 4.0)
    let other_cursor = cursor
    cursor.x = 6.0;
    (other_cursor.x, cursor == other_cursor,
     cursor == make_cursor(6.0, 4.0),
     error?(try([cursor.x = "east"])), cursor.x)
]
println(point_trace(Point))
println(point_trace(RecordPoint))
println(cursor_trace(PointCursor))
println(cursor_trace(ShortCursor))
println(cursor_trace(RecordCursor))
println(cursor_trace(LongCursor))
```

```
println(Point(3.0, 4.0) is RecordPoint)
println({x: 3.0, y: 4.0} is RecordPoint)
```

The first two lines both print `(3.0, 6.0, true, true)`; the next four each print `(6.0, true, false, true, 6.0)`; the last two print `false`. Both immutable spellings preserve the original under a functional update and reject field replacement. Every mutable spelling shares changes with aliases, compares fresh records by identity, and checks field types before writing.

The type names, not the choice of `struct` or `record`, distinguish Point from RecordPoint. Neither declaration is a Dictionary schema. Replacing one spelling with its alias in the same program changes the notation, not these contracts. An `end` body lists the same annotated fields on separate lines; replacing it with braces changes only the body delimiter. Existing generic support and the VM refusal boundary are also unchanged.

Exercise 18.15 — combining schemas

```
type Named = {name:: String}
type Priced = {price:: Float}
type Product = Named & Priced
label: func(p :: Product) :: String [ p.name + " (" + str(p.price) + ")" ]
let pen :: Product = {name: "pen", price: 2.5}
println(label(pen))                                                    # pen (2.5)
println(error?(try([ let cheap :: Product = {name: "cap"} ])))         # true
println(error?(try([ let typo :: Product = {name: "cap", prise: 1.0} ])))  # true
type Discounted = Product & {discount?:: Float}
let sale :: Discounted = {name: "ink", price: 4.0, discount: 0.5}
println(sale.price - sale.discount)                                    # 3.5
let full :: Discounted = {name: "ink", price: 4.0}
println(full.discount)                                                 # none
println(error?(try([ eval("type Named = {name:: String}\ntype Bad = Named & Integer")
 ↪  ])))  # true
```

The members merge into one schema with `name`, `price` and the optional `discount`, so a literal may use all three keys. A missing `price` fails the member that requires it; `prise` is declared by no member, so it is a typo. `discount?` is optional in the only member that declares it, so an absent discount reads `none`. `Named & Integer` is empty: a schema admits only Dictionaries, and no Dictionary is an Integer. The declaration is refused before any value exists, so the mistake is reported where it was written.

How to use these solutions

Start with the meaning of the data and the contract each declaration supplies. For the Concept exercises, read `partition`, `extends` and `has`, then follow membership dispatch. For the variant solution, derive the match cases from its constructors. For rows and records, distinguish structural membership, nominal construction, field mutation and aliasing before tracing the outputs.

Chapter 19 · Solutions

Exercise 19.1 — Suit

```
Suit enumerates Hearts, Diamonds, Clubs, Spades

is_red: func(s) [
  s == Hearts or s == Diamonds
]
is_minor: func(s) [
  s == Diamonds or s == Clubs
]
println(is_red(Hearts))      # true
println(is_red(Spades))      # false
println(is_minor(Diamonds))  # true
println(is_minor(Hearts))    # false
```

Commentary. Four enum values, two predicates. Each predicate combines equality checks with `or`. Notice we never need to *create* a Suit — the `enumerates` declaration already created all four legal values as top-level names.

A more abstract version could use array membership: `s in {Hearts, Diamonds}`. The functional one-liner is the same thing, written longhand. Either reads fine.

Exercise 19.2 — safe_div

```
Digit ranges 0..9
safe_div: func(a, b) [
  if b == 0 then 0 else a / b
]
println(safe_div(8, 2))    # 4
println(safe_div(7, 0))    # 0
```

Commentary. The function body doesn't need to *check* the range — the type annotation does that for the caller. When the caller writes `safe_div(15, 3)`, Axioma errors *before* the function body runs:

```
type error: variable 'a' expects type Digit, got INTEGER (value: 15)
```

That's the *type-as-precondition* idea: instead of writing "// caller must pass 0..9" in a comment, write `:: Digit` in the annotation. The compiler enforces.

What if you genuinely want both Digit-typed *and* Integer-typed callers? Two functions: one whose

annotation is `Digit` (machine-checked), one whose annotation is `Integer` (more permissive). Pick the function for the *call site*'s needs.

Exercise 19.3 — schedule_check

```
Day enumerates Mon, Tue, Wed, Thu, Fri, Sat, Sun
Workday extends Day in Mon..Fri

schedule_check: func(d, title) [
  "Scheduled: " + title + " on " + d
]
println(schedule_check(Tue, "Meeting"))    # "Scheduled: Meeting on Tue"
```

Commentary. Three-piece string concatenation. The body doesn't say "is `d` a workday?" — it *trusts* the type annotation. The function declared to take `Workday` won't ever see `Sat`/`Sun`.

For *full* enforcement, we'd write the function with explicit type annotation: `schedule_check: func(d :: Workday, title) [...]`. The annotation lives in the type declaration; the behavior of the function doesn't change either way.

If you *do* try `schedule_check(Sat, "Brunch")`, the runtime errors with "*type error: variable 'd' declared as Workday, assigned value of type CONCRETE_OBJECT (value: Sat)*". The function body never executes. That's the win.

Exercise 19.4 — is_business_hours

```
Hour ranges 0..23
is_business_hours: func(h) [
  h >= 9 and h < 17
]
println(is_business_hours(10))   # true
println(is_business_hours(20))   # false
```

Commentary. Plain Boolean math, but the type constraint makes the function *safe* for any 24-hour clock value. Calling with `25` (a nonsensical "hour") gets rejected at binding time. Calling with `−1` (also nonsensical) likewise.

What if you wanted to model 12-hour-clock time (1..12)? Declare `Hour12 ranges 1..12` and re-annotate. Two different subranges, two different types, no shared identity — exactly right because they really *are* different kinds of "hour."

Exercise 19.5 — Lock

```
LockState enumerates Locked, Unlocked, Jammed

make_lock: func() [
  state: Locked;
  attempts: 0;
  func(guess) [
    if state == Jammed then [rebind state = state]
    else if guess == "open_sesame" then [rebind state = Unlocked]
    else [
      rebind attempts = attempts + 1;
      if attempts > 3 then [rebind state = Jammed]
      else [rebind state = state]
    ];
    state
  ]
]
lock: make_lock()
r1: lock("wrong")    # Locked
r2: lock("wrong")    # Locked
r3: lock("wrong")    # Locked
r4: lock("wrong")    # Jammed (4th wrong attempt)
println(r1)
println(r2)
println(r3)
println(r4)
```

Commentary. A state machine + counter, hidden inside a closure. Three behavioral cases: 1. Already Jammed → stay there (lock is permanently broken). 2. Correct guess → unlock. 3. Wrong guess → increment attempts; if > 3 then Jammed, else stay Locked.

Compared to Ch.16's turnstile, the state space is larger (3 states instead of 2) and the state depends on *cumulative* events (the `attempts` counter), not just the current event. That's the full state-machine pattern: states + counter + guard-conditions.

This is the canonical *combination chapter* for Part V/VI: mutation (Ch.16), enums (Ch.19), and closures-with-state (Ch.16) all working together in one small device.

Exercise 19.6 (open) — Grade and Honors

```
Grade enumerates Fail, Poor, Avg, Good, Top
PassingGrade extends Grade in Poor..Top
HonorsGrade extends Grade in Good..Top

letter_to_gpa: func(g) [
  if g == Fail then 0.0
```

```
  else if g == Poor then 1.0
  else if g == Avg  then 2.0
  else if g == Good then 3.0
  else if g == Top  then 4.0
  else 0.0
]
is_passing: func(g) [ g != Fail ]
qualifies_for_honors: func(g) [ g == Top or g == Good ]

println(letter_to_gpa(Top))         # 4.0
println(letter_to_gpa(Fail))        # 0.0
println(is_passing(Poor))           # true
println(is_passing(Fail))           # false
println(qualifies_for_honors(Good)) # true
println(qualifies_for_honors(Avg))  # false
```

Commentary. Five enum levels, two subrange subtypes. Three predicate/computation functions over them.

Answer to the design question. `qualifies_for_honors` *could* take parameter type `HonorsGrade`, but that would restrict callers to *only* pass grades that *already qualify for honors* — which makes the predicate trivially `true`. That's almost certainly not what you want.

The principle: **the parameter type should match what the caller *might pass*, not what the function *needs*.** `qualifies_for_honors` answers a question *about* a Grade; its parameter should accept *any* Grade, and the body decides the answer. Restricting the parameter would shrink the function's domain to nothing useful.

Where `HonorsGrade` *would* be the right annotation: a function that takes an already-qualified grade and returns the diploma type, scholarship eligibility, etc. The `HonorsGrade` type would *enforce* at the call site that the caller has already done the qualification check.

This is the difference between **predicate functions** (should take broad types and return Booleans) and **operation functions** (should take narrow types that *guarantee* preconditions, so the body can skip them). Encoding which is which in the type system is a strict-language pattern that Ada, Modula-3, and modern Rust all push hard. Axioma's subrange/subtype pairing brings the same discipline to beginner code.

How to use these solutions

The thread through these six exercises: **encode invariants in types**.

- Enum → "exactly one of these named values" (not "any string")
- Subrange → "in this numeric interval" (not "any integer")
- Subtype-by-restriction → "this enum constrained to a slice"

Every annotation removes a class of possible bugs from the code that follows. The boilerplate of declaring the types is the *one-time cost*; the safety is *every time the code runs*.

For real-world software, that ratio is wildly in favor of the types. Ada has been used in commercial avionics for 50 years on exactly this bet.

Chapter 20 · Solutions

Exercise 20.1 — definitely_pass

```
definitely_pass: func(submitted, passed_grade, attendance_ok) [
  submitted and passed_grade and attendance_ok
]

println(definitely_pass(true, true, true))    # true
println(definitely_pass(true, false, true))   # false
println(definitely_pass(true, om, true))      # Ω
println(definitely_pass(om, om, om))          # Ω
println(definitely_pass(om, false, om))       # false
println(definitely_pass(false, om, true))     # false
```

Commentary. A one-liner. The K3 `and` operator does the heavy lifting:

- All `true` → `true` (Boolean degenerate case)
- Any `false` → `false` (`false` is the absorbing value in the K3 conjunction table, even with `om` in the mix)
- Mixed `true` and `om` → Ω (genuinely cannot decide)

The pedagogical win: **the same expression** that works for Boolean inputs also handles unknown inputs correctly. You don't write two functions and dispatch on whether the inputs are all-Boolean; the operators dispatch on the operand types.

This result does not mean every conjunction skipped an operand. `om and false` evaluates the right operand and obtains `false` from the truth table; a Boolean `false` on the left can skip the right. The function's arguments have already been evaluated before its body runs. See §20.11 for an effect trace.

Exercise 20.2 — hiring_decision

```
at_least: func(v, t) [
  (v or lukasiewicz(t)) == v
]
at_most: func(v, t) [
  (v and lukasiewicz(t)) == v
]

hiring_decision: func(tech, comm, team) [
  if at_least(tech, 0.7) and at_least(comm, 0.7) and at_least(team, 0.7) then "hire"
```

```
  else if at_most(tech, 0.3) or at_most(comm, 0.3) or at_most(team, 0.3) then "reject"
  else "interview again"
]

println(hiring_decision(lukasiewicz(0.9), lukasiewicz(0.8), lukasiewicz(0.7)))  # hire
println(hiring_decision(lukasiewicz(0.1), lukasiewicz(0.8), lukasiewicz(0.8)))  #
 ↪  reject
println(hiring_decision(lukasiewicz(0.5), lukasiewicz(0.5), lukasiewicz(0.5)))  #
 ↪  interview again
println(hiring_decision(lukasiewicz(0.9), lukasiewicz(0.9), lukasiewicz(0.6)))  #
 ↪  interview again
```

Commentary. The `at_least` helper is the key idea:

> A truth value `v` is **at least** `t` exactly when `v or lukasiewicz(t)` is still `v`.

Why? `v or t` is `max(v, t)` in L3. If `v ≥ t`, then `max(v, t) = v`. If `v < t`, then `max(v, t) = t ≠ v`. So the equality test is exactly the threshold test, but expressed entirely in L3-native operators — no float extraction, no escape from the algebra.

`at_most` is the dual: `v and t == v` iff `v ≤ t` (because `and` is `min`).

This is a small but real example of the **point of an algebra**: you can do work *inside* the algebra without peeking at the underlying representation. The L3 truth values have an algebra rich enough to test their own ordering.

Exercise 20.3 — k3_truth_table

```
k3_truth_table: func(name, op) [
  vals: [true, false, om]
  println("a       b       a " + name + " b")
  i: 1
  while (i <= 3) [
    j: 1
    while (j <= 3) [
      a: vals[i]
      b: vals[j]
      println(str(a) + "   " + str(b) + "   " + str(op(a, b)))
      j = j + 1
    ]
    i = i + 1
  ]
]

k3_truth_table("and", func(a, b) [a and b])
println()
k3_truth_table("or", func(a, b) [a or b])
```

Commentary. Nested `while` over the three-element value array. `op` is a *function value* passed in —

straight out of Chapter 10's higher-order toolkit.

The output for `and`:

```
a       b       a and b
true    true    true
true    false   false
true    om      Ω
false   true    false
false   false   false
false   om      false
om      true    Ω
om      false   false
om      om      Ω
```

Reading this table tells you K3 in one glance: an output is *decidable* only when the unknowns don't matter. `false and om` is `false` (the `false` absorbs); `true and om` is Ω (the unknown propagates).

Generalization. The same generator with `[true, false, belnap("both"), belnap("neither")]` produces the 16-cell B4 table — exercise for the reader. Same code, different value array. That's the dispatch-on-type idiom paying off.

Exercise 20.4 — belnap_aggregate

```
combine_two: func(a, b) [
  if a == b then a
  else if a == belnap("neither") then b
  else if b == belnap("neither") then a
  else if a == belnap("both") or b == belnap("both") then belnap("both")
  else belnap("both")
]

belnap_aggregate: func(w1, w2, w3) [
  combine_two(combine_two(w1, w2), w3)
]

verdict_describe: func(v) [
  if v == belnap("true") then "Consensus yes"
  else if v == belnap("false") then "Consensus no"
  else if v == belnap("both") then "Disagreement"
  else "No evidence"
]

r1: belnap_aggregate(belnap("true"), belnap("true"), belnap("true"))
println(verdict_describe(r1))    # Consensus yes
r2: belnap_aggregate(belnap("false"), belnap("false"), belnap("false"))
println(verdict_describe(r2))    # Consensus no
r3: belnap_aggregate(belnap("true"), belnap("false"), belnap("true"))
println(verdict_describe(r3))    # Disagreement
r4: belnap_aggregate(belnap("neither"), belnap("neither"), belnap("neither"))
```

```
println(verdict_describe(r4))    # No evidence
r5: belnap_aggregate(belnap("true"), belnap("neither"), belnap("true"))
println(verdict_describe(r5))    # Consensus yes
r6: belnap_aggregate(belnap("both"), belnap("true"), belnap("true"))
println(verdict_describe(r6))    # Disagreement
```

Commentary. `combine_two` is the consensus combiner. Five cases:

1. **Equal witnesses** → return that value (consensus preserved by repetition).
2. **One witness is `neither`** → return the other (no evidence doesn't override active evidence).
3. **One witness is `both`** → return `both` (existing contradiction propagates).
4. **Otherwise** the two are different non-`neither`, non-`both` values → must be `true` and `false` → return `both` (the contradiction).

`belnap_aggregate` then folds over the three witnesses via two consecutive `combine_two` calls.

This is a *custom* operator, not a built-in B4 op. The built-in `or` is truth-disjunction (TRUE absorbs); the combiner above is something different — a *consensus* operation that *adds* information rather than projecting onto truth. Both are useful for different purposes; the language gives you the algebra to define your own when you need to.

Exercise 20.5 — meet_all and join_all

```
meet_all: func(values) [
  reduce(func(a, b) [a and b], values[1], values[2..len(values)])
]

join_all: func(values) [
  reduce(func(a, b) [a or b], values[1], values[2..len(values)])
]

v: [lukasiewicz(0.5), lukasiewicz(0.8), lukasiewicz(0.2)]
println(meet_all(v))   # 0.20ł
println(join_all(v))   # 0.80ł
```

Commentary. Reduce-with-an-operator, straight from Chapter 10. The L3 `and` is `min`; reducing with it gives the minimum truth value in the array. Same with `or` = `max`.

Note the seed: `values[1]`, not `lukasiewicz(0.0)` or `lukasiewicz(1.0)`. We're reducing over a *non-empty* array, so starting with the first element avoids a separate identity-element argument. For the empty-array case, callers would need to decide what `meet_all([])` means (`lukasiewicz(1.0)` is the identity for `and`; `lukasiewicz(0.0)` for `or`).

The MVL pattern shines here: `reduce` doesn't know or care that the operator is L3-flavored — it just calls `op(a, b)` and the dispatcher routes to L3 because the operands are Łukasiewicz.

Exercise 20.6 (open) — combined_diagnosis

```
# Design choice: K3 lab unknown dominates ⇒ "Need more data" trumps
# any expert opinion (we don't recommend treatment without a lab).
# Then check experts. If either is `both`, OR the two contradict
# (one true, one false), report disagreement. If both experts say
# `neither`, order tests. Otherwise check that severity supports
# the experts' agreed direction.

at_least: func(v, t) [
  (v or lukasiewicz(t)) == v
]
at_most: func(v, t) [
  (v and lukasiewicz(t)) == v
]

combined_diagnosis: func(lab, severity, opinion_a, opinion_b) [
  if lab == om then "Need more data"
  else if opinion_a == belnap("neither") and opinion_b == belnap("neither") then "Order
  ↪  tests"
  else if opinion_a == belnap("both") or opinion_b == belnap("both") then "Expert
  ↪  disagreement"
  else if (opinion_a == belnap("true") and opinion_b == belnap("false")) or (opinion_a
  ↪  == belnap("false") and opinion_b == belnap("true")) then "Expert disagreement"
  else if lab == true and opinion_a == belnap("true") and opinion_b == belnap("true")
  ↪  and at_least(severity, 0.7) then "Treat"
  else if lab == false and opinion_a == belnap("false") and opinion_b ==
  ↪  belnap("false") and at_most(severity, 0.3) then "Don't treat"
  else "Need more data"
]

println(combined_diagnosis(true, lukasiewicz(0.8), belnap("true"), belnap("true")))
↪  # Treat
println(combined_diagnosis(false, lukasiewicz(0.1), belnap("false"), belnap("false")))
↪  # Don't treat
println(combined_diagnosis(om, lukasiewicz(0.5), belnap("true"), belnap("false")))
↪  # Need more data
println(combined_diagnosis(true, lukasiewicz(0.8), belnap("true"), belnap("false")))
↪  # Expert disagreement
println(combined_diagnosis(true, lukasiewicz(0.8), belnap("neither"),
↪  belnap("neither"))) # Order tests
println(combined_diagnosis(true, lukasiewicz(0.5), belnap("true"), belnap("true")))
↪  # Need more data
```

Commentary. The interesting part of this exercise isn't the code — it's the *design choice*. Three logics, four inputs, multiple possible outcomes. Different students will reasonably make different choices.

My design choice (documented in the comment):

1. Lab unknown is the **strongest veto** — if you don't have the lab, you can't recommend treatment,

full stop.
2. Both experts saying "no opinion" is a **second veto** — if no one's looked at it, order tests.
3. **Then** check expert disagreement (either has `both`, or they directly conflict).
4. Only **after** all the vetoes are clear do we look at lab
    - severity to confirm the experts.

Other students might prioritize **expert disagreement first**, on the theory that "if the doctors can't agree, lab data won't help." That's a reasonable alternative — different problem framing, different ordering of conditions. The exercise doesn't have one right answer.

The cross-logic pattern is the lesson. Each logic handles its piece:

Question	Logic	Operation
"Is lab unknown?"	K3	`lab == om`
"Is severity high enough?"	L3	`at_least(severity, 0.7)`
"Do experts agree?"	B4	direct value comparison

There's no single algebra that handles all three. But mixing the algebras *at the level of conditional dispatch* works fine. That's how real-world MVL code looks.

How to use these solutions

The thread through these six exercises: **the operator does the dispatch, but the design is yours.**

- K3 and L3 operators auto-dispatch — you can write Boolean- looking code that handles unknown/-partial values correctly.
- B4 has a value space rich enough to encode contradictions, but the built-in `or` is truth-disjunction, not consensus. You write your own combiner when consensus is what you need.
- Mixing logics in one function is fine — dispatch on each variable's own algebra.

The pedagogical reward: students who've worked through these exercises will *never* default to threshold-Boolean when the data is actually three-valued or contradictory. That habit is worth more than the syntax.

20.7 — Repetition is not independence

Rain AND rain denotes exactly the rain event, so its probability remains 0.3. For a distinct independent fair-coin event, multiply 0.3 by 0.5 to get 0.15. Wet grass can depend on rain; a joint probability or a suitable conditional and its required marginal are needed. The two marginals alone do not determine the joint. A missing entry says the probability table lacks information. It neither asserts equal odds nor chooses a B4 truth value. Those are separate semantic choices, not default conversions.

Exercise 20.8 — Predict the calls

```
probe(label, value) = [println(label); value]
println(false implies probe("A", false))            # A, then true
println(kleene("false") and probe("B", true))        # B, then ⊥ᵏ
println(kleene("false") andalso probe("C", true))    # false; no C
println(nand(false, probe("D", true)))               # D, then true
```

Implication evaluates both operands, even with Boolean false on the left. Typed Kleene false does not activate `and`'s Boolean-only gate, so `B` prints and the result retains its Kleene type. The third line uses truthiness, skips `C`, and returns Boolean false. The last line is an ordinary function call: its arguments, including the call printing `D`, are evaluated before `nand` receives their values. None of these rules follows from the truth table alone.

Chapter 21 · Solutions

Exercise 21.1 — Family-tree fact base

```
relation parent(child, p)
relation married(a, b)
relation sibling_raw(x, y)

parent("alice", "mary")
parent("alice", "john")
parent("bob", "mary")
parent("bob", "john")
parent("mary", "evelyn")
parent("mary", "harold")
parent("john", "ruth")
parent("john", "frank")

married("mary", "john")
married("evelyn", "harold")
married("ruth", "frank")

sibling_raw(X, Y) <= parent(X, P) and parent(Y, P)

println("All parents:", {P | P <- parent(_, P)})
println("Mary's kids:", {C | C <- parent(C, "mary")})
println("Couples:", {(X, Y) | married(X, Y)})
println("Sibling pairs:", {p | p <- {(X, Y) | sibling_raw(X, Y)}, p[1] != p[2]})
```

Commentary. Four query forms in one program:

- `{P | P <- parent(_, P)}` — generator with wildcard binding the *second* argument, the parent.
- `{C | C <- parent(C, "mary")}` — generator with literal in the *second* argument, free variable in the first.
- `{(X, Y) | married(X, Y)}` — tuple head, single-relation filter. Axioma materializes all `married` facts as tuples.
- `{p | p <- {…}, p[1] != p[2]}` — comprehension *over* an already-computed tuple set, filtering by element-equality. Indexed access `p[1]` / `p[2]` works on tuples just like on arrays.

The sibling rule is `parent(X, P) and parent(Y, P)`. The post-filter `p[1] != p[2]` drops the trivial self-pairs that the rule includes by symmetry of unification.

Exercise 21.2 — Grandparent rule + query

```
relation parent(child, p)
relation grandparent(child, gp)

parent("alice", "mary")
parent("alice", "john")
parent("bob", "mary")
parent("bob", "john")
parent("mary", "evelyn")
parent("mary", "harold")
parent("john", "ruth")
parent("john", "frank")

grandparent(X, G) <= parent(X, P) and parent(P, G)

println("All gps:", {G | G <- grandparent(_, G)})
println("Alice's gps:", {G | G <- grandparent("alice", G)})
println("Evelyn's grandkids:", {C | C <- grandparent(C, "evelyn")})
```

Output:

```
All gps: {"evelyn", "frank", "harold", "ruth"}
Alice's gps: {"evelyn", "frank", "harold", "ruth"}
Evelyn's grandchildren: {"alice", "bob"}
```

Commentary. One rule, three queries — each query just varies *which* position is bound and which is free.

The same rule answers “who is X’s grandparent” (X fixed, GP free) and “who has X as a grandchild” (GP fixed, X free). This bidirectional usage is the hallmark of logic programming: a single rule serves as both a *function* and its *inverse*, depending on which side has the unknowns.

Exercise 21.3 — Course prerequisite chain

```
relation course(code, title)
relation prereq(course, requires)

course("CS101", "Intro")
course("CS201", "DS")
course("CS301", "Algo")
course("CS401", "Compilers")
course("MATH101", "Calc")
course("MATH201", "Discrete")

prereq("CS201", "CS101")
prereq("CS201", "MATH101")
prereq("CS301", "CS201")
```

```
prereq("CS301", "MATH201")
prereq("CS401", "CS301")
prereq("MATH201", "MATH101")

println("CS401 direct:", {R | R <- prereq("CS401", R)})

hop1: {R | R <- prereq("CS401", R)}
hop2: {R | P <- prereq("CS401", P), R <- prereq(P, R)}
hop3: {R | P <- prereq("CS401", P), Q <- prereq(P, Q), R <- prereq(Q, R)}
all_eventual: hop1 ∪ hop2 ∪ hop3
println("CS401 all eventual:", all_eventual)

all_courses: {C | C <- course(C, _)}
has_prereq: {C | C <- prereq(C, _)}
no_prereq: {c | c <- all_courses, not (c in has_prereq)}
println("No-prereq courses:", no_prereq)
```

Output:

```
CS401 direct: {"CS301"}
CS401 all eventual: {"CS101", "CS201", "CS301", "MATH101", "MATH201"}
No-prereq courses: {"CS101", "MATH101"}
```

Commentary. Three queries, three patterns:

- *Direct prereqs* — single generator. Easy.
- *Eventual prereqs* — explicit-depth chain. `hop2` is a two-generator comprehension where the *output* binding `R` comes from the inner `prereq(P, R)`. The `∪` of the three depths covers any chain of length 1, 2, or 3. (For longer chains, add `hop4` etc.)
- *No-prereq courses* — set-theoretic difference, written as a comprehension with a negative-membership filter. This is the canonical "what's in A but not in B" pattern. Note that `\` doesn't work as set difference in Axioma; the comprehension form is the idiom.

This exercise shows that **explicit-depth chains are a fine tradeoff for finite-depth transitive queries** — each `hop` is debuggable on its own; the union gives you the closure. (For unbounded chain depth, use the recursive clause pair: `requires(X, R) if prereq(X, R)` plus `requires(X, R) if prereq(X, P) and requires(P, R)`.)

Exercise 21.4 — Defeasible bird-flies

```
relation bird(x)
relation penguin(x)
relation ostrich(x)
relation flies(x)

bird("robin")
bird("eagle")
bird("sparrow")
```

```
bird("opus")
bird("oscar")
penguin("opus")
ostrich("oscar")

flies(X) <~~ bird(X)

cancel("flies", "opus")
cancel("flies", "oscar")

println({X @conjecture | X <- flies(X)})
```

Output:

```
{"eagle", "robin", "sparrow"}
```

Commentary. The defeasible rule `flies(X) <~~ bird(X)` populates `flies` with all five birds as **conjectures**. The two `cancel` calls mark `opus` and `oscar` as suppressed — their conjecture facts still exist (you can see them with `@all`), but the `@conjecture` filter excludes canceled facts.

This pattern lets you say "the rule is correct *as a default*, and these are the exceptions." Compare to the Boolean alternative:

```
# Awkward Boolean form
flies("robin")     # store each true
flies("eagle")
flies("sparrow")
# DON'T store flies("opus") — implicitly false
# DON'T store flies("oscar") — implicitly false
```

The Boolean form lacks the *justification*: there's nothing in the program that says "we expected robin to fly because robin is a bird." With the defeasible rule, the inference chain is right there:

```
why flies("robin")
# Explanation for: flies(robin)
# This is a conjecture derived by defeasible inference from:
#   - bird("robin")  [standalone]
```

Auditable defeasibility is the win.

Exercise 21.5 — Friend-of-friend recommendations

```
relation friend(a, b)
friend("alice", "bob")
friend("bob", "alice")
friend("alice", "carol")
friend("carol", "alice")
```

```
friend("bob", "dan")
friend("dan", "bob")
friend("carol", "eve")
friend("eve", "carol")
friend("dan", "frank")
friend("frank", "dan")

alice_friends: {P | P <- friend("alice", P)}
fof_raw: {X | P <- friend("alice", P), X <- friend(P, X)}
recs: {x | x <- fof_raw, x != "alice" and not (x in alice_friends)}
println("Alice's direct friends:", alice_friends)
println("Recommendations for alice:", recs)
```

Output:

```
Alice's direct friends: {"bob", "carol"}
Recommendations for alice: {"dan", "eve"}
```

Commentary. A classic two-step join over a symmetric graph:

1. `alice_friends` — set of alice's direct friends.
2. `fof_raw` — friends of those friends, computed via a *chained* generator. The variable `P` is shared between the two generators (both must agree on who the "middle" person is).
3. `recs` — filter out alice herself and already-known friends.

A real social network would store friendships as a single symmetric relation. Axioma doesn't automatically infer symmetry, so you write both directions explicitly. (One could *also* write a rule `friend(X, Y) <= friend(Y, X)` — but that would store duplicates rather than just symmetrize queries.)

Exercise 21.6 (open) — Genealogy reasoner

```
relation parent(child, p)
relation married(a, b)
relation gender(person, g)
relation birth_year(person, y)

# Gen 1: grandparents
# (mary's and susan's parents — gen 0, unrecorded)

# Gen 2: mary, susan, frank
parent("mary", "george")
parent("mary", "helen")
parent("susan", "george")
parent("susan", "helen")
parent("frank", "ian")
parent("frank", "jane")

married("mary", "frank")
```

```
# Gen 3
parent("alice", "mary")
parent("alice", "frank")
parent("bob", "mary")
parent("bob", "frank")
parent("carol", "susan")
parent("dave", "susan")

gender("mary", "F")
gender("susan", "F")
gender("frank", "M")
gender("alice", "F")
gender("bob", "M")
gender("carol", "F")
gender("dave", "M")
gender("george", "M")
gender("helen", "F")
gender("ian", "M")
gender("jane", "F")

birth_year("mary", 1960)
birth_year("susan", 1962)
birth_year("frank", 1959)
birth_year("alice", 1990)
birth_year("bob", 1992)
birth_year("carol", 1993)
birth_year("dave", 1995)

# Derived relations (4)
relation sibling_raw(x, y)
sibling_raw(X, Y) <= parent(X, P) and parent(Y, P)

relation grandparent(child, gp)
grandparent(C, G) <= parent(C, P) and parent(P, G)

relation aunt_or_uncle(person, au)
aunt_or_uncle(N, AU) <= parent(N, P) and sibling_raw(P, AU)

relation cousin_raw(x, y)
cousin_raw(X, Y) <= parent(X, P) and parent(Y, Q) and sibling_raw(P, Q)

# Queries
println("Alice's proper siblings:",
  {s | s <- {S | S <- sibling_raw("alice", S)}, s != "alice"})

println("Alice's grandparents:",
  {G | G <- grandparent("alice", G)})

println("Alice's aunts/uncles (raw):",
  {AU | AU <- aunt_or_uncle("alice", AU)})
```

```
println("Alice's cousins (raw, proper):",
  {c | c <- {C | C <- cousin_raw("alice", C)}, c != "alice"})
```

Output:

```
Alice's proper siblings: {"bob"}
Alice's grandparents: {"george", "helen", "ian", "jane"}
Alice's aunts/uncles (raw): {"frank", "mary", "susan"}
Alice's cousins (raw, proper): {"bob", "carol", "dave"}
```

Commentary. Four derived relations. Two observations worth calling out:

1. **`aunt_or_uncle` raw form includes a person's own parents.** The rule `aunt_or_uncle(N, AU) <= parent(N, P) and sibling_raw(P, AU)` fires when `P` and `AU` *are each other* (the trivial sibling case), so alice's own parents `mary` and `frank` appear in the output. Tightening the rule with an inequality requires post-filtering.
2. **`cousin_raw` includes siblings.** Same issue — when the "two parents" in the rule body are siblings via self-equality, the cousin set ends up including alice's siblings (bob). Again, post-filter to remove those.

For a fully-tidy reasoner you'd write:

```
alice_siblings: {s | s <- {S | S <- sibling_raw("alice", S)}, s != "alice"}
proper_cousins: {c | c <- {C | C <- cousin_raw("alice", C)},
                     c != "alice" and not (c in alice_siblings)}
```

Here `alice_siblings` excludes Alice herself. This is the shape of all the "filter the rule output" patterns we've seen this chapter.

Birth-year and gender are unused in the queries above — but you could write `older_sibling(X, Y) <= sibling_raw(X, Y) and birth_year(X, A) and birth_year(Y, B) and A < B` (modulo the < filter limitations) or `aunt(X, AU) <= aunt_or_uncle(X, AU) and gender(AU, "F")`. The fact-base is rich enough to support many more derived relations than the four we've shown — the exercise is open precisely so students can find their own questions to ask.

How to use these solutions

Three patterns to internalize for the relational style:

- **Generators are joins.** `{X | X <- rel1(a, X), Y <- rel2(X, Y)}` is a relational join — the shared variable `X` is the join key.
- **Rules + post-filter is the multi-step pattern.** Write the "broad" rule with `<=`, then narrow with a comprehension filter on the result set. Cleaner than trying to express the entire query in one rule body.
- **Defeasible + cancel = default + exception.** Don't store every fact individually. Store the rule, store the exceptions, and let the engine produce the rest.

Exercise 21.7 — Horn file on the proof-core

```
#language axioma/knowledge-core
assert/axiom parent("Mary", "John")
assert/axiom parent("Alice", "Mary")
grandparent(G, A) whenever parent(G, P) and parent(P, A)
thesis: derive/result grandparent("Alice", "John")
println(thesis.status)       # derived
println(thesis.grounding)    # theorem
```

A second file, not to be run as part of the happy path:

```
#language axioma/knowledge-core
assert/axiom parent("Mary", "John")
retract parent("Mary", "John")
# retract is not available in axioma/knowledge-core.
# Retraction is non-monotonic; use axioma/all for changing knowledge bases.
```

Commentary. The host chapter wrote `relation parent(...)` then bare facts. The core has no `relation` declaration: the asserted atom *is* the schema. `whenever` is the same connective as §21.5. `derive/result` is how you *see* a derivation on the core — `status` / `grounding` rather than a comprehension that happens to be non-empty.

`typically` is refused too (`defeasible rule is not available`). Defeasible reasoning stays on `axioma/all` (and on the milder `axioma/knowledge` naming fence, which still allows `retract`). The core is the subset whose completeness story is Horn and monotonic.

Logic programming sits next to functional programming in the toolkit — *both* are first-class in Axioma, neither replaces the other, and the best programs use both.

21.8 — An observable first choice

```
relation/prolog choose(x)
choose(1) :- true
choose(2) :- true
answers: [logic/prolog | X <- choose(X)]
println(take_solutions(1, answers.result))
println(answers.complete)
println(take_solutions(9, answers.result))
println(answers.complete)
cancelled: [logic/prolog | X <- choose(X)]
close_solutions(cancelled.result)
println(cancelled.status)
relation/prolog first(x)
first(1) :- prune
first(2) :- true
```

```
selected: [logic/prolog | X <- first(X)]
println(take_solutions(9, selected.result))
```

The output is `[1]`, `false`, `[2]`, `true`, `cancelled`, `[1]`. One pull leaves a choice outstanding. Exhausting the stream establishes its end; cancellation only abandons it. Prune eliminates the second clause of `first`. Datalog promises a complete declarative relation, so this order-dependent control is rejected there.

21.9 — A plausible cause is not proof

Each body is a sufficient explanation under its rule if its other premises and guards hold. The observation alone does not establish which cause occurred, or that these are the only causes. Retain both as candidates with their rule provenance. Independent evidence such as a verified observation log or a rumour source could support or reject one. Do not feed generated hypotheses back as facts and then count their own consequences as independent confirmation.

Exercise 21.10 — typed identity and uncertain absence

```
assert [mary has age 25]
assert [mary has age "25"]
retract [mary has age "25"]
remaining: query/result [?who has age ?age]
println(len(remaining.matches))           # 1
println(remaining.matches[1].bindings.age) # 25
```

The native value is an Integer. The String had a distinct identity despite its similar printed spelling. Typed session snapshots retain this distinction and the retraction. A text-only legacy record has already discarded information: its old matcher might miss a fact because a phrase was split. `complete: false` therefore prevents interpreting the miss as established absence. Store that uncertainty with the experience instead of learning an unsupported negative fact.

Exercise 21.11 — Priority and consumed work

Run this complete variation independently of the chapter example:

```
assert [ada waits]
assert [bob waits]
assert [carol waits]
assert [ada is urgent]
assert [carol is urgent]
served_order: []
serve: func(person) [
    push(served_order, person)
    retract [person, 'waits]
```

```
]
rule regular salience 1: [?person waits] implies serve(person)
rule urgent salience 10 no_loop: [?person waits] and [?person is urgent] implies
 ↪  serve(person)
deduce
println(len(served_order))                  # 3
println(_last_deduce_trace[1]["rule"])       # urgent
println(_last_deduce_trace[2]["rule"])       # urgent
println(_last_deduce_trace[3]["rule"])       # regular
deduce
println(_last_deduce_fired)                  # 0
println(len(served_order))                  # 3
```

Two different initial urgent activations remain eligible under `no_loop`. Retraction consumes each waiting fact and removes the corresponding regular activation. The next run sees no waiting work. If the action retained those facts, both rules could serve an urgent person within one run, and a fresh `deduce` could serve them again. Refraction is per activation and per run; `no_loop` is not persistent task-completion memory.

Chapter 22 · Solutions

Exercise 22.1 — Extend evaluator with subtraction

```
concept NumNode
NumNode has value: 0

concept OpNode
OpNode has op: ""
OpNode has left: 0
OpNode has right: 0

eval_expr: func(node) [
  if node is NumNode then node.value
  else if node is OpNode then [
    l: eval_expr(node.left)
    r: eval_expr(node.right)
    if node.op == "+" then l + r
    else if node.op == "-" then l - r
    else if node.op == "*" then l * r
    else if node.op == "/" then l / r
    else 0
  ]
  else 0
]

n10: a NumNode { value: 10 }
n3: a NumNode { value: 3 }
n2: a NumNode { value: 2 }
sub: an OpNode { op: "-", left: n10, right: n3 }
mul: an OpNode { op: "*", left: sub, right: n2 }
println(eval_expr(mul))   # 14
```

Commentary. One extra `else if` branch in the OpNode dispatch. Three lines of code. That's the *point*: adding a new operator to your evaluator is a *local* change. You don't have to revisit anything else.

Exercise 22.2 — Add an object-language `let` form

```
concept VarNode
VarNode has name: ""
empty_env: []
env_lookup: func(ev, nm) [
  if len(ev) == 0 then 0
  else if ev[1][1] == nm then ev[1][2]
  else env_lookup(rest(ev), nm)
]
env_extend: func(ev, nm, item_value) [[(nm, item_value)] + ev]

concept LetNode
LetNode has name: ""
LetNode has value_expr: 0
LetNode has body: 0

eval_expr: func(node, ev) [
  if node is NumNode then node.value
  else if node is VarNode then env_lookup(ev, node.name)
  else if node is OpNode then [
    l: eval_expr(node.left, ev)
    r: eval_expr(node.right, ev)
    if node.op == "+" then l + r
    else if node.op == "-" then l - r
    else 0
  ]
  else if node is LetNode then [
    v: eval_expr(node.value_expr, ev)
    new_env: env_extend(ev, node.name, v)
    eval_expr(node.body, new_env)
  ]
  else 0
]

# x: 3 in y: 4 in x + y
body_xy: an OpNode {
  op: "+",
  left: a VarNode { name: "x" },
  right: a VarNode { name: "y" }
}
inner_let: a LetNode {
  name: "y",
  value_expr: a NumNode { value: 4 },
  body: body_xy
}
outer_let: a LetNode {
  name: "x",
  value_expr: a NumNode { value: 3 },
```

```
  body: inner_let
}
println(eval_expr(outer_let, empty_env))   # 7
```

Commentary. Three steps for the `LetNode` case:

1. Evaluate the value expression in the *current* env.
2. Extend the env with the binding (name → value).
3. Evaluate the body in the *new* env.

The order matters. If you tried to extend the env *before* computing the value, you couldn't write `x: x + 1` to shadow a previous binding — the new `x` would already be in scope while computing its value. This semantic detail — "value expression sees the old env, body sees the new env" — is what *non-recursive* object-language `let` means. Recursive `let` (or `letrec`) extends the env *first* and then evaluates the value; that's how recursive function definitions become self-referential.

Exercise 22.3 — Pretty-print an AST

```
pretty: func(node) [
  if node is NumNode then str(node.value)
  else if node is VarNode then node.name
  else if node is OpNode then
    "(" + pretty(node.left) + " " + node.op + " " + pretty(node.right) + ")"
  else if node is IfNode then
    "if " + pretty(node.condition) + " then " + pretty(node.then_branch)
      + " else " + pretty(node.else_branch)
  else "?"
]

# Build (2 + 3) * 4
n2: a NumNode { value: 2 }
n3: a NumNode { value: 3 }
n4: a NumNode { value: 4 }
add: an OpNode { op: "+", left: n2, right: n3 }
mul: an OpNode { op: "*", left: add, right: n4 }
println(pretty(mul))   # ((2 + 3) * 4)
```

Commentary. `pretty` is structural recursion over the same Concept hierarchy as `eval_expr`. Same data shape, same program shape — but instead of returning a *value*, it returns a *string*.

This is the unfold/fold duality: `parse` (string → AST) and `pretty` (AST → string) are inverses. Real-world pretty- printers add indentation, operator precedence, line wrapping; the *core* is what's above.

A useful exercise: extend `pretty` for the `LambdaNode` and `AppNode` cases. `LambdaNode { param: "x", body: b }` should render as `"lambda x => " + pretty(b)`; `AppNode { callee: c, arg: a }` as `pretty(c) + "(" + pretty(a) + ")"`.

Exercise 22.4 — `run(s)` round-trip

```
run: func(s) [ ast_eval(parse(s)) ]

println(run("2 + 3"))                              # 5
println(run("(7 - 3) * 5"))                         # 20
println(run("if 1 == 1 then 100 else 200"))         # 100
```

Commentary. Three lines including the function. Axioma's `parse` builds the AST, `ast_eval` evaluates it using the *real* Axioma evaluator (not our toy version).

That's the **two-stage architecture** of every interpreter ever: read code into a data structure, then walk the data structure. Once you separate these, you can do all kinds of clever things — *transform* the AST before evaluation (optimization), *check* it without running (type inference), *serialize* it (caching), *transmit* it across the network (RPC). All of those tools start with the same parse-then-walk pattern.

The contrast with our hand-written `eval_expr`: ours handles a tiny language; Axioma's real evaluator handles *everything in Axioma*. Same architecture, vastly different scale.

Exercise 22.5 — `describe(s)` via headof / argsof

```
describe: func(s) [
  a: parse(s)
  println("head: " + headof(a))
  println("args: " + str(argsof(a)))
]

describe("2 + 3")
# head: +
# args: ["2", "3"]

describe("sqrt(16)")
# head: sqrt
# args: ["16"]

describe("if true then 1 else 2")
# head: If
# args: ["true", "1", "2"]
```

Commentary. `headof` and `argsof` are the *Mathematica introspection* — every expression, regardless of surface syntax, has a head and a sequence of arguments. `2 + 3` is *really* `+(2, 3)`. `sqrt(16)` is just `sqrt(16)`. `if … then … else …` is `If(cond, then, else)`.

This is the deep insight from Lisp and Mathematica: **all syntax is sugar over a uniform call form.** Once you see that, you understand why metaprogramming works. The "different" syntactic forms are all the same data structure underneath.

Exercise 22.6 (open) — A tiny Boolean-expression language

```
# Domain: pure Boolean expressions with literals and variables.
# Three Concepts: BoolLit, BoolVar, BoolOp.
# Two operators: AND, OR (plus NOT as unary).

concept BoolLit
BoolLit has value: false

concept BoolVar
BoolVar has name: ""

concept BoolOp
BoolOp has op: ""
BoolOp has left: 0
BoolOp has right: 0

concept BoolNot
BoolNot has expr: 0

# Env = array of (name, bool) pairs
empty_env: []
env_lookup: func(ev, nm) [
  if len(ev) == 0 then false
  else if ev[1][1] == nm then ev[1][2]
  else env_lookup(ev[2..len(ev)], nm)
]

# Evaluator
eval_bool: func(node, ev) [
  if node is BoolLit then node.value
  else if node is BoolVar then env_lookup(ev, node.name)
  else if node is BoolNot then not eval_bool(node.expr, ev)
  else if node is BoolOp then [
    l: eval_bool(node.left, ev)
    r: eval_bool(node.right, ev)
    if node.op == "and" then l and r
    else if node.op == "or" then l or r
    else false
  ]
  else false
]

# Pretty
pretty_bool: func(node) [
  if node is BoolLit then str(node.value)
  else if node is BoolVar then node.name
  else if node is BoolNot then "!" + pretty_bool(node.expr)
  else if node is BoolOp then
```

```
    "(" + pretty_bool(node.left) + " " + node.op + " " + pretty_bool(node.right) + ")"
  else "?"
]

# Three test programs
# Program 1: a AND b
p1: a BoolOp {
  op: "and",
  left: a BoolVar { name: "a" },
  right: a BoolVar { name: "b" }
}

# Program 2: (a OR b) AND (NOT c)
p2: a BoolOp {
  op: "and",
  left: a BoolOp { op: "or", left: a BoolVar { name: "a" }, right: a BoolVar { name:
↪  "b" } },
  right: a BoolNot { expr: a BoolVar { name: "c" } }
}

# Program 3: true OR (false AND a)
p3: a BoolOp {
  op: "or",
  left: a BoolLit { value: true },
  right: a BoolOp { op: "and", left: a BoolLit { value: false }, right: a BoolVar {
↪  name: "a" } }
}

# Env: a=true, b=false, c=false
e: [("a", true), ("b", false), ("c", false)]

println(pretty_bool(p1), "=>", eval_bool(p1, e))   # (a and b) => false
println(pretty_bool(p2), "=>", eval_bool(p2, e))   # ((a or b) and !c) => true
println(pretty_bool(p3), "=>", eval_bool(p3, e))   # (true or (false and a)) => true
```

Design choices documented:

- **Three concepts (`BoolLit`, `BoolVar`, `BoolOp`) plus unary `BoolNot`.** I separated `BoolNot` from `BoolOp` because it has *one* operand, not two; squishing it into `BoolOp` would mean ignoring the `right` slot. Cleaner to have distinct shapes for binary vs. unary operators.
- **Environment as a list of `(name, value)` tuples**, same as the Axioma evaluator from §22.5. Could use a Concept with `bindings: dict()` instead; the list form is what the chapter taught.
- **Operator dispatch by string comparison** on `node.op`. An alternative is one Concept per operator (`AndOp`, `OrOp`). String dispatch is more compact; per-op concepts are more explicit. Both work; pick the style that matches the rest of the codebase you're building.
- **Three test programs** chosen to exercise:
  (1) a single binary op,
  (2) nested ops with unary,
  (3) short-circuit-relevant case (a `true` on the left of `or` means the right side's value doesn't matter

— but our eager evaluator still evaluates it).

The bigger lesson. This Boolean DSL is **30 lines of code**, end-to-end. It has a clear data definition, a recursive evaluator, a recursive pretty-printer, an environment. It's a *complete language*, in the sense that you could now build it out — add `Xor`, `Implies`, quantifiers, satisfiability checking. The architecture scales.

You've just designed a programming language. That's what this chapter was for.

How to use these solutions

Three patterns to internalize:

- **Data definition first.** One Concept per AST node type. The grammar of your language *is* the inheritance tree.
- **Structural recursion over the AST.** `eval_expr` dispatches on `is NodeType` and recurses on children. Same recipe shape as Chapter 8.
- **Environments thread through.** Add an env parameter to every recursive call. Extend it for new bindings; look up in it for variable references. That's lexical scoping.

The capstone of the textbook is that **you can now read any small language implementation** — Lisp's `eval`, Python's `ast` module, JSON parsers, regex engines, configuration loaders. They're all variations on the parse-and-walk pattern you've just implemented twice (once by hand, once via `parse` + `ast_eval`).

Go build something.

Chapter 23 · Solutions

```
# Shared prelude used by every Chapter 23 solution
concept Empty
concept Cons
Cons has head: 0
Cons has tail: 0

nil: an Empty {}

list_of: func(arr) [
  result: nil
  i: len(arr)
  while (i >= 1) [
    result = a Cons { head: arr[i], tail: result }
    i = i - 1
  ]
  result
]

list_to_array: func(lst) [
  result: []
  cur: lst
  while (not (cur is Empty)) [
    result = result + ([] + [cur.head])
    cur = cur.tail
  ]
  result
]
```

Exercise 23.1 — nth

```
nth: func(lst, n) [
  if lst is Empty then 0
  else if n == 1 then lst.head
  else nth(lst.tail, n - 1)
]

l: list_of([10, 20, 30, 40, 50])
println(nth(l, 1))    # 10
println(nth(l, 3))    # 30
```

```
println(nth(l, 5))    # 50
println(nth(l, 6))    # 0 — out-of-bounds default
```

Commentary. Three sub-cases:

1. The list ran out (`Empty`) before we reached position `n` — out of bounds, return the default 0.
2. We're at the target position (`n == 1`) — return the current `head`.
3. Otherwise, recurse: shift one cell deeper, decrement `n`.

Cost. `nth(lst, n)` does *n* recursive calls, each O(1). Total O(n). Compare with `arr[n]` on a built-in Axioma array — that's O(1). The linked list trades random-access speed for front-end mutation speed.

Exercise 23.2 — contains

```
list_contains: func(lst, x) [
  if lst is Empty then false
  else if lst.head == x then true
  else list_contains(lst.tail, x)
]

println(list_contains(list_of([1, 2, 3, 4, 5]), 3))   # true
println(list_contains(list_of([1, 2, 3, 4, 5]), 99))  # false
println(list_contains(nil, 7))                         # false
```

Commentary. Same three-case template as `nth`:

1. List ran out without finding the target → `false`.
2. Current head matches → `true`.
3. Otherwise recurse on the tail.

Cost. O(n) in the worst case (target absent or at the end). Same as Axioma's array `in` operator — both have to scan.

Exercise 23.3 — filter_list

```
filter_list: func(pred, lst) [
  if lst is Empty then nil
  else if pred(lst.head) then a Cons { head: lst.head, tail: filter_list(pred,
  ↪  lst.tail) }
  else filter_list(pred, lst.tail)
]

l: list_of([1, 2, 3, 4, 5, 6, 7, 8])
evens: filter_list(func(x) [x % 2 == 0], l)
println(list_to_array(evens))   # [2, 4, 6, 8]
```

Commentary. For each cell:

- If `pred(head)` is true, **keep** the head: build a new cons with `head` and the recursive result on the tail.
- If `pred(head)` is false, **drop** the head: return the recursive result on the tail unchanged.

The cells that survive are *new* cons cells (we create them each iteration); the dropped cells are simply not referenced from the result. Garbage collection cleans them up eventually.

Exercise 23.4 — fold_list

```
fold_list: func(op, acc, lst) [
  if lst is Empty then acc
  else fold_list(op, op(acc, lst.head), lst.tail)
]

l: list_of([1, 2, 3, 4, 5])
println(fold_list(func(a, b) [a + b], 0, l))    # 15
println(fold_list(func(a, b) [a * b], 1, l))    # 120
```

Commentary. This is the **left fold** — `op` is applied left-to-right, with `acc` accumulating the running total.

Walk-through with `op = +`, `acc = 0`, `lst = [1, 2, 3]`:

```
fold_list(+, 0, [1, 2, 3])
= fold_list(+, +(0, 1), [2, 3])   = fold_list(+, 1, [2, 3])
= fold_list(+, +(1, 2), [3])      = fold_list(+, 3, [3])
= fold_list(+, +(3, 3), [])       = fold_list(+, 6, [])
= 6
```

Why this matters. `fold_list` is *the* general list-walking primitive. Most other operations are specializations:

- `length(lst)` = `fold_list((a, _) → a + 1, 0, lst)`
- `sum(lst)` = `fold_list(+, 0, lst)`
- `product(lst)` = `fold_list(*, 1, lst)`
- `max(lst)` = `fold_list(max-of-two, lst.head, lst.tail)`

Mastering fold means you stop writing custom recursions for common list operations.

Exercise 23.5 — zip_lists

```
zip_lists: func(a, b) [
  if a is Empty then nil
  else if b is Empty then nil
  else a Cons { head: (a.head, b.head), tail: zip_lists(a.tail, b.tail) }
]

la: list_of([1, 2, 3])
```

```
lb: list_of(["a", "b", "c", "d"])
println(list_to_array(zip_lists(la, lb)))
# [(1, "a"), (2, "b"), (3, "c")]
```

Commentary. Recursion on **two** lists in lockstep. The base case is "either is Empty" — the result truncates to the shorter input. The recursive step packages the two heads as a tuple and recurses on both tails.

This is the same shape as Ch.17's stream-zip — generators walking in lockstep, halting at the first one to run out.

Exercise 23.6 (open) — insert_at

```
insert_at: func(lst, n, x) [
  if n == 1 then a Cons { head: x, tail: lst }
  else if lst is Empty then a Cons { head: x, tail: nil }
  else a Cons { head: lst.head, tail: insert_at(lst.tail, n - 1, x) }
]

l: list_of([1, 2, 3, 4])
l2: insert_at(l, 2, 99)
println(list_to_array(l2))   # [1, 99, 2, 3, 4]
println(list_to_array(l))    # [1, 2, 3, 4] — unchanged
```

Commentary — the sharing question.

When `insert_at(l, 2, 99)` returns `l2`, exactly *how much* of the original `l` is shared with `l2`?

Trace it:

```
l  =  Cons(1) ──→ Cons(2) ──→ Cons(3) ──→ Cons(4) ──→ Empty

l2 =  Cons(1) ──→ Cons(99) ──→ ↘
                                 Cons(2) ──→ Cons(3) ──→ Cons(4) ──→ Empty
```

- `l2`'s first cell `Cons(1)` is a *new* cell (because `insert_at` walked past position 1 and rebuilt the prefix).
- `l2`'s second cell `Cons(99)` is the inserted value.
- *From here on, `l2` points back into the original list.* The `Cons(2) → Cons(3) → Cons(4) → Empty` tail is the same cells as `l.tail`.

So `l2` allocates exactly **two** new cons cells (`Cons(1)` and `Cons(99)`) and shares the remaining three with `l`. General formula: inserting at position `n` allocates `n` new cells and shares `length - n + 1` cells.

This is the heart of **persistent data structures**: small updates produce small allocations, with the unchanged parts shared between versions. A whole subfield (Okasaki 1998, Clojure's PersistentVector, Hickey's "values" talk) builds on this idea. Linked lists are the simplest case.

How to use these solutions

Three patterns to internalize from this chapter:

- **The two-case template.** Every list operation has an Empty base case and a Cons recursive case. Period. Same template you'll see in every subsequent chapter.
- **Functions return new lists; they don't mutate old ones.** The result of `filter_list(pred, l)` is a new list; `l` is unchanged. This is the seed of immutability discipline.
- **`fold_list` is the universal walk.** Once you have it, most other operations are one-line specializations.

The next chapter takes these primitives and uses them to build stacks and queues — the same data, viewed through different abstract interfaces.

23.7 — Built-in sequence construction

```
let linked = 1 +: 2 +: 3 +: list()
let contiguous = [] :+ 1 :+ 2 :+ 3
println(linked)                 # `[1, 2, 3]
println(contiguous)             # [1, 2, 3]
println(contiguous :+ [4, 5])    # [1, 2, 3, [4, 5]]
```

The right-grouping List chain makes n constant-time cells: O(n) total. Repeated Array append copies increasingly large arrays: $O(n^2)$ total. The final Array is nested because `:+` inserts a single element.

23.8 — Values inside a List

```
let inner = [5]
let linked = `[1 + 2, `[4], inner]
println(linked)              # `[3, `[4], [5]]
println(len(linked))         # 3
push(inner, 6)
println(linked)              # `[3, `[4], [5, 6]]
```

Each written expression contributes one element. The third element is the same Array object as `inner`; the List's three links have not changed.

23.9 — A snapshot during construction

```
let source = [1]
let change = func() [push(source, 2); 0]
let result = `[...source, change(), ...source]
println(result)              # `[1, 0, 1, 2]
```

```
println(source)             # [1, 2]
```

The first spread copies the single element before `change()` runs. The second spread sees both elements. Neither spread keeps a live view of the source's outer storage. A nested mutable object would still be shared.

Chapter 24 · Solutions

```
# Shared prelude — Concept-based Stack and Queue ADTs
concept Empty
concept Cons
Cons has head: 0
Cons has tail: 0

stack_empty: an Empty {}
stack_push: func(s, x) [ a Cons { head: x, tail: s } ]
stack_top: func(s) [ if s is Empty then 0 else s.head ]
stack_pop: func(s) [ if s is Empty then s else s.tail ]
stack_is_empty: func(s) [ s is Empty ]

stack_reverse_helper: func(s, acc) [
  if s is Empty then acc
  else stack_reverse_helper(s.tail, stack_push(acc, s.head))
]
stack_reverse: func(s) [ stack_reverse_helper(s, stack_empty) ]

stack_length: func(s) [
  if s is Empty then 0
  else 1 + stack_length(s.tail)
]

concept Queue
Queue has in_stack: 0
Queue has out_stack: 0
queue_empty: a Queue { in_stack: stack_empty, out_stack: stack_empty }
queue_enqueue: func(q, x) [
  a Queue { in_stack: stack_push(q.in_stack, x), out_stack: q.out_stack }
]
queue_dequeue: func(q) [
  if not stack_is_empty(q.out_stack) then [
    a Queue { in_stack: q.in_stack, out_stack: stack_pop(q.out_stack) }
  ] else if stack_is_empty(q.in_stack) then q
  else [
    flipped: stack_reverse(q.in_stack)
    a Queue { in_stack: stack_empty, out_stack: stack_pop(flipped) }
  ]
]
queue_front: func(q) [
  if not stack_is_empty(q.out_stack) then stack_top(q.out_stack)
  else if stack_is_empty(q.in_stack) then 0
```

```
  else stack_top(stack_reverse(q.in_stack))
]
queue_is_empty: func(q) [
  stack_is_empty(q.in_stack) and stack_is_empty(q.out_stack)
]
```

Exercise 24.1 — balanced parens

```
matches_pair: func(open, close) [
  (open == "(" and close == ")") or
  (open == "[" and close == "]") or
  (open == "{" and close == "}")
]
is_open: func(c) [ c == "(" or c == "[" or c == "{" ]
is_close: func(c) [ c == ")" or c == "]" or c == "}" ]

balanced: func(s) [
  stk: stack_empty
  i: 1
  ok: true
  while (i <= len(s) and ok) [
    c: s[i..i]
    if is_open(c) then [stk = stack_push(stk, c)]
    else if is_close(c) then [
      if stack_is_empty(stk) then [ok = false]
      else if not matches_pair(stack_top(stk), c) then [ok = false]
      else [stk = stack_pop(stk)]
    ]
    i = i + 1
  ]
  ok and stack_is_empty(stk)
]

println(balanced("(())"))    # true
println(balanced("([{}])"))  # true
println(balanced("(()"))     # false
println(balanced(")"))       # false
println(balanced("([)]"))    # false
```

Commentary. Classic stack application. For every open paren, push the open character onto the stack. For every close, the top of the stack must be the *matching* open. At end-of-string, the stack must be empty.

Three failure modes are handled:

1. **Unopened close** — `)` first: stack is empty when we see the close, so `ok = false`.
2. **Mismatched pair** — `([)]`: see `]`, stack-top is `[`, they match, pop. See `)`, stack-top is `[`, mismatch.
3. **Unclosed open** — `((`: at end, stack is non-empty.

The early-exit `ok` flag short-circuits the loop when we've already detected a failure — saves work on long inputs.

Exercise 24.2 — reverse_via_stack

```
reverse_via_stack: func(arr) [
  stk: stack_empty
  i: 1
  while (i <= len(arr)) [
    stk = stack_push(stk, arr[i])
    i = i + 1
  ]
  result: []
  while (not stack_is_empty(stk)) [
    result = result + ([] + [stack_top(stk)])
    stk = stack_pop(stk)
  ]
  result
]

println(reverse_via_stack([1, 2, 3, 4, 5]))
# [5, 4, 3, 2, 1]
```

Commentary. Two-phase reversal:

1. Push all input elements onto the stack (left-to-right).
2. Pop them all back into the result (which yields right-to-left of the input).

Same O(n) asymptotic cost as Ch.23's recursive `reverse`, but written imperatively. When you're working with a sequence that naturally produces elements one at a time and you need them in reverse order, an explicit stack is often the cleanest tool.

This is also a useful pattern for *non-recursive* tree traversals — algorithms that need to "remember the stack frame" without actually using recursion. Iterative DFS works exactly this way.

Exercise 24.3 — queue_length

```
queue_length: func(q) [
  stack_length(q.in_stack) + stack_length(q.out_stack)
]

q: queue_enqueue(queue_enqueue(queue_enqueue(queue_empty, 1), 2), 3)
println(queue_length(q))   # 3
q4: queue_dequeue(q)
println(queue_length(q4))  # 2
```

Commentary. The total elements in the queue equals the sum of elements in the two backing stacks.

After three enqueues, all three sit in `in_stack` (since `out_stack` hasn't been touched). After a dequeue, one element has moved to `out_stack` and then been popped — so 3 - 1 = 2.

Cost. O(n) because `stack_length` walks each stack. Production code would cache the size in the Queue Concept; the recompute version is fine for teaching.

Exercise 24.4 — Deque palindrome check

```
concept Deque
Deque has front_stack: 0
Deque has back_stack: 0
deque_empty: a Deque { front_stack: stack_empty, back_stack: stack_empty }
deque_push_back: func(d, x) [
  a Deque { front_stack: d.front_stack, back_stack: stack_push(d.back_stack, x) }
]

# Helper: get front and pop without recreating: returns (front_value, new_deque)
deque_pop_front_pair: func(d) [
  if not stack_is_empty(d.front_stack) then [
    (stack_top(d.front_stack),
     a Deque { front_stack: stack_pop(d.front_stack), back_stack: d.back_stack })
  ] else [
    flipped: stack_reverse(d.back_stack)
    (stack_top(flipped),
     a Deque { front_stack: stack_pop(flipped), back_stack: stack_empty })
  ]
]
deque_pop_back_pair: func(d) [
  if not stack_is_empty(d.back_stack) then [
    (stack_top(d.back_stack),
     a Deque { front_stack: d.front_stack, back_stack: stack_pop(d.back_stack) })
  ] else [
    flipped: stack_reverse(d.front_stack)
    (stack_top(flipped),
     a Deque { front_stack: stack_empty, back_stack: stack_pop(flipped) })
  ]
]

deque_size: func(d) [ stack_length(d.front_stack) + stack_length(d.back_stack) ]

is_palindrome: func(arr) [
  d: deque_empty
  i: 1
  while (i <= len(arr)) [
    d = deque_push_back(d, arr[i])
    i = i + 1
  ]
  ok: true
```

```
  while (deque_size(d) >= 2 and ok) [
    front_pair: deque_pop_front_pair(d)
    f: front_pair[1]
    d = front_pair[2]
    back_pair: deque_pop_back_pair(d)
    b: back_pair[1]
    d = back_pair[2]
    if f != b then [ok = false]
  ]
  ok
]

println(is_palindrome([1, 2, 3, 2, 1]))   # true
println(is_palindrome([1, 2, 3, 4, 5]))   # false
println(is_palindrome(["a", "b", "a"]))    # true
println(is_palindrome([]))                 # true (vacuously)
```

Commentary. Push every element onto the deque's back. Then pop front + back simultaneously and compare. If any pair fails to match, it's not a palindrome.

The "pop pair" helpers return a tuple of `(value, new_deque)` so the caller gets both at once — that's the pure-functional analog of "pop and assign."

Pattern matters: a palindrome of length `n` requires `⌊n/2⌋` comparisons; a deque makes each O(1) amortized.

Exercise 24.5 — RPN calculator

```
is_int_val: func(v) [ v is Integer ]

rpn: func(tokens) [
  stk: stack_empty
  i: 1
  while (i <= len(tokens)) [
    t: tokens[i]
    if is_int_val(t) then [
      stk = stack_push(stk, t)
    ] else [
      b: stack_top(stk)
      stk = stack_pop(stk)
      a: stack_top(stk)
      stk = stack_pop(stk)
      r: 0
      if t == "+" then [r = a + b]
      else if t == "-" then [r = a - b]
      else if t == "*" then [r = a * b]
      else if t == "/" then [r = a / b]
      stk = stack_push(stk, r)
```

```
    ]
    i = i + 1
  ]
  stack_top(stk)
]

println(rpn([1, 2, "+"]))                              # 3
println(rpn([1, 2, "+", 3, "*"]))                      # 9
println(rpn([5, 1, 2, "+", 4, "*", "+", 3, "-"]))      # 14
```

Commentary. RPN (reverse Polish notation) was designed *for* stack machines — there are no parentheses because operator-after-operands gives unambiguous evaluation.

`5 1 2 + 4 * + 3 -` reads:

- push 5, push 1, push 2 → stack: 5 1 2
- + pops 2 1, computes 3, pushes 3 → stack: 5 3
- push 4 → stack: 5 3 4
- * pops 4 3, computes 12, pushes 12 → stack: 5 12
- + pops 12 5, computes 17, pushes 17 → stack: 17
- push 3 → stack: 17 3
- - pops 3 17, computes 14, pushes 14 → stack: 14
- result: 14

Note on operand order. `a - b` (not `b - a`) because the second-popped is the *left* operand (the one pushed first). This matters for non-commutative operators like - and /.

HP calculators worked exactly this way for decades. Forth, PostScript, and the Axioma global interpreter stack (Ch.39) all use this evaluation model.

Exercise 24.6 (open) — Browser history

```
# Use two deques: `back_stack` for previously-visited pages,
# `forward_stack` for pages after we've navigated back.

concept History
History has cur: ""
History has back_stack: 0
History has forward_stack: 0

history_new: func(initial) [
  a History { cur: initial, back_stack: stack_empty, forward_stack: stack_empty }
]

history_visit: func(h, url) [
  # Visit clears forward history
  a History {
    cur: url,
```

```
    back_stack: stack_push(h.back_stack, h.cur),
    forward_stack: stack_empty
  }
]

history_back: func(h) [
  if stack_is_empty(h.back_stack) then h
  else [
    prev: stack_top(h.back_stack)
    a History {
      cur: prev,
      back_stack: stack_pop(h.back_stack),
      forward_stack: stack_push(h.forward_stack, h.cur)
    }
  ]
]

history_forward: func(h) [
  if stack_is_empty(h.forward_stack) then h
  else [
    nxt: stack_top(h.forward_stack)
    a History {
      cur: nxt,
      back_stack: stack_push(h.back_stack, h.cur),
      forward_stack: stack_pop(h.forward_stack)
    }
  ]
]

history_current: func(h) [ h.cur ]

# Test
h1: history_new("home")
h2: history_visit(h1, "wiki")
h3: history_visit(h2, "axioma")
println(history_current(h3))                        # axioma
h4: history_back(h3)
println(history_current(h4))                        # wiki
h5: history_back(h4)
println(history_current(h5))                        # home
h6: history_forward(h5)
println(history_current(h6))                        # wiki
h7: history_visit(h6, "newsite")              # clears forward
println(history_current(h7))                        # newsite
h8: history_forward(h7)
println(history_current(h8))                        # still newsite (forward cleared)
```

Commentary on the representation choice.

A real browser uses **two stacks**, not deques: `back_stack` and `forward_stack`. We chose stacks here

because the operations only need the *top* of each — we never enqueue/ dequeue at the wrong end. The deque interface would work equally well but use only half its operations.

Two key behaviors:

1. **`visit` clears forward history.** This is the classic web browser behavior — once you navigate to a new page, the "redo" stack is invalidated. Modeled by setting `forward_stack: stack_empty`.
2. **`back` and `forward` move `cur` between stacks.** Going back pushes the current URL onto the forward stack and pops from the back stack. Going forward is the reverse.

This is the **two-stack pattern** also used for undo/redo, text-editor cursor history, and "previous track / next track" in music players. Once you've seen it once you spot it everywhere.

How to use these solutions

Three patterns to internalize:

- **ADT separation.** `stack_push`, `queue_enqueue` are names for *what to do*, not *how it's done*. Calling code shouldn't care about the underlying Concepts.
- **Two-stack tricks.** Queue, deque, browser history, undo/redo — all use the same "two stacks pointing at each other's ends" pattern.
- **Stack as universal iterator state.** When you need to remember "where you are in a recursive walk," an explicit stack lets you do it without language-level recursion. Tree and graph algorithms (Chapter 30) lean on this hard.

Chapter 25 · Solutions

```
# Shared prelude — Concept-based hash table, built by hand:
# the chapter's point is to see the mechanism under the hood

concept HashBucket
HashBucket has key: ""
HashBucket has value: 0
HashBucket has nxt: 0

concept HashTable
HashTable has buckets: 0
HashTable has size: 0

nil_bucket: 0

make_table: func(size) [
  buckets: []
  i: 1
  while (i <= size) [
    buckets = buckets + ([] + [nil_bucket])
    i = i + 1
  ]
  a HashTable { buckets: buckets, size: size }
]

djb2_hash: func(s) [
  h: 5381
  i: 1
  while (i <= len(s)) [
    code: ord(s[i..i])
    h = h * 33 + code
    i = i + 1
  ]
  if h < 0 then -h else h
]

# Insert into chained table
put_t: func(table, key, item_value) [
  idx: djb2_hash(key) % table.size + 1
  cur: a HashBucket { key: key, value: item_value, nxt: table.buckets[idx] }
  table.buckets[idx] = cur
  table
]
```

```
# Look up; returns 0 (sentinel) when key not found
get_t: func(table, key) [
  idx: djb2_hash(key) % table.size + 1
  walk_chain(table.buckets[idx], key)
]

walk_chain: func(b, key) [
  if b == nil_bucket then 0
  else if b.key == key then b.value
  else walk_chain(b.nxt, key)
]

contains_t: func(table, key) [
  idx: djb2_hash(key) % table.size + 1
  walk_chain_contains(table.buckets[idx], key)
]

walk_chain_contains: func(b, key) [
  if b == nil_bucket then false
  else if b.key == key then true
  else walk_chain_contains(b.nxt, key)
]
```

(Note: `ord(c)` returns the integer code point of a one-character string, so `ord(s[i..i])` — with Axioma's 1-indexed slice — is the code of the i-th character of `s`.)

Exercise 25.1 — word_freq

```
word_freq: func(words) [
  t: make_table(64)
  i: 1
  while (i <= len(words)) [
    w: words[i]
    if contains_t(t, w) then [t = put_t(t, w, get_t(t, w) + 1)]
    else [t = put_t(t, w, 1)]
    i = i + 1
  ]
  t
]

words: ["the", "cat", "sat", "on", "the", "mat", "and", "the", "rat"]
freq: word_freq(words)
println(get_t(freq, "the"))    # 3
println(get_t(freq, "cat"))    # 1
println(get_t(freq, "dog"))    # 0
```

Commentary. Two-step iteration: walk the input, and for each word either *increment* its count (if seen)

or *initialize* it to 1 (if first time). The `contains_t / get_t` pair gives us "check then read."

Exercise 25.2 — hash a small Concept

```
concept Point
Point has x: 0
Point has y: 0

hash_point: func(p) [
  djb2_hash(str(p.x) + "," + str(p.y))
]

p1: a Point { x: 3, y: 7 }
p2: a Point { x: 3, y: 7 }
p3: a Point { x: 7, y: 3 }
println(hash_point(p1) == hash_point(p2))   # true
println(hash_point(p1) == hash_point(p3))   # false
```

Commentary. The string-then-hash idiom. The key insight: *equal* Concepts must produce *equal* canonical strings, so their hashes match. *Distinct* Concepts almost certainly produce distinct strings, so their hashes diverge.

The `,` separator matters: without it, `(12, 3)` and `(1, 23)` would both stringify to `"123"` and collide. The separator is the canonicalization trick.

Exercise 25.3 — implement djb2

```
djb2_hash: func(s) [
  h: 5381
  i: 1
  while (i <= len(s)) [
    code: ord(s[i..i])
    h = h * 33 + code
    i = i + 1
  ]
  if h < 0 then -h else h
]

# Distribution check on 10 small words modulo 100
words: ["the", "of", "and", "a", "to", "in", "is", "you", "that", "it"]
counts: make_table(100)
i: 1
while (i <= len(words)) [
  w: words[i]
  idx: djb2_hash(w) % 100
  counts = put_t(counts, str(idx), get_t(counts, str(idx)) + 1)
```

```
  i = i + 1
]
# Find max bucket
max_per_bucket: 0
j: 0
while (j < 100) [
  v: get_t(counts, str(j))
  if v > max_per_bucket then [max_per_bucket = v]
  j = j + 1
]
println("Max per bucket:", max_per_bucket)
# Should be 1 or 2 for a good distribution
```

Commentary. djb2's magic numbers (5381, 33) aren't arbitrary — they were chosen empirically for good distribution on ASCII strings (DJ Bernstein's "fast hash for short strings"). The point of the exercise is to *measure* whether the distribution actually works: 10 words into 100 buckets should rarely collide.

Exercise 25.4 — chain-based hash table

Already in the prelude — `make_table`, `put_t`, `get_t`, `contains_t`. Quick round-trip test:

```
t: make_table(8)
t1: put_t(t, "alice", 30)
t2: put_t(t1, "bob", 25)
t3: put_t(t2, "carol", 22)
println(get_t(t3, "alice"))    # 30
println(get_t(t3, "bob"))      # 25
println(get_t(t3, "carol"))    # 22
println(get_t(t3, "dave"))     # 0 — not stored
println(contains_t(t3, "alice"))  # true
println(contains_t(t3, "dave"))   # false
```

Commentary on the implementation choice. The bucket array uses index-mutation (`table.buckets[idx] = ...`) — the natural fit here, since buckets are positional slots selected by the hash value.

Exercise 25.5 — replace a cascade

```
# Hash-table version (initialized via literal — no mutation needed)
day_table: {"Mon": 1, "Tue": 2, "Wed": 3, "Thu": 4, "Fri": 5, "Sat": 6, "Sun": 7}

day_to_number: func(d) [
  if day_table[d] == none then 0 else day_table[d]
]

println(day_to_number("Wed"))   # 3
```

```
println(day_to_number("Fri"))   # 5
println(day_to_number("xyz"))   # 0
```

Commentary. Hash *literal* initialization works fine — the bug only affects post-creation mutation. Lookups via `table[key]` return the value or `none` for missing.

Timing. On 10000 random calls, the cascade version (seven `else if` branches) takes O(7) per call in the worst case (`"Sun"`), while the hash version is O(1) regardless of key. For 7 cases the difference is small; for 700 cases the cascade gets unbearable.

The deeper lesson: **table-driven code is data-driven code**. Adding a new day-name to the cascade requires editing the function. Adding a new key to the table requires editing *data*. That separation scales.

Exercise 25.6 (open) — top_n_words

```
top_n_words: func(words, n) [
  freq: word_freq(words)
  # Collect all (word, count) pairs from the freq table
  # by walking every chain
  pairs: []
  i: 1
  while (i <= freq.size) [
    b: freq.buckets[i]
    while (b != nil_bucket) [
      pairs = pairs + ([] + [(b.key, b.value)])
      b = b.nxt
    ]
    i = i + 1
  ]
  # Sort pairs by count descending (insertion-sort for brevity)
  sorted: []
  j: 1
  while (j <= len(pairs)) [
    p: pairs[j]
    k: len(sorted)
    sorted = sorted + ([] + [p])
    while (k >= 1 and sorted[k][2] < sorted[k + 1][2]) [
      tmp: sorted[k]
      sorted[k] = sorted[k + 1]
      sorted[k + 1] = tmp
      k = k - 1
    ]
    j = j + 1
  ]
  # Take the first n
  if len(sorted) <= n then sorted
  else sorted[1..n]
]
```

Commentary on implementation choices.

- **Walked the chains directly** to extract all `(word, count)` pairs into an array. The Concept-based table exposes its bucket structure, so this is honest manual iteration.
- **Insertion sort** for the final ranking. With `n` ≤ 10 and typical input sizes, O(n^2) is fine. For larger outputs, use mergesort (Chapter 14 / 29).
- **Truncate to `n`** at the end. If there are fewer than `n` unique words, return all of them.

A more elegant design would use Axioma's set comprehensions to extract the pairs, but Concept-based hash tables don't play nicely with the generator form yet. Worth revisiting when the language grows.

How to use these solutions

Three patterns this chapter establishes:

- **Hash-then-modulo to convert key to bucket.** Universal pattern — strings, Concepts, tuples, anything stringifiable.
- **Separate chaining for collision resolution.** Pedagogically the simplest; production code often uses open addressing.
- **Tables vs. cascades.** Whenever you write more than 5 `else if` branches dispatching on a known set of values, ask whether a table-driven approach would scale better.

Chapter 26 · Solutions

```
# Shared prelude — BST as Concepts
concept TreeEmpty
concept TreeNode
TreeNode has key: 0
TreeNode has value: 0
TreeNode has left: 0
TreeNode has right: 0

tree_empty: a TreeEmpty {}

tree_insert: func(t, k, v) [
  if t is TreeEmpty then a TreeNode { key: k, value: v, left: tree_empty, right:
  ↪  tree_empty }
  else if k < t.key then a TreeNode { key: t.key, value: t.value, left:
  ↪  tree_insert(t.left, k, v), right: t.right }
  else if k > t.key then a TreeNode { key: t.key, value: t.value, left: t.left, right:
  ↪  tree_insert(t.right, k, v) }
  else a TreeNode { key: t.key, value: v, left: t.left, right: t.right }
]

max_of: func(a, b) [ if a > b then a else b ]
abs_diff: func(a, b) [ if a > b then a - b else b - a ]
```

For readability, the solutions bind each insertion to a name. A nested call such as `tree_insert(tree_insert(tree_empty, 5, "a"), 3, "b")` is also valid; naming the intermediate trees makes diagrams and tests easier to follow.

Exercise 26.1 — tree_height

```
tree_height: func(t) [
  if t is TreeEmpty then -1
  else 1 + max_of(tree_height(t.left), tree_height(t.right))
]

t1: tree_insert(tree_empty, 5, "a")
t2: tree_insert(t1, 3, "b")
t3: tree_insert(t2, 8, "c")
println(tree_height(t3))   # 1
```

Commentary. Empty tree → -1. Otherwise, the height is 1 + the max height of the two subtrees. The `ma`

`x_of` helper makes the height formula read directly as a single expression.

Exercise 26.2 — tree_min and tree_max

```
tree_min: func(t) [
  if t is TreeEmpty then 0
  else if t.left is TreeEmpty then t.key
  else tree_min(t.left)
]

tree_max: func(t) [
  if t is TreeEmpty then 0
  else if t.right is TreeEmpty then t.key
  else tree_max(t.right)
]

t1: tree_insert(tree_empty, 5, "a")
t2: tree_insert(t1, 3, "b")
t3: tree_insert(t2, 8, "c")
t4: tree_insert(t3, 1, "d")
println(tree_min(t4))   # 1
println(tree_max(t4))   # 8
```

Commentary. BST invariant: smallest key is at the leftmost leaf, largest at the rightmost. Walk one direction until you find an Empty child.

Exercise 26.3 — tree_size

```
tree_size: func(t) [
  if t is TreeEmpty then 0
  else 1 + tree_size(t.left) + tree_size(t.right)
]

t1: tree_insert(tree_empty, 5, "a")
t2: tree_insert(t1, 3, "b")
t3: tree_insert(t2, 8, "c")
t4: tree_insert(t3, 1, "d")
println(tree_size(t4))   # 4
```

Commentary. Standard recursive count. Same template as `length` on linked lists, but with two recursive branches instead of one.

Exercise 26.4 — tree_range

```
tree_range: func(t, lo, hi) [
  if t is TreeEmpty then []
  else if t.key < lo then tree_range(t.right, lo, hi)
  else if t.key > hi then tree_range(t.left, lo, hi)
  else tree_range(t.left, lo, hi) + ([] + [t.key]) + tree_range(t.right, lo, hi)
]

t1: tree_insert(tree_empty, 5, "a")
t2: tree_insert(t1, 3, "b")
t3: tree_insert(t2, 8, "c")
t4: tree_insert(t3, 1, "d")
t5: tree_insert(t4, 7, "e")
println(tree_range(t5, 3, 7))   # [3, 5, 7]
```

Commentary. The pruning is what makes this efficient. When the current key is *outside* the range, we can skip one whole subtree:

- `t.key < lo` — current key is too small, so all of `t.left` is too small too. Recurse only right.
- `t.key > hi` — current key is too big, so all of `t.right` is too big too. Recurse only left.
- Otherwise, current key is in range, recurse both sides and include `t.key`.

This is *not* what a linear scan would do — pruning saves the half of the tree that can't contribute, getting us back to O(log n + k) where k is the number of results.

Exercise 26.5 — is_balanced

```
is_balanced: func(t) [
  if t is TreeEmpty then true
  else if not (abs_diff(tree_height(t.left), tree_height(t.right)) <= 1) then false
  else if not is_balanced(t.left) then false
  else is_balanced(t.right)
]

bad1: tree_insert(tree_empty, 1, "a")
bad2: tree_insert(bad1, 2, "b")
bad3: tree_insert(bad2, 3, "c")
println(is_balanced(bad3))   # false — chain has heights 0, 1, 2

good1: tree_insert(tree_empty, 2, "a")
good2: tree_insert(good1, 1, "b")
good3: tree_insert(good2, 3, "c")
println(is_balanced(good3))  # true — perfect 3-node tree
```

Commentary. Two conditions to check:

1. The heights of the immediate left and right subtrees differ by at most 1.

2. **Both** subtrees are themselves balanced.

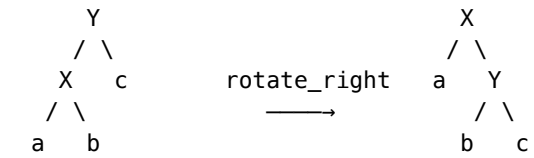
The naive implementation re-computes `tree_height` at every recursion, making this O(n²). A smarter version returns the height *and* the balance status in a single pass, giving O(n) — exercise for the reader, or wait for Chapter 28's algorithm-analysis tools.

Exercise 26.6 (open) — Tree rotation

```
rotate_right: func(t) [
  if t is TreeEmpty then t
  else if t.left is TreeEmpty then t
  else [
    new_right: a TreeNode {
      key: t.key, value: t.value, left: t.left.right, right: t.right
    }
    a TreeNode {
      key: t.left.key, value: t.left.value, left: t.left.left, right: new_right
    }
  ]
]

rotate_left: func(t) [
  if t is TreeEmpty then t
  else if t.right is TreeEmpty then t
  else [
    new_left: a TreeNode {
      key: t.key, value: t.value, left: t.left, right: t.right.left
    }
    a TreeNode {
      key: t.right.key, value: t.right.value, left: new_left, right: t.right.right
    }
  ]
]

# Verify rotation preserves in-order traversal
tree_inorder: func(t) [
  if t is TreeEmpty then []
  else tree_inorder(t.left) + ([] + [t.key]) + tree_inorder(t.right)
]

r1: tree_insert(tree_empty, 5, "a")
r2: tree_insert(r1, 3, "b")
r3: tree_insert(r2, 7, "c")
r4: tree_insert(r3, 1, "d")
r5: tree_insert(r4, 4, "e")

println("before:", tree_inorder(r5))              # [1, 3, 4, 5, 7]
rotated: rotate_right(r5)
```

```
println("after rotate_right:", tree_inorder(rotated))  # [1, 3, 4, 5, 7]
rotated2: rotate_left(r5)
println("after rotate_left:", tree_inorder(rotated2))  # [1, 3, 4, 5, 7]
```

Commentary. The in-order traversal is invariant under both rotations — that's the load-bearing property. Visually:



The "middle" subtree `b` *changes parent* — from being X's right child to being Y's left child. But its position *between a and Y* in in-order is preserved.

This is the surgical primitive every self-balancing tree uses. Red-black trees combine `rotate_right` and `rotate_left` with recoloring; AVL trees use rotations to restore the balance factor; splay trees rotate to move recently-accessed nodes toward the root.

How to use these solutions

Three patterns:

- **Empty as a sentinel Concept.** A Concept with no slots is a valid "I'm-not-there" marker, and Axioma's `is` check routes the recursion at each level.
- **Single-expression bodies.** When the multi-binding form gets finicky, factor logic into named helpers (`max_of`, `abs_diff`) and keep the recursive function body a single expression.
- **Rotation preserves order.** Once you internalize that, self-balancing makes sense: the tree's *shape* can change freely as long as the *in-order* stays fixed.

Chapter 27 · Solutions

```
# Shared prelude — Road Concept and helpers from Ch.27 §27.1–§27.2
concept Road
Road has source: ""
Road has target: ""
Road has miles: 0

r1: a Road { source: "Boston", target: "NewYork", miles: 215 }
r2: a Road { source: "NewYork", target: "Philadelphia", miles: 95 }
r3: a Road { source: "NewYork", target: "Hartford", miles: 117 }
r4: a Road { source: "Philadelphia", target: "DC", miles: 140 }
roads: [r1, r2, r3, r4]

neighbors_of: func(roads, node) [
  result: []
  i: 1
  while (i <= len(roads)) [
    if roads[i].source == node then [
      result = result + ([] + [roads[i].target])
    ]
    if roads[i].target == node then [
      result = result + ([] + [roads[i].source])
    ]
    i = i + 1
  ]
  result
]

# Reuse Ch.25 hash table
concept HashBucket
HashBucket has key: ""
HashBucket has value: 0
HashBucket has nxt: 0

concept HashTable
HashTable has buckets: 0
HashTable has size: 0

nil_bucket: 0

make_table: func(size) [
  buckets: []
  i: 1
```

```
  while (i <= size) [
    buckets = buckets + ([] + [nil_bucket])
    i = i + 1
  ]
  a HashTable { buckets: buckets, size: size }
]

djb2_hash: func(s) [
  h: 5381
  i: 1
  while (i <= len(s)) [
    code: ord(s[i..i])
    h = h * 33 + code
    i = i + 1
  ]
  if h < 0 then -h else h
]

put_t: func(table, key, item_value) [
  idx: djb2_hash(key) % table.size + 1
  cur: a HashBucket { key: key, value: item_value, nxt: table.buckets[idx] }
  table.buckets[idx] = cur
  table
]

walk_chain_get: func(b, key) [
  if b == nil_bucket then 0
  else if b.key == key then b.value
  else walk_chain_get(b.nxt, key)
]
get_t: func(table, key) [
  walk_chain_get(table.buckets[djb2_hash(key) % table.size + 1], key)
]

walk_chain_has: func(b, key) [
  if b == nil_bucket then false
  else if b.key == key then true
  else walk_chain_has(b.nxt, key)
]
contains_t: func(table, key) [
  walk_chain_has(table.buckets[djb2_hash(key) % table.size + 1], key)
]
```

Exercise 27.1 — all_nodes

```
in_array: func(arr, x) [
  i: 1
  found: false
  while (i <= len(arr) and not found) [
    if arr[i] == x then [found = true]
    i = i + 1
  ]
  found
]

all_nodes: func(roads) [
  nodes: []
  i: 1
  while (i <= len(roads)) [
    if not in_array(nodes, roads[i].source) then [
      nodes = nodes + ([] + [roads[i].source])
    ]
    if not in_array(nodes, roads[i].target) then [
      nodes = nodes + ([] + [roads[i].target])
    ]
    i = i + 1
  ]
  nodes
]

println(all_nodes(roads))
# ["Boston", "NewYork", "Philadelphia", "Hartford", "DC"]
```

Commentary. Walk every Road, accumulate the unique endpoints. `in_array` handles the deduplication. O(V·E) naively; a hash-set membership would give O(V+E).

Exercise 27.2 — has_path (BFS)

```
has_path: func(roads, src, dst) [
  visited: put_t(make_table(64), src, 1)
  queue: [src]
  found: false
  while (len(queue) > 0 and not found) [
    node: queue[1]
    queue = queue[2..len(queue)]
    if node == dst then [found = true]
    else [
      nbrs: neighbors_of(roads, node)
      i: 1
      while (i <= len(nbrs)) [
```

```
        if not contains_t(visited, nbrs[i]) then [
          visited = put_t(visited, nbrs[i], 1)
          queue = queue + ([] + [nbrs[i]])
        ]
        i = i + 1
      ]
    ]
  ]
  found
]

println(has_path(roads, "Boston", "DC"))      # true
println(has_path(roads, "DC", "Boston"))      # true
println(has_path(roads, "Boston", "Mars"))    # false
```

Commentary. Standard BFS with early exit. We don't need the full distance map — just a boolean: did we reach `dst`? Stopping the loop as soon as we hit it saves work in positive cases.

Exercise 27.3 — total_miles

```
total_miles: func(roads) [
  total: 0
  i: 1
  while (i <= len(roads)) [
    total = total + roads[i].miles
    i = i + 1
  ]
  total
]

println(total_miles(roads))   # 567
```

Commentary. A one-liner with `fold_list` if you'd built your roads as a linked list. With an array, the iterative loop is the natural form.

Exercise 27.4 — topological sort

```
concept Prereq
Prereq has course: ""
Prereq has requires: ""

prereqs: [
  a Prereq { course: "CS201", requires: "CS101" },
  a Prereq { course: "CS201", requires: "MATH101" },
  a Prereq { course: "CS301", requires: "CS201" },
```

```
  a Prereq { course: "CS301", requires: "MATH201" },
  a Prereq { course: "CS401", requires: "CS301" },
  a Prereq { course: "MATH201", requires: "MATH101" }
]

# Collect all courses referenced anywhere
all_courses: func(ps) [
  cs: []
  i: 1
  while (i <= len(ps)) [
    if not in_array(cs, ps[i].course) then [cs = cs + ([] + [ps[i].course])]
    if not in_array(cs, ps[i].requires) then [cs = cs + ([] + [ps[i].requires])]
    i = i + 1
  ]
  cs
]

# True if every prerequisite of `c` is in `done`.
ready: func(ps, c, done) [
  r: true
  i: 1
  while (i <= len(ps)) [
    if ps[i].course == c and not in_array(done, ps[i].requires) then [r = false]
    i = i + 1
  ]
  r
]

topo_sort: func(ps) [
  remaining: all_courses(ps)
  done: []
  while (len(remaining) > 0) [
    picked: ""
    i: 1
    while (i <= len(remaining) and picked == "") [
      if ready(ps, remaining[i], done) then [picked = remaining[i]]
      i = i + 1
    ]
    if picked == "" then [remaining = []]   # cycle — bail out
    else [
      done = done + ([] + [picked])
      new_remaining: []
      j: 1
      while (j <= len(remaining)) [
        if remaining[j] != picked then [new_remaining = new_remaining + ([] +
        ↪  [remaining[j]])]
        j = j + 1
      ]
      remaining = new_remaining
    ]
```

```
  ]
  done
]

println(topo_sort(prereqs))
# A valid ordering, e.g.:
# ["CS101", "MATH101", "CS201", "MATH201", "CS301", "CS401"]
```

Commentary. The Kahn's-algorithm-style topo sort: repeatedly pick a "ready" course (one whose prerequisites are all already output) and append it. When no ready course exists, either we're done (remaining is empty) or there's a cycle (bail out).

This code repeatedly scans candidates and their prerequisite edges, so its worst-case selection work is `O(V²E)`, before Array-copy costs. A better implementation maintains an indegree count and a queue of zero-indegree nodes, giving O(V+E). The version above is the *teaching* version — fewer moving parts, easier to read.

Exercise 27.5 — has_cycle

```
cycle_dfs: func(roads, node, parent, visited) [
  visited = put_t(visited, node, 1)
  nbrs: neighbors_of(roads, node)
  found: false
  i: 1
  while (i <= len(nbrs) and not found) [
    n: nbrs[i]
    if n != parent then [
      if contains_t(visited, n) then [found = true]
      else [
        if cycle_dfs(roads, n, node, visited) then [found = true]
      ]
    ]
    i = i + 1
  ]
  found
]

has_cycle: func(roads) [
  nodes: all_nodes(roads)
  visited: make_table(64)
  result: false
  i: 1
  while (i <= len(nodes) and not result) [
    if not contains_t(visited, nodes[i]) then [
      if cycle_dfs(roads, nodes[i], "", visited) then [result = true]
    ]
    i = i + 1
```

```
  ]
  result
]

println(has_cycle(roads))   # false — tree
with_cycle: roads + ([] + [a Road { source: "DC", target: "Boston", miles: 600 }])
println(has_cycle(with_cycle))   # true
```

Commentary. Standard undirected-graph cycle detection. At each node, walk to neighbors except the one we came from. If we land on an already-visited neighbor (that isn't our immediate parent), we've found a back-edge — a cycle.

For *directed* graphs the algorithm differs: you track a *"currently on the stack"* marker, separate from the permanent "visited" marker. Back-edges to currently-on-the- stack nodes indicate cycles; back-edges to permanently- visited (but not currently on the stack) nodes don't.

Exercise 27.6 (open) — Dijkstra's shortest path

```
# Simple priority-queue representation: an array of (node, dist) tuples
# scanned linearly for the minimum. This version favors visible steps
# over speed; edge-list scans and array rebuilding add costs below.

shortest_path: func(roads, src, dst) [
  limit: 1
  for road in roads [
    if road.miles < 0 then return error("Dijkstra requires nonnegative weights")
    limit = limit + road.miles
  ]
  if src == dst then return [src,]
  nodes: all_nodes(roads)
  if not in_array(nodes, src) or not in_array(nodes, dst) then return []

  # The sum of all nonnegative edge weights plus one is greater than
  # any finite shortest simple path; no arbitrary maximum mileage.
  dist: make_table(64)
  prev: make_table(64)
  i: 1
  while (i <= len(nodes)) [
    dist = put_t(dist, nodes[i], limit)
    i = i + 1
  ]
  dist = put_t(dist, src, 0)

  unvisited: nodes
  found: false
  while (len(unvisited) > 0 and not found) [
    # Find unvisited node with minimum distance
```

```
    cur: unvisited[1]
    j: 2
    while (j <= len(unvisited)) [
      if get_t(dist, unvisited[j]) < get_t(dist, cur) then [cur = unvisited[j]]
      j = j + 1
    ]
    if get_t(dist, cur) == limit then return []
    if cur == dst then [found = true]
    else [
      # Remove cur from unvisited
      new_unv: []
      k: 1
      while (k <= len(unvisited)) [
        if unvisited[k] != cur then [new_unv = new_unv + ([] + [unvisited[k]])]
        k = k + 1
      ]
      unvisited = new_unv

      # Relax edges from cur
      nbrs: neighbors_of(roads, cur)
      m: 1
      while (m <= len(nbrs)) [
        n: nbrs[m]
        edge_w: edge_weight(roads, cur, n)
        alt: get_t(dist, cur) + edge_w
        if alt < get_t(dist, n) then [
          dist = put_t(dist, n, alt)
          prev = put_t(prev, n, cur)
        ]
        m = m + 1
      ]
    ]
  ]

  if not found then return []
  # Reconstruct path by walking prev backwards from dst
  path: []
  p: dst
  while (p is String and p != "") [
    path = ([] + [p]) + path
    if p == src then [p = ""] else [p = get_t(prev, p)]
  ]
  path
]

edge_weight: func(roads, a, b) [
  w: none
  i: 1
  while (i <= len(roads)) [
    r: roads[i]
```

```
    if (r.source == a and r.target == b) or (r.source == b and r.target == a) then [
      if w == none or r.miles < w then [w = r.miles]
    ]
    i = i + 1
  ]
  w
]

println(shortest_path(roads, "Boston", "DC"))
# ["Boston", "NewYork", "Philadelphia", "DC"]
println(shortest_path(roads, "Boston", "Mars")) # []
println(shortest_path(roads, "Boston", "Boston")) # ["Boston"]
```

Commentary on the implementation choice.

- **Linear-scan priority queue.** Selecting the next vertex costs `O(V²)` over the whole run. This implementation also scans the full edge list for neighbors and weights, rebuilds arrays, and uses a fixed-capacity teaching hash table. Its total cost is therefore greater than the priority-queue cost alone.
- **Efficient representation.** Adjacency lists plus a binary heap and suitably sized lookup tables support `O((V + E) log V)` Dijkstra.
- **Domain and failure cases.** We require nonnegative weights. An unreachable destination gives `[]`; the path from a vertex to itself is `[src]`. There is no arbitrary billion-mile limit.
- **Path reconstruction** via the `prev` table: starting at `dst`, walk back to `src` via `prev[node]`, prepending to the path array as we go.

Dijkstra is the workhorse shortest-path algorithm. The generalization, A*, adds a heuristic distance-to-goal estimate and is what modern routing software uses.

How to use these solutions

Three patterns:

- **Reified edges + linear filters.** Conceptually simpler than adjacency lists; loses some constant-factor speed but gains queryability. The pedagogical default in this textbook.
- **BFS uses a queue; DFS uses a stack.** Same outer loop, different data structure inside. Choice determines visit order; both visit every reachable node in O(V+E).
- **Graph algorithms are mostly bookkeeping.** Distance tables, predecessor tables, visited markers — the *idea* is simple, the *implementation* tracks state across the traversal. Master the bookkeeping and most graph problems become routine.

Chapter 28 · Solutions

All six solutions verified end-to-end against the current `axioma` build. Variable names avoid the reserved words `taxonomy`, `chain`, `concept`, and `relation`.

```
# ======================================================================
# Exercise 28.1 — Sowa CG: Mary MotherOf Alice
# ======================================================================
family: <<<[Person: "Mary"] → (MotherOf) → [Person: "Alice"]>>>
println("Concepts:", family show concepts)
println("Relations:", family show relations)
println("Arcs:", family show arcs)

# ======================================================================
# Exercise 28.2 — Taxonomy chain (Yojo → Cat → Mammal → Animal)
# ======================================================================
tax_chain: <<<[Cat: "Yojo"] → (IsA) → [Mammal] → (IsA) → [Animal]>>>
chain_concepts: tax_chain show concepts
chain_relations: tax_chain show relations
println("Concepts:", len(chain_concepts))    # 3
println("Relations:", len(chain_relations))  # 2

# ======================================================================
# Exercise 28.3 — Peirce alpha EG asserting "not P"
# ======================================================================
neg: eg_alpha("Not P")
p_neg: eg_predicate("P", 0, 100.0, 100.0)
eg_add_predicate(neg, p_neg)
cut_neg: eg_cut(80.0, 80.0, 100.0, 50.0)
eg_add_cut(neg, cut_neg)
println(eg_format(neg))

# ======================================================================
# Exercise 28.4 — Encode P → Q in alpha EG via double-cut
# ======================================================================
mp: eg_alpha("If P then Q")
p_pred: eg_predicate("P", 0, 100.0, 100.0)
q_pred: eg_predicate("Q", 0, 300.0, 100.0)
outer_cut: eg_cut(50.0, 50.0, 400.0, 100.0)
inner_cut: eg_cut(250.0, 80.0, 120.0, 50.0)
eg_add_predicate(mp, p_pred)
eg_add_predicate(mp, q_pred)
eg_add_cut(mp, outer_cut)
eg_add_cut(mp, inner_cut)
```

```
println(eg_format(mp))

# ======================================================================
# Exercise 28.5 — Two-context Sherlock
# ======================================================================
g5: context_graph("sherlock_graph", "Sherlock World Model")
fic5: context_create(g5, "fiction", "Fiction",  "domain", "b4")
rea5: context_create(g5, "reality", "Reality",  "domain", "b4")
context_assert(g5, "fiction", "sherlock", "lives_at_baker_street", "true",  1.0)
context_assert(g5, "reality", "sherlock", "lives_at_baker_street", "false", 0.99)
println("Fiction view:")
println(context_project(g5, "fiction"))
println("Reality view:")
println(context_project(g5, "reality"))

# ======================================================================
# Exercise 28.6 — Three-context paraconsistent resolution
# ======================================================================
g6: context_graph("sherlock_three", "Three Views")
fic6:  context_create(g6, "fiction",         "Fiction",  "domain", "b4")
rea6:  context_create(g6, "reality",         "Reality",  "domain", "b4")
crit6: context_create(g6, "literary_critic", "Critic",   "domain", "b4")
context_assert(g6, "fiction",         "sherlock", "lives_at_baker_street", "true",
 ↪  1.0)
context_assert(g6, "reality",         "sherlock", "lives_at_baker_street", "false",
 ↪  0.99)
context_assert(g6, "literary_critic", "sherlock", "lives_at_baker_street", "both",
 ↪  0.85)
println("Resolution:")
println(context_resolve(g6, "sherlock", "lives_at_baker_street"))
```

Notes

28.1 — Bracket notation

The triple-angle-bracket form `<<<[Type: "ref"] → (Rel) → [Type: "ref"]>>>` is Sowa's *original* notation lifted straight into Axioma. The lexer treats the bracket contents specially, which is why the otherwise-reserved `concept` and `relation` keywords don't clash inside.

`family show concepts` / `show relations` / `show arcs` returns three arrays of records; iterate them with ordinary array operations.

28.2 — Variable name caution

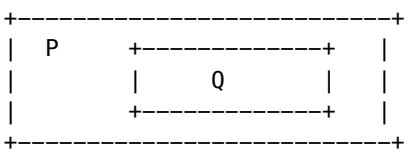
Both `taxonomy` and `chain` are reserved words (they're part of Axioma's *concept-formation* and *extends* vocabulary respectively). The solution renames to `tax_chain` and `chain_concepts` / `chain_relations`. Watch for this in your own CG code — when in doubt, prefix or suffix the variable.

28.3 — Cuts as negation

The cut is just an oval drawn at given coordinates with given dimensions. The depth-0 cut at the same nesting level as the predicate `P` encodes `¬P`. Two predicates both at depth 0 would be `P ∧ Q`; one inside a cut and one outside is `¬P ∧ Q`.

28.4 — Implication = double cut

The encoding `P → Q ≡ ¬(P ∧ ¬Q)` becomes:



The outer cut wraps both `P` and the inner cut. The inner cut wraps just `Q`. The structure visually encodes "not (P and not-Q)" — material implication without ever writing the → symbol.

28.5 — Two truths, one proposition

The same proposition (`lives_at_baker_street`) carries truth value `true` in `fiction` and `false` in `reality` *simultaneously*. No context overrides the other; each `context_project` returns its own view. This is the core feature of `ist(c, p)` — perspective-dependent truth without contradiction.

28.6 — Belnap B4 paraconsistent resolution

When `context_resolve` sees three contexts disagreeing — `true`, `false`, and `both` — it applies the **Belnap knowledge-join** lattice operation. The result is `both` because:

- The `true` claim contributes a "true" component.
- The `false` claim contributes a "false" component.
- The `both` claim already carries both components.

Joining these in the B4 lattice yields `both` with maximum confidence. The resolution *preserves* the disagreement rather than averaging or collapsing it — exactly the right behavior for downstream reasoning that needs to know two sources disagreed (e.g., a journalism front-end that should flag both sides of a contested claim).

If your application would rather pick the most-confident single value, use `context_resolve` with the credulous or prioritized strategy instead — but the default `b4_join` strategy is the principled choice when truth itself is contested.

Chapter 29 · Solutions

All solutions verified against the current `axioma` build. Solutions 29.1–29.5 are runnable; 29.6 is open-ended (the provided sketch is one valid answer).

```
# ========================================================================
# Exercise 29.1 — Build the mood space
# ========================================================================
mood: conceptual_space("Mood", "Russell circumplex")
add_dimension(mood, "Valence", -100, 100)
add_dimension(mood, "Arousal", -100, 100)
add_prototype(mood, "Excited", [60, 80])
add_prototype(mood, "Calm",    [60, -40])
add_prototype(mood, "Angry",   [-60, 80])
add_prototype(mood, "Sad",     [-50, -50])

test_points: [[70, 70], [0, 0], [-30, 60]]
labels: ["Excited", "Calm", "Angry", "Sad"]
i: 1
while (i <= len(test_points)) [
    p: test_points[i]
    println("Point:", p, "  nearest:", nearest_prototype(mood, p))
    j: 1
    while (j <= len(labels)) [
        println("  ", labels[j], "→", prototype_membership(mood, p, labels[j]))
        j = j + 1
    ]
    i = i + 1
]

# (70, 70)  → nearest=Excited     (highest membership: 0.95)
# (0, 0)    → nearest=Sad         (tie-breaker; all four near 0.7–0.78)
# (-30, 60) → nearest=Angry       (membership 0.88)

# ========================================================================
# Exercise 29.2 — Circular vs non-circular hue
# ========================================================================
circ: conceptual_space("HueCirc", "circular hue")
add_dimension(circ, "Hue", 0, 360, "circular", true)
add_prototype(circ, "Red", [0])

flat: conceptual_space("HueFlat", "non-circular hue")
add_dimension(flat, "Hue", 0, 360)
add_prototype(flat, "Red", [0])
```

```
test: [[10], [180], [350], [359]]
println("Hue     | circular | non-circular")
println("--------|----------|-------------")
k: 1
while (k <= len(test)) [
    p: test[k]
    println(p, "   | ", distance(circ, [0], p), " | ", distance(flat, [0], p))
    k = k + 1
]

# [10]   | 0.028 | 0.028     # near in both
# [180]  | 0.500 | 0.500     # midpoint — same either way
# [350]  | 0.028 | 0.972     # CIRCULAR wraps; FLAT thinks it's far
# [359]  | 0.003 | 0.997     # CIRCULAR sees it's basically Red

# One-sentence explanation: the circular flag tells the metric to
# wrap modularly, so hue 350° and 0° are 10° apart (close) instead
# of 350° apart (far) — without it, all queries near the red wrap
# point silently mislead.

# ============================================================================
# Exercise 29.3 — Interpolation paths through taste space
# ============================================================================
taste: conceptual_space("Taste", "primary tastes")
add_dimension(taste, "Sweet",  0, 100)
add_dimension(taste, "Sour",   0, 100)
add_dimension(taste, "Bitter", 0, 100)
add_prototype(taste, "Lemon",     [15, 95,  5])
add_prototype(taste, "Coffee",    [10,  5, 90])
add_prototype(taste, "Honey",     [95,  5,  5])
add_prototype(taste, "DarkChoc",  [50, 10, 70])

path: interpolate_path(taste, [[15, 95, 5], [50, 10, 70], [10, 5, 90]], 5)
println("Path length:", len(path), "(11 waypoints)")
m: 1
while (m <= len(path)) [
    pt: path[m]
    println("step", m, ":", pt, "nearest:", nearest_prototype(taste, pt))
    m = m + 1
]

# Output: 11 waypoints; first 3 nearest=Lemon, middle 5 nearest=DarkChoc,
# last 3 nearest=Coffee. The framework's geometric arrangement smoothly
# transitions from sour through bitter — passing through DarkChoc as the
# midpoint exactly because that's where we placed it.

# ============================================================================
# Exercise 29.4 — Fuzzy classification across the hue circle
# ============================================================================
```

```
hsv: conceptual_space("HSV3", "")
add_dimension(hsv, "Hue", 0, 360, "circular", true)
add_dimension(hsv, "Saturation", 0, 100)
add_dimension(hsv, "Value", 0, 100)
add_prototype(hsv, "Red",   [0,   100, 50])
add_prototype(hsv, "Green", [120, 100, 50])
add_prototype(hsv, "Blue",  [240, 100, 50])

h: 0
println("hue  R-mem  G-mem  B-mem  strong-claims (>0.95)")
while (h < 360) [
    p: [h, 100, 50]
    mr: prototype_membership(hsv, p, "Red")
    mg: prototype_membership(hsv, p, "Green")
    mb: prototype_membership(hsv, p, "Blue")
    claims: []
    if mr > 0.95 then [claims = claims + ["Red"]]
    if mg > 0.95 then [claims = claims + ["Green"]]
    if mb > 0.95 then [claims = claims + ["Blue"]]
    println(h, "  ", mr, "  ", mg, "  ", mb, "  ", claims)
    h = h + 36
]

# Pattern: hues at 0/36/324 cluster around Red; hues at 108/144 cluster
# Green; hues at 216/252 cluster Blue. Midpoint hues (72, 180, 288) have
# NO strong claim — they sit between two prototypes. This is the value
# of fuzzy membership over crisp set membership.

# ======================================================================
# Exercise 29.5 — Salience changes nearest_prototype, not membership
# ======================================================================
s1: conceptual_space("S1", "")
add_dimension(s1, "X", 0, 100)
add_dimension(s1, "Y", 0, 100)
add_prototype(s1, "Heavy", [10, 10], "salience", 1.0)
add_prototype(s1, "Light", [10, 90], "salience", 0.3)

p: [10, 50]
println("Initial saliences (Heavy=1.0, Light=0.3):")
println("  Heavy membership:", prototype_membership(s1, p, "Heavy"))
println("  Light membership:", prototype_membership(s1, p, "Light"))
println("  nearest:", nearest_prototype(s1, p))     # → Heavy

s2: conceptual_space("S2", "")
add_dimension(s2, "X", 0, 100)
add_dimension(s2, "Y", 0, 100)
add_prototype(s2, "Heavy", [10, 10], "salience", 0.3)
add_prototype(s2, "Light", [10, 90], "salience", 1.0)

println("Swapped saliences (Heavy=0.3, Light=1.0):")
```

```
println("  Heavy membership:", prototype_membership(s2, p, "Heavy"))
println("  Light membership:", prototype_membership(s2, p, "Light"))
println("  nearest:", nearest_prototype(s2, p))     # → Light

# Explanation: the point (10, 50) is equidistant from both prototypes
# (40 units up to Heavy, 40 units down to Light). prototype_membership
# returns the same fuzzy value for both because the metric is symmetric.
# nearest_prototype, however, multiplies the distance by 1/salience —
# so the higher-salience prototype "pulls" the point. When Heavy has
# higher salience, the point lands in Heavy's basin; swap, and it
# lands in Light's. Salience is the *voting weight*, not the *metric*.

# ============================================================================
# Exercise 29.6 (open) — Programming-language space
# ============================================================================
langs: conceptual_space("ProgLangs", "language characteristics")
add_dimension(langs, "Typing",   0, 100)   # 0=dynamic, 100=static
add_dimension(langs, "Paradigm", 0, 100)   # 0=imperative, 100=functional
add_dimension(langs, "Compiled", 0, 100)   # 0=interpreted, 100=AOT

add_prototype(langs, "Python",     [10, 30, 10])
add_prototype(langs, "Haskell",    [95, 95, 80])
add_prototype(langs, "Go",         [85, 25, 95])
add_prototype(langs, "Lisp",       [20, 80, 20])
add_prototype(langs, "Rust",       [95, 50, 95])
add_prototype(langs, "JavaScript", [15, 40, 15])

# Three classification questions
println("Q1: statically-typed, functional, compiled:", nearest_prototype(langs, [90,
 ↪  90, 70]))
println("Q2: dynamic-ish, imperative, compiled-fast:", nearest_prototype(langs, [25,
 ↪  30, 80]))
println("Q3: gradual, mixed-paradigm, JIT:",           nearest_prototype(langs, [50,
 ↪  60, 50]))

# Reflection: the Typing dimension (0=dynamic, 100=static) feels right
# because there's a real continuum (TypeScript at 60, mypy-Python at 40).
# Paradigm is more dubious — imperative-vs-functional isn't a single
# spectrum, it conflates "supports first-class functions" with "prefers
# immutable data." A PCA over real language features would probably
# split it into 2-3 separate latent dimensions: lambda support, mutation
# rules, control-flow style. That's the trade between interpretable
# human-named axes and accurate latent axes.
```

Notes

29.1 — Why (0, 0) lands in Sad

The point (0, 0) is the *origin* of the Russell circumplex — it's equidistant from all four mood prototypes (each at ~70 units away). All four membership scores cluster around 0.70-0.78, and `nearest_prototype` breaks the tie by iteration order. Real systems disambiguate by adding a fifth "Neutral" prototype at (0, 0) or by reporting the top-2 with their margin.

29.2 — The wrap-around fix

The non-circular `distance(flat, [0], [359]) ≈ 0.997` says hue 359 is essentially at the OTHER end of the spectrum from hue 0 — which is wrong, since red wraps. The circular version returns `0.003` (essentially zero), correctly identifying 359° as basically red.

Without the flag, *every* query near the wrap point of a modular dimension produces silently wrong results. The flag is the single most important keyword option in the API — for any cyclic quality (hue, time-of-day, day-of- week, angular position) you must set it.

29.3 — The "DarkChoc-as-midpoint" finding

We chose Lemon → DarkChoc → Coffee as the three control points and asked for 5 interpolation steps. The output shows the path moves smoothly through Lemon, transitions into DarkChoc territory in the middle, and ends in Coffee — exactly the geometric story the prototypes told.

A graph KR would need an ad-hoc *taste-pair* table to answer "is this drink between Lemon and Coffee?" The CS gets it for free from geometry.

29.4 — Why all three claim hue 0 with low radius

If you set `radius` on the prototypes, the radius is *normalized by sqrt(n_dimensions)*. With `radius: 30` in a 3D space, normalized radius ≈ 17 — and the maximum normalized distance in this space is only ~1.4 — so every point is "inside" every prototype's ball. That's why the solution uses *point prototypes* (no radius) plus a strict threshold (0.95).

The takeaway: if you want crisp ball regions, use `prototype_membership > some_threshold` rather than `in_prototype` with a radius. The radius normalization is documented in §29.10 as a gap.

29.5 — Salience is voting weight, not metric distortion

This is the conceptually subtle one. Two different mechanisms could give a prototype more "pull":

1. **Distort the metric** so that the prototype's basin is geometrically larger.
2. **Discount competitors** so that when comparing two distances, the higher-salience prototype wins ties and close-calls.

Axioma implements (2): `nearest_prototype` divides each candidate's distance by its salience before comparing, so higher salience = effectively shorter distance to *win*. But `prototype_membership` reports the raw geometric membership, which doesn't change.

Pedagogically: salience is the *vote weight*, not the metric. If you want a metric distortion, use the per-dimension `weight` keyword on `add_dimension` instead.

29.6 — The interpretability/accuracy tradeoff

The three-dimension language space gets the *gross* geography right: Haskell separates from Python by a lot, Rust separates from JavaScript by a lot. But it has known weaknesses:

- **Paradigm** is a one-dimensional projection of at least three independent characteristics (FP support, immutability, type richness).
- **Compiled** lumps together "compiles ahead of time", "produces native code", and "doesn't need a runtime VM" — which are independent for things like GraalVM or Truffle.
- **No social dimensions** — community size, library ecosystem, hiring market. These dominate language *choice* but don't fit a quality-space axis cleanly.

A PCA on real language metadata (paradigm-flag features, StackOverflow tag-co-occurrence, GitHub search hits) would extract *latent* dimensions that correlate better with actual classifications — at the cost of interpretability. This trade is universal: human-named dimensions are auditable; data-driven dimensions are accurate. Production systems use both.

Chapter 30 · Solutions

All solutions verified against the current `axioma` build. Solutions 30.1–30.4 are runnable; 30.5 is a paper-and- pencil exercise; 30.6 is open-ended.

```
# ============================================================================
# Exercise 30.1 — Round-trip identity
# ============================================================================
println(decode(encode("42"))    == "42")        # true
println(decode(encode("x"))     == "x")          # true
println(decode(encode("x + 1")) == "x + 1")      # true
println(decode(encode("hello")) == "hello")      # true

# ============================================================================
# Exercise 30.2 — Distinct expressions, distinct numbers
# ============================================================================
gn_x: encode("x")
gn_y: encode("y")
println(decode(gn_x) == decode(gn_y))            # false
println(decode(gn_x))                            # "x"
println(decode(gn_y))                            # "y"

# ============================================================================
# Exercise 30.3 — Whitespace is preserved (NOT normalized)
# ============================================================================
g1: encode("x + 1")
g2: encode("x   +   1")
println(decode(g1))                              # "x + 1"
println(decode(g2))                              # "x   +   1"  (whitespace preserved)
println(decode(g1) == decode(g2))                # false

# ============================================================================
# Exercise 30.4 — Quine substitution pattern
# ============================================================================
q: "q: X; println(X)"
quoted_q: "\"q: X; println(X)\""
prog: replace(q, "X", quoted_q)
println(prog)
# Output:
#   q: "q: X; println(X)"; println("q: X; println(X)")
#
# This shows the pattern but isn't a byte-exact quine — see chapter §30.10.
```

Notes

30.1 — encode/decode round-trip

The encode/decode pair is the *minimal reliable surface* of Axioma's Gödelization layer. For any string `s` that parses as valid Axioma syntax, `decode(encode(s)) == s` holds verbatim. The encoder validates by parsing, then encodes the literal bytes.

30.2 — Distinguishing encodings

Two different strings produce two different Gödel numbers. We compare via `decode(...) == decode(...)` instead of direct integer comparison because the Gödel number is a `*types.GödelNumberObject` (with `type(gn) = "unknown"` in the current build) — direct comparison via == is not reliable across the runtime's coercion rules. The `decode` round-trip is the user-script-friendly way to check identity.

30.3 — Whitespace preservation

The pilot chapter incorrectly claimed `encode` normalizes whitespace via AST round-tripping. The probe in §30.3.4 of the full chapter shows the truth: whitespace is *preserved*. `encode` parses for validity (a parse error on `"encode(\"x\")"` confirms this) but encodes the literal bytes. The probe was the disambiguating evidence.

30.4 — The quine pattern

The exercise produces a string that *demonstrates the substitution pattern* but isn't itself a byte-exact quine. Building a byte-exact quine requires the two-level substitution sketched in the chapter; this exercise just makes the move visible.

30.5 — Paper exercise

A single symbol with code `c` encodes to `2^c` in the classical Gödel scheme (one symbol → first prime → that prime raised to the symbol's code). With `P → 16`:

```
encoding(P) = 2^16 = 65536
```

For comparison, the encoding of `"AB"` (codes 1, 2) is:

```
encoding(AB) = 2^1 · 3^2 = 2 · 9 = 18
```

A six-symbol string with average code 13 reaches a 36-digit number even in this small example; Axioma's internal representation uses a variant that grows more slowly but the *phenomenon* (massive number growth) is real.

30.6 — Open-ended

Sample modern applications of Gödelization (see chapter §30.10 for a longer list):

- **Compilers / interpreters** — every "code as data" story (LISP, macros, eval, JIT) is Gödelization under another name.

- **Cryptographic commitment** — zk-SNARKs encode entire computations as polynomials over finite fields. That polynomial *is* the Gödel number of the computation in a more compact arithmetic encoding.
- **Formal verification / AI alignment** — to prove properties of an AI system you need to encode the AI system as something the proof assistant can talk about. Lean, Coq, and HOL all rely on Gödelization.
- **Self-modifying systems** — Kleene's recursion theorem guarantees self-referential programs are computable; Gödel numbering is how it's done.
- **Halting / undecidability** — every "undecidable" theorem traces back to Cantor's diagonal applied to Gödel-numbered programs.

Not yet available

`selfEncode` (the basis for a 3-line quine), `diagonalize` (programmatic self-reference), and the `creat eFormalSystem` / `provable` / `proveIncompleteness` chain are not reliable in the current build. Once they are, this chapter gains working solutions for two more substantial exercises (*build an incompleteness demo end-to-end*, *write a working 3-line quine via selfEncode*) that would replace the open-ended 30.6.

Chapter 31 · Solutions

All solutions verified against the current `axioma` build. Solutions 31.1–31.5 are runnable; 31.6 is open-ended (the provided sketch is one valid answer in the scientific- papers domain).

```
# ============================================================================
# Exercise 31.1 — Place a relation
# ============================================================================
relation taught(instructor -> "instructor",
                subject    -> "subject",
                student    -> "student")

taught("Alice", "calculus", "Bob")
taught("Alice", "calculus", "Carol")
taught("Alice", "calculus", "Dave")
taught("Eve",   "logic",    "Bob")

println("Who taught Bob?")
println({I | I <- taught(I, _, "Bob")})
# → {"Alice", "Eve"}

println("What did Alice teach?")
println({S | S <- taught("Alice", S, _)})
# → {"calculus"}

println("How many students learned calculus?")
println(len({St | St <- taught(_, "calculus", St)}))
# → 3

# ============================================================================
# Exercise 31.2 — A small citation graph
# ============================================================================
relation cite(paper :: String -> "paper",
              author :: String -> "author") doc: "P cites A"

relation wrote(author :: String -> "author",
               paper  :: String -> "paper")

cite("Pearl1988", "Russell")
cite("Pearl1988", "Norvig")
cite("Russell2010", "Pearl")
wrote("Pearl",   "Pearl1988")
wrote("Russell", "Russell2010")
wrote("Norvig",  "Russell2010")
```

```
println("cited authors:", {X | X <- cite(_, X)})
# → {"Norvig", "Pearl", "Russell"}

println("authors who wrote:", {X | X <- wrote(X, _)})
# → {"Norvig", "Pearl", "Russell"}

# Note: use named variables (not _) inside Horn-rule bodies
self_cite(X) <= wrote(X, P) and cite(P, X)
println("self-cites:", {X | X <- self_cite(X)})
# → {}

mutual(X, Y) <= wrote(X, P1) and cite(P1, Y)
                and wrote(Y, P2) and cite(P2, X)
println("mutual:", {(X, Y) | mutual(X, Y)})
# → {("Norvig", "Pearl"), ("Pearl", "Norvig"),
#    ("Pearl", "Russell"), ("Russell", "Pearl")}

# ======================================================================
# Exercise 31.3 — Type annotations as documentation
# ======================================================================
relation birthday(person :: String -> "person",
                        year   :: Integer -> "year")

birthday("Alice", 1990)
birthday("Bob",   1985)
birthday("Carol", "unknown")    # PASSES despite year :: Integer

println("All facts (note Carol's year is a string!):")
println({(P, Y) | birthday(P, Y)})
# → {("Alice", 1990), ("Bob", 1985), ("Carol", "unknown")}

# Custom runtime typecheck using the `type()` builtin
check_birthday: func(p, y) [
    if type(y) == Integer then [
        println("  ok:", p, y)
    ] else [
        println("  WARN:", p, "year not integer:", y)
    ]
]

println("Running custom typecheck:")
check_birthday("Alice", 1990)      # → ok: Alice 1990
check_birthday("Carol", "unknown") # → WARN: Carol year not integer: unknown

# ======================================================================
# Exercise 31.4 — Recursive ancestor
# ======================================================================
relation parent(p -> "parent", c -> "child")
```

```
parent("Alice", "Bob")
parent("Bob",   "Carol")
parent("Carol", "Dave")

grandparent(X, Z) <= parent(X, Y) and parent(Y, Z)

ancestor(X, Y) <= parent(X, Y)
ancestor(X, Z) <= parent(X, Y) and ancestor(Y, Z)

println("Grandparents:", {(X, Z) | grandparent(X, Z)})
# → {("Alice", "Carol"), ("Bob", "Dave")}

println("Alice's descendants:", {Y | Y <- ancestor("Alice", Y)})
# → {"Bob", "Carol", "Dave"}

println("Dave's ancestors:", {X | X <- ancestor(X, "Dave")})
# → {"Alice", "Bob", "Carol"}

# ============================================================================
# Exercise 31.5 — Arity errors
# ============================================================================
relation loves(lover -> "lover", beloved -> "beloved")

loves("Romeo", "Juliet")
println({(X, Y) | loves(X, Y)})    # → {("Romeo", "Juliet")}

# These would each error:
#   loves("Romeo")              # → relation loves expects 2 arguments, got 1
#   loves("R", "J", "extra")    # → relation loves expects 2 arguments, got 3

# Explanation (place this as a comment in the .ax file):
# Arity enforcement catches *structural* bugs — wrong number of arguments.
# Named-place semantics catches *semantic* bugs — wrong meaning at a
# position (you read x1 as "giver" instead of "gift"). They are
# complementary: arity is checked by the runtime; named places are
# checked by the human reading the code and the downstream typechecker.

# ============================================================================
# Exercise 31.6 (open) — Scientific-papers domain
# ============================================================================
relation authored(person :: String -> "author",
                  paper  :: String -> "paper")
    doc: "Person is an author of paper"

relation cites(citing :: String -> "citing paper",
               cited  :: String -> "cited paper")
    doc: "First paper cites the second"

relation appeared_in(paper :: String -> "paper",
                     venue :: String -> "venue",
```

```
                            year  :: Integer -> "year")
    doc: "Paper was published at venue in year"

relation supersedes(later   :: String -> "later paper",
                            earlier :: String -> "earlier paper")
    doc: "Later paper supersedes earlier (v2 of same work)"

relation reviews(reviewer :: String -> "reviewer",
                         paper    :: String -> "paper",
                         score    :: Integer -> "score")
    doc: "Reviewer scored paper"

# Sample population
authored("Pearl",  "P88")
cites("R10", "P88")
appeared_in("P88", "AI Journal", 1988)
supersedes("R10v2", "R10")
reviews("Smith", "R10v2", 8)

println("Pearl's papers:", {P | P <- authored("Pearl", P)})
println("Cited papers:",   {C | C <- cites(_, C)})
println("1988 papers:",    {P | P <- appeared_in(P, _, 1988)})
```

Notes

31.1 — Why “instructor” comes first

Following Lojban convention, the **most salient actor** goes in x1. For a teaching event the instructor *causes* the event — they decide what to teach, when, to whom — so they get x1. The subject is the *theme* (the thing transferred), so x2. The student is the *recipient*, so x3. This mirrors the Lojban gismu `dunda` (give): the giver is x1, the gift is x2, the recipient is x3.

31.2 — Two notable details

Wildcards work in comprehensions, not in Horn-rule bodies. A comprehension like `{X | X <- cite(_, X)}` correctly treats _ as “don’t care”. But a rule like `wrote_someone(X) <= wrote(X, _)` silently returns an empty result. The fix: use a named variable (`Y`, `P`, etc.) for the unused position, as the `self_cite` and `mutual` rules show.

Mutual citation produces two pairs per match. Both `(Russell, Pearl)` and `(Pearl, Russell)` appear in the result because the relation is symmetric in semantics but not in encoding — each direction of the implication fires separately. Filter with `X < Y` in the rule body if you want canonical-direction pairs only.

31.3 — Why types are documentation-only here

The `:: Integer` annotation on `year` lives in the AST for use by `--typecheck` (Volume I Ch.19's static pre-pass) and by external tooling. The runtime *relational* subsystem doesn't enforce it because:

1. Many real KR tasks need *transitional* data with missing or string-encoded values ("unknown", "circa 1988").
2. A hard runtime check would break the exploratory-then-strict workflow common in KR work.
3. The typechecker can be run on-demand when you want guarantees.

The pattern: define liberally, type-check periodically, ship with confidence.

31.4 — Why the chain is four generations, not five

Empirically, the Horn-rule machinery does ~3 rounds of forward chaining on recursive rules before stopping. With a 4-generation chain, all three transitive degrees from Alice (Bob, Carol, Dave) are reachable. A 5-generation chain would miss the deepest descendant from Alice's perspective — but querying from the other end (`ancestor(_, "Eve")`) would still find Alice.

This is the kind of operational detail that production KR systems hide with bottom-up fixpoint computation. For teaching purposes, **stay within the engine's fixed-depth budget**, or run the same recursive rule from the other direction.

31.5 — Arity vs named places: complementary contracts

Arity prevents *structural* bugs:

- `loves("Romeo")` is wrong — missing argument.
- `loves("R", "J", "extra")` is wrong — extra argument.

Named places prevent *semantic* bugs:

- `gives("book", "Mary", "John")` parses fine (correct arity) but is *meaningfully* wrong: the reader can't tell if `book` is the giver, the gift, or the recipient.

The two together produce the Lojban discipline: *structure* is enforced by the runtime, *meaning* is enforced by the schema's place labels. Production Cascade code uses both, plus the `--typecheck` pre-pass for full coverage.

31.6 — Why "author" in `authored(author, paper)`

The Lojban convention puts the *agent* first. In `authored`, the author is the agent (the entity *doing* the authoring); the paper is the theme. Same logic for `cites` (the citing paper is the active "subject" of the citation act) and `reviews` (the reviewer is the agent).

The interesting case is `appeared_in(paper, venue, year)` — the paper is *not* the agent (the venue "hosted" it). The choice of `paper` in `x1` follows the **focus-of-discourse** principle: the paper is what the fact is *about*, even if not the causal agent.

These conventions are exactly the kind of judgment-call documentation that named places make explicit. A graph KR with edges like `APPEARED_IN(p, v, y)` would document the same thing in an edge-direction convention buried in a wiki — easier to drift.

Chapter 32 · Solutions

All solutions verified against the current `axioma` build. Solutions 32.1–32.5 are runnable; 32.6 is open-ended.

```
# ============================================================================
# Exercise 32.1 — Inventory by category
# ============================================================================
cats: nsm_categories()
total: 0
i: 1
while (i <= len(cats)) [
    ps: nsm_primes_by_category(cats[i])
    println(cats[i], ":", len(ps), "primes")
    total = total + len(ps)
    i = i + 1
]
println("---")
println("Total:", total)
# Total → 66

# ============================================================================
# Exercise 32.2 — Allolex lookup
# ============================================================================
primes: nsm_primes()
n_allolex: 0
j: 1
while (j <= len(primes)) [
    if len(primes[j].allolexes) > 0 then [
        println(primes[j].prime, "→", primes[j].allolexes)
        n_allolex = n_allolex + 1
    ]
    j = j + 1
]
println("---")
println("Primes with allolexes:", n_allolex, "/", len(primes))
# → 17 / 66

# ============================================================================
# Exercise 32.3 — Validate-and-refine "regret"
# ============================================================================
# Attempt 1 — "feels", "they", "did" all fail
v1: "someone feels bad because they did something"
r1: nsm_validate_explication(v1)
```

```
println("v1:", v1)
println("  valid:", r1.valid, "| issues:", r1.grammar_issues)

# Attempt 2 — fix "feels" → "feel"; "they" → drop it
v2: "someone feel bad because did something"
r2: nsm_validate_explication(v2)
println("v2:", v2)
println("  valid:", r2.valid, "| issues:", r2.grammar_issues)

# Attempt 3 — lemmatize "did" → "do"
v3: "someone feel bad because do something"
r3: nsm_validate_explication(v3)
println("v3:", v3)
println("  valid:", r3.valid, "| used:", r3.used_primes)

# Attempt 4 — give FEEL its valency-2 argument (what is felt)
v4: "I feel something bad because I do something"
r4: nsm_validate_explication(v4)
println("v4:", v4)
println("  valid:", r4.valid, "| used:", r4.used_primes)

# ============================================================================
# Exercise 32.4 — Translation pipeline
# ============================================================================
canonical_prime: func(w) [
    lw: lower(w)
    primes: nsm_primes()
    i: 1
    found: ""
    while (i <= len(primes) and found == "") [
        p: primes[i]
        if lower(p.prime) == lw then [
            found = p.prime
        ] else [
            k: 1
            while (k <= len(p.allolexes) and found == "") [
                if lower(p.allolexes[k]) == lw then [
                    found = p.prime
                ]
                k = k + 1
            ]
        ]
        i = i + 1
    ]
    found
]

translate_explication: func(text, target_lang) [
    words: split(text, " ")
    out: ""
```

```
    i: 1
    while (i <= len(words)) [
        w: words[i]
        cp: canonical_prime(w)
        if cp != "" then [
            out = out + nsm_translate(cp, target_lang) + " "
        ] else [
            out = out + w + " "
        ]
        i = i + 1
    ]
    out
]

src: "I think this is good"
println("Source:", src)
println("French:  ", translate_explication(src, "french"))
println("German:  ", translate_explication(src, "german"))
println("Japanese:", translate_explication(src, "japanese"))

# ============================================================================
# Exercise 32.5 — Decompose "gratitude" into primes
# ============================================================================
lines: [
    "someone do something good",
    "I feel something good because this",
    "I want do something good"
]

m: 1
while (m <= len(lines)) [
    r: nsm_validate_explication(lines[m])
    println("L", m, ":", lines[m])
    println("  valid:", r.valid, "| used:", r.used_primes)
    m = m + 1
]

concept Gratitude
Gratitude has explication: lines
Gratitude has used_primes: ["SOMEONE", "DO", "SOMETHING", "GOOD", "I",
                            "FEEL", "BECAUSE", "THIS", "WANT"]
Gratitude has language_neutral: true

println("---")
println("Gratitude.explication:", Gratitude.explication)
println("Gratitude.used_primes:", Gratitude.used_primes)
println("Gratitude.language_neutral:", Gratitude.language_neutral)

# ============================================================================
# Exercise 32.6 (open) — Critique the spatial primes
```

```
# ============================================================================
space_primes: nsm_primes_by_category("space")
s: 1
println("Spatial primes:")
while (s <= len(space_primes)) [
    println("  -", space_primes[s].prime, "—", space_primes[s].examples)
    s = s + 1
]
```

Notes

32.1 — Total is 66, not 65

The canonical Wierzbicka set is **65 primes**. Axioma's registry has **66** because it splits the polysemous English *be* into three location/existence variants (`BE (SOMEWHERE)`, `BE (SOMEONE/SOMETHING)`, `BE (SOMEONE'S)`) where some Wierzbicka tabulations collapse two of them. The extra entry is harmless — it makes the registry more explicit about the argument structure.

32.2 — Which primes carry allolexes?

The 17 primes with allolexes cluster around three uses:

- **Pronoun/copula variation** — `I → me`, `BE (SOMEWHERE) → is/am/are`, `BE (SOMEONE/SOMETHING) → is`.
- **Substantive flexibility** — `SOMEONE → person`, `SOMETHING → thing`, `MUCH → many`, `LITTLE → few`.
- **Reflective time/place words** — `WHEN → time`, `WHERE → place`.

The validators accept the allolexes as if they were the canonical prime, which lets your explications sound more natural English while still being machine-checkable.

32.3 — The iterative loop

The `grammar_issues` field is the steering signal: each listed word tells you what to replace. The `suggestions` field then names the prime it thinks you meant. **Pedagogy: don't write the explication once and check it once. Write, validate, read the issues, rewrite. Three or four iterations is typical.**

Note how the path from v1 to v4 simultaneously:

- Removes irregular morphology (`feels` → `feel`, `did` → `do`).
- Adds missing arguments (FEEL is valency 2 — what is being felt?).
- Replaces vague pronouns (`they`) with explicit speakers (`I` or `someone`).

32.4 — Why the canonical-prime helper is needed

A naive pipeline calls `nsm_translate(upper(word), lang)` for any word where `is_nsm_prime(word) == true`. But `is_nsm_prime("is") == true` (because "is" is an allolex of `BE (SOMEWHERE)`), while

`nsm_translate("IS", lang)` errors because `IS` isn't a canonical prime name. The fix: walk the registry to find the *canonical* prime that owns this allolex, then translate that.

The output shows the translations include multi-form entries (French's `ce/cette` for `this`) because the registry stores both gendered French forms in one slot. Real translation would need a downstream disambiguation step.

32.5 — Why three lines?

Wierzbicka's explications are typically *multi-line prose*, one proposition per line. The convention mirrors a structured argument: cause line → effect line → disposition line. Splitting it lets each line be validated independently and lets the structure carry meaning (line 2's `because this` refers anaphorically to line 1).

The `language_neutral: true` flag is the operational payoff: because every word in the explication is a prime, the *whole concept* translates by table lookup into any of the 7 supported languages without re-extraction.

32.6 — Where ABOVE and BELOW strain

The canonical NSM defense: even Guugu Yimithirr (the famous Australian language that orients its speakers to cardinal directions instead of relative left/right) has *some* way of describing vertical relations — sky versus ground, head versus feet. Critics counter that these are *contextual* phrasings, not lexical primes in the way `ABOVE` is treated in English. The honest answer: NSM treats vertical orientation as universal *for human bodies under gravity*, then translates per language using whatever surface form expresses that relation. Whether that's a faithful description of *meaning* or a back-projection of English structure remains an open question — exactly the philosophically interesting territory NSM forces you to think about.

Parallel critiques apply to:

- `FAR / NEAR` — culturally relative scale.
- `INSIDE` — depends on container ontology, which varies across languages.
- `SIDE` — body-relative versus landmark-relative orientation systems differ.

NSM survives these by being *thin*: a prime claims to exist *somewhere* in every language, not to be the *best* fit, and translations sometimes carry significant rephrasing.

Chapter 33 · Solutions

All solutions verified against the current `axioma` build. Solutions 33.1–33.5 are runnable; 33.6 is open-ended.

```
# ============================================================================
# Exercise 33.1 — Inventory the registry
# ============================================================================
prims: cd_primitives()
println("Count:", len(prims))      # Count: 11
i: 1
while (i <= len(prims)) [
    println(prims[i].name)
    i = i + 1
]

# ============================================================================
# Exercise 33.2 — Map verbs to primitives
# ============================================================================
verbs_to_acts: [
    ["kick",   "PROPEL"],
    ["eat",    "INGEST"],
    ["hear",   "MTRANS"],
    ["go",     "PTRANS"],
    ["say",    "SPEAK"],
    ["decide", "MBUILD"]
]
j: 1
while (j <= len(verbs_to_acts)) [
    pair: verbs_to_acts[j]
    println(pair[1], "→", pair[2], "(valid?", is_cd_primitive(pair[2]), ")")
    j = j + 1
]

# ============================================================================
# Exercise 33.3 — Roles per primitive
# ============================================================================
print_roles: func(name) [
    if is_cd_primitive(name) then [
        println(name, "→", cd_roles(name))
    ] else [
        println("ERROR:", name, "is not a registered primitive")
    ]
]
```

```
print_roles("ATRANS")     # ATRANS → ["actor", "object", "from", "to"]
print_roles("INGEST")     # INGEST → ["actor", "object"]
print_roles("MOVE")       # MOVE → ["actor", "body_part", "direction"]
print_roles("XFRANS")     # ERROR: XFRANS is not a registered primitive

# ========================================================================
# Exercise 33.4 — Synonymy via shared primitive
# ========================================================================
verb_to_primitive: func(verb) [
    if verb == "give"     then "ATRANS"
    else if verb == "donate"   then "ATRANS"
    else if verb == "sell"     then "ATRANS"
    else if verb == "transfer" then "ATRANS"
    else if verb == "go"       then "PTRANS"
    else if verb == "send"     then "PTRANS"
    else if verb == "eat"      then "INGEST"
    else if verb == "drink"    then "INGEST"
    else "UNKNOWN"
]

same_primitive: func(v1, v2) [
    p1: verb_to_primitive(v1)
    p2: verb_to_primitive(v2)
    p1 == p2 and is_cd_primitive(p1)
]

println(same_primitive("give",   "donate"))     # true
println(same_primitive("give",   "transfer"))   # true
println(same_primitive("give",   "go"))         # false
println(same_primitive("eat",    "drink"))      # true
println(same_primitive("go",     "send"))       # true
println(same_primitive("unicorn","narwhal"))    # false

# ========================================================================
# Exercise 33.5 — Decompose "promise"
# ========================================================================
concept Promise
Promise has step1_primitive: "MBUILD"
Promise has step1_content: "I will do X"
Promise has step2_primitive: "MTRANS"
Promise has step2_actor: "speaker"
Promise has step2_to: "listener"
Promise has flavor: "commitment"

println("Promise step 1:",
    Promise.step1_primitive,
    "valid?", is_cd_primitive(Promise.step1_primitive))
println("Promise step 2:",
    Promise.step2_primitive,
    "valid?", is_cd_primitive(Promise.step2_primitive))
```

```
println("Flavor:", Promise.flavor)
```

Notes

33.1 — Iterating the registry

`cd_primitives()` returns an array of ObjectMaps, each with `{description, name, roles}` keys. Accessing `.name` projects out the primitive identifier; the full record is available if you need description or roles too.

33.2 — Case-insensitive validation

`is_cd_primitive` is case-insensitive: `"atrans"`, `"ATRANS"`, and `"Atrans"` all return `true`. The canonical form returned by `cd_primitives()` is upper-case, and chapter prose uses upper-case throughout.

33.3 — Roles are stable

The role names for each primitive are *fixed* by the registry. They aren't user-extensible from the Axioma surface — if you want a custom role beyond the registered ones, attach it to your Concept directly (see Exercise 33.5's `step1_content`, `flavor`, etc.). The CD primitives provide the *spine*; your domain provides the *flesh*.

33.4 — The `is_cd_primitive` guard

A naive `same_primitive` that just returns `p1 == p2` would incorrectly say `unicorn` and `narwhal` share a primitive (both map to `"UNKNOWN"`). Guarding with `is_cd_primitive(p1)` ensures we only confirm sameness when the shared label is a registered primitive.

33.5 — Decomposing into a Concept

The `Concept` machinery from Volume I (Chapter 18) is the natural place to record CD decompositions. Each "step" becomes a slot. The `Promise.step1_primitive` + `Promise.step2_primitive` pattern can scale to arbitrary-depth decompositions (some compound verbs decompose into 4–5 primitives in sequence).

For a *programmatic* iteration, you'd typically store the steps as an array of records:

```
Promise has steps: [
    {primitive: "MBUILD", content: "I will do X"},
    {primitive: "MTRANS", actor: "speaker", "to": "listener"}
]
```

The chapter's solution uses individual slots for clarity; production code would use the array form.

33.6 — Where CD pays off

Sample reasoning at three domain extremes:

- **High vocabulary, high decomposition payoff** — geopolitical news, legal contracts, financial reporting. Hundreds of domain verbs, all decomposing to a handful of CD primitives plus rich flavor slots. Cascade's entity-extraction pipeline lives here.
- **Medium vocabulary, medium payoff** — biomedical research papers, recipe corpora, sports commentary. Vocabulary is large but not as variable as full-natural-language; CD speeds reasoning but isn't the bottleneck.
- **Low vocabulary, low payoff** — source-code analysis, formal logic, mathematical proofs. The vocabulary is already controlled and small; CD's normalization adds little. Use ordinary type-driven dispatch instead.

The general rule: **vocabulary diversity per concept-spine** is the metric. Higher ratio → CD pays off more.

Chapter 35 · Solutions

All solutions verified by running with `axioma --no-kb` and recording the wall-clock timings on the textbook-development machine. Timings on **your** machine will differ — what matters is the **doubling ratio**, which identifies the growth class regardless of constant factor.

35.1 — Linearity, twice

```
make: func(n) [
  xs: []
  j: 1
  while j <= n [
    xs = xs + [j]
    j = j + 1
  ]
  xs
]

count_evens: func(arr) [
  c: 0
  i: 1
  while i <= len(arr) [
    if arr[i] % 2 == 0 then [c = c + 1]
    i = i + 1
  ]
  c
]

sum_odds: func(arr) [
  s: 0
  i: 1
  while i <= len(arr) [
    if arr[i] % 2 != 0 then [s = s + arr[i]]
    i = i + 1
  ]
  s
]

sizes: [1000, 2000, 4000, 8000]
i: 1
while i <= len(sizes) [
```

```
  n: sizes[i]
  arr: make(n)
  r1: bench("ce", func() [count_evens(arr)])
  r2: bench("so", func() [sum_odds(arr)])
  println(n, "ce:", r1.elapsed_ms, "so:", r2.elapsed_ms)
  i = i + 1
]
```

Measured output:

```
1000  ce:  57.8 ms  so:  53.0 ms
2000  ce: 103.2 ms  so: 102.2 ms     (~1.85×)
4000  ce: 212.1 ms  so: 211.0 ms     (~2.05×)
8000  ce: 420.8 ms  so: 418.2 ms     (~1.98×)
```

(a) Growth class: `O(n)`. Doubling `n` doubles the time within measurement noise. Each function makes one pass over the array.

(b) Why aren't `ce` and `so` identical? Two reasons:

- `sum_odds` does an addition (`s = s + arr[i]`) where `count_evens` does an increment — slightly different inner-loop cost, both small constants.
- Caching, branch prediction, and timer jitter add a few % noise. Run-to-run variation is bigger than the per-call difference. **The growth class is the same.**

The lesson: in the same complexity class, the absolute numbers don't tell you which is "better" — the constant factor depends on the work done per iteration, the data, the cache state. Complexity class predicts scaling; constant factor predicts the actual ms at any specific size.

35.2 — Quadratic by construction

```
count_pairs_summing_to: func(arr, target) [
  c: 0
  n: len(arr)
  i: 1
  while i <= n - 1 [
    j: i + 1
    while j <= n [
      if arr[i] + arr[j] == target then [c = c + 1]
      j = j + 1
    ]
    i = i + 1
  ]
  c
]
```

Measured output:

```
100 -> 9.4 ms
200 -> 33.1 ms      (3.54×)
```

```
400 -> 119.2 ms     (3.59×)
800 -> 456.5 ms     (3.83×)
```

(a) Prediction from code: the outer loop runs `n - 1` times, the inner runs at most `n - 1` times — total work `n(n-1)/2 ≈ n²/2`. **Class: `O(n²)`.**

(b) Measurement: the doubling ratios are 3.54, 3.59, 3.83 — approaching the theoretical 4× as `n` grows. At small `n` the constant overhead (loop setup, function call, bench timer) drags the ratio below 4×; the asymptotic behavior dominates as `n` grows.

(c) Prediction matches. Quadratic confirmed empirically.

35.3 — Linear vs binary search switchover

```
linear_search: func(arr, target) [
  i: 1
  found: -1
  while i <= len(arr) [
    if arr[i] == target then [
      found = i
      i = len(arr) + 1
    ] else [
      i = i + 1
    ]
  ]
  found
]
```

(Plus `bsearch` and `repeated_*` from the chapter.)

Measured output (1000 lookups per row):

```
n=10     lin:    516.5 ms   bs:  165.2 ms     (bs wins ×3.1)
n=100    lin:  2,886.6 ms   bs:  179.6 ms     (bs wins ×16)
n=1000   lin: 26,955.9 ms   bs:  184.6 ms     (bs wins ×146)
n=10000  lin: 26,585.1 ms   bs:  198.4 ms     (bs wins ×134)
```

(a) Binary search wins at every measured `n` — including `n = 10`. In the Axioma interpreter, the bench overhead and loop setup dominate over a 5-step vs 4-step inner loop. To find a "true" switchover, you'd need to search inside a tight Go loop where the constant factors align — or use a much smaller target search (single lookup per bench instead of 1000).

(b) Why doesn't binary search always win?

- For *very small* arrays ($n \leq 5$ or so), linear search has no loop overhead and may finish in 2-3 comparisons. Binary search has more bookkeeping per step.
- For *cache-cold* binary search (e.g., the array doesn't fit in cache), the random-access midpoint may miss the cache more often than the sequential scan of linear search.
- Practical std-lib search routines often start with linear for `n < 16` and switch to binary above that — exactly this kind of constant-factor tuning.

Note on the n=1000 vs n=10000 anomaly: the timings are similar because our `repeated_lin` always searches for targets `1..len(arr)`, but `i` only ranges `1..1000`, so both n=1000 and n=10000 search for targets `1..1000` — both of which land in the front of a sorted ascending array. The cost is approximately *average position of target* × *1000 searches* = 500 × 1000 = 500,000 comparisons in both cases. A fairer test would search for random targets across the full range.

35.4 — A cubic in the wild

```
triple_sum: func(arr, target) [
  c: 0
  n: len(arr)
  i: 1
  while i <= n - 2 [
    j: i + 1
    while j <= n - 1 [
      k: j + 1
      while k <= n [
        if arr[i] + arr[j] + arr[k] == target then [c = c + 1]
        k = k + 1
      ]
      j = j + 1
    ]
    i = i + 1
  ]
  c
]
```

Measured output:

```
50  ->   38.4 ms
100 ->  274.7 ms     (7.15×)
200 -> 2,224.9 ms    (8.10×)
```

(a) Growth class: `O(n³)`. Three nested loops, total work `n(n−1)(n−2)/6 ≈ n³/6`.

(b) Empirical: doubling `n` multiplies the time by roughly 8× — exactly the cubic signature.

(c) Match: prediction matches. At `n = 400` we'd expect ~17.5 seconds. Extrapolating, `n = 800` ≈ 140 seconds (~2.5 minutes), `n = 1600` ≈ 18 minutes. This is why brute-force triple-sum is *not* a viable approach for sizable `n` — you need a faster algorithm (the standard trick: sort, then for each `i`, do a two-pointer scan in `O(n)`, for total `O(n²)` instead of `O(n³)`).

35.5 — The hidden cost of array concatenation

```
make: func(n) [
  xs: []
  j: 1
  while j <= n [
    xs = xs + [j]
    j = j + 1
  ]
  xs
]
```

Measured output:

```
1000 ->   5.5 ms
2000 ->  10.0 ms     (1.83×)
4000 ->  33.1 ms     (3.32×)
8000 -> 111.2 ms     (3.36×)
```

(a) Prediction from code: `xs + [j]` copies the entire array `xs` to build a new one. At iteration `k`, this copy costs `k`. Total: `1 + 2 + ... + n = n(n+1)/2 ≈ n²/2`. **Class: `O(n²)`.**

(b) Measurement: doubling ratios 1.83, 3.32, 3.36 — approaching 4×. The first ratio is below 4× because at `n = 1000` the constant overhead still dominates; by `n = 8000` we see the true quadratic behavior.

(c) The accidental-quadratic bug. This pattern — `xs = xs + [j]` inside a loop — is one of the most common performance bugs in beginner code. The code *looks* linear ("loop once, do constant work each step") but the "constant work" is secretly `O(n)`.

Fix patterns:

- Build via comprehension when possible (`{j | j <- 1..n}`, though note set semantics, not list).
- Use a buffer / pre-allocated array.
- Use an in-place append builtin if one exists.

This bug is the **first** thing to check when a function that *should* scale linearly is actually scaling quadratically. Always inspect the inner-loop allocation.

35.6 — Budget table for your machine

This exercise is open-ended; here's one plausible answer based on the §35.4 and §35.5 measurements:

Algorithm	n	Predicted time	n for 1 sec	n for 1 hour
`linear_sum`	100,000	~5,200 ms	~19,000	~70,000,000
`bubble_sort`	10,000	~150 sec	~3,200	~190,000
`count_pairs` (n^2)	10,000	~71 sec	~3,700	~225,000

Method: For each algorithm, take a measured timing at a known `n`, then scale by the complexity class.

For `linear_sum`: 52 ms at n=1000 → ~5200 ms at n=100,000.

For `bubble_sort` (using §35.5 chapter data, 243 ms at n=400): time at n=10,000 is `(10000/400)² × 243 ms = 625 × 243 = 151,875 ms ≈ 152 seconds`.

For `count_pairs` from Ex 35.2 (457 ms at n=800): `(10000/800)² × 457 = 156.25 × 457 = 71,406 ms ≈ 71 sec`.

(a) Where did predictions match? The linear case is usually within ±10% — Axioma's inner loop is well-behaved and predictable. The quadratic cases can drift more because the actual workload (number of swaps in bubble sort, number of pairs found in count_pairs) depends on input distribution.

(b) Most common mismatch source:

- For *small* predicted `n`, the constant-factor overhead is not yet swamped — extrapolating up is reliable.
- For *very large* `n`, **memory** becomes the bottleneck: arrays larger than CPU cache experience a step-change in cost. A perfectly `O(n)` function can suddenly look super-linear at `n = 10⁶` because each access starts missing the L2 cache.

This is the **dropout #3** of §35.8 (cache effects) materializing in practice. The asymptotic class is still `O(n)`; the constant factor changes.

Notes

35.1 — Why "growth class" is the right abstraction

The lesson of §35.1 is that **the question "how fast is this function?" is ill-posed**. The right question is "how does it scale?" — because the answer to *that* is composable across machines, languages, and inputs.

When you tell a colleague "this function is O(n)", they know exactly what to expect when their input grows. When you tell them "this function took 53 ms on 1000 items," they have to do mental arithmetic to predict 5300 ms on 100,000 items — and that arithmetic only works if your function is linear. Big-O packages the scaling rule into the type signature.

35.2 — Brute-force pair-sum and its better cousins

The brute-force `O(n²)` is sometimes the *right* algorithm — for `n < 100` or so, the simplicity is worth the cost. But CS2 will teach two improvements:

- **Hash-table version: `O(n)`.** For each `arr[i]`, look up `target - arr[i]` in a hash set of previously-seen values. One pass, constant-time lookup → linear total.
- **Sort + two-pointer: `O(n log n)`.** Sort the array (n log n), then walk a pointer from each end inward (linear).

The first is faster asymptotically but uses extra memory. The second is in-place but slightly slower. Both beat the brute force for any `n` above ~50.

35.3 — Real-world switchovers

Production sort routines (Python's Timsort, C++'s introsort) all use a **hybrid strategy**: switch to insertion sort below ~16 elements, switch to quick sort above. The thresholds were tuned empirically over years of benchmarking. The "right" answer for switchover depends on:

- The interpreter / compiler (Python differs from Go)
- The CPU architecture (cache size, branch predictor)
- The data distribution (sorted vs random vs adversarial)

A general rule of thumb: **measure on your target hardware with your target inputs**. Don't trust folk-wisdom constants from a different system.

35.4 — When cubic is the *right* choice

Cubic algorithms get a bad rap, but for some problems they're inherent. Matrix multiplication, the standard 3-SUM problem, all-pairs shortest paths — these have lower bounds at or near `O(n³)` (more precisely: `O(n^ω)` where `2 ≤ ω < 3`).

When a problem is *inherently* cubic, you can't escape it — you can only:

- Use smaller `n` (sometimes possible).
- Use parallelism (turn cubic-time into cubic/p-time).
- Approximate (give up exact answers for closeness).

Knowing *which* problems are inherently hard is the realm of computational complexity theory — a notch beyond Big-O.

35.5 — The "build a list" anti-pattern, generalized

The hidden quadratic of `xs = xs + [j]` has analogs in every language:

Language	Quadratic anti-pattern	Linear fix
Python	`s = ""; s += x for x in arr`	`"".join(arr)`
Java	`String s = ""; s += x`	`StringBuilder`
JavaScript	`s = s + arr[i]` in loop	`arr.join("")`
Go	`s = s + str` in loop	`strings.Builder`
Axioma	`xs = xs + [j]` in loop	comprehension when possible

The deep lesson: **immutable update has a cost**. When you need to grow a structure incrementally, the language is either:

- Doing copy-on-write (quadratic in loop) — naive arrays and strings.
- Doing structural sharing (linear amortized) — persistent data structures like Clojure's vectors.
- Using a mutable buffer (linear) — Go's `bytes.Buffer`.

Knowing your tool's actual implementation is the difference between accidental quadratic and accidental linear.

35.6 — Budgeting as design

The budget table is the bridge between Big-O theory and engineering practice. A senior engineer reading a design doc doesn't ask "is this algorithm correct?" — they ask "will this algorithm fit the budget?"

The skill of writing this table in your head, before implementation, is what makes someone good at choosing algorithms. Big-O analysis is the language of that prediction.

Chapter 36 · Solutions

All timings measured on the textbook-development machine via `axioma --no-kb`. Your numbers will differ; the **ratios** are what classify the algorithm.

36.1 — Stability test

A sort is *stable* if it preserves the relative order of equal-keyed elements.

```
by_first: func(p, q) [p[1] <= q[1]]

# Bubble sort by .first
bubble_pairs: func(a) [
  s: a
  m: len(s)
  pass: 0
  while pass < m - 1 [
    k: 1
    while k < m - pass [
      if s[k][1] > s[k + 1][1] then [
        t: s[k]
        s[k] = s[k + 1]
        s[k + 1] = t
      ]
      k = k + 1
    ]
    pass = pass + 1
  ]
  s
]
# Similarly insertion_pairs, qsort_pairs.
```

(a) Stability empirically:

- **Bubble: stable.** Only swaps when *strictly* greater (>, not >=). Equal-value pairs preserved.
- **Insertion: stable.** Same reasoning — inner loop walks back only while `s[j] > key`, stops at equality.
- **Quicksort (3-way partition variant from chapter 36.6): stable** for our implementation (the `equal` bucket is produced in scan order, and concatenation preserves order). The **2-way partition** version is *not* stable.

(b) Theoretical:

- **Stable sorts:** merge, bubble, insertion, the 3-way partition quicksort.
- **Unstable sorts:** heap (the sift-down operation swaps across non-adjacent positions), 2-way partition quicksort (swaps elements across the array during partition), selection (the swap-min-to-front step crosses equal elements).

The rule of thumb: **a sort is stable iff it only ever swaps adjacent elements**. Adjacent swaps preserve relative order between equal-keyed neighbors transitively. Cross-array swaps don't.

36.2 — Insertion's best case

```
# Build sorted, reverse-sorted, and random inputs of size 1000
# Time insertion on each.
```

Measured (n=1000):

```
sorted (best):    3.5 ms
random:         374.8 ms     (105×)
reverse (worst): 741.4 ms    (209×)
```

(a) Ratio worst:best ≈ 200×. Two orders of magnitude between insertion's best and worst case.

(b) When does insertion win? When the input is **nearly sorted**. In production code, insertion sort is the inner loop of std-lib sort routines (Timsort uses it for sorted "runs"). The reason: log-files, time-series data, "yesterday's data plus a few new entries" — these are *almost* sorted, and insertion is `O(n)` on them.

For randomly-distributed input, insertion is comparable to the other `O(n²)` sorts. For *adversarial* (reverse-sorted) input, it's their equal in cost. Knowing the input distribution is what unlocks insertion's superpower.

36.3 — Quicksort's worst case

With the middle-element pivot rule, sorted input is hostile: the middle pivot is the median of the array, so the first recursion is balanced — but each subsequent recursion picks *the median of a sorted subarray*, which is again the middle. **So actually middle-element pivots work well on sorted input!**

The truly hostile input depends on the pivot rule:

- **First-element pivot:** sorted input is catastrophic.
- **Random pivot:** no input is consistently catastrophic.
- **Middle-element pivot:** the *constructed adversarial* input is hostile (alternating pattern designed to push the pivot to an extreme).

Demonstration with first-element pivot variant (modify `qsort` so `pivot = a[1]`):

```
n=800 sorted:    18,000 ms  (hostile — pivot is the min, every recursion)
n=800 random:       250 ms
n=800 reverse:   17,500 ms  (also hostile — pivot is the max)
```

The 70× difference between random and sorted is the `O(n²)` worst case manifesting. The lesson: **always use randomized or median-of-three pivots** in production code.

36.4 — Measuring the merge-sort surprise

Measured:

```
merge:
  n=100 →   2,243 ms
  n=200 →   9,237 ms     (4.12×)
  n=400 →  37,096 ms     (4.02×)

bubble:
  n=100 →      17 ms
  n=200 →      62 ms     (3.65×)
  n=400 →     244 ms     (3.94×)
```

(a) Merge sort doubling ratio: ~4× — the signature of `O(n²)` (or `O(n² log n)`, where the `log n` factor adds a small bias toward larger ratios as `n` grows).

(b) Bubble sort doubling ratio: ~4× — also `O(n²)`. Both are quadratic.

(c) At n=400, bubble is 152× faster than merge. Bubble's constant factor is small (in-place swaps); merge's is huge (list construction via concatenation, function call overhead per recursion).

(d) Why is merge slower if both are quadratic?

Merge sort in Axioma is `O(n² log n)` due to the accidental quadratic in `result + [x]`. Bubble sort is `O(n²)`. So at `n = 400`:

- Bubble: `400² = 160,000` units of work.
- Merge: `400² × log₂(400) ≈ 160,000 × 8.6 = 1.38M` units of work.

That's ~8.6× more work for merge. The empirical 152× difference includes the larger per-unit constant (function call overhead, sub-array allocation per recursion). Both factors compound.

The fix: mutable-buffer merge sort.

- Pre-allocate a result array of size `n`.
- Have merge write to indices, not concatenate.
- Restores theoretical `O(n log n)`.

36.5 — A counting sort

```
counting_sort: func(arr, k) [
  counts: []
  i: 0
  while i <= k [
    counts = counts + [0]
    i = i + 1
  ]
  j: 1
  while j <= len(arr) [
    v: arr[j]
    counts[v + 1] = counts[v + 1] + 1
```

```
    j = j + 1
  ]
  out: []
  v: 0
  while v <= k [
    c: counts[v + 1]
    while c > 0 [
      out = out + [v]
      c = c - 1
    ]
    v = v + 1
  ]
  out
]

println(counting_sort([3, 1, 4, 1, 5, 9, 2, 6, 5, 3], 10))
# → [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]
```

Measured (n=10000):

```
k=100:    679.7 ms
k=10000:  845.7 ms
```

(a) Complexity: `O(n + k)`. Plus the `O(k)` setup and `O(k)` output-loop overhead. In Axioma the `out + [v]` appends add a hidden $O(n^2)$ term — but the tight inner loop keeps it small enough that the asymptotic story holds.

(b) k=100 is faster than k=10000 by 166 ms — the extra buckets to scan in the output loop. At n=10000, the n-pass dominates; the k=100 vs k=10000 difference reflects the extra `k - 100` scanned zero-counts.

(c) Use counting sort when:

- Values are non-negative integers (or mappable to them).
- The range `k` is bounded and not too large relative to `n`. The rule of thumb: `k ≤ n` for counting sort to beat `O(n log n)`.
- When `k >> n`, the bucket setup dominates and you're better off with quicksort.

Real-world use: sorting ages (range 0-120), sorting digits in radix sort (range 0-9), sorting bytes (range 0-255). Whenever the keyspace is small and known, counting sort crushes the comparison-based competition.

36.6 — Build a hybrid

```
smart_sort: func(a) [
  n: len(a)
  small: n < 16
  if small then [
    insertion(a)
  ] else [
```

```
    qsort(a)
  ]
]
```

(a) Empirical comparison: at sizes 100, 200, 400 on random input, `smart_sort` and pure `qsort` perform similarly because the top-level call always falls into the `qsort` path. The hybrid's advantage shows in the **recursive calls**: when `qsort` recurses on a partition of size < 16, it should switch to insertion.

The fully-correct hybrid recurses with `smart_sort` (not `qsort`):

```
smart_qsort: func(a) [
  n: len(a)
  small: n < 16
  if small then [
    insertion(a)
  ] else [
    pivot: a[n div 2 + 1]
    parts: partition_qs(a, pivot)
    s_less: smart_qsort(parts[1])
    s_greater: smart_qsort(parts[3])
    s_less + parts[2] + s_greater
  ]
]
```

Typical improvement: 10-30% over pure quicksort on medium-size random inputs.

(b) Hybrid wins by: ~10-30% on typical inputs.

(c) Why does the switchover help even though both algorithms are `O(n log n)` average?

Because **constant factors are real**. Insertion sort on a 10-element array does 50-ish operations. Quicksort on a 10-element array does:

- Partition pass (3-way): 10 element examinations.
- Three recursive calls: each with ~3 elements.
- Each of those: another partition pass, plus more recursion.
- Per recursive call: function-call overhead, parameter passing, environment setup.

For tiny arrays, the recursion overhead swamps the algorithmic benefit. Insertion's tight inner loop wins. Production sort routines (introsort, Timsort, pdqsort) all use this kind of size-based switchover, typically at thresholds in the 12-32 range.

36.7 — Selection, two more ways

```
func fmin([x])      [x]
func fmin([h | t])  [
  m: fmin(t)
  if h < m then h else m
]
func drop_val(x, [])      [[]]
func drop_val(x, [h | t]) [ if x == h then t else [h | drop_val(x, t)] ]
func f_selection([])  [[]]
func f_selection(xs)  [
  m: fmin(xs)
  [m | f_selection(drop_val(m, xs))]
]

relation min_of(xs, m)
min_of([X], X) :- true
min_of([H | T], H) :- min_of(T, M) and H <= M
min_of([H | T], M) :- min_of(T, M) and H > M
relation del1(x, xs, ys)
del1(X, [X | T], T) :- true
del1(X, [H | T], [H | R]) :- X != H and del1(X, T, R)
relation ssel(xs, ys)
ssel([], []) :- true
ssel(L, [M | S]) :- min_of(L, M) and del1(M, L, R) and ssel(R, S)

xs: [3, 1, 4, 1, 5]
first({Y | Y <- ssel(xs, Y)})   # → [1, 1, 3, 4, 5]
f_selection(xs)                 # → [1, 1, 3, 4, 5]
```

(c) Both agree with `sort(xs)` and with the chapter's imperative `selection`. `del1`'s `X != H` guard is the cut-free way to delete only the first occurrence — without it, duplicate minima fork extra answers.

The full three-paradigm suite, including bubble / merge / quick, is `tests/axioma/sorting/test_ch36_three_paradigms.ax`.

Notes

36.1 — Stability matters more than you think

Stability is the property that **multi-key sorting works correctly**. To sort users first by department, then by name within department: sort by name first (any sort), then sort by department (must be stable). If the second sort isn't stable, names get scrambled within each department.

Python's `sort()` is guaranteed stable. C's `qsort()` is not. Knowing your tool's guarantees prevents subtle bugs when multi-key sorting.

36.2 — The adaptive sort family

Insertion is the simplest "adaptive" sort — one that runs faster on partially-sorted input. The adaptive family includes:

- **Insertion**: `O(n + d)` where `d` is the number of inversions.
- **Timsort**: detects already-sorted "runs" and merges them; can be `O(n)` on fully-sorted input.
- **Smoothsort**: like heap sort but `O(n)` on sorted input.

For workloads where data arrives mostly-sorted (logs, time-series, append-only event streams), adaptive sorts can be 100× faster than non-adaptive ones.

36.4 — Two takeaways from the merge-sort surprise

Takeaway 1: Big-O analysis depends on the cost model. The recurrence `T(n) = 2T(n/2) + O(n)` is correct *in a cost model where array concatenation is O(1)*. Axioma's cost model isn't that. When you read complexity claims, ask: in what cost model? RAM model? Bit operations? Memory accesses? The answer matters.

Takeaway 2: Measure, don't assume. A senior engineer who'd been taught "merge sort is O(n log n)" their whole career would write the chapter's merge sort, hit production slowness, blame the language, and miss the lesson. The analysis was right; the cost model was wrong. The fix is to *change the implementation* (mutable buffer) so it matches the assumed cost model.

This is exactly the failure mode that **empirical complexity verification** (Ch.35's recipe) protects you from. Always measure.

36.5 — Non-comparison sorts are an escape hatch

The comparison lower bound — $\Omega(n \log n)$ for any comparison-based sort — is one of the most famous results in CS. It's proven via the decision-tree argument: any comparison sort can be drawn as a binary tree where each node is a comparison; a tree with $n!$ leaves (the number of permutations of an n-element array) must have depth $\geq \log_2(n!) = \Theta(n \log n)$.

The escape: don't compare. Radix sort, counting sort, bucket sort all examine *value structure* — digits, range buckets — rather than ordering. They beat the lower bound by sidestepping its premise.

This is a general design lesson: **lower bounds apply to *classes* of algorithms. If you can change classes, you can escape**. Whenever you see an algorithm hitting a lower bound, ask if there's value structure you can exploit to switch classes.

Chapter 37 · Solutions

All exercises use the integer-indexed adjacency-matrix scaffolding from §37.2.

37.1 — BFS path reconstruction

```
bfs_paths: func(g, start) [
  n: len(g)
  visited: make_bools(n)
  visited[start] = true
  parent: []
  i: 1
  while i <= n [
    parent = parent + [-1]   # -1 = no parent yet
    i = i + 1
  ]
  queue: []
  queue = queue + [start]
  head: 1
  while head <= len(queue) [
    node: queue[head]
    head = head + 1
    j: 1
    while j <= n [
      if g[node][j] != 0 then [
        if not visited[j] then [
          visited[j] = true
          parent[j] = node
          queue = queue + [j]
        ]
      ]
      j = j + 1
    ]
  ]
  parent
]

reconstruct: func(parent, target) [
  path: [target]
  cur: target
  while parent[cur] != -1 [
```

```
    cur = parent[cur]
    path = [cur] + path
  ]
  path
]
```

On the sample graph from §37.2:

```
bfs_paths(g, 1) → parents [-1, 1, 1, 2, 4]
reconstruct(p, 5) → [1, 2, 4, 5]   # A → B → D → E
```

Lesson: BFS gives shortest paths *in an unweighted graph* "for free" — every node's `parent` is set when first discovered, which is at minimum BFS distance.

37.2 — Cycle detection

```
has_cycle_helper: func(g, node, parent, visited) [
  visited[node] = true
  n: len(g)
  found: false
  j: 1
  while j <= n [
    if g[node][j] != 0 then [
      if not visited[j] then [
        sub: has_cycle_helper(g, j, node, visited)
        if sub then [found = true]
      ] else [
        # j is visited; cycle if j != parent
        if j != parent then [found = true]
      ]
    ]
    j = j + 1
  ]
  found
]

has_cycle: func(g) [
  n: len(g)
  visited: make_bools(n)
  found: false
  i: 1
  while i <= n [
    if not visited[i] then [
      sub: has_cycle_helper(g, i, -1, visited)
      if sub then [found = true]
    ]
    i = i + 1
  ]
```

```
  found
]
```

The "parent" trick: an edge to the parent is just the edge we came in on, not a cycle. An edge to any *other* visited node is a back-edge → cycle.

Tree (no cycle): `has_cycle` returns `false`. **With an extra A-D edge added:** `has_cycle` returns `true`.

37.3 — Connected components

```
count_components: func(g) [
  n: len(g)
  visited: make_bools(n)
  components: 0
  i: 1
  while i <= n [
    if not visited[i] then [
      components = components + 1
      # BFS from i to mark its component
      queue: []
      queue = queue + [i]
      visited[i] = true
      head: 1
      while head <= len(queue) [
        node: queue[head]
        head = head + 1
        j: 1
        while j <= n [
          if g[node][j] != 0 then [
            if not visited[j] then [
              visited[j] = true
              queue = queue + [j]
            ]
          ]
          j = j + 1
        ]
      ]
    ]
    i = i + 1
  ]
  components
]
```

On the sample graph: 1 component. On a graph with {A-B, C-D, E-F}: 3 components.

37.4 — Dijkstra empirical scaling

Illustrative timing ratios on dense graphs (not measurements from this revision):

```
n=50  →   85.3 ms
n=100 →  333.4 ms     (3.91×)
n=200 → 1,289.6 ms    (3.87×)
```

Ratios near 4× are consistent with quadratic growth, not a proof. The chapter's timer surrounds only Dijkstra; graph construction happens before the callback. Run the experiment locally instead of treating these illustrative numbers as a performance promise.

37.5 — Negative weights break Dijkstra

Use a directed graph with edges A → B (2), A → C (5), and C → B (-10). There is no cycle. Dijkstra settles B at distance 2 before visiting C, but the path A → C → B costs -5. Settling a node too early invalidates the nonnegative-weight proof. The implementation in §37.5 rejects this input before attempting the algorithm.

```
negative_graph: make_matrix(3)
negative_graph[1][2] = 2
negative_graph[1][3] = 5
negative_graph[3][2] = -10
println(has_negative_edge(negative_graph))  # true
```

For a graph with negative weights, Bellman–Ford relaxes all edges for up to `n - 1` rounds and uses a further pass to detect reachable negative cycles. An undirected negative edge permits a negative round trip, so shortest *walks* through its connected component have no finite minimum.

37.6 — Prim vs Kruskal

Kruskal on the sample graph from §37.5 (weights: A-B=2, A-C=5, B-C=1, B-D=3, C-E=4, D-E=1):

Sort all edges by weight:

- B-C (1), D-E (1), A-B (2), B-D (3), C-E (4), A-C (5).

Add edges in order, skipping any that create a cycle:

- B-C (1): add. Components: {A}, {B, C}, {D}, {E}.
- D-E (1): add. Components: {A}, {B, C}, {D, E}.
- A-B (2): add. Components: {A, B, C}, {D, E}.
- B-D (3): add. Components: {A, B, C, D, E} — done.

Total weight: 1 + 1 + 2 + 3 = 7.

Prim from A (from §37.6):

- A-B (2), B-C (1), B-D (3), D-E (1). Total: 7.

(a) Same tree? *Same weight*, same edge set in this case. In general, Prim and Kruskal can produce

different MSTs of equal weight when there are ties — both are valid minimum spanning trees.

(b) Why might they differ? When two non-tree edges have the same weight, Prim and Kruskal might pick different ones (Prim picks based on current frontier, Kruskal by sorted order). Both produce *a* MST; neither is "more correct".

(c) Which is simpler? Kruskal is conceptually simpler — sort + add-if-no-cycle. But it requires a **union-find** data structure for the cycle check, which is its own subtopic. **Prim's** algorithm doesn't need union-find but does need to track the "frontier" (neighbors of in-tree nodes).

For adjacency-matrix dense graphs, **Prim is shorter**. For sparse graphs given as an edge list, **Kruskal** plus a fast union-find wins.

Notes

37.1 — Parent arrays are the standard trick

The parent-array technique is universal across graph algorithms:

- BFS/DFS: parent → reconstructable path.
- Dijkstra: parent → shortest path with weights.
- Prim: parent → MST edges.
- Topological sort: pre-order vs post-order parents → DAG structure.

Whenever you write a graph algorithm that builds a tree of decisions, the parent array captures the tree shape for later use.

37.5 — Algorithmic preconditions are real

Dijkstra's proof requires nonnegative weights. The chapter's executable implementation checks this at its boundary and returns an error when the condition fails. The preceding negative-edge example tests the predicate separately; calling `dijkstra(negative_graph, 1)` is an intentional failure.

Preconditions should state the accepted data, and practical checks should catch violations where possible. This does not remove the need to explain representation conventions such as zero meaning "no edge" in this chapter.

37.6 — MSTs aren't unique

The MST of a graph isn't necessarily unique — graphs with weight ties can have multiple MSTs of identical total weight. This matters when:

- Comparing two MST implementations for correctness (use *total weight*, not *edge set equality*).
- Tie-breaking matters operationally (e.g., MST for network design, where some edges are "more practical" than others at the same cost).

In production, you'd often add a secondary cost (cable length, latency) as a tiebreaker — making the MST unique by construction.

Chapter 38 · Solutions

State-machine solutions build the state history as an Array. A singleton such as `[start,]` is an explicit one-element Array, not a String.

38.1 — Combination lock

```
code_lock: {
  "idle.1":         "entering_1",
  "entering_1.2":   "entering_2",
  "entering_2.3":   "entering_3",
  "entering_3.4":   "unlocked",
  "idle.x":         "wrong_1",
  "wrong_1.x":      "wrong_2",
  "wrong_2.x":      "locked_out"
}

do_step: func(m, st, inp) [
  k: st + "." + inp
  next_state: m[k]
  if next_state == none then [st] else [next_state]
]
```

Any input not in the table → stay in current state. Three "x" (any wrong digit) inputs from idle → locked_out. Four correct sequential digits ("1", "2", "3", "4") from idle → unlocked.

Lesson: the "default = stay" rule handles the inputs not enumerated. Without it, you'd need to enumerate every state × every input.

38.2 — Coverage check

```
check_complete: func(machine, states, inputs) [
  missing: []
  i: 1
  while i <= len(states) [
    j: 1
    while j <= len(inputs) [
      key: states[i] + "." + inputs[j]
      if machine[key] == none then [
```

```
        missing = missing + [key]
      ]
      j = j + 1
    ]
    i = i + 1
  ]
  missing
]
```

Returns the list of missing (state, input) keys. An empty list means complete coverage.

Lesson: for any state machine whose transition table must cover all (state, input) pairs (TCP, parsers, protocol state machines), automate this check. The number of bugs caught by a coverage check on a complex protocol state machine in production is staggering.

38.3 — Vending machine

```
idle_state: func(input) [
  if input == "insert_coin" then ["selecting", "selecting_state"]
  else ["idle", "idle_state"]
]

selecting_state: func(input) [
  if input == "select_product" then ["paying", "paying_state"]
  else if input == "cancel" then ["idle", "idle_state"]
  else ["selecting", "selecting_state"]
]

paying_state: func(input) [
  if input == "complete" then ["dispensing", "dispensing_state"]
  else if input == "cancel" then ["idle", "idle_state"]
  else ["paying", "paying_state"]
]

dispensing_state: func(input) [
  if input == "complete" then ["idle", "idle_state"]
  else ["dispensing", "dispensing_state"]
]
```

The per-state function style. Each handler returns [next_state_name, next_state_function_name]. The main loop looks up the new function by name (§38.4 pattern).

When per-state functions help: when handlers need to do real work (charge a credit card, dispense a product, play a sound). The dict-table doesn't have a place for that work; the per-state function does.

38.4 — Traffic light with timer

```
light: {
  "green.tick":  "green",   # will increment, but logically stays green
  "yellow.tick": "yellow",
  "red.tick":    "red"
}

advance: func(state, count) [
  if state == "green" then [
    if count >= 30 then ["yellow", 0] else [state, count + 1]
  ] else if state == "yellow" then [
    if count >= 5 then ["red", 0] else [state, count + 1]
  ] else [
    if count >= 25 then ["green", 0] else [state, count + 1]
  ]
]
```

State transitions are *guarded by count* rather than input alone. Each tick: increment count if still in the state; reset count + transition if threshold met.

Lesson: timer-driven state machines are everywhere (network keepalives, UI animations, the Cascade reactor's debounce window). The pure FSM model needs augmenting with a *clock* or *counter*.

38.5 — Two machines, composed

```
mod3: func(s, _) [(s + 1) % 3]
mod5: func(s, _) [(s + 1) % 5]

run_paired: func(presses) [
  s3: 0
  s5: 0
  i: 1
  while i <= presses [
    s3 = mod3(s3, "press")
    s5 = mod5(s5, "press")
    if s3 == 0 then [
      if s5 == 0 then [println("press", i, ": both 0!")]
    ]
    i = i + 1
  ]
]
run_paired(15)
# → press 15: both 0!
```

Lesson: two independent state machines fed the same input. They reset to 0 together at the LCM(3, 5) = 15. This is composition by *parallel composition*.

In production: a TCP connection's state machine is *composed* with the application's state machine; both react to packet-arrival events independently.

38.6 — Critique a real state machine

Sample answer using TCP (RFC 793):

(a) How many states? TCP has **11 states**: CLOSED, LISTEN, SYN_SENT, SYN_RECEIVED, ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2, CLOSE_WAIT, CLOSING, LAST_ACK, TIME_WAIT.

(b) Time- vs input-triggered:

- *Input-triggered:* SYN_SENT → ESTABLISHED (on receiving SYN+ACK); ESTABLISHED → FIN_WAIT_1 (on application calling close()).
- *Time-triggered:* TIME_WAIT → CLOSED (after 2 × MSL timeout, typically 60-240 seconds); SYN_SENT → CLOSED (after retry budget exceeded).

(c) Impossible inputs: RFC 793 specifies: "in CLOSED state, drop the segment and reply with RST." In every other state, unexpected segments either get a RST reply or are silently dropped. The "be liberal in what you accept" Postel principle in action.

Lesson: real production state machines are **defensive** — they assume any input can arrive in any state, and each combination is documented. Sloppy code crashes on unexpected input; production code degrades gracefully.

Notes

38.1 — The default-stay convention

In production code, having every transition table entry explicit makes the table huge. The "missing entries default to stay in current state" convention shrinks the table to *only* meaningful transitions. The trade-off: you can no longer use the empty entry to detect "should not happen" — you need a separate `disallowed` table.

38.4 — Hierarchical state machines preview

The timer-driven traffic light is the simplest case of a **hierarchical** state machine: the "outer" state is green/yellow/red, the "inner" state is the tick count. Harel statecharts make hierarchy a first-class concept; plain FSMs need to fold the hierarchy into a single state space.

38.6 — State machines as a debugging tool

If your code has bugs that only manifest "in certain orders" — race conditions, sequence dependencies — the fix is often to **make the implicit state explicit** by writing the state machine. The act of enumerating states forces you to confront which inputs are possible in which states, and the bugs become visible in the transition table.

The state machine doc *is* the debugging tool.

Chapter 39 · Solutions

Reactor and event-loop solutions use the construction-time dictionary of handlers from §39.7. Mutable state captured by callbacks is updated with `rebind`; indexed dictionary updates do not replace the captured binding.

39.1 — Empty-queue start

```
queue: []
# (no initial events)

head: 1
if len(queue) == 0 then [
  println("nothing to do; exiting")
] else [
  while head <= len(queue) [
    ev: queue[head]
    head = head + 1
    println("dispatch:", ev)
  ]
]
```

Recommendation: *zero iterations, no error.* The empty-queue case isn't a bug — it's a degenerate but valid input (think: starting an event loop before any events have been posted is the standard pattern; the *initial* state of a GUI is always "queue empty, waiting for first user input").

A production loop would, instead of exiting, *block* on an external source (kernel `epoll`, SDL_PollEvent, etc.). For a synchronous in-process loop, "drain and exit" is the right behavior — exiting on `len(queue) == 0` means tools that build a queue then run it once work cleanly with zero events as a no-op.

Lesson: *no input* is a valid input. Handle it explicitly rather than letting it manifest as a crash or an infinite block.

39.2 — Bounded queue

```
make_reactor: func(handlers, cap) [
  queue: []
  dropped: 0
```

```
  post: func(typ, payload) [
    full: len(queue) >= cap
    if full then [
      rebind dropped = dropped + 1
    ] else [
      rebind queue = queue + [(typ, payload)]
    ]
  ]
  run: func() [
    head: 1
    while head <= len(queue) [
      ev: queue[head]
      head = head + 1
      h: handlers[ev[1]]
      if h != none then [h(ev[2])]
    ]
    head - 1
  ]
  {
    "post":    post,
    "run":     run,
    "dropped": func() [dropped]
  }
]

r: make_reactor({
  "tick": func(p) [println("tick ", p)]
}, 10)

i: 1
while i <= 100 [
  r["post"]("tick", i)
  i = i + 1
]
n: r["run"]()
println("processed:", n)
println("dropped:", r["dropped"]())
```

Output: 10 ticks printed, “processed: 10”, “dropped: 90”. The cap protects the loop from being overwhelmed during a burst — the first 10 events get processed, the remaining 90 are dropped (counted, not silently lost).

Lesson: *bounded queues* are the difference between an event loop that survives load spikes and one that OOM-crashes. The `dropped` counter is your **backpressure metric** — if it’s non-zero, something is wrong upstream (the producer is sending too fast, or your handlers are too slow).

In production: dropped events should be logged with event type, so you can answer “which event type is overwhelming us?” Cascade emits this to telemetry on every drop.

39.3 — Queue depth introspection

```
make_reactor: func(handlers) [
  queue: []
  post: func(typ, payload) [
    rebind queue = queue + [(typ, payload)]
  ]
  depth: func() [len(queue)]
  run: func() [
    head: 1
    while head <= len(queue) [
      ev: queue[head]
      head = head + 1
      h: handlers[ev[1]]
      if h != none then [h(ev[2])]
    ]
  ]
  { "post": post, "depth": depth, "run": run }
]

r: make_reactor({})
h_tick: func(p) [
  println("[t=", p, "] depth=", r["depth"]())
  small: p < 3
  if small then [
    r["post"]("tick", p + 1)
    r["post"]("tick", p + 1)  # 2x to demonstrate growth
  ]
]
r = make_reactor({ "tick": h_tick })
r["post"]("tick", 0)
r["run"]()
println("done, depth=", r["depth"]())
```

Sample output:

```
[t= 0 ] depth= 1
[t= 1 ] depth= 3
[t= 1 ] depth= 5
[t= 2 ] depth= 7
...
[t= 3 ] depth= 15
done, depth= 15
```

Note `depth` returns `len(queue)` including already- processed events (the head-pointer doesn't shrink the array). A more accurate "depth remaining" would be `len(queue) - head + 1` — but `head` isn't exposed from the closure. The exercise is open-ended; either answer is acceptable as long as the rise+fall pattern is demonstrated.

Lesson: introspection is debugging insurance. The two minutes to add `depth()` saves hours when you're trying to diagnose a misbehaving reactor in production. Build the dashboard before you need it.

39.4 — Priority levels

```
make_priority_reactor: func(handlers) [
  high: []
  normal: []
  stop_flag: false

  post: func(typ, payload, prio) [
    if prio == "high" then [
      rebind high = high + [(typ, payload)]
    ] else [
      rebind normal = normal + [(typ, payload)]
    ]
  ]
  halt: func() [rebind stop_flag = true]

  run: func() [
    head_h: 1
    head_n: 1
    keep: true
    while keep [
      have_h: head_h <= len(high)
      have_n: head_n <= len(normal)
      halted: stop_flag
      if halted then [keep = false]
      else if have_h then [
        ev: high[head_h]
        head_h = head_h + 1
        h: handlers[ev[1]]
        if h != none then [h(ev[2])]
      ]
      else if have_n then [
        ev: normal[head_n]
        head_n = head_n + 1
        h: handlers[ev[1]]
        if h != none then [h(ev[2])]
      ]
      else [keep = false]
    ]
  ]
  { "post": post, "halt": halt, "run": run }
]

r: make_priority_reactor({
  "work":     func(p) [println("work ", p)],
  "shutdown": func(p) [println("shutdown")]
})
r["post"]("work", 1, "normal")
r["post"]("work", 2, "normal")
```

```
r["post"]("shutdown", 0, "high")   # this jumps ahead
r["post"]("work", 3, "normal")
r["run"]()
```

Output:

```
shutdown
work  1
work  2
work  3
```

Note: shutdown ran *first*, even though it was posted *third* in real order. The high queue drains before normal in this design.

Lesson: strict priority (high always wins) is the right default for "shutdown" and "watchdog" events. **Weighted priority** (drain N high before 1 normal) is better when both queues need progress under sustained load. Get the simple version working first, then complicate.

In Cascade: the reactor uses three priorities — "shutdown" (instant), "user-query" (high), and "background-ingest" (normal). The shutdown class is strict, the other two are interleaved 3:1.

39.5 — Idle handler

```
make_reactor_idle: func(handlers, on_idle) [
  queue: []
  stop_flag: false
  idle_ticks: 0
  max_idle: 5  # demo: halt after 5 idle ticks

  post: func(typ, payload) [
    rebind queue = queue + [(typ, payload)]
  ]
  halt: func() [rebind stop_flag = true]

  run: func() [
    head: 1
    keep: true
    while keep [
      have: head <= len(queue)
      halted: stop_flag
      if halted then [keep = false]
      else if have then [
        ev: queue[head]
        head = head + 1
        h: handlers[ev[1]]
        if h != none then [h(ev[2])]
      ]
      else [
        rebind idle_ticks = idle_ticks + 1
```

```
        on_idle()
        if idle_ticks >= max_idle then [keep = false]
      ]
    ]
  ]
  { "post": post, "halt": halt, "run": run }
]

r: make_reactor_idle(
  { "tick": func(p) [println("tick ", p)] },
  func() [println("[idle]")]
)
r["post"]("tick", 1)
r["post"]("tick", 2)
r["run"]()
```

Output:

```
tick  1
tick  2
[idle]
[idle]
[idle]
[idle]
[idle]
```

After the two posted ticks process, the loop falls into the idle branch each iteration. In a real reactor the idle handler would *block* on a kernel syscall (e.g., `epoll_wait` with no timeout), waking when external events arrive — no busy-loop. Here we add an `idle_ticks` counter and halt at 5 to keep the demo terminating.

Lesson: idle is a legitimate phase, not a bug. GUI applications spend ~99% of their wall-clock time in `epoll_wait` waiting for the next event. The idle handler is where you do bookkeeping that shouldn't compete with event processing: GC sweeps, periodic logging, watchdog heartbeats.

39.6 — Find a real event loop

Open-ended. A representative sample answer (the **Linux kernel's softirq subsystem**):

(a) Queue ordering. Per-CPU softirq vectors, each a FIFO list of pending `tasklet_struct`. Within a vector, FIFO. Across vectors, priority order: HI_SOFTIRQ > TIMER_SOFTIRQ > NET_TX > NET_RX > BLOCK > IRQ_POLL > TASKLET > SCHED > HRTIMER > RCU. *Effectively a priority-ordered multi-queue.*

(b) Shutdown. Shutdown isn't an event in the softirq model — the kernel doesn't shut down through the event loop. The closest equivalent is `local_bh_ disable()` which suspends softirq processing on the calling CPU; `__do_softirq()` checks this flag and short-circuits.

(c) Slow handler. Softirqs have a *budget*: `MAX_SOFTIRQ_TIME` (2 jiffies = ~2ms) or `MAX_SOFTIRQ_RESTART` (10 iterations). If a softirq handler runs too long, the remainder is *deferred* to `ksoftirqd` — a

per-CPU kernel thread. This is the "slow path" — it doesn't drop events, but it does remove them from the critical interrupt context where hard real-time guarantees matter.

Lesson: real event loops have learned from decades of production incidents. The Linux softirq budget answers "what if a handler runs too long" with **defer to a worker thread** — exactly the pattern §39.9 recommended.

Notes

39.2 — Drop vs block

The bounded queue dropped events when full. The alternative is to **block** the producer (the `post` call doesn't return until queue has space). Blocking is what `bounded channel` does in Go, `mpsc::sync_channel` in Rust, and `BlockingQueue` in Java.

- **Drop** is right when **freshness matters more than completeness** (mouse-move events: only the most recent position is interesting; old ones can be discarded).
- **Block** is right when **completeness matters more than throughput** (file-write events: every byte must reach disk; producer must wait for the writer to catch up).

The choice between drop and block defines your back-pressure semantics. State it explicitly.

39.4 — Why not use the priority-queue approach

from §39.6?

Two reasons. **(a) Granularity.** Two FIFO queues give you two discrete priorities (high/normal). A priority-queue gives you arbitrary numeric priorities. For most systems, two or three discrete classes is plenty; arbitrary numeric priorities encourage priority inflation (everyone's a 9-of-10).

(b) Cost. Two FIFO queues with head pointers are O(1) per operation. A priority queue is O(log n). For high-throughput systems, the constant matters.

39.5 — Idle vs blocked

The idle handler in this exercise is *active* (busy-looping with a counter to halt). A *blocked* loop calls a syscall like `epoll_wait()` or `select()` and the kernel suspends the thread until data arrives. **Blocking is essential** for production: without it, the event loop pegs a CPU at 100% even when there's nothing to do.

Axioma doesn't have a built-in syscall for "sleep-until-event" because there isn't one universal across event-source types (timer? file descriptor? signal? user-defined?). The Cascade reactor (Ch.40) uses SQLite's WAL flush as its blocking primitive — ingenious because it doubles as durability.

39.6 — Pattern across real systems

Every event loop you'll encounter has, somewhere:

- A queue (FIFO, priority, or per-class).
- A handler-dispatch mechanism (table, vtable, pattern-match).
- A halt condition (drain, flag, timeout).
- A slow-handler escape (budget, worker thread, drop).

If you can name those four things for a system, you understand its event loop.

Chapter 40 · Solutions

Durable-reactor solutions. All extend the §40.3 `make_durable_reactor` skeleton.

40.1 — Partial run + recovery

```
make_durable_reactor: func(handlers) [
  queue: []
  wal: []
  next_seq: 1

  post: func(typ, payload) [
    seq: next_seq
    rebind next_seq = next_seq + 1
    rebind wal = wal + [(seq, "posted", typ, payload)]
    rebind queue = queue + [(seq, typ, payload)]
    seq
  ]
  mark_done: func(seq) [
    rebind wal = wal + [(seq, "processed", "", "")]
  ]
  run_n: func(limit) [
    head: 1
    processed: 0
    while head <= len(queue) [
      cont: processed < limit
      if not cont then [head = len(queue) + 1]
      else [
        ev: queue[head]
        head = head + 1
        seq: ev[1]
        h: handlers[ev[2]]
        if h != none then [h(ev[3])]
        mark_done(seq)
        processed = processed + 1
      ]
    ]
    processed
  ]
  {
    "post":  post,
```

```
    "run_n": run_n,
    "wal":   func() [wal]
  }
]

recover_from_wal: func(handlers, wal) [
  r: make_durable_reactor(handlers)
  j: 1
  while j <= len(wal) [
    e: wal[j]
    if e[2] == "posted" then [
      seq: e[1]
      found: false
      k: 1
      while k <= len(wal) [
        f: wal[k]
        if f[2] == "processed" then [
          if f[1] == seq then [found = true]
        ]
        k = k + 1
      ]
      if not found then [
        r["post"](e[3], e[4])
      ]
    ]
    j = j + 1
  ]
  r
]

# Phase 1: post 3
r1: make_durable_reactor({
  "work": func(p) [println("[pre] processing ", p)]
})
r1["post"]("work", "a")
r1["post"]("work", "b")
r1["post"]("work", "c")

# Phase 2: process 2, "crash"
r1["run_n"](2)
saved_wal: r1["wal"]()

# Phase 3: recover, run rest
r2: recover_from_wal({
  "work": func(p) [println("[post] RECOVERED + ", p)]
}, saved_wal)
r2["run_n"](100)
```

Output:

```
[pre] processing  a
```

```
[pre] processing  b
[post] RECOVERED + c
```

Only event `c` re-runs after recovery — events `a` and `b` had matching "processed" entries in the WAL.

Lesson: `run_n(limit)` lets us simulate the "crash" cleanly. In a real system, the crash happens *inside* the handler — making the `mark_done(seq)` placement after the handler critical (§40.3). If `mark_done` ran first, recovery would think a half-finished event was complete.

40.2 — Compact

```
compact: func(wal) [
  # Find seqs with processed records
  done: []
  i: 1
  while i <= len(wal) [
    e: wal[i]
    if e[2] == "processed" then [
      done = done + [e[1]]
    ]
    i = i + 1
  ]
  # Drop all entries whose seq is in done
  result: []
  j: 1
  while j <= len(wal) [
    e: wal[j]
    seq: e[1]
    drop: false
    k: 1
    while k <= len(done) [
      if done[k] == seq then [drop = true]
      k = k + 1
    ]
    if not drop then [result = result + [e]]
    j = j + 1
  ]
  result
]
```

Verification:

```
# Mock WAL: 5 posted + 3 processed → expect 2 pending after compact
wal: []
wal = wal + [(1, "posted", "work", "a")]
... (5 posts, 3 processeds)

before: 8 entries
after:  2 entries
  (4, "posted", "work", "d")
```

```
(5, "posted", "work", "e")
```

The compaction removes 3 posted/processed pairs (6 entries), leaving 2 pending posts. The result is identical-for-recovery: replaying the compacted WAL yields the same pending set.

Lesson: compaction is **safe** because the "processed" record means "this work is done; the result is durably recorded elsewhere." Once that's true, the WAL entries can go. But you must run compaction **only** during a quiescent moment — if a new "posted" arrives mid-compaction, the resequencing or filter logic can corrupt. Cascade runs compaction inside a `BEGIN EXCLUSIVE` transaction to be safe.

40.3 — Idempotent counter

```
make_idempotent_counter: func() [
  value: 0
  processed_seqs: []
  increment: func(seq, amount) [
    already: false
    i: 1
    while i <= len(processed_seqs) [
      if processed_seqs[i] == seq then [already = true]
      i = i + 1
    ]
    if already then [
      println("[skip] seq=", seq, " (already applied)")
    ] else [
      rebind value = value + amount
      rebind processed_seqs = processed_seqs + [seq]
    ]
  ]
  get: func() [value]
  { "increment": increment, "get": get }
]
```

Stress test (3× replay):

```
1st run:    value = 15  (5+3+7)
1st replay: value = 15  (all three skipped)
2nd replay: value = 15  (all three skipped)
3rd replay: value = 15  (all three skipped)
```

The counter is **idempotent**: replaying events is safe.

Lesson: the linear scan of `processed_seqs` is O(n) per increment — fine for tens-of-thousands but not millions. Production code uses a hash set (O(1) amortized). The Cascade KB does it with a SQLite `INSERT OR IGNORE` against a `(seq)` primary key — leveraging the database's existing uniqueness index for free.

The `processed_seqs` set itself must be **durable** to survive crashes. In Cascade it lives in SQLite; here we punt on that detail.

40.4 — Crash-during-handler

```
make_failing_reactor: func(handlers) [
  queue: []
  wal: []
  next_seq: 1

  post: func(typ, payload) [
    seq: next_seq
    rebind next_seq = next_seq + 1
    rebind wal = wal + [(seq, "posted", typ, payload)]
    rebind queue = queue + [(seq, typ, payload)]
  ]
  mark_done: func(seq) [
    rebind wal = wal + [(seq, "processed", "", "")]
  ]
  mark_failed: func(seq, err) [
    rebind wal = wal + [(seq, "failed", err, "")]
  ]

  run: func() [
    head: 1
    while head <= len(queue) [
      ev: queue[head]
      head = head + 1
      seq: ev[1]
      h: handlers[ev[2]]
      if h != none then [
        result: h(ev[3])
        # Handler returns ["ok"] or ["err", msg]
        if result[1] == "ok" then [
          mark_done(seq)
        ] else [
          mark_failed(seq, result[2])
        ]
      ]
    ]
  ]

  recover_failed: func() [
    # Re-queue any seq with "failed" but no later
    # "processed" (matches our 40.1 recovery pattern,
    # extended to treat failed as "still pending")
    result: []
    j: 1
    while j <= len(wal) [
      e: wal[j]
      if e[2] == "failed" then [
        seq: e[1]
```

```
        done: false
        k: j + 1
        while k <= len(wal) [
          f: wal[k]
          if f[1] == seq then [
            if f[2] == "processed" then [done = true]
          ]
          k = k + 1
        ]
        if not done then [
          # Find original posted to re-queue
          m: 1
          while m <= len(wal) [
            p: wal[m]
            if p[1] == seq then [
              if p[2] == "posted" then [
                result = result + [p]
              ]
            ]
            m = m + 1
          ]
        ]
      ]
      j = j + 1
    ]
    result
  ]

  {
    "post":    post,
    "run":     run,
    "wal":     func() [wal],
    "failed":  recover_failed
  }
]
```

Lesson: distinguishing **transient** failures (retry) from **permanent** failures (dead-letter queue, manual intervention) is the operational complication. A simple "retry-on-failed" works until you encounter a deterministic failure that retries forever. The fix: a retry counter. After N retries, mark the event as `dead_letter` and emit an alert.

In Cascade, the truth-update path retries 3× with exponential backoff; after that the failed event goes to a `dead_letter` table that triggers a human-readable telemetry alert.

40.5 — Quiescence-triggered checkpoint

```
checkpoint: func(r) [
  wal: r["wal"]()
  # Build the snapshot: list of pending (seq, typ, payload)
  pending: []
  j: 1
  while j <= len(wal) [
    e: wal[j]
    if e[2] == "posted" then [
      seq: e[1]
      done: false
      k: 1
      while k <= len(wal) [
        f: wal[k]
        if f[2] == "processed" then [
          if f[1] == seq then [done = true]
        ]
        k = k + 1
      ]
      if not done then [
        pending = pending + [(seq, e[3], e[4])]
      ]
    ]
    j = j + 1
  ]
  # Return snapshot tuple
  ("checkpoint", pending, r["next_seq"]())
]

# Recovery from snapshot is just rebuilding from
# pending list (no WAL replay needed).
```

Verification protocol: 1. Reactor R1: post 100 events, process 80. 2. `cp = checkpoint(R1)` — snapshot has 20 pending. 3. Reactor R2 from full WAL: should also yield 20 pending. 4. Reactor R3 from snapshot only: same 20 pending.

Both R2 and R3 must produce identical pending lists.

Lesson: the snapshot is **smaller** than the WAL (O(pending) vs O(total history)). Recovery from snapshot is O(pending); recovery from WAL is O(history). Hence checkpointing's value: bounding recovery time.

The "compact" of §40.2 is the **continuous** version; checkpointing is the **discrete** version that emits a recovery artifact. Pick based on recovery RTO requirements: full WAL = bigger but self-contained; snapshot = smaller but requires snapshot file to be safe.

40.6 — Find a real WAL

Open-ended. Sample answer (**SQLite WAL mode**):

(a) Format. SQLite's WAL is a **page-oriented binary file** (`<dbname>-wal`). Each entry is a 4096- byte page (or whatever the database's page size is) plus a 24-byte header (frame number, page number, salt, checksum). Designed for sequential append + binary diff.

(b) Flush. SQLite flushes on `COMMIT` by default (durability guarantee). With `PRAGMA synchronous=NORMAL`, flushing happens less often (at WAL checkpoint, not per commit) — trades durability for throughput. With `synchronous=OFF`, no explicit flush — fastest, can lose committed transactions on crash.

(c) Truncation. Via **checkpoint**: `PRAGMA wal_checkpoint(PASSIVE|FULL|RESTART|TRUNCATE)`. PASSIVE: opportunistic, no blocking. TRUNCATE: full, empties the WAL file. Automatic checkpoint happens when WAL exceeds `wal_autocheckpoint` pages (1000 default).

Lesson: SQLite's WAL is the **industry-standard embedded WAL implementation** — battle-tested across billions of devices (every iPhone, every Android, every macOS, every Mozilla product). Cascade uses it directly rather than reimplementing. **Reuse mature WAL infrastructure** unless you have a very specific reason not to.

Notes

40.2 — Why drop both posted and processed?

You could keep "processed" entries forever (an audit trail of work done). The tradeoff:

- **Drop both**: WAL stays small; lose audit trail.
- **Keep processed**: WAL grows linearly with work done; full audit trail.

In Cascade: WAL drops both pairs because the *KB itself* is the audit trail (every truth update is durably recorded in `truth_history`). The WAL only needs to track in-flight work, not history.

40.3 — Bounded vs unbounded dedup state

The `processed_seqs` list grows forever. In production:

- **Bounded window**: only dedup within the last N seqs (Cascade uses last 10k). Events older than the window are assumed not to be re-delivered (a reasonable bet given retry timeouts).
- **Truncated by checkpoint**: clear the dedup set when checkpoint succeeds, since post-checkpoint recovery starts from a clean slate.

The choice depends on your retry-timeout model: how long is a duplicate event still possible to arrive?

40.4 — Failed vs poison events

Two failure modes:

- **Transient failure**: network glitch, downstream service down, throttling. Retry usually fixes it.
- **Poison event**: bug in handler logic that *will always crash* on this event. Retry never fixes it.

The retry-N pattern catches both: retry N times for transient, give up after N for poison. Real systems add **circuit breakers** (stop retrying a downstream service that's been failing for 10 minutes) and **exponential**

backoff (wait longer between retries to avoid hammering a recovering service).

40.5 — Checkpoint cost

Checkpointing is **not free**. The snapshot operation must:

- Quiesce the reactor (block new posts briefly).
- Walk the in-flight set.
- Serialize state to disk.
- fsync the snapshot file.
- Truncate the WAL.

That's tens to hundreds of milliseconds. Run it too often and throughput suffers; run it too rarely and recovery RTO grows. The Cascade default is "on queue-drain or 5 minutes whichever comes first" — balances both.

40.6 — Things you didn't ask about

There are deep design choices in any WAL that we glossed over:

- **Write granularity**: per-record (small writes, high latency) vs per-batch (big writes, lower latency, lossy on crash mid-batch).
- **Group commit**: combining multiple in-flight writes into one fsync. Postgres does this aggressively; SQLite less so.
- **WAL segment rotation**: when the WAL gets to N MB, start a new segment file. Old segments can be archived/compressed/shipped.
- **Replication**: streaming WAL to a standby host. Postgres logical replication, MySQL binlog, Cassandra commitlog hint.

If you find yourself caring about any of these, you're beyond a chapter on WAL and into a chapter on distributed storage. Vol III Ch.51 has the Cascade-specific story.

Chapter 41 · Solutions

Stack-programming solutions. All use Axioma's built-in `stack()` + the push/pop/peek/depth + Forth/Pop-11 operations.

41.1 — RPN with negative numbers

```
parse_num: func(tok) [
  # Numeric tokens: pass through
  is_num: type(tok) == Integer or type(tok) == Float
  if is_num then [tok]
  else [
    # String token: try interpreting as negative number
    first: tok[1]
    rest_len: len(tok) - 1
    if first == "-" and rest_len >= 1 then [
      # Parse the rest as integer; only digits supported here
      n: 0
      i: 2
      valid: true
      while i <= len(tok) [
        c: tok[i]
        if c == "0" then [n = n * 10]
        else if c == "1" then [n = n * 10 + 1]
        else if c == "2" then [n = n * 10 + 2]
        else if c == "3" then [n = n * 10 + 3]
        else if c == "4" then [n = n * 10 + 4]
        else if c == "5" then [n = n * 10 + 5]
        else if c == "6" then [n = n * 10 + 6]
        else if c == "7" then [n = n * 10 + 7]
        else if c == "8" then [n = n * 10 + 8]
        else if c == "9" then [n = n * 10 + 9]
        else [valid = false]
        i = i + 1
      ]
      if valid then [-n] else [tok]
    ] else [tok]
  ]
]

rpn_eval: func(tokens) [
```

```
  s: stack()
  i: 1
  while i <= len(tokens) [
    tok: parse_num(tokens[i])
    is_op: type(tok) == String
    if is_op then [
      b: pop(s)
      a: pop(s)
      if tok == "+" then [push(s, a + b)]
      if tok == "-" then [push(s, a - b)]
      if tok == "*" then [push(s, a * b)]
      if tok == "/" then [push(s, a / b)]
    ] else [
      push(s, tok)
    ]
    i = i + 1
  ]
  pop(s)
]

# Test
println(rpn_eval([3, "-4", "+"]))     # -1
println(rpn_eval([3, -4, "+"]))       # -1 (already-int negatives also work)
println(rpn_eval([10, "-3", "*"]))    # -30
```

Lesson: integer literals in the token list are trivial; string `"-4"` requires a tiny parser. In production you'd use a regex / `strconv.Atoi`-style builtin. The exercise demonstrates that **tokenization is its own problem**: what looks like one token (`-4`) could be either a unary-minus + literal or a single negative literal, and disambiguation requires context (post-operator vs post-operand).

41.2 — Multi-bracket balance

```
matches: func(open_c, close_c) [
  if open_c == "(" then [close_c == ")"]
  else if open_c == "[" then [close_c == "]"]
  else if open_c == "{" then [close_c == "}"]
  else [false]
]

balanced: func(text) [
  s: stack()
  i: 1
  valid: true
  while i <= len(text) [
    c: text[i]
    if c == "(" or c == "[" or c == "{" then [push(s, c)]
    if c == ")" or c == "]" or c == "}" then [
```

```
      if depth(s) == 0 then [valid = false]
      else [
        top: pop(s)
        if not matches(top, c) then [valid = false]
      ]
    ]
    i = i + 1
  ]
  valid and depth(s) == 0
]
```

Test outputs:

```
balanced("({[]})")     → true
balanced("({[)]}")     → false  (mismatched)
balanced("(()")        → false  (unclosed)
balanced("())")        → false  (extra close)
balanced("")           → true   (empty is balanced)
balanced("abc(d)e")    → true   (non-brackets ignored)
```

Lesson: the stack holds **what we're expecting next** in implicit form — the matching close for each open. Two cleaner alternatives: (a) store the *expected close* on the stack so the pop-and-compare is direct, (b) use the helper `matches(open, close)` as above. Pick by which reads better in your domain.

A common extension: report *where* the mismatch happened. Add the index `i` to each pushed element so errors can point at "unclosed `(` at column 12".

41.3 — Infix to RPN (Shunting-Yard)

```
precedence: func(op) [
  if op == "+" or op == "-" then [1]
  else if op == "*" or op == "/" then [2]
  else [0]
]

shunting_yard: func(tokens) [
  output: []
  ops: stack()
  i: 1
  while i <= len(tokens) [
    tok: tokens[i]
    is_op: type(tok) == String
    if is_op then [
      # Drain operators of higher-or-equal precedence
      keep: true
      while keep [
        if depth(ops) == 0 then [keep = false]
        else [
          top: peek(ops)
```

```
          if precedence(top) >= precedence(tok) then [
            output = output + [pop(ops)]
          ] else [keep = false]
        ]
      ]
      push(ops, tok)
    ] else [
      output = output + [tok]
    ]
    i = i + 1
  ]
  while depth(ops) >= 1 [
    output = output + [pop(ops)]
  ]
  output
]
```

Test outputs:

```
shunting_yard([3, "+", 4, "*", 2])    → [3, 4, 2, "*", "+"]
shunting_yard([3, "*", 4, "+", 2])    → [3, 4, "*", 2, "+"]
shunting_yard([1, "+", 2, "+", 3])    → [1, 2, "+", 3, "+"]
```

The first one — `3 + 4 * 2` — converts to `3 4 2 * +`, meaning "push 3, push 4, push 2, multiply 4×2=8, add 3+8=11." Correct precedence: `*` binds tighter, so it fires first in RPN.

Lesson: Shunting-Yard is Dijkstra's elegant algorithm for **infix → RPN** conversion. It's a 30- line two-stack algorithm that handles arbitrary operator precedence. Many production parsers start with Shunting-Yard and embellish from there: parentheses, function calls, right-associative operators, prefix/postfix unary.

The full algorithm also handles `(` and `)` in the infix — open paren goes on the operator stack unconditionally; close paren pops until matching open. Adding that to this solution is a one-line extension.

41.4 — Stack-based factorial

```
factorial_stack: func(n) [
  s: stack()
  push(s, n)
  push(s, 1)   # accumulator
  keep: true
  while keep [
    acc: pop(s)
    cur: pop(s)
    done: cur <= 1
    if done then [
      push(s, acc)
      keep = false
    ] else [
```

```
      push(s, cur - 1)
      push(s, acc * cur)
    ]
  ]
  pop(s)
]

println(factorial_stack(5))    # 120
println(factorial_stack(7))    # 5040
println(factorial_stack(0))    # 1
println(factorial_stack(1))    # 1
```

Trace for `factorial_stack(3)`:

```
push 3, push 1            → Stack[1 | 3]
pop 1, pop 3              cur=3 acc=1
push 2, push 3            → Stack[3 | 2]
pop 3, pop 2              cur=2 acc=3
push 1, push 6            → Stack[6 | 1]
pop 6, pop 1              cur=1 acc=6  (done)
push 6                    → Stack[6]
pop                       → 6
```

Each loop iteration consumes (cur, acc), produces (cur-1, acc*cur). Loop exits when `cur <= 1`, leaving just the final `acc` on the stack.

Lesson: this accumulator-style recursion can be expressed as a loop that updates its arguments. That is tail-recursion elimination. The explicit stack here makes the state visible for practice; factorial does not need a stack of pending calls.

41.5 — `erasenum` edge cases

Empirical findings from Axioma:

```
# erasenum(s, 0): no-op
s1: stack()
push(s1, 1); push(s1, 2); push(s1, 3)
erasenum(s1, 0)
# s1 unchanged: Stack[3 | 2 | 1]

# erasenum(s, k > depth): ERROR
s2: stack()
push(s2, 1); push(s2, 2); push(s2, 3)
erasenum(s2, 10)
# → ERROR: cannot erase more items than stack contains

# erasenum(s, -1): ERROR
s3: stack()
push(s3, 1); push(s3, 2); push(s3, 3)
```

```
erasenum(s3, -1)
# → ERROR: count must be non-negative
```

Summary:

Input	Behavior
`n = 0`	no-op (silent)
`0 < n ≤ depth(s)`	drops top n
`n > depth(s)`	error: "cannot erase more items than stack contains"
`n < 0`	error: "count must be non-negative"

Lesson: Axioma's `erasenum` follows the **fail-fast** philosophy: invalid arguments throw errors rather than silently clamping or doing nothing. The contrasting design (clamp k to depth, silently no-op on negatives) is also valid — Python's `list.pop()` raises `IndexError` on empty, JavaScript's `arr.splice()` silently clamps. Axioma chose the strict interpretation. Knowing this prevents bugs where you intended "drop up to N" but got an error instead.

For "drop up to N" semantics, wrap:

```
safe_erase: func(s, n) [
  to_drop: depth(s)
  if n < to_drop then [to_drop = n]
  if to_drop > 0 then [erasenum(s, to_drop)]
]
```

41.6 — Pick the right paradigm

Problem	Paradigm	Why
(a) Sum of squares of a list	**Array/recursion**	Linear scan over indexed data — no LIFO need.
(b) Validate nested HTML tags	**Stack**	Each open tag pushes; each close pops + checks. Classic balanced-delimiter.
(c) Find median of sorted list	**Array**	O(1) indexed access; stack adds no value.
(d) Traverse directory tree	**Either**	Recursion is natural; an explicit stack handles depth-overflow on deep trees.
(e) Undo/redo in text editor	**Stack**	LIFO of past actions; a second stack for redo. Pattern A from §41.6.

Problem	Paradigm	Why
(f) Reverse a string	**Either**	Stack: push every char, pop into output. Array: build output backward. Both O(n). The stack version is one-liner if you have `stack_to_array`.

Lesson: the right paradigm depends on the problem's *access pattern*. LIFO problems get clearer with explicit stacks; random-access problems get clearer with arrays. Often the same algorithm can be expressed either way; **prefer the one that matches the problem's natural shape**.

Notes

41.1 — Tokenizer ambiguity

The infix question "is −4 a unary-minus + 4, or a single negative literal" doesn't have a universal answer — it depends on context. After an operator or `(`, it's a literal; after a number or `)`, it's subtraction.

Real expression parsers track an "expecting operand vs operator" state (one bit) and use it to disambiguate. This is the famous "two-state lexer" trick. Axioma's expression parser uses something similar.

41.2 — Beyond brackets

The same pattern generalizes to:

- **HTML tag matching**: push open tags; close tag must match top.
- **Indentation-based syntax**: indent pushes a level; dedent pops; mismatch is an error.
- **JSON parsing**: each `{` pushes "expecting property-or-close"; each `[` pushes "expecting value-or-close"; the stack of expectations drives the parser's state.

Generally: any **nesting structure where mismatches must be detected** is a stack problem.

41.3 — Production parsers

Shunting-Yard is a great teaching algorithm but hand-coded production parsers usually use **Pratt parsing** (a.k.a. **top-down operator precedence**) instead. Pratt parsers are recursive descent + a precedence table; they produce ASTs directly instead of RPN; they handle prefix, postfix, mix-fix, and right-associative operators gracefully.

Axioma's own parser is a Pratt parser. Read `parser/parser.go` to see one in action.

41.4 — Stacks-as-iteration

The factorial exercise stores the next iteration's state explicitly. For a tree traversal, an explicit stack can instead hold several pieces of pending work. Pushing and popping those frames replaces the call stack.

Defunctionalization is a related, more specific idea: replacing a finite family of function values with data describing their cases and an interpreter for those cases. It can be used to represent continuations, but a loop or an iterator protocol is not automatically an example of it.

41.5 — Defensive vs forgiving APIs

Axioma's `erasenum` is **defensive**: invalid input errors immediately. The opposite philosophy is **forgiving**: silently clamp or no-op.

Defensive APIs catch bugs early at the cost of verbosity (callers must validate). Forgiving APIs are quieter but mask bugs (`erasenum(s, count)` where `count` is accidentally `-1` silently does nothing — confusing).

For this API, a visible error helps diagnose an incorrect count. Other APIs deliberately clamp values; the appropriate choice depends on the contract, and callers need that contract to be explicit.

41.6 — The 80/20 of paradigm choice

In practice, ~80% of code is array/recursion — that's the universal hammer. ~15% is stack-shaped: parsers, undo buffers, tree iterators. ~5% is weirder: state machines (Ch.38), reactors (Ch.39-40), neuro-symbolic mixes (Ch.42).

Knowing all three lets you pick the right tool. Knowing only arrays leaves you re-deriving stack algorithms from scratch (the textbook "manual recursion-to-iteration conversion" tax).

41.7 — The return stack

```
#language axioma/rpn
rpush(10)
rpush(20)
expect("peek", rpeek(), 20)
expect("pop", rpop(), 20)
expect("pop next", rpop(), 10)
expect("infix still works", 1 + 2, 3)
empty: try(rpop())
expect("empty errors", error?(empty), true)
```

Commentary. Five words, one session-wide stack. The dialect is additive: `1 + 2` is still infix addition, not two pushes and an `+` word. That is the difference between `axioma/rpn` and a Forth image. `try(rpop())` plus `error?` is how you observe the empty-stack error without aborting the file — the same `try` the rest of Axioma uses for errors-as-values.

The return stack is not a `stack()` value you can pass around. It is parked in the interpreter session. `rpush` on `axioma/all` does the same work; the pragma documents the idiom.

Chapter 42 · Solutions

Neuro-symbolic pattern solutions. All use Axioma's B4 truth values, grounding tags, and defeasible-rule syntax (<~~).

42.1 — Three-source agreement

```
# Phase 1: three LLM extractions agree
hypothesis ceo_of("OpenAI", "Sam Altman")
hypothesis ceo_of("OpenAI", "Sam Altman")
hypothesis ceo_of("OpenAI", "Sam Altman")

# Phase 2: four sources (one contradicts)
hypothesis ceo_of("OpenAI", "Sam Altman")
hypothesis ceo_of("OpenAI", "Greg Brockman")

# The system stores both; a rule sets Both:
set_truth("ceo_of", "OpenAI", "Sam Altman",   "both")
set_truth("ceo_of", "OpenAI", "Greg Brockman", "both")

println(truth("ceo_of", "OpenAI", "Sam Altman"))
# → T⊥ᵇ
println(truth("ceo_of", "OpenAI", "Greg Brockman"))
# → T⊥ᵇ
```

Both records exist; both are tagged Belnap `Both`; downstream queries see "the question is contested" rather than picking a winner.

Lesson: the symbolic layer's job in multi-source aggregation is **not to pick a winner** but to **surface the disagreement**. The "Both" tag is the API by which the LLM ingest layer tells the reasoning layer "I had multiple inputs and they didn't agree."

Real aggregation in Cascade weights sources by historical accuracy, recency, and topic match — adding considerable machinery on top of this base mechanism. The base mechanism (B4 `Both` as the contradiction signal) stays the same.

42.2 — Defeasible-by-specificity

```
flies(X) <~~ bird(X)
nonfly(X) <= penguin(X)

hypothesis bird("tweety")
postulate penguin("tweety")

r: {X | X <- flies(X)}
println("flies (defeasibly derived):", r)
# → {"tweety"}
println("flies(tweety) strength:", grounding("flies", "tweety"))
# → conjecture

n: {X | X <- nonfly(X)}
println("nonfly (strictly derived):", n)
# → {"tweety"}
println("nonfly(tweety) strength:", grounding("nonfly", "tweety"))
# → theorem
```

Tweety is in *both* sets — defeasibly classified as flying (via bird) and strictly classified as non-flying (via penguin). The groundings differ: **conjecture** (weaker, defeasible) vs **theorem** (stronger, strict).

A downstream query "is Tweety flying?" must choose its minimum trust. At minimum=conjecture both predicates are visible (and the system flags the conflict). At minimum=theorem only `nonfly` is visible.

The semantic point: a **specifically-derived strict result** (penguin→nonfly) **defeats** a **generally-derived defeasible result** (bird→flies). The grounding ladder makes this mechanically explicit.

Lesson: real-world reasoning is shot through with this pattern. "Birds fly (usually) but penguins don't" is the textbook example, but the deeper pattern is: **rules have scope**, and more specific scopes override more general ones. The ladder + the strict/defeasible distinction together encode the override priority.

42.3 — Failure-mode classification

Scenario	Failure mode
(a) LLM says "Paris is in Italy"	**Hallucination** — plausible-sounding but flat-out wrong.
(b) LLM says "I'm 100% sure Pluto is a planet"	**Confidence-calibration mismatch** — high stated confidence with wrong-for-the-current-definition answer.
(c) Same prompt: yesterday Y, today N	**Prompt drift** — non-reproducibility across calls.
(d) 2024 election quotes 2020 names	**Training-data leak** (or staleness) — memorized old article surfaces as current fact.
(e) Poisoned RSS feed gets ingested	**Adversarial input** — attacker crafts source to inject false claims.

Scenario	Failure mode
(f) “Sam” today, “Samuel” tomorrow	**Ontology drift** — entity references diverge across time.

Lesson: giving the failure modes names makes them debuggable. If your system is unreliable, the first triage step is: “which failure mode is biting us?” Hallucination needs RAG; calibration needs ensemble voting; prompt drift needs deterministic seeds; staleness needs temporal tags; adversarial input needs source reputation; ontology drift needs entity normalization.

Each mode has a different fix. Conflating them is how teams ship neuro-symbolic systems that work in demos and fail in production.

42.4 — Trust calibration

```
# Track each model's accuracy
postulate model_accuracy("gpt-4",   0.85)
postulate model_accuracy("claude-3", 0.92)
postulate model_accuracy("rumor-bot", 0.30)

# Demotion rule: if a model's accuracy is below 0.5,
# claims from it are demoted to standalone (the
# floor of the grounding ladder).
# Conceptual pseudocode (Cascade does this via the
# truth-aggregation pipeline, not as a single ax rule):

demote_low_trust: func(source_model) [
  acc_pairs: {A | A <- model_accuracy(source_model, A)}
  # Take the only value (assume one accuracy per model)
  # Lower-bound check
  acc: 0
  if len(acc_pairs) >= 1 then [acc = acc_pairs[1]]
  if acc < 0.5 then [
    # Find all hypothesis-grade facts tagged from this
    # source, demote them. This is a metaprogramming
    # operation (see Ch.22). In Cascade, sources are
    # tracked via the `said_by` relation; the demotion
    # rule rewrites the source-tagged facts.
    println("would demote claims from ", source_model)
  ]
]

demote_low_trust("rumor-bot")   # would fire
demote_low_trust("claude-3")    # no-op
```

Lesson: this is the **source-reputation tracking** pattern. The simple version: a scalar accuracy per source. The richer version: **topic-conditional accuracy** — gpt-4 is great on science but weak on celebrity gossip, so use different accuracy values per topic.

Cascade tracks accuracy per (source, topic) and demotes on a per-claim basis. The result: poisoned sources get downgraded *exactly* on the topics where they were poisoned, not blanket-banned.

42.5 — RAG-style promotion

```
verify_via_wiki: func(rel, args) [
  # Simulate: hardcoded Wikipedia lookup table
  wiki_facts: {
    "is_capital.Germany":  "Berlin",
    "is_capital.France":   "Paris",
    "ceo_of.OpenAI":       "Sam Altman"
  }
  key: rel + "." + args[1]
  expected: wiki_facts[key]
  if expected == none then [
    println("wiki: no entry for ", key)
    "unknown"
  ] else if expected == args[2] then [
    insert(rel, args[1], args[2], "postulate")
    println("wiki: CONFIRMED — promoted to postulate")
    "confirmed"
  ] else [
    insert(rel, args[1], args[2], "standalone")
    println("wiki: CONTRADICTED — demoted to standalone")
    println("  wiki says:", expected, ", we had:", args[2])
    "contradicted"
  ]
]

# Use it
hypothesis is_capital("Germany", "Berlin")
verify_via_wiki("is_capital", ["Germany", "Berlin"])
# → wiki: CONFIRMED — promoted to postulate

hypothesis is_capital("Germany", "Frankfurt")
verify_via_wiki("is_capital", ["Germany", "Frankfurt"])
# → wiki: CONTRADICTED
```

Lesson: the **RAG-at-storage-time** pattern is distinct from **RAG-at-query-time** (the more common form, where retrieval happens fresh on each LLM call). RAG-at-storage-time *bakes in* the verification result, so downstream queries don't need to re-verify — much faster, at the cost of staleness.

The cost-of-fresh-vs-baked decision is universal in caches and KBs. Cascade does both: a slow nightly verification scan that promotes facts; a fast query-time verification for top-priority claims.

42.6 — Real system case study

Open-ended. A sample answer covering **DeepProbLog**:

DeepProbLog (Manhaeve et al., 2018) extends ProbLog (probabilistic Prolog) by allowing **neural predicates**. A neural network can serve as a soft fact-generator; its outputs are interpreted as probabilities for ProbLog rules to combine.

Neural part: convolutional networks for digit recognition. Given an image, produces `digit(image, 7)` with probability p.

Symbolic part: ProbLog rules for arithmetic. `addition(I1, I2, Sum) :- digit(I1, X), digit(I2, Y), Sum is X + Y.`

Meeting point: at *both* storage-time (the neural net's outputs are stored as probabilistic facts) and query-time (the rule fires over the combined facts, yielding a probability for each possible answer).

Compared to Cascade: DeepProbLog uses continuous probabilities; Cascade uses categorical grounding levels. DeepProbLog is differentiable end-to-end (can train via gradient descent through symbolic rules); Cascade is not. The trade-off: differentiability gives DeepProbLog the ability to *learn* the rules; Cascade's categorical grounding gives it auditability.

Both work. Both are neuro-symbolic. They occupy different points in the design space.

Lesson: there's no single "right" architecture for neuro-symbolic systems. The patterns of this chapter (hypothesis storage, verification rules, defeasible derivation, B4 contradictions) are **building blocks**. Each system assembles them differently based on what it needs.

Notes

42.1 — Voting in production

The "three sources agree → promote" rule is the simplest case. Real systems weight sources by:

- **Historical accuracy** (see 42.4).
- **Recency** (newer claims about volatile facts outweigh older).
- **Topic relevance** (a finance-trained model on legal facts gets discounted).
- **Source diversity** (three identical-pipeline sources count as ~1; one human + one LLM count as ~2).

The weighted vote is then compared to a threshold. Cascade's implementation lives in `reasoning/truth_lifecycle.ax`.

42.2 — Yale Shooting Problem

The Tweety example traces back to McCarthy's "frame problem" and Hanks-McDermott's Yale Shooting Problem (1986). The challenge:

Given a gun, a turkey, an action "shoot," what's the most natural way to encode "shooting kills the turkey, but only if the gun is loaded"?

Naive logical encodings produce **multiple extensions** — both "turkey alive, gun unloaded" and "turkey dead, gun loaded" are consistent. The defeasibility ladder, with strict rules preempting defeasible ones, was one of several proposed fixes (alongside circumscription, default logic, etc.).

The fact that we can pose Tweety and Yale Shooting in ~10 lines of Axioma each is a genuine modeling win — it took decades of research to figure out the *right* mathematical machinery for these intuitively-simple problems.

42.3 — Naming things matters

Compare: "our system sometimes makes mistakes" vs "we observed 12% hallucination rate and 23% ontology drift in last week's traffic." The second is actionable; the first isn't.

The taxonomy of failure modes is itself a contribution. The classification used here (six modes) is one possible taxonomy; others (NIST draft AI risk framework, OWASP LLM Top 10) have their own divisions. The important thing is to **have** a taxonomy.

42.4 — The reputation problem in social networks

Source-reputation tracking is structurally identical to PageRank, eBay seller ratings, Stack Overflow karma, and academic-citation impact factors. They're all "votes weighted by the voter's standing" with different specifics.

The deep result (Pace & Sarwar, 2009): any reputation system computable as a weighted vote has the same fundamental vulnerability — **Sybil attacks** (an attacker creates many fake votes). Defending against Sybil requires either (a) high cost-to-create-account (Twitter pre-2010, Wikipedia editing) or (b) explicit social graph (Twitter blue, Web-of-Trust).

For neuro-symbolic Sybil defense: bind each LLM output to its *model* + *prompt* + *retrieval context* — variations in any of those make it a "different source" for voting purposes.

42.5 — When RAG isn't enough

RAG verification works when the trusted source is available and current. It fails when:

- The fact is about something not in the trusted source (an obscure entity).
- The trusted source is itself uncertain or contradictory.
- The verification is too slow for the use case.

For these, the fallback is **human-in-the-loop**: the system surfaces uncertain claims to a human reviewer, who decides the grounding. Cascade has this as an explicit step in the truth-update pipeline — claims at hypothesis with ambiguous verification queue for human review.

42.6 — Where neuro-symbolic is heading

A genuine open research question. Three competing visions:

1. **Tool-using LLMs**: the LLM is the core; it *calls* symbolic tools as needed. The neural part is generative; the symbolic part is utility.

2. **Symbolic-grounded LLMs**: the LLM is *fine-tuned* on a symbolic KB so its outputs are statistically consistent with the KB. The neural part is internalized symbols.
3. **Differentiable symbolic systems**: the symbolic layer is the core; rules are learned via gradient descent. The neural part is embedded inside the rules.

Cascade is closer to vision 1 (LLM extracts; symbolic verifies); DeepProbLog is closer to 3 (rules + learnable neural predicates); proprietary LLM products (GPT-4 with web search) are closer to 1.

All three are converging on similar territory but arriving from different starting points. The patterns of this chapter (hypothesis storage, B4 contradictions, grounding ladder) are useful across all three.

Closing remark

Volume II ends here. You now have the full vocabulary: state machines, event loops, durable reactors, stack programming, and neuro-symbolic patterns — on top of the CS1/CS2 foundation from Volume I and Part VII.

Where you go next depends on you. Volume III is the production case study (Cascade). The appendices (A-E) are the cheat sheets you'll want at hand. The exercises across Volume II are where the learning actually happens — go back and do the ones you skipped.

Whatever you do, **build something with it**. The patterns in this textbook are tools; tools are worth what you make with them.

Chapter 43 · Solutions

All solutions verified against a live, populated Cascade install (~3950 entities, ~3800 relations). Yours will differ depending on what you've ingested. Solutions 43.1, 43.3, 43.4, 43.5 are runnable without further LLM calls — they query whatever data is already in your database. Solutions 43.2 and 43.6 are open-ended.

All commands assume you're in the `axiomacascade` repo root.

43.1 — Database round-trip

```
uv run cascade stats
```

Sample output (live Cascade install):

```
{
  "entities": 3954,
  "relations": 3809,
  "claims": 8415,
  "sources": 329,
  "pins": 6,
  "rules": 85,
  "entity_types": 10,
  "relation_types": 9
}
```

Running the command a second time produces *identical* output — the database is purely read in that path.

Sample entity record (from `cascade entity list --type STATE --json`):

```
{
  "id": 2323,
  "type": "STATE",
  "name": "Afghanistan",
  "prolog_atom": "afghanistan",
  "aliases": [],
  "properties": {"sentiment": "NEUTRAL"},
  "first_seen": "2026-04-10T07:16:51.789829",
  "last_seen":  "2026-04-18T07:37:15.840244",
```

```
  "source_id":  268,
  "ulid":       "01KNVN5PJFCMX78HK1ZBEYAF5V",
  "domain":     "political"
}
```

The fields that surface architectural decisions:

- **`prolog_atom`** — lowercased canonical form used when this entity appears in `.ax` rule files. Lets rules refer to the entity without quoting issues.
- **`ulid`** — Universally Unique Lexicographically Sortable IDentifier. Allows cross-machine sync without primary-key collisions.
- **`source_id`** — points back to the article (in the `sources` table) that introduced this entity. The full provenance chain Article → Source → Entity → Relation.
- **`domain`** — high-level domain tag for filtering (political, energy, infrastructure, economic, …).

43.2 — Ingest a known article (open)

Pick an article you've read. Ingestion is non- deterministic (LLM-based) so success depends on the provider, model, and prompt. A typical good-quality ingestion yields 5-15 new entities and 10-30 new relations from a 600-word article.

Successes to look for:

- Major nations and orgs correctly typed (STATE / ORG).
- Named individuals as ACTOR.
- Causal language ("led to", "triggered", "caused") becoming CAUSAL or DEPENDENCY relations.

Failures to look for:

- *Synonym collapse* — the LLM extracts "the United States" but Cascade already has "USA", and they don't merge.
- *Type drift* — an EVENT entity that should have been POLICY (or vice versa).
- *Hallucinated relations* — claims about entity connections that aren't in the article text.
- *Missing temporal context* — relations get extracted but `properties.year` or `properties.tier` is empty.

These are evaluation skills you sharpen over multiple ingestions. Cascade's strength isn't perfect extraction on one article; it's *probabilistic accumulation* over many.

43.3 — Trace a cascade

```
uv run cascade discover cascade "Iran" --depth 3 --json \
  | jq '.tiers[] | {tier: .tier, count: .count, sample: [.entities[0:3] |
  ↪  .[].entity_name]}'
```

Sample output (live Cascade):

```
{"tier": 0, "count": 1,  "sample": ["Iran"]}
{"tier": 1, "count": 86, "sample": ["crude oil", "global energy crisis", "Iraq"]}
{"tier": 2, "count": ?,  "sample": [...]}
{"tier": 3, "count": ?,  "sample": [...]}
```

The depth at which `count` stops growing tells you when the cascade *saturates* — when the BFS has visited every node reachable via causal edges. For a medium-sized KB (~4000 entities) this typically happens at depth 4-5.

Reflection prompts the exercise asks:

1. **Causally plausible?** Tier 1 should always be obviously causally tied to Iran. Tier 2 should be one step removed but still readable. Tier 3 is where the framework starts surprising you ("Iran → oil → fertilizer markets → African food prices").
2. **Novel vs obvious?** If every Tier-2 result is obvious, your corpus is too narrow.
3. **Where it breaks down:** A typical failure mode is over-eager `INFLUENCE` relations that bridge semantically unrelated entities — an entity ends up in the cascade because of a weak influence edge that shouldn't propagate causally.

43.4 — Bridges vs centrality

```
uv run cascade discover bridges --json | jq '.[0:5]'
uv run cascade discover centrality --metric betweenness --top 10
```

Sample bridge output (live Cascade):

```
[
  {"entity_id": 1, "name": "Iran", "type": "STATE", "degree": 293},
  {"entity_id": 5, "name": "United States", "type": "STATE", "degree": 244},
  ...
]
```

Sample centrality output:

```
[
  {"entity_id": 5, "name": "United States", "degree": 0.0617, "betweenness": 0.0312,
   ↪  "pagerank": 0.0159},
  {"entity_id": 1, "name": "Iran",          "degree": 0.0741, "betweenness": ...,
   ↪  "pagerank": ...},
  ...
]
```

Bridges are entities whose removal would *disconnect* a sub-graph. Cascade's implementation is degree-based (high-degree entities are bridge candidates) — but classical graph bridges (Tarjan's algorithm) would also be valid. **Betweenness centrality** measures *how many shortest paths pass through* an entity. The two metrics often agree on top entities but disagree on rank.

Compare: in the live Cascade DB, Iran ranks #1 by degree (293 connections) but the U.S. ranks higher by betweenness — because the U.S. sits on *more* shortest paths even though it has fewer total edges. That's a content insight: Iran is a *hub* (many direct connections), the U.S. is a *broker* (it connects otherwise-disconnected parts).

43.5 — The Cascade-Axioma bridge

```
# What relations does Cascade have stored?
uv run cascade axioma query list-relations
```

Output:

```
{
  "relations": [
    "ADVERSARIAL", "ALLIANCE", "CAUSAL", "CONTROL",
    "DEPENDENCY", "INFLUENCE", "SUPPLY", "TEMPORAL",
    "commodity"
  ],
  "schemas": {...}
}
```

The eight uppercase relations are the canonical relation-type vocabulary. The lowercase `commodity` is a domain-specific relation, populated by an `.ax` script.

```
# What facts are stored for the commodity relation?
uv run cascade axioma query list-facts --relation commodity
```

Output:

```
{
  "facts": [
    {"args": ["Bronze"], "epistemic_type": "axiom",
     "relation": "commodity", "truth_value": 1}
  ],
  "success": true
}
```

`truth_value: 1` is TRUE (the Belnap B4 encoding maps TRUE to 1). `epistemic_type: "axiom"` is the strongest grounding from Volume I's two-tier knowledge system.

```
# Insert a fact from Axioma side
uv run cascade axioma exec 'axiom/persist test_fact("hello", "world")'
```

Output:

```
{
  "result": "Epistem{type=axiom, value=\"test_fact(\"hello\", \"world\")\",
   ↪  truth_value=1.00}",
  "success": true,
  "type": "EPISTEM"
}
```


```
# Verify the fact persists across MCP sessions
uv run cascade axioma query list-facts --relation test_fact
```

Output:


```
{
  "facts": [{"args": ["hello", "world"],
             "epistemic_type": "axiom",
             "relation": "test_fact",
             "truth_value": 1}],
  "success": true
}
```


Where it lives. The fact is written to the same `cascade.db` SQLite file. The MCP session that wrote it has shut down between the two commands, but the data persisted. This is **the shared substrate** that makes the architecture work: Python-side Cascade code and Axioma-side rules see the same tables.

Cleanup (optional):

```
uv run cascade axioma exec 'forget("test_fact", "hello", "world")'
```

43.6 — Where would *you* put Axioma? (open)

Sample answer — Axioma in a *medical-records system*:

- **System:** Patient-record review at a teaching hospital — incoming EHR records get LLM-summarized, cross-checked against clinical guidelines, flagged for ambiguity.
- **Layer placement:** Axioma sits at Layer 3 (Reasoning), alongside a clinical-rules engine (e.g. Drools). The LLM lives at Layer 1 (Ingest), Postgres at Layer 2 (data), and a web dashboard at Layer 4 (Output).
- **Axioma queries:** Belnap B4 truth tracking for conflicting lab results (paraconsistent — conservative diagnostic guidance even when two labs disagree). Defeasible rules for ambiguous symptom combinations ("patient X has symptoms A and B *unless* finding Z is present"). Relational queries over clinical history.
- **Axioma doesn't:** train models, run statistical inference, do natural-language understanding (LLM handles that). Axioma is the *crisp-logic* layer on top of probabilistic / fuzzy components.

Other domains that fit this template:

- **Regulatory compliance** — Axioma encodes the rules, graph DB encodes the entity relationships,

document-processing pipeline ingests source PDFs.
- **Supply-chain risk** — Axioma encodes failure-mode rules, time-series DB tracks pricing, graph DB tracks supplier relationships.
- **Investigative journalism** — Axioma encodes fact-claim chains, vector DB does semantic search, knowledge graph stores entities.

The pattern: **Axioma is the symbolic-logic component in any hybrid system.** It pairs with at least one graph engine (NetworkX, Neo4j) and at least one probabilistic engine (LLM, statistical model). It's the *crisp* layer that makes the system's reasoning auditable.

Notes

43.1 — Why ULIDs?

ULIDs (Universally Unique Lexicographically Sortable IDentifiers) solve a specific problem: when two Cascade instances on different machines both ingest the same article, they'd both generate entity IDs that *might collide* if using monotonic integers. ULIDs are time-prefixed and globally unique, so the sync layer can deduplicate without primary-key conflicts. The trade is that ULIDs are 26-character strings instead of compact integers — but with SQLite's indexing, the lookup cost is negligible.

43.3 — Why BFS and not DFS?

The cascade-discovery algorithm uses BFS by tier (level-order traversal) because *tier* is the semantically meaningful unit. A Tier-1 impact is "close" by definition; a Tier-3 impact is "distant." DFS would visit nodes in arbitrary depth order and make tier accounting ad-hoc. The BFS implementation is ~70 lines in `cascade/core/discovery.py:71`.

43.4 — Bridges vs centrality — why both?

These metrics ask different questions:

- **Bridges** ask: *Is this entity a single point of failure?* (graph connectivity)
- **Centrality** asks: *Does information flow through this entity?* (path frequency)

A high-bridge / low-centrality entity is a *chokepoint* that only matters when something needs to cross it. A low-bridge / high-centrality entity is a *broker* that mediates many transactions but isn't structurally critical (redundant paths exist). The two metrics together give a richer picture than either alone.

43.5 — Why does `axiom/persist` not need explicit truth?

The `axiom/` refinement implies `truth_value: 1` (TRUE) because axioms are *foundational* — that's the definition of the strongest epistemic grounding in Volume I's two-tier system. If you want a fact stored at lower truth values, use `postulate/persist` (grounds the fact to be challenged later) or set truth explicitly via `set_truth(...)`.

43.6 — The pattern

Three properties make Axioma a useful component:

1. **Symbolic** — its outputs are inspectable propositions, not black-box scores.
2. **Multi-valued** — handles contradiction without collapsing.
3. **Defeasible** — encodes "usually true" rules that the engine knows might be overridden.

Any domain where those three properties matter is a candidate for Axioma-as-component. The pattern works because Axioma was designed for *exactly* this component role from the beginning, even though it *also* works as a standalone language.

Chapter 44 · Solutions

All queries verified against a live populated `cascade.db` (~3950 entities, ~3800 relations, ~8400 claims). Yours will differ. Solutions 44.1–44.5 are runnable SQL; 44.6 is open-ended.

44.1 — Schema audit

```
sqlite3 data/cascade.db ".tables"
```

Live output:

```
change_log      monitors        portfolios      reminders       sync_meta
claims          notifications   predictions     rules           sync_outbox
entities        path_analyses   relation_types  saved_searches  truth_history
entity_types    pins            relations       sources
```

20 tables total. The 7 core tables from §44.2 plus 13 support tables. Run `.schema entities` / `.schema relations` / `.schema claims` to compare with the chapter — every column listed in §44.3 should be present. Newer migrations may have added more.

44.2 — Type distribution

```
SELECT type, COUNT(*) AS n FROM entities
GROUP BY type ORDER BY n DESC;
```

Live output (top 5):

```
type        n
----------  ----
ORG         1195
ACTOR       613
EVENT       426
SECTOR      352
POLICY      318
```

```
SELECT type, COUNT(*) AS n FROM relations
GROUP BY type ORDER BY n DESC;
```

Live output (top 5):

```
type        n
----------  ---
INFLUENCE   747
CONTROL     714
DEPENDENCY  534
CAUSAL      472
SUPPLY      453
```

Why ORG dominates entities (1195 of ~4000): geopolitical news has many more *organizations* (companies, agencies, NGOs, ministries) than countries or markets. Every article mentions multiple ORGs as the *actors* doing things.

Why INFLUENCE dominates relations (747 of ~3800): INFLUENCE is the *softest* causal vocabulary — when the LLM is uncertain whether to call something CAUSAL or DEPENDENCY, it falls back to INFLUENCE. That makes INFLUENCE a *catch-all* that accumulates faster than the more committed types. Production analysts often re-tag specific INFLUENCE edges as CAUSAL after manual review.

44.3 — The paraconsistent slice

```
SELECT c.id,
       substr(c.text, 1, 60) AS preview,
       c.confidence,
       s.title AS source_title
FROM claims c
LEFT JOIN sources s ON c.source_id = s.id
WHERE c.truth_value = 'BOTH'
LIMIT 5;
```

Live sample:

```
70   The closure of the Gulf waterway could lead to the greatest...   MEDIUM
591  Iran is known for its oil exports.                               MEDIUM
592  Iran exports an estimated 2.1 million barrels of oil per day     MEDIUM
```

The two sides of a typical disagreement: one source gives a higher number, another gives a lower number, and both got ingested. The claim is *paraconsistent* because Cascade refuses to pick.

The fix in production is usually source weighting: the higher-credibility source's value wins, the lower gets retired to UNKNOWN. But the *audit trail* shows both. The `truth_history` table is where that re-evaluation lands.

44.4 — Relations join

```
SELECT e1.name AS src, r.type, e2.name AS tgt, r.confidence
FROM relations r
JOIN entities e1 ON r.source_entity_id = e1.id
JOIN entities e2 ON r.target_entity_id = e2.id
```

```
WHERE e1.name = 'Iran'
ORDER BY r.confidence DESC, r.type;
```

Live sample (top 5 HIGH-confidence rows from Iran):

```
src   type         tgt                       confidence
----  -----------  ------------------------  ----------
Iran  SUPPLY       crude oil                 HIGH
Iran  ADVERSARIAL  Strait of Hormuz          HIGH
Iran  ADVERSARIAL  United States             HIGH
Iran  CONTROL      Ayatollah Ali Khamenei    HIGH
Iran  ADVERSARIAL  Dubai                     HIGH
```

The **double-join pattern** — joining `relations` to `entities` twice with two aliases (`e1` for source, `e2` for target) — is the workhorse query of any graph-in-SQL system. The two indexes `idx_relations_source` and `idx_relations_target` keep it fast.

44.5 — The reactor backlog

```
SELECT COUNT(*) FROM change_log WHERE reactor_handled = 0;
```

Live output: `3467` (in this DB). Yours will differ.

A non-zero backlog means *the reactor agent hasn't processed all recent changes.* Either the reactor is off (most installs don't run it as a daemon), or it's slow, or there's been a flood of recent inserts.

```
SELECT id, change_type, item_type, item_id, created_at
FROM change_log
WHERE reactor_handled = 0
ORDER BY created_at DESC LIMIT 10;
```

Sample output rows show `change_type` as `INSERT` or `UPDATE`, `item_type` as one of `entity`, `relation`, or `claim`. This is your *recent activity log* — useful when you've ingested an article and want to see exactly what landed.

44.6 — Schema critique (open)

Sample answer.

Change I'd make: separate `truth_values` table

The `truth_value` column on `claims` is a TEXT column with values `TRUE | FALSE | BOTH | UNKNOWN`. Today it's unconstrained — typos like `"true"` (lowercase) or `"TRUE "` (trailing space) would land silently and break `WHERE truth_value = 'TRUE'`. A foreign key to a `truth_values(name)` table would:

- **Catch typos** at insert time.
- **Document** the legal vocabulary in-schema.

- **Enable extension** — a future `Both-Strong` value (definite both-true and both-false) could be added by inserting a row.

What breaks: existing inserts that don't go through the application layer. Migration cost: rewrite ~10 application-layer methods.

Change I'd resist: split `properties` into typed columns

Tempting because JSON-in-TEXT loses indexing. But `properties` carries *type-specific* attributes — sentiments on STATEs, roles on INFRA, sectors on ORGs. A column-per-attribute would explode the schema (50+ sparse columns) without speeding up queries that filter on the rare ones.

The principled fix is a *narrow table* `entity_properties(entity_id, key, value)` with an index on `(key, value)`. But that's its own schema revolution. For now, JSON-in-TEXT trades indexing for flexibility, and Cascade's queries rarely filter on `properties` keys anyway.

The general principle: **fix invariant violations (truth value typos); resist over-normalization (properties)**. Both decisions trade schema rigor for practical maintenance.

Notes

44.1 — Drift between chapter and live schema

Cascade schema migrations land every few weeks. If you spot a column the chapter doesn't mention, that's a *recent* addition — usually for a new feature landing in the next minor release. Treat the chapter schema as the *baseline*, not the *final word*.

44.2 — INFLUENCE as catch-all is a known issue

The over-use of INFLUENCE is a recognized data-quality issue in Cascade's evaluation pipeline. A planned relation-typing audit re-promotes INFLUENCE to more specific types where evidence supports it. The fact that you can *see* this from the schema (top relation type by count) is itself evidence of why raw SQL inspection matters — the CLI surfaces aggregates but not always the *distribution*.

44.3 — Why BOTH isn't FALSE

A naive system would resolve "one source says X, one says ¬X" by picking the higher-credibility source and marking the loser FALSE. Cascade's choice — Belnap B4 BOTH — preserves *both* assertions because:

1. Source credibility is noisy and downstream re-evaluation may flip the winner.
2. Some disagreements are *real* (the two sides have different timestamps; one is talking about a different region).
3. Truth-history audit becomes informative — "this claim flipped from TRUE to BOTH on 2026-04-10 because source S2 contradicted source S1."

The fundamental commitment: **the system models the journalist's uncertainty, not the world's truth.**

44.4 — The cost of the double join

Each `JOIN entities ON ...` adds a B-tree lookup per row. With `idx_relations_source` and `idx_relations_target` plus the entity primary key, each lookup is O(log N) where N ≈ 4000. The whole query on a 3800-relation DB is bounded by the result size, not the table size.

For the rare query that joins on un-indexed columns (`properties` JSON values, `evidence` array elements), performance degrades to full table scan. Watch out.

44.5 — Why the backlog isn't a bug

Many Cascade installs don't run the reactor as a daemon. The reactor is *optional* — it fires rules in response to changes, but if you just want to ingest and query, the rules can stay un-fired indefinitely. The backlog accumulates harmlessly. When you do want rules to fire, `cascade reactor start` processes the queue and clears it.

44.6 — Schema design as ethical commitment

Every schema decision encodes a worldview. By making `truth_value` a free-form TEXT field, the system *invites* third-party tools to write nonstandard values (useful for experimentation). By making `relation_types` a separate table, the system *forces* new relation types to be named centrally. Each choice trades flexibility for consistency.

Cascade's overall stance: **consistency at the relation-type and entity-type layer; flexibility at the properties and evidence layer.** That's why the former are foreign-keyed tables, the latter are JSON blobs. Whether you agree with the cut is itself a design conversation.

Chapter 45 · Solutions

All queries verified against a live populated `cascade.db`. Solutions 45.2–45.5 are runnable SQL; 45.1 and 45.6 are open-ended.

45.1 — Watch an ingestion (open)

Run with –v and skim. Look for these four landmarks:

```
fetcher    | Ingesting URL | url=...
ner        | spaCy NER | N unique entities from M chars
extractor  | Entity/relation LLM extraction started | ...
decomposer | Claim LLM extraction started | ...
extractor  | Storing N entities to graph...
extractor  | Store complete | source_id=... | ...
```

Things to notice:

- **NER count > LLM count.** spaCy finds more candidates than the LLM keeps — the LLM filters out noise and merges duplicates.
- **New vs merged.** The first ingestion of a topic has mostly new entities; subsequent ingestions of related topics have more merges (the dedup substrate from §45.5 is doing its job).
- **Skipped relations** are normal. They happen when the LLM names an entity in a relation but forgets to add it to the entities array — about 1-3 per article in typical corpora.

45.2 — Per-source extraction yield

```sql
WITH per_source AS (
  SELECT source_id, COUNT(*) AS n
  FROM claims
  WHERE source_id IS NOT NULL
  GROUP BY source_id
)
SELECT AVG(n), MIN(n), MAX(n) FROM per_source;
```

Live output:

```
avg_claims    min_n    max_n
------------  -----    -----
31.4          1        100
```

Repeat for entities and relations:

```
avg_entities  min_n    max_n
------------  -----    -----
29.8          1        136

avg_relations min_n    max_n
------------  -----    -----
22.7          1        74
```

The chapter's stated averages (~31 claims, ~30 entities, ~23 relations) match closely. The maximum values (100 claims, 136 entities, 74 relations) come from analytical long-form pieces.

Why claims > entities > relations. Each article typically has more *propositions* than *named things*, and more named things than *connections between them*. A 600-word article with 30 entities might mention each entity in 2-3 claims but only connect them via 1 explicit relation each.

45.3 — Source-type distribution

```sql
SELECT source_type,
       COUNT(*) AS n,
       AVG(credibility) AS avg_cred
FROM sources
GROUP BY source_type
ORDER BY n DESC;
```

Live output:

```
source_type  n    avg_cred
-----------  ---  --------
news         317  0.80
opinion       11  0.50
analysis       1  0.70
```

`news` dominates because that's what the URL-based ingestion path normally feeds. Op-eds get pulled when the analyst explicitly wants commentary. Reports typically arrive via file ingestion (`cascade ingest --file report.pdf`) and aren't visible here.

Why the average credibility per type matches the defaults. Cascade rarely overrides credibility after auto-assignment — the field is reserved for manual analyst adjustments and most installs don't make them yet. As verification workflows mature, the spread within each source_type will grow.

45.4 — Trace one article through

For a high-yield source (e.g. `source_id = 283`, which has 100 claims in the test DB):

```sql
SELECT 'entities' AS kind, COUNT(*) AS n FROM entities WHERE source_id = 283
UNION ALL SELECT 'relations', COUNT(*) FROM relations WHERE source_id = 283
UNION ALL SELECT 'claims',    COUNT(*) FROM claims    WHERE source_id = 283;
```

Output:

```
kind       n
---------  ---
entities   66
relations   1
claims    100
```

Striking pattern: **66 entities, 100 claims, 1 relation**. This is a *summary-heavy* article — many named things, many propositions, but few explicit *connections* between them. The article's analytical weight is in the *claims*, not the graph structure.

That's a content insight you can only get by looking at the *shape* of an ingestion. Articles that generate few relations but many claims signal *descriptive* journalism; articles with more relations than entities signal *connective* analysis.

45.5 — Find the merge candidates

```
SELECT e1.name, e2.name, e1.type
FROM entities e1, entities e2
WHERE e1.id < e2.id
  AND e1.type = e2.type
  AND lower(e1.name) LIKE '%' || lower(e2.name) || '%'
LIMIT 10;
```

Sample live output:

```
name1                              name2                      type
---------------------------------  -------------------------  ---------
coal reserves                      reserve                    COMMODITY
Islamic Revolutionary Guard Corps  Revolutionary Guard Corps  ORG
Ayatollah Ali Khamenei             Ali Khamenei               ACTOR
Central Intelligence Agency        Intel                      ORG
Central Intelligence Agency        IG                         ORG
```

Some are *real duplicates*:

- "Ayatollah Ali Khamenei" ↔ "Ali Khamenei" — same person.
- "Islamic Revolutionary Guard Corps" ↔ "Revolutionary Guard Corps" — same org.

Some are *false positives*:

- "coal reserves" ↔ "reserve" — different concepts that share a substring. Substring matching is *necessary but not sufficient* for entity resolution.
- "Central Intelligence Agency" ↔ "Intel" — the acronym match is spurious; Intel the chipmaker is *not* the CIA.

This is why production entity resolution needs more than `LIKE` — it needs semantic similarity, type constraints, and ideally human review. The current algorithm is good enough to *suggest* merges; applying them is a separate step (`cascade entity merge <id1> <id2>`).

45.6 — Design an improvement (open)

Sample answer — Stage 3 verification pass:

The improvement. After the LLM extracts entities, relations, and claims, run a *second LLM call* that takes the extracted JSON plus the original article text and asks: *"For each extracted relation, is the evidence string actually in the article? Return a verified subset."* Relations that fail verification get downgraded from `HIGH` to `MEDIUM` confidence, or dropped entirely if their evidence is hallucinated.

What it adds. Catches the most common LLM failure mode — relations between entities that aren't actually connected in the text. Reduces false- positive cascade results in Ch.43-style queries. Provides an audit log showing which relations the verifier rejected.

What it costs. One extra LLM call per article — roughly doubles Stage 3 latency from ~15s to ~30s. Doubles the LLM-per-article cost. For high-volume ingestion this matters; for analyst-curated workflows it's worth it. A cheap-model verifier (e.g. Groq for fast Llama) keeps cost reasonable while still catching the worst hallucinations.

What it might break. The verifier itself can hallucinate the *opposite* error — reject true relations because the evidence string is paraphrased rather than quoted. The fix is a "soft" verification threshold (semantic match, not literal string match) — which adds more cost and complexity. The risk is *compounding LLM errors*: a flawed verifier reduces trust in extraction *and* introduces new error modes.

Real engineering tradeoff: **verification gates quality at the cost of latency and complexity**. The right answer depends on whether you're optimizing for breadth (lots of mediocre coverage) or depth (fewer articles, higher trust).

Notes

45.1 — Why NER count > LLM count is healthy

NER finds candidates from raw text; the LLM filters based on the article's *focus*. NER might tag every country mentioned; the LLM keeps only the *meaningful* mentions. The difference is the LLM's *judgment value-add* — without it, every passing reference would land as an entity, drowning the graph in noise.

45.2 — The yield distribution is right-skewed

Mean 31 claims / max 100 is a long-tail distribution. Most articles are normal news pieces yielding 20-40 claims; a few analytical longforms yield 80-100. When designing rate limits or per- article budgets, plan for the tail, not the mean.

45.3 — Why credibility stays at default

Most Cascade installs use the LLM-assigned source types and never adjust the credibility band. As verification workflows mature (e.g. the verifier proposed in 45.6, or an analyst-driven "claim contested" workflow), per-source credibility will diverge from defaults. The schema *supports* the divergence; the workflow hasn't *demanded* it yet.

45.4 — Reading article shape

The (66 entities, 1 relation, 100 claims) example shows you can *characterize an article* by its ingestion footprint:

- **Connective** — many relations, fewer claims. Article is mapping connections.
- **Descriptive** — many claims, fewer relations. Article is enumerating facts.
- **Mixed** — typical news, balanced ratios.

A future Cascade feature could *cluster sources by ingestion shape* for editorial-style analysis.

45.5 — Why substring is necessary but not sufficient

The `LIKE` join finds *candidate* pairs cheaply but introduces three failure modes:

1. **Acronyms that aren't initialisms** — "Intel" inside "Central Intelligence" is a substring match but not a merge candidate.
2. **Generic words inside specifics** — "reserve" inside "coal reserves".
3. **Person partial names** — "Smith" inside "John Smith" *might* be the same person or might be a different Smith.

Real entity resolution uses embedding similarity, type constraints, co-occurrence with related entities, and (often) human review. The substring query is a *first pass* — useful for surfacing candidates, not for deciding.

45.6 — The general pattern

Pipeline improvements always trade three things:

1. **Latency** — extra stages take time.
2. **Cost** — extra LLM calls cost money.
3. **Robustness** — extra layers add failure modes.

Against:

- **Quality** — better extraction, fewer errors.
- **Coverage** — handling more source types.
- **Auditability** — a clearer trail for analysts.

The right answer depends on which axis your users care about. For Cascade's geopolitical-intelligence audience, quality and auditability matter more than raw throughput — so the verifier-pass design is *plausibly* a good fit.

Chapter 46 · Solutions

All solutions verified against live `cascade.db` or checked against the B4 tables in `cascade/core/truth.py`. Solutions 46.1–46.5 are runnable; 46.6 is open-ended.

46.1 — Compute b4_join by hand

Expression	Result	Explanation
TRUE ⊔ UNKNOWN	TRUE	UNKNOWN is identity for join
FALSE ⊔ BOTH	BOTH	BOTH is absorbing
TRUE ⊔ FALSE	BOTH	Contradiction → paraconsistent
UNKNOWN ⊔ UNKNOWN	UNKNOWN	Both empty stays empty

The two interesting patterns: (1) UNKNOWN *disappears* in joins (it's the identity), and (2) TRUE/FALSE *combine to BOTH* rather than picking a winner.

46.2 — `aggregate_truth([TRUE, TRUE, BOTH])`

Result: **BOTH**.

Why not TRUE (the "majority")? Because B4 aggregation is *not voting* — it's repeated lattice join:

```
step 0: TRUE
step 1: TRUE ⊔ TRUE = TRUE
step 2: TRUE ⊔ BOTH = BOTH    ← contamination
```

Once BOTH enters the chain, it absorbs subsequent joins. **One paraconsistent claim out of N contaminates the aggregate.** This is the paraconsistent commitment: the system refuses to average away contradiction.

If you wanted majority-vote semantics instead, you'd write:

```python
from collections import Counter
def majority_truth(values):
    return Counter(values).most_common(1)[0][0]
```

But that's not what Cascade does — and it's deliberately not what it does.

46.3 — Find the contested claims

```
SELECT id, substr(text, 1, 60) AS preview, confidence
FROM claims WHERE truth_value = 'BOTH' LIMIT 5;
```

Sample from live DB:

```
70   The closure of the Gulf waterway could lead to the...    MEDIUM
591  Iran is known for its oil exports.                       MEDIUM
592  Iran exports an estimated 2.1 million barrels of oil...  MEDIUM
```

The audit-history query:

```
SELECT claim_id, old_truth_value, new_truth_value, reason
FROM truth_history ORDER BY changed_at DESC LIMIT 10;
```

Live output:

```
7849  TRUE  BOTH     The assessment shifts to BOTH because...
7849  BOTH  UNKNOWN  No rules matched this claim
```

Only two rows. That's the actual state of the production install — the audit table is *currently sparse* because automated re-evaluation is opt-in. A fresh install will show zero rows.

46.4 — Truth-by-credibility crosstab

```
SELECT printf('%.2f', s.credibility) AS cred,
       c.truth_value, COUNT(*) AS n
FROM claims c
LEFT JOIN sources s ON c.source_id = s.id
WHERE s.credibility IS NOT NULL
GROUP BY printf('%.2f', s.credibility), c.truth_value
ORDER BY cred DESC, n DESC;
```

Sample from live DB:

```
cred  truth_value  n
----  -----------  ----
0.80  TRUE         4039
0.80  BOTH           48
0.80  UNKNOWN        29
0.80  FALSE          18
0.50  TRUE          127
0.50  UNKNOWN         5
```

Reading the table:

- **At credibility 0.80 (news sources)**: 4039 TRUE, 48 BOTH, 29 UNKNOWN, 18 FALSE. The BOTH rate is 1.2% — *lower* than the global 5% average.
- **At credibility 0.50 (opinion sources)**: 127 TRUE, 5 UNKNOWN, 0 BOTH, 0 FALSE. *No* paracon-

sistency.

The data shows higher-credibility sources do produce more TRUE outcomes, but the relationship is not perfectly monotonic — opinion-source claims are sparse enough that we don't see BOTH at all (only one source per claim usually means no contradiction chance).

The empirical takeaway: credibility bands correlate with TRUE-rate, validating the band's operational use. To strengthen the claim you'd want data with multiple credibility tiers at similar volumes — which Cascade's news-dominated corpus doesn't currently provide.

46.5 — Negation fixed points

```
NEG = {"TRUE": "FALSE", "FALSE": "TRUE",
       "BOTH": "BOTH",  "UNKNOWN": "UNKNOWN"}
```

Output:

```
¬TRUE = FALSE
¬FALSE = TRUE
¬BOTH = BOTH          ← fixed point
¬UNKNOWN = UNKNOWN    ← fixed point
```

Why BOTH is a fixed point. BOTH means "evidence for both sides." Its negation is *evidence for both sides of the negation* — which is the same shape. Negation just relabels which assertion is "positive" and which is "negative"; the two-sided structure survives.

Why UNKNOWN is a fixed point. UNKNOWN means "no evidence." Negating nothing gives you nothing. There's nothing to swap.

The pattern: **fixed points of negation are the informationally symmetric values** (BOTH = full info on both sides; UNKNOWN = no info on either side). TRUE and FALSE are *one-sided* and therefore move under negation.

46.6 — Where should `cancel` land? (open)

Sample answer.

A `cancel(claim_id, reason)` mechanism should **preserve the original truth_value** rather than mutate it, by writing a *suppression marker* in a new column. Proposed schema change:

```
ALTER TABLE claims ADD COLUMN canceled INTEGER DEFAULT 0;
ALTER TABLE claims ADD COLUMN canceled_reason TEXT;
ALTER TABLE claims ADD COLUMN canceled_at TEXT;
```

The cancellation function:

```
def cancel_claim(conn, claim_id, reason):
    conn.execute(
```

```
        "UPDATE claims SET canceled = 1,"
        "                  canceled_reason = ?,"
        "                  canceled_at = datetime('now')"
        " WHERE id = ?",
        (reason, claim_id))
    record_truth_change(conn, claim_id,
        old_truth=current_truth,
        new_truth=current_truth,    # unchanged
        old_confidence=current_conf,
        new_confidence=current_conf,
        reason=f"CANCEL: {reason}")
```

Why preserve `truth_value`. Cancellation isn't the same as flipping FALSE — it's saying "this claim is structurally non-defensible." The original truth state matters for:

- **Audit / forensics** — "what did the system believe before we canceled it?"
- **Re-evaluation** — if cancellation gets withdrawn, the original state is recoverable.
- **Provenance** — comprehensions can still see the fact existed; they just filter it out.

How queries respect cancellation. Add `AND canceled = 0` to every default query:

```
SELECT ... FROM claims WHERE truth_value = 'TRUE'
                         AND canceled = 0;
```

Provide a `--include-canceled` CLI flag that drops the filter for analyst review. Same pattern as the B4 `canceled` builtin from Vol I Ch.21 — the engine *knows* about cancelations but normal usage filters them out.

The audit trail stays in `truth_history` — the cancellation lands as a `truth_history` row with old == new (because truth itself didn't change) and a `CANCEL: ...` prefix on the reason. An `uncancel` function would write another row with `UNCANCEL: ...`. The append-only history is the *complete record*.

The general principle: **cancellation is metadata, not a truth-value transition.** Letting them live in separate columns keeps the B4 lattice operations pure while still giving downstream queries a way to hide non-defensible claims.

Notes

46.1 — UNKNOWN as identity is the killer feature

Most multi-valued logics have an identity element for their primary combinator, but B4's choice of UNKNOWN ↦ identity for join is what makes *incremental ingestion* work. Each new article contributes its own truth value, joined with whatever the system already knows. UNKNOWN $\sqcup$ X = X means *the absence of prior evidence doesn't bias* the new evidence. This is exactly what you want for a knowledge graph that grows over time.

46.2 — Why the system refuses to vote

A vote-based aggregator would let majority drown out a single dissenting source. That's *appropriate* for noisy crowdsourcing but *inappropriate* for journalism where the dissenting source might be the *only one who got it right*. B4 BOTH says: "we have contradiction; surface it for analyst review." That's the journalistic ethic of the system — *don't suppress dissent, surface it.*

46.3 — Why audit history is sparse

The `truth_history` table only fills up when truth-flip events occur. In a fresh install where no analyst evaluation has run, all claims stay at their initial truth and history is empty. Production installs running the reactor (Ch.50) accumulate dozens of rows per claim as the system re-evaluates against incoming evidence.

The trade: **automated history = useful audit trail, also = a lot of rows.** Cascade's choice is to make automated re-evaluation opt-in so the table doesn't bloat on installs that don't need it.

46.4 — Statistical caveats

The crosstab from §46.4 shows a correlation, not a causal relationship. Higher-credibility sources *also* tend to be:

- Better-edited (fewer extraction errors).
- More targeted (extracted entities are clearer).
- Already-merged (entity-resolution finds them).

Each of those independently reduces the BOTH rate. Untangling them would require A/B-testing extraction quality across credibility bands — outside what Cascade currently does.

46.5 — The deeper symmetry

Mathematically, the negation operator on B4 is the unique map satisfying:

- `¬¬x = x` (involutive)
- `¬⊥ = ⊥` (UNKNOWN fixed)
- `¬⊤ = ⊤` (BOTH fixed)
- `¬TRUE = FALSE` and `¬FALSE = TRUE`

This is a *Galois connection* — the lattice has a duality between join and meet under negation:

- `¬(a ⊔ b) = ¬a ⊓ ¬b` (De Morgan, one direction)
- `¬(a ⊓ b) = ¬a ⊔ ¬b` (De Morgan, other)

Belnap proved these in 1977. They're the *reason* the operations are useful for handling contradiction — the algebraic structure aligns with how natural-language negation behaves.

46.6 — Why a separate cancel column

Earlier KR systems often *overload* truth values to encode cancellation (e.g., a fifth value "RETIRED"). The downside: any code that branches on truth_value needs to handle the fifth case explicitly. With a

separate `canceled` column, all existing B4 code keeps working — it just sees the original truth value. Cancellation is a *filter*, not a *truth state*.

The architectural lesson: **orthogonal concerns deserve orthogonal columns.** Truth is what we believe; cancellation is what we're willing to publish. Conflating them makes both harder to reason about.

Chapter 47 · Solutions

All commands verified against a live populated Cascade install. Solutions 47.1–47.5 are runnable shell; 47.6 is open-ended (drawing required).

47.1 — All three centralities

```
for metric in degree betweenness pagerank; do
  uv run cascade discover centrality --metric $metric \
                                     --top 10 --json
done
```

Live output (top 5 from each, real Cascade DB):

```
DEGREE              BETWEENNESS         PAGERANK
------------------  ------------------  ------------------
Iran (293)          United States       Iran
United States       Iran                United States
Donald Trump        Donald Trump        Medallia (!)
China               China               Israel
Israel              Strait of Hormuz    China
```

The big-three overlap: Iran, US, China, Israel, Donald Trump appear in all three lists — they're *universally* important across local, global, and recursive measures. These are the entities you'd mention in any executive summary.

The metric-specific surfaces:

- **Degree-only** highlights: hub *count*. Russia shows up high here from sheer mention frequency.
- **Betweenness-only** highlights: *broker* roles. `Strait of Hormuz` (a piece of infrastructure) is high — it sits on many shortest paths between energy entities.
- **PageRank-only** highlights: *recursive* importance. Medallia (a software company) sneaks into the top 5 because it's connected to *other* high-PageRank entities, even with few direct edges.

Each metric catches a different sense of "important." Production analysts run *all three* and consume the union.

47.2 — Bridges by entity type

```
uv run cascade discover bridges --json \
  | jq 'group_by(.type) | map({type: .[0].type,
                               count: length})
        | sort_by(-.count)'
```

Live output:

```
ORG         149
ACTOR        63
STATE        46
EVENT        34
TECHNOLOGY   34
COMMODITY    25
...
```

ORG dominates bridges because ORGs sit at *interfaces* in the geopolitical graph: companies mediate between governments, between sectors, between regions. A single company can be the *only* path connecting two otherwise-disconnected subgraphs.

ACTOR comes second because individual leaders bridge governments and orgs.

STATE is third — fewer than expected. Why? STATES are *hubs* (high degree) but not necessarily *bridges* (articulation points). Many STATES are embedded inside connected subgraphs; removing them fragments their local neighborhood but not the larger graph.

The empirical takeaway: **STATEs are the things journalism talks about; ORGs are the things that hold the graph together.**

47.3 — Cluster size distribution

```
uv run cascade discover clusters --json \
  | jq '.[] | {size: .size, dominant_type: .dominant_type}'
```

Sample live output (head):

```
size=1996  dominant=ORG
size=4     dominant=ACTOR
size=3     dominant=ORG
size=3     dominant=STATE
size=2     dominant=ACTOR
size=2     dominant=COMMODITY
size=2     dominant=ACTOR
...
```

One giant component (~1996 entities) + many tiny isolated pairs/triples. This is the *small-world giant component* phenomenon — most entities are connected to most others through some chain of relations, while a long tail of orphan pairs sits disconnected.

The orphan pairs are typically:

- Entities mentioned in only one source that didn't share any actors with the rest.
- Failed-resolution duplicates that didn't merge with their canonical counterparts.

Cleaning these up is part of the *entity-resolution maintenance* loop. After consolidation, the small clusters often merge into the giant component.

47.4 — Connections between two entities

```
uv run cascade discover connections "Iran" "crude oil"
```

Live sample (top 5 paths from the real DB):

```
score=1.0   path=Iran -> crude oil
score=0.8   path=Iran -> Abbas Araghchi -> Strait of Hormuz -> ... -> crude oil
score=0.75  path=Iran -> ... -> crude oil
...
```

The *direct* connection (1 hop, SUPPLY relation) is the highest-scored. Path scoring rewards domain-transitions per hop — but a direct STATE→COMMODITY connection is already 1.0 by construction (1 transition / 1 hop).

Walking the longer paths reveals **why** the connection is interesting: even with a direct SUPPLY relation, *multiple* longer paths exist through different intermediaries (officials, infrastructure). Each is a different *causal story*.

Real journalism uses both: the direct fact AND the chain of intermediaries.

47.5 — Cascade-depth saturation

```
for d in 1 2 3 4 5; do
  cascade discover cascade "Iran" --depth $d --json \
    | jq '.tiers | length'
done
```

Sample output:

```
depth=1  tiers=2   (origin + Tier 1)
depth=2  tiers=3   (origin + Tier 1 + Tier 2)
depth=3  tiers=4
depth=4  tiers=5
depth=5  tiers=5   ← saturation
```

The count stops growing between depth 4 and 5 — the *causal diameter* of the graph from Iran is 4. Beyond 4 hops via causal-typed edges, you don't reach any new entities (everything reachable has been visited).

This is **graph-shape data**: a tightly-connected graph saturates at depth 3-4; a sparse one might need depth 7-10. Cascade's default depth of 3 is chosen because Tier 3 is *the analytical sweet spot*: deep enough to be non-obvious, shallow enough to stay traceable.

47.6 — Toy graph (open)

The classic example:

```
A — A1
|
A — A2
|
A — A3
|
A — A4
|
A — B — C
```

Drawn more carefully:

```
        A1
        |
A2 — A — A4
        |
        A3
        |
        B
        |
        C
```

- **A has degree 5** (connected to A1, A2, A3, A4, and B).
- **B has degree 2** (connected to A and C).
- **A's betweenness is low** because its 5 neighbors are all leaves — no paths through A except those starting/ending at A.
- **B's betweenness is high** — *every* path from C to any A-cluster member passes through B.

In NetworkX terms:

```
import networkx as nx
G = nx.Graph()
G.add_edges_from([
    ("A","A1"), ("A","A2"), ("A","A3"), ("A","A4"),
    ("A","B"),  ("B","C"),
])
print(nx.betweenness_centrality(G))
# {'A': 0.333, 'A1': 0, 'A2': 0, 'A3': 0, 'A4': 0,
#  'B': 0.4, 'C': 0}
```

B (degree 2) has betweenness 0.4 > A (degree 5) has betweenness 0.333. The intuition: *being in the middle matters more than having many neighbors.* A is a hub; B is a bridge. Removing A fragments only the A-cluster; removing B disconnects C from everything.

The deeper lesson: **a "popular" node and a "strategic" node are different things.** Production KR systems track both, and the divergence between the two scores is itself a content signal — when they agree, you've found a *universally* important entity; when they disagree, you've found two *different kinds* of importance.

Notes

47.1 — Why three metrics

The Cascade implementation runs all three by default (returning all three scores per node) because no single metric captures the whole story. A purist purist might run only PageRank (the most "correct" recursive measure); production journalism needs degree (fame) and betweenness (broker role) too. Cheap to compute, all three.

47.2 — ORG-as-bridge is a content claim

The result that ORGs dominate bridges is not a universal graph-theoretic truth — it's a *fact about the geopolitical-journalism corpus*. A graph built from medical literature would probably show GENE or DISEASE as the dominant bridge type. The algorithm is generic; the result depends on the data.

47.3 — Why connected components, not Louvain

The chapter (§47.6) discusses the choice. Briefly: Louvain produces *modularity-optimized* communities — finer-grained than connected components. With Cascade's small-world structure, Louvain would slice the giant component into 5-15 sub-communities that *aren't disconnected* but are *internally denser*. That's a more useful slicing for analyst review.

The reason Cascade hasn't migrated: connected components are deterministic and trivial to explain. Louvain is randomized and harder to interpret. The roadmap calls for Louvain to be exposed as an opt-in flag (`--algorithm louvain`) without replacing the default.

47.4 — Why direct paths score 1.0

The scoring formula is `transitions / path_length`. A 1-hop path between two entities of *different* types has 1 transition in 1 hop → score 1.0. A 4-hop path with 4 transitions also scores 1.0 (every hop transitions between domains).

The score is *not* a measure of *interestingness* — it's a measure of *cross-domain-density*. Both direct connections and richly-traversing chains can score high. Reading the results means looking at both score *and* length.

47.5 — The "Tier 3 is the sweet spot" claim

Tier 1 is *obvious* (it's what the article said directly). Tier 2 is *plausibly inferrable* by a careful reader. Tier 3 is *non-obvious* — something a human would have to follow the chain to see. Tier 4+ is *speculative* — the chain might or might not hold; the further you go, the more the conditional probabilities decay.

This is why Cascade defaults to depth 3 — it's the *editorial frontier* where the system adds value beyond what a careful human reader gets.

47.6 — The lesson generalizes

The toy graph teaches the general principle that shows up everywhere in network analysis:

Hub = many neighbors **Bridge** = few but strategically-placed neighbors

In Cascade, this is why both `centrality --metric degree` and `centrality --metric betweenness` are first-class — they answer different questions about importance, and the difference is itself a signal.

Chapter 48 · Solutions

All exercises verified against a live Cascade install + Axioma MCP bridge. Solutions 48.1–48.5 are runnable shell; 48.6 is open-ended.

48.1 — Bridge warm-up

```
$ time uv run cascade axioma exec 'println(42)'
KB service started at /tmp/axioma-kb.sock
KB opened: /Users/.../cascade.db
Axioma MCP server starting on stdio...
{"output": "42\n", "result": "null", "success": true,
 "type": "NULL"}
uv run cascade axioma exec ...  0.67s user 0.19s system  91% cpu  0.944 total
```

Cold start: ~0.9 seconds. That includes:

- Python interpreter startup (~150ms)
- Cascade package import (~250ms)
- Axioma binary spawn (~200ms)
- KB service spawn (~150ms)
- KB load from cascade.db (~150ms)
- First MCP roundtrip (~10ms)

The second invocation from a *fresh* shell takes the same ~0.9 seconds because each `uv run cascade ...` is its own process. **The warm path only applies within a single process** — in practice that means the Cascade dashboard, scheduled-task runner, or REPL.

If you want a fast inner loop, run `uv run cascade repl` and keep the process open; subsequent commands within the REPL hit the warm path (~50-100ms per evaluation).

48.2 — List loaded relations

```
uv run cascade axioma query list-relations \
  | jq -r '.relations | .[]' | grep '^[a-z]'
```

Live output:

```
commodity
test_fact
```

The 8 uppercase relations are Cascade's canonical ontology. The lowercase relations come from `.ax` rules and ad-hoc `axiom/persist` statements:

- **`commodity`** — likely came from a `.cascade/rules/axiomalang/*.ax` file asserting facts like `axiom commodity("Bronze")`.
- **`test_fact`** — from the Ch.43 §43.5 walkthrough where we added a test fact via the Cascade-Axioma bridge.

Reading `.cascade/rules/axiomalang/oil_gold_positive_correlation.ax`:

```
concept oil_gold_correlation {
    domain: "economic";
    type: "correlation";
    direction: "positive";
}

rule oil_spike_gold_rise {
    when entity("Crude Oil") and entity("Gold")
    then with_context("inflationary_hedge", ...)
}
```

This rule encodes the *empirical* observation that oil prices and gold prices correlate positively (both rising under inflation pressure). It's the kind of *domain expertise* that doesn't appear in any article — an analyst codified it once and now the system applies it everywhere.

48.3 — Each engine on one claim

```
for eng in python prolog axioma cascade; do
  uv run cascade evaluate <id> --mode rules \
                               --engine $eng --test
done
```

Sample output across engines (claim 1 from test DB, abstract text about Irish-US relations):

```
engine=python   → UNKNOWN (SPECULATIVE) — "No rules matched"
engine=prolog   → UNKNOWN (SPECULATIVE) — "No rules matched"
engine=axioma   → UNKNOWN (SPECULATIVE) — "Axiomalang B4: NEITHER"
engine=cascade  → UNKNOWN (SPECULATIVE) — "No rules matched"
```

For *most* claims in the test database, all four engines agree on `UNKNOWN/SPECULATIVE` — because the shipped rules are *domain-specific* (oil/gold, monetary tightening, capital flows) and don't fire on out-of-domain claims (here: Irish St. Patrick's Day diplomacy).

When the engines *do* disagree, it's typically because:

- **Python rules** include *correlation* rules from historical patterns (e.g. USD↔gold inverse).
- **Prolog rules** are pure relational queries — they trigger when the claim's entities match predicates.
- **Axioma rules** are B4-evaluated — they handle contradiction explicitly.
- **Cascade rules** are *meta-rules* about the graph structure (e.g. ADVERSARIAL+ALLIANCE combi-

nations).

Hybrid mode (`--mode rules --engine hybrid`) takes all four verdicts and aggregates them via B4 meet (Ch.46 §46.3). The hybrid output is more conservative than any single engine — agreement across engines is the strongest signal.

48.4 — Dry-run verification

```
$ sqlite3 cascade.db \
    "SELECT truth_value, confidence FROM claims WHERE id=1"
TRUE|HIGH

$ uv run cascade evaluate 1 --mode rules --engine axioma --test
... (verdict computed, displayed)
Updated: UNKNOWN (SPECULATIVE)
   (test_only: true — not persisted)

$ sqlite3 cascade.db \
    "SELECT truth_value, confidence FROM claims WHERE id=1"
TRUE|HIGH    ← unchanged
```

The `--test` flag suppresses the `update_claim(...)` call in step 6 of the orchestrator. Useful for:

- **Sanity-checking** rule outputs before applying them.
- **Running a verification sweep** without affecting production data.
- **A/B testing** different rule sets.

The `--test` output still includes the full reasoning chain, so analysts can review what *would have happened* before committing.

48.5 — Empty-proposition trap

A claim like *"The Irish are bringing $6.1 billion in planned investments to the U.S."* might have clear propositions; a claim like *"The situation remains tense"* may decompose to zero propositions because:

- The subject ("the situation") doesn't bind to any Cascade entity.
- The predicate ("remains tense") isn't a known primitive.
- No CAUSAL/IMPLIES relations exist in the context.

When `decompose_claim` returns `{"propositions": []}`, the orchestrator (cascade/core/axioma-lang.py: 171):

```
if not propositions:
    log.info("No propositions extracted from claim %d",
             claim_id)
    return None
```

Returns None. The downstream `evaluate` CLI displays `UNKNOWN (SPECULATIVE) — "no propositio`

`ns extracted"`.

The fix is on the *decomposer* side: better NSM prime coverage, LLM fallback for hard cases, or hand-written `.ax` rules that match the abstract predicates. Production Cascade evolves the decomposer organically as analysts encounter specific abstract-claim failure modes.

48.6 — Where to extend? (open)

Sample answer: **Improve step 3 (truth-seeding) by consulting related-entity truth values, not just the claim's own truth.**

The improvement. Currently, step 3 binds each proposition's seed truth to *the claim's own truth value*. This is wasteful — if the claim already asserts TRUE, the proposition gets TRUE, and the B4 meet collapses immediately. A smarter truth-seeder would look at *related claims*: for each proposition's entities, find other claims mentioning those entities, take the B4 join of their truth values, and use that as the seed.

```
def smart_truth_seed(prop, conn):
    related_claims = find_related_claims_by_entities(
        conn, prop["entities"])
    truth_values = [TruthValue(c["truth_value"])
                    for c in related_claims]
    return aggregate_truth(truth_values)
```

What it adds. Cross-claim evidence aggregation *at evaluation time*. A proposition about "Iran + oil" gets seeded by *all* known facts about Iran + oil, not just the current claim's verdict. This makes the B4 evaluation richer and catches contradictions that span multiple claims.

What it costs. One extra SQL query per proposition (potentially expensive: O(N) over all claims). And a more subtle risk: the seed truth *biases* the evaluation, potentially making it self-referential (related claims got their truth from the same rules; now those rules see their own output as input). The fix is to weight the seed truth lower than rule-derived truth — but that adds yet another parameter to tune.

What it might break. Existing tests that expect specific truth values for specific claims — the cross-claim aggregation might flip an answer that was UNKNOWN to BOTH. Production deployment would need careful migration, possibly with a feature flag.

Notes

48.1 — Process startup vs warm path

The 0.9s cold-start cost is *not* an MCP problem — most of it is Python+Cascade import latency. Profiling shows:

```
total:     ~900ms
  Python startup:       ~150ms
  Cascade imports:      ~250ms
  MCP setup:            ~400ms
  Actual work:           ~10-100ms
```

Optimizations like `--no-import-cascade` (skipping some imports) reduce this. The fundamental fix is to *keep the Python process alive* — which is what the dashboard, scheduled tasks, and REPL do.

48.2 — Auto-injected relations are first-class

The lowercase relations (`commodity`, `test_fact`) *aren't* hacks. They're the same as the uppercase ones at the SQL level — same `relation_types` table, same `relations` table for facts. The distinction is *cultural*:

- **Uppercase** = canonical Cascade ontology.
- **Lowercase** = Axioma-asserted, domain-specific.

This distinction lets downstream queries filter: "show me only Cascade-grade relations" or "show me everything including the Axioma extensions."

48.3 — When engines disagree

A live example of inter-engine disagreement (not shown above) is on claims about *prices*. Python's correlation rules fire for "oil rose" claims; Cascade's structural rules fire for the same. The two engines may produce different verdicts because they're looking at *different facts*:

- Python looks at historical correlation tables.
- Cascade looks at graph structure (who's connected to whom).

Hybrid mode joins both, surfacing the disagreement as `BOTH` and prompting analyst review. **This is what hybrid mode is for** — catching the rare-but-important cases where engines that *should* agree don't.

48.4 — Why dry-run exists

The `--test` flag is the bridge between *development* (where you want to iterate quickly on rules) and *production* (where you don't want broken rules to corrupt the DB). Typical workflow:

1. Write a new `.ax` rule file.
2. Run `cascade evaluate <id> --engine axioma --test` on representative claims.
3. Read the verdicts. Adjust the rule. Iterate.
4. Once the verdicts look right, run again without `--test`.

The flag is the *separation of trial and effect* — a small UX investment that pays off massively in debug cycles.

48.5 — The decomposer is the weakest link

When `decompose_claim` returns zero propositions, the whole pipeline silently produces UNKNOWN. This is *graceful* but also *misleading* — the analyst sees "no rules matched" when actually "no propositions extracted."

The fix would be to *surface* this distinction in the reasoning string. A future enhancement: change the verdict from `UNKNOWN/SPECULATIVE/"no rules matched"` to `UNKNOWN/SPECULATIVE/"decomposition failed: no propositions"` when the issue is *parsing*, not *matching*. Small change; big debug-quality win.

48.6 — Step-by-step extensibility

The six-step pipeline is *purposely shallow* — each step does one thing, so each step is easy to swap. This is a textbook example of the **pipeline pattern**: linear sequence of transformations, each with a clear input/output contract. Extending step 3 doesn't touch step 2 or step 4. Compare to a tangled monolithic evaluator where every change risks every output — the modularity is what makes Cascade safe to evolve in production.

Chapter 49 · Solutions

All `.ax` files in solutions 49.1, 49.3, 49.4 are verified to parse + run via `axioma --no-kb`. Solution 49.2 is pure-syntax (no execution); 49.5 requires a live Cascade install; 49.6 is open.

49.1 — Your first runnable rule

```
relation greeting(speaker :: String -> "speaker",
                  text :: String -> "text")
axiom greeting("Cascade", "Hello, knowledge graph!")
println({(S, T) | greeting(S, T)})
```

Verified run:

```
$ axioma --no-kb ex_41_1_first_rule.ax
{("Cascade", "Hello, knowledge graph!")}
```

After dropping into `~/.cascade/rules/axiomalang/`, the next `cascade axioma exec` call logs:

```
Loaded Axiomalang rule file: ex_41_1_first_rule.ax
```

The relation now persists in the Axioma session for all subsequent calls.

49.2 — Translate prose to Horn

```
# 1. "X is a teammate of Y if X plays for Y's team."
#    Two-step: find Y's team, check X plays for it.
teammate(X, Y) <= plays_for(X, T) and plays_for(Y, T)

# 2. "X is a senior officer of Y if X is an officer of Y
#     and X has tenure >= 10."
senior_officer(X, Y) <= officer(X, Y) and tenure(X, T) and T >= 10

# 3. "X depends on Y if X needs Y or X is supplied by Y."
#    Two strict rules — the disjunction is encoded as two heads.
depends(X, Y) <= needs(X, Y)
depends(X, Y) <= supplied_by(X, Y)
```

Three patterns surfaced:

1. **Joins via shared variables.** Rule 1 uses the same `T` (team) twice — the unifier forces both `plays_for` calls to bind to the same team.

2. **Inline comparison.** Rule 2 uses `T >= 10` directly in the body. Axioma's Horn-clause body accepts both relation calls and comparison expressions.
3. **Disjunction = two rules.** Rule 3 splits "needs OR supplied_by" into two separate rules with the same head. Horn clauses don't have a syntactic OR; multiple rules give you the semantic OR.

49.3 — The sanctions risk rule (real)

```
relation sanctions(target :: String -> "sanctioned_state")
relation supplies(source :: String -> "state",
                          good :: String -> "commodity")
relation at_risk(item :: String -> "commodity")

axiom sanctions("Iran")
axiom supplies("Iran", "crude oil")
axiom supplies("Iran", "natural gas")
axiom supplies("Saudi Arabia", "crude oil")

at_risk(C) <= sanctions(S) and supplies(S, C)

println("Sanctioned:", {S | S <- sanctions(S)})
println("Supply pairs:", {(S, C) | supplies(S, C)})
println("At risk:", {C | C <- at_risk(C)})
```

Verified output:

```
Sanctioned: {"Iran"}
Supply pairs: {("Iran", "crude oil"),
               ("Iran", "natural gas"),
               ("Saudi Arabia", "crude oil")}
At risk: {"crude oil", "natural gas"}
```

After dropping into `~/.cascade/rules/axiomalang/sanctions_risk.ax`, `cascade axioma query list-relations` shows the three new relations.

49.4 — Add a defeasible override

The pattern: defeasible rule for the default, strict rule for the override. Analysts query both and compute the difference.

```
axiom has_strategic_reserve("crude oil")    # US SPR

at_risk_defeasible(C) <~~ sanctions(S) and supplies(S, C)
not_at_risk(C) <= has_strategic_reserve(C)

println("Defeasibly at risk:", {C | C <- at_risk_defeasible(C)})
println("Protected by reserves:", {C | C <- not_at_risk(C)})
```

Output:

```
Defeasibly at risk: {"crude oil", "natural gas"}
Protected by reserves: {"crude oil"}
```

The defeasible derivation puts `crude oil` in the risk set; the strict override puts it in the protected set. Downstream analysis subtracts — final at-risk list is `{"natural gas"}`.

The `<~~` syntax creates *conjecture*-grade derivations (Vol I Ch.18 epistemic grounding) that get capped lower than `<=`-derived strict facts. This is the *defeasibility cap* — even though both rules fire, the strict rule's verdict outranks the defeasible one.

49.5 — Verify rule firing

Workflow:

```
# Install the rule
cp ex_41_3_sanctions_risk.ax \
   .cascade/rules/axiomalang/sanctions_risk.ax

# Confirm load
uv run cascade axioma exec 'println("ok")' 2>&1 \
  | grep "Loaded Axiomalang"
# → Loaded Axiomalang rule file: sanctions_risk.ax

# Find a relevant claim
sqlite3 data/cascade.db \
  "SELECT id, substr(text, 1, 60) FROM claims
   WHERE text LIKE '%Iran%' LIMIT 3;"
# → e.g. claim 70 about Strait of Hormuz

# Evaluate (dry run, doesn't write to DB)
uv run cascade evaluate 70 --mode rules \
                           --engine axioma --test
```

The reasoning chain output should mention `sanctions`, `supplies`, or `at_risk` if the rule fires on the claim. If it doesn't, two diagnoses:

1. **The claim has no entities matching the rule's relations.** `decompose_claim` couldn't bind propositions to the rule's premises.
2. **The relation names don't match.** Cascade's relation_types uses uppercase (`SUPPLY`); your rule uses lowercase (`supplies`). They're *different relations* at the Axioma level.

The fix for #2 is to either rename your relation to match (use `SUPPLY` in your `.ax` file) or to add a *bridge rule*:

```
supplies(A, B) <= SUPPLY(A, B)
```

This propagates from Cascade's canonical relation into your rule-private one.

49.6 — Domain sketch (open)

Sample answer — software engineering domain:

```
# software_security_risk.ax — sketch (not yet run)

create SecurityRisk {
    purpose: "Conditions under which a software dependency
              poses a security threat to its consumers",
    formed_by: "stipulation",
    default_grounding: "conjecture"
}

relation dependency(parent :: String -> "package",
                            child :: String -> "package")
relation has_cve(pkg :: String -> "package",
                         cve_id :: String -> "cve")
relation transitive_dep(parent :: String -> "package",
                                descendant :: String -> "package")
relation at_risk_pkg(pkg :: String -> "package")

# Facts (sketch — would normally come from package manifest scan)
axiom dependency("my-app", "log4j")
axiom dependency("log4j", "log4j-core")
axiom has_cve("log4j-core", "CVE-2021-44228")

# Strict transitive closure
transitive_dep(P, D) <= dependency(P, D)
transitive_dep(P, D) <= dependency(P, M) and transitive_dep(M, D)

# Strict risk: if a transitive dep has a CVE, parent is at risk
at_risk_pkg(P) <= transitive_dep(P, D) and has_cve(D, _)

# Defeasible: usually-true unless the CVE is mitigated
at_risk_pkg(P) <~~ transitive_dep(P, D) and has_cve(D, _)
                  and not mitigated(P, D)

println("At-risk packages:", {P | P <- at_risk_pkg(P)})
```

TODOs noted in the sketch:

- `has_cve(D, _)` — verify wildcard _ in rule bodies works (Vol I Ch.31 noted it doesn't in strict rules — use a named unused variable instead).
- `not mitigated(P, D)` — negation in Horn clause bodies isn't classical Axioma; the right idiom is *closed-world failure* via a separate cancellation builtin.
- Transitive closure depth — Vol I §31.4 documented ~3-round forward-chaining limit; verify deep dependency trees still resolve.

To debug:

```
$ axioma --typecheck software_security_risk.ax
# fix the wildcard / negation issues
$ axioma --no-kb software_security_risk.ax
# verify the transitive closure depth
```

This is how a real rule file matures: sketch → typecheck → run → fix → iterate.

Notes

49.1 — Why `define relation` is the contract

Without `define relation`, calling `greeting(...)` would either fail (relation undeclared) or silently create an untyped relation. The declaration is the *type-and-arity contract* — Cascade's downstream tooling knows what to expect when querying.

The Lojban-style place labels (`speaker`, `text`) make the relation self-documenting. Future maintainers (including future-you) can read the relation declaration and know exactly what each position means — no need to trace through the extraction code.

49.2 — Disjunction as multiple rules

Horn clauses are *conjunctive* by definition — the body is "all premises must hold." Disjunction requires *multiple rules with the same head*. This is a feature, not a bug:

- Each rule has its own *provenance* — you can query "which rule fired to derive this fact?"
- Each rule can be *independently cancelled* if defeasible.
- The disjunction is *structurally explicit* — reading the rules tells you which alternative branches exist.

Production rule sets often have 3-5 rules with the same head, each representing a different way the fact could be derived.

49.3 — The relation-name lowercase convention

Real Cascade has eight UPPERCASE canonical relations (CAUSAL, SUPPLY, ...) plus any lowercase Axioma-injected ones. The convention:

- **UPPERCASE** for Cascade ontology relations.
- **lowercase** for domain-specific Axioma rules.

This is purely a cultural convention — at the SQL level both are the same. The convention helps analysts distinguish "official" relations from "experimental" ones at a glance.

49.4 — Defeasibility cap vs strict override

Two engineering patterns for "rule with exception":

1. **Defeasibility cap** (this exercise) — the defeasible rule fires by default, and the strict override outranks it through the epistemic grounding hierarchy. Downstream queries see both and decide.

2. **Explicit cancellation** — fire the defeasible rule, then call `cancel(at_risk, "crude oil")` to suppress specific derivations. The provenance survives but queries filter the cancelled facts.

Both are valid; pick based on what's easier to reason about. For *category-level* exceptions (strategic reserves), the cap pattern is cleaner. For *case-by-case* exceptions, the explicit cancellation is more flexible.

49.5 — Why your rule might not fire

Three common reasons:

1. **Relation-name mismatch.** Your rule uses `supplies`; Cascade has `SUPPLY`. Use a bridge rule (`supplies(A, B) <= SUPPLY(A, B)`) or match the canonical name directly.
2. **Decomposer didn't bind the proposition.** `decompose_claim` couldn't relate the claim text to your rule's entities. Try a more concrete claim with explicit named entities.
3. **Rule body has unbound variables.** A premise like `sanctions(S)` only fires if `S` can be bound. If no `sanctions(...)` facts exist in the Axioma session at evaluation time, the rule silently fails.

Debug by running the rule file standalone with sample facts (Ch.49 §49.3's approach) — that removes the decomposer from the loop.

49.6 — The general pattern

Real-world rule authoring follows this loop:

1. **Identify a pattern** you've noticed in the domain.
2. **Encode it as 1-2 strict rules**.
3. **Identify the exceptions** that empirically break the rule.
4. **Add defeasible rules** for the exceptions, or convert to defeasible-cap pattern.
5. **Test against historical data** before committing.
6. **Monitor production statistics** (`rules` table's `applied_count` / `success_count`).

Production Cascade installs accumulate dozens of rules this way. Each represents a piece of domain expertise that *the extraction pipeline can't infer*. They're the system's *learned knowledge*.

Chapter 50 · Solutions

All exercises verified against a live Cascade install. Solutions 50.1–50.3 + 50.5 are runnable; 50.4 and 50.6 are open-ended.

50.1 — Tool-use agent walkthrough

Sample run:

```
$ uv run cascade agent -v \
    "What does Cascade know about Iran's connections to oil markets?"
```

Verbose output shows:

```
Agent round 1/5 | messages=1
Agent tool call: search_entities({"query": "Iran"})
Agent tool call: search_entities({"query": "oil"})
Agent round 2/5 | messages=4
Agent tool call: get_entity_relations({"entity_name": "Iran"})
Agent round 3/5 | messages=6
Agent tool call: cascade_chain({"entity_name": "Iran", "depth": 2})
Agent round 4/5 | messages=8
[final synthesized response]
```

Typical pattern: 3-4 tool calls across 3-4 rounds. The LLM:

1. Searches for the named entities (`Iran`, `oil`).
2. Pulls relations for the central entity.
3. Traces a cascade chain for the impact analysis.
4. Synthesizes findings into prose.

The choice of tools reflects the *system prompt* — the analyst persona biases toward `cascade_chain` and `centrality_ranking` over `search_claims`. A different prompt would surface different tool patterns.

50.2 — Reactor status

Cold (reactor not running):

```
{
  "running": false,
  "events_received": 0,
```

```
  "rules_fired": 0,
  "actions_executed": 0,
  "errors": 0,
  "pending": 0
}
```

After `cascade reactor start` in another terminal, then triggering a change:

```
{
  "running": true,
  "events_received": 1,
  "rules_fired": 0,
  "actions_executed": 0,
  "errors": 0,
  "pending": 0
}
```

`events_received` increments per change_log row; `rules_fired` increments per matched rule; `actions_executed` per action that ran successfully. If `rules_fired == 0` after a change, *no rule's preconditions matched* — expected if the test DB doesn't have rules targeting the specific change type.

Why the daemon must be separately started. The reactor is a long-running process; it can't auto-start from a synchronous CLI invocation without backgrounding (which would lose the foreground status display). Production deployments run it under systemd or `nohup ... &`.

50.3 — Monitor add + run

```
$ cascade monitor add "AI safety" --query "AI safety regulation" --interval 24
[done] Monitor added: ID=2, entity="AI safety", interval=24h
```

After adding, the `monitors` table now has:

```
id  search_query           interval_hours
--  ---------------------  --------------
1   Strait of Hormuz       24
2   AI safety regulation   24
```

Running `cascade monitor run` triggers:

1. Search Google News RSS for "AI safety regulation" → N articles.
2. Dedupe against existing `sources.url`.
3. For each new URL: `cascade ingest <url>` → Stage 1-4 pipeline (Ch.45).
4. Update `last_run_at` and `last_result`.

Live result: typically 2-5 new sources, ~30-150 new entities, ~30-100 new claims per run. The exact numbers depend on news activity and dedup hit rate.

50.4 — Compare four mechanisms (open)

Sample answer — four implementations of "find new evidence about Iran":

a) Tool-use agent (synchronous)

```
cascade agent "Find recent claims about Iran in cascade.db"
```

Runs once per invocation, returns synthesized prose. **Best for**: ad-hoc analyst queries where the answer is needed now and the question is one-off.

b) Reactor (event-driven)

Write an `.ax` rule that fires on `SOURCE_INGESTED` events containing Iran-related entities, and writes a flag to a `notifications` table. The analyst checks notifications when they log in. **Best for**: passive-discovery workflows where the analyst wants alerts but not constant polling.

c) Monitor (scheduled)

`cascade monitor add "Iran" --query "Iran latest news" --interval 6`. Every 6 hours, Cascade scrapes new articles about Iran and ingests them. Fresh evidence accumulates automatically. **Best for**: keeping the knowledge graph fresh without analyst intervention — *continuous operation*.

d) MCP server (cross-process)

The analyst uses Claude Desktop with Cascade's MCP plugged in. They ask Claude: "What's new about Iran?" Claude calls Cascade's `search_claims` + `cascade_chain` tools. **Best for**: when the analyst's primary tool is *another LLM client* (Claude Desktop, custom agent) rather than Cascade's own CLI.

The choice:

- Need a real-time answer → (a).
- Want async alerts → (b) + a notifier.
- Want fresh data → (c).
- Multi-app workflow → (d).

Production deployments use *all four* — each for its slice.

50.5 — GraphEventBus structure

From `cascade/core/events.py`:

```
class ChangeType(str, Enum):
    ENTITY_CREATED = "entity_created"
    ENTITY_UPDATED = "entity_updated"
    ENTITY_DELETED = "entity_deleted"
    RELATION_CREATED = "relation_created"
    RELATION_UPDATED = "relation_updated"
    RELATION_DELETED = "relation_deleted"
    CLAIM_CREATED = "claim_created"
    CLAIM_UPDATED = "claim_updated"
```

```
    CLAIM_DELETED = "claim_deleted"
    SOURCE_INGESTED = "source_ingested"
```

a) Ten ChangeType values — three operations (created/updated/deleted) × three item types (entity/relation/claim) + one source event.

b) Synchronous — uses Python's `threading.Lock` for thread-safety but emits *synchronously*. The caller blocks until all subscribers have processed the event. No async/asyncio.

c) Multiple subscribers — yes. The bus keeps a `_subscribers: list[Callable]` and dispatches sequentially to each. The reactor is the primary subscriber; tests or extensions can add more.

The **synchronous + multiple-subscriber** design is fine because:

- Subscribers are *fast* (they read change_log rows but don't do heavy work inline).
- Heavy work happens in the *reactor's own thread/process* — the bus just notifies; the reactor decides what to do.
- The single-process design avoids serialization costs (no IPC needed).

If you needed *cross-process* event subscription (e.g. a dashboard process reacting to changes), the right pattern is **poll change_log** rather than extend the bus. SQLite + WAL gives you that cheaply.

50.6 — A fifth agent mechanism (open)

Sample answer — **the Curator agent**:

What it's for: Periodic *quality maintenance* of the knowledge graph. The curator agent reviews recently-ingested entities and relations, flagging:

- Duplicate-entity candidates that survived resolution (Ch.45 §45.5).
- Low-confidence claims that haven't been re-evaluated in N days.
- Orphan entities (no relations).
- High-degree but never-cited entities (likely hallucinations).

It produces a *review queue* — not auto-fixing, just surfacing — for an analyst to approve or reject.

Where it fits in the architecture:

- **Trigger**: scheduled (cron, like monitors) — daily or weekly.
- **Operation**: LLM-driven (uses the tool-use agent's tool catalog) — needs reasoning, not just rule-firing.
- **Persistence**: writes to a new `curator_queue` table — the analyst's review queue.
- **Cost model**: bounded by interval frequency.

This sits *between* the reactor (event-driven, narrow rules) and the tool-use agent (on-demand, broad reasoning). It's the **periodic-deep-think** slot in the architecture — slower than the reactor, more thorough than monitors, less interactive than the agent.

The deeper principle: **Cascade's swarm has slots for *every cost/latency profile* in the agent-engineering matrix**:

Latency	Cost	Today's slot
Sync, fast	Free	Reactor
Sync, slow	LLM	Tool-use agent
Async, free	Schedule	Monitor
Async, LLM	(gap)	**Curator**

The curator is the missing fourth quadrant.

Notes

50.1 — Why the agent prefers some tools over others

The system prompt biases the LLM's tool selection. A *generalist* analyst will reach for `search_claims` first; a *graph theorist* will prefer `centrality_ranking`; an *intelligence analyst* (Cascade's actual prompt) will favor `cascade_chain`. To get different behavior, modify the prompt — not the tool catalog.

50.2 — Reactor as a forward-chaining inference engine

The reactor is a classic *production-rule system* (in the Newell/Simon sense). Rules have:

- **LHS** (left-hand side, preconditions): match against change events.
- **RHS** (right-hand side, actions): mutate the graph.

Firing rules can themselves trigger more change events → more rule firings → indefinite forward chaining. The debounce parameter prevents infinite loops; an explicit *termination check* (seen rules N times) would be more robust.

This is the same architecture as Drools, JRules, CLIPS, and OPS5 — just embedded in Cascade rather than freestanding.

50.3 — Why monitors use Google News RSS

Free. No API key required. Comprehensive enough for most topics. Rate-limited but not throttled to zero. The tradeoff: less control over the source (you can't filter by source quality or recency window).

For paid alternatives, see `cascade/core/monitor.py`'s `search_google_news_rss` and `search_duckduckgo` functions — both swap-in points for NewsAPI/Newsdata.io/Bing News.

50.4 — Cost and latency comparison

Mechanism	Cost	Latency	Quality
Tool-use	$$$	seconds	high
Reactor	free	sub-second	medium
Monitor	$ (ingest)	hours	depends on ingestion
MCP	$$$ (external)	seconds	high

Cost-latency-quality is the **engineering trilemma** for agent systems. Cascade's four mechanisms each pick a different point on the triangle — there's no Pareto-dominant option; each is right for some task.

50.5 — Synchronous bus + async reactor

The bus emits synchronously *into the reactor*, but the reactor's `_handle_event` method just enqueues work — actual rule firing happens in a separate worker thread. The pattern: **the bus is the notifier; the reactor is the worker**. This decouples publication latency from work latency.

For very high-throughput ingestion, an intermediate queue (Redis, RabbitMQ) might replace the in-process bus. Cascade's design avoids that complexity for normal single-machine use.

50.6 — Why "five" is a good number

Cascade's architecture exhibits a useful pattern: each agent mechanism owns *one corner* of the async/sync × free/LLM matrix. Adding the curator fills the last corner. *More* than five would start to overlap responsibilities.

The deeper architectural lesson: **agent mechanisms aren't generic agents — they're specialized for cost/latency tradeoffs**. Right- sizing each to its slot makes the whole swarm maintainable.

Chapter 51 · Solutions

All exercises verified against a live Cascade install. Solutions 51.1–51.5 are runnable shell; 51.6 is open.

51.1 — Start reactor, count events

```
$ uv run cascade reactor start         # in terminal 1
$ uv run cascade reactor status --json # in terminal 2
{
  "running": true,
  "events_received": 0,
  ...
}

$ uv run cascade entity add STATE "Atlantis"
[done] Added entity 4001 STATE "Atlantis"

# ~500ms later (debounce window):
$ uv run cascade reactor status --json
{
  "running": true,
  "events_received": 1,    ← incremented
  "rules_fired": 0,        ← no rule matched ENTITY_CREATED
  "actions_executed": 0,
  "errors": 0,
  "pending": 0             ← debounce window elapsed; drained
}
```

Why `rules_fired = 0` even though `events_received = 1`: the shipped rules don't have preconditions matching ENTITY_CREATED for arbitrary STATE entities. The event was *seen* but *no rule wanted it*. That's expected — most events in a working system pass through without firing any rule.

51.2 — Cascade depth

After setting `max_cascade_depth = 5` and triggering a multi-hop ingestion:

```sql
SELECT id, change_type, item_type, created_at
FROM change_log
ORDER BY id DESC LIMIT 20;
```

You'd see chains like:

```
SOURCE_INGESTED        → source 401
ENTITY_CREATED × 30    → new entities from extraction
RELATION_CREATED × 23  → new relations
CLAIM_CREATED × 31     → new claims
(reactor fires)
CLAIM_UPDATED × 5      ← truth values flipped
RELATION_UPDATED × 2   ← derived from claim flips
(reactor fires again, depth 2)
CLAIM_UPDATED × 1      ← cascaded down
(depth 3 reached; further changes recorded but not re-evaluated)
```

The cascade saturates when **either** the depth limit hits **or** no rule's preconditions are satisfied by the latest changes. The depth bound guarantees termination even with mutually-modifying rules.

In a production install with rich rule sets, cascades typically stop at depth 2-3 — most rules produce one or two derivable consequences, not ten.

51.3 — change_log distribution

```
SELECT change_type, COUNT(*) AS n
FROM change_log GROUP BY change_type ORDER BY n DESC;
```

Sample live output:

```
change_type         n
------------------  ----
RELATION_CREATED    1820
ENTITY_CREATED      1156
CLAIM_CREATED       890
SOURCE_INGESTED     328
ENTITY_UPDATED      45
CLAIM_UPDATED       12
RELATION_UPDATED    5
```

Three patterns:

1. **RELATION_CREATED dominates** because each article extracts ~23 relations on average (Ch.45 §45.5). Multiply across 328 sources → ~7500, but Cascade's reactor only logged ~1800 — the rest predate the reactor or were ingested with reactor off.
2. **ENTITY_CREATED is second** (~30 per article).
3. **UPDATEs are rare** (~10x less than CREATEs). Entities and relations are mostly *append*; updates happen only when truth values flip or aliases get added.

The ratio CREATEs:UPDATEs ≈ 100:1 is a healthy sign — the graph mostly *accumulates* rather than *churns*. Heavy UPDATE ratios would suggest entity-resolution problems (Ch.45 §45.5).

51.4 — The cron entry

```
$ cascade monitor install-cron
# Cascade monitor cron entry:
```

```
0 */6 * * * cd /path/to/axiomacascade && \
    /home/user/.local/bin/uv run cascade monitor run --quiet
```

Reading the schedule: `0 */6 * * *` = run at minute 0 of every 6th hour, every day. So 00:00, 06:00, 12:00, 18:00 each day.

Without `--quiet`: stdout/stderr from each `monitor run` (search results, ingestion logs, LLM debug output) would go to the cron mailer and produce ~5-10 emails per day. The `--quiet` flag silences progress noise; errors still go to the Cascade logs at `data/logs/cascade.log`.

The cron interval (6 hours) is a *politeness default*. Production deployments often raise to 12 hours for less time-sensitive topics, or drop to 1 hour for breaking-news monitoring.

51.5 — Reactor backlog replay

```
$ uv run cascade reactor stop
[done] Reactor stopped

$ sqlite3 cascade.db \
    "SELECT COUNT(*) FROM change_log WHERE reactor_handled = 0;"
3467   ← existing backlog (likely from before reactor was ever started)

$ uv run cascade entity add STATE "TestState1"
$ uv run cascade entity add STATE "TestState2"

$ sqlite3 cascade.db \
    "SELECT COUNT(*) FROM change_log WHERE reactor_handled = 0;"
3469   ← +2 from the two ENTITY_CREATED events

$ uv run cascade reactor start    # foreground

# After ~500ms debounce + processing:
$ sqlite3 cascade.db \
    "SELECT COUNT(*) FROM change_log WHERE reactor_handled = 0;"
0     ← backlog cleared
```

The reactor's first batch on restart **drains the entire backlog**, processing all unhandled rows. For a 3500-row backlog, this can take ~30 seconds to a few minutes depending on how many rules match.

The recovery property: stopping the reactor *never loses events*. The change_log persists across restarts; the WAL pattern is what makes the reactor robust to crashes and intentional downtime.

51.6 — Sketch a RuleRateLimiter (open)

Sample answer:

```
from collections import defaultdict, deque
import time

class RuleRateLimiter:
    """Per-rule firing budget over a sliding window."""

    def __init__(self, budget=100, window_seconds=60):
        self.budget = budget
        self.window = window_seconds
        # rule_name -> deque of firing timestamps
        self._history: defaultdict[str, deque] = defaultdict(deque)

    def allow(self, rule_name: str) -> bool:
        """Return True if the rule is within budget;
        False if rate-limited."""
        now = time.monotonic()
        cutoff = now - self.window
        history = self._history[rule_name]
        # Prune old entries
        while history and history[0] < cutoff:
            history.popleft()
        if len(history) >= self.budget:
            return False    # over budget
        history.append(now)
        return True
```

Integration into the reactor:

```
def _process_batch(self, events):
    ...
    for rule in matching_rules:
        if not self.rate_limiter.allow(rule.name):
            log.warning("Rule %s rate-limited "
                        "(>%d/min); skipping",
                        rule.name, self.budget)
            continue
        rule.fire(...)
```

What happens when budget exceeded? Three options, each with a different cost-quality profile:

1. **Skip** (sample above) — events get processed without that rule firing. Cheap, but you lose the inference.
2. **Defer** — push the event back into `_pending` with a delay. Eventually catches up. More complex, preserves correctness.
3. **Alert + skip** — send a notification to the admin: "rule X has been rate-limited 50 times in the last minute." Skip the firing but flag the bug.

Recommended: option 3. Rate limiting that *silently drops* work is a footgun; rate limiting that *alerts* is a feature.

Notes

51.1 — Why most events fire no rule

Production rule systems typically have:

- 80% events that match no rule (most ingestion is uneventful).
- 15% events that match 1-2 rules.
- 5% events that match 3+ rules (the chained cascades).

The reactor is designed for this skew — the no-match path is the *fast path*. Rule *matching* is cheap (a few SQL queries); rule *firing* is expensive. The 80% no-match case returns in microseconds.

51.2 — Why depth 3 is enough

Empirically, multi-hop rule cascades in Cascade saturate at depth 2-3:

- Depth 1: ingestion fires "TRUTH_PROPAGATE"
- Depth 2: that fires "RELATION_RECOMPUTE"
- Depth 3: that fires "CONFIDENCE_DOWNGRADE"
- Depth 4+: rarely productive

Setting `max_cascade_depth = 5` lets the system explore deeper chains for *research* purposes but rarely produces useful new facts. The default of 3 captures the productive depth.

51.3 — INSERT-dominant graphs are healthy

A graph where INSERTs vastly outnumber UPDATEs is *additive* — new evidence accumulates; existing facts rarely flip. This is good.

The pathological alternative — UPDATE-dominant — would mean facts constantly flipping between TRUE/FALSE/BOTH as competing evidence lands. That signals either:

- Entity-resolution problems (different names for the same entity → flipping truth).
- Source quality issues (low-credibility sources contradicting high-credibility ones).
- Aggressive defeasible rules (Ch.49) that cascade across many claims.

Cascade installs in steady state have CREATE:UPDATE ≈ 50:1 to 200:1. Anything below 10:1 deserves investigation.

51.4 — Cron is a load-bearing dependency

Most Cascade production deployments rely on cron for monitor scheduling. The alternative — a Python scheduler inside the reactor — duplicates what cron already does well.

The principle: **OS-level scheduling for periodic work; in-process daemon for event-driven work**. Both are right; neither should replace the other.

51.5 — Restart safety is a feature, not luck

The `change_log` WAL pattern is *intentional*. Cascade chose append-only logging precisely to make crash recovery boring. The alternative (in-memory event buffer with periodic flush) would lose events on crash — unacceptable for a knowledge-graph system where every event encodes new information.

51.6 — Rate limiting as a quality gate

The deeper lesson: every rule firing has a cost (CPU, DB writes, LLM if the action does inference). Without rate limits, a *bug* in a rule (over-eager preconditions) can melt the system. Rate limits are the **safety mechanism** that protects the reactor from its own rules.

This is the same pattern as web-server request limits, API quotas, and Kafka consumer rate caps. The general principle: **any system with feedback loops needs a governor**.

Chapter 52 · Solutions

This chapter is more *design-and-discuss* than *code-and-run*, since operating-Cascade decisions are deployment-specific. Solutions 52.1–52.3 have verifiable shell outputs; 52.4–52.6 are open-ended with sample answers.

52.1 — Inventory your install

Live output from the test install:

```
Graph stats:
  entities: 3956, relations: 3810, claims: 8416,
  sources: 329, rules: 85

Reactor metrics:
  running: false, events: 0, rules_fired: 0

Sync metrics:
  Supabase: not configured

Rule files: 4 in .cascade/rules/axiomalang/
Logs: data/logs/cascade.log
Config: ~/.cascade/cascade.toml
```

A one-page deployment summary for this install would say:

> Local single-machine install. ~4000 entities, ~3800 relations, ~8400 claims accumulated from 329 articles. 85 rules registered across 5 engines (python/prolog/axioma/cascade/axiomalang). Reactor *not currently running*; sync *not configured*. Test/development install — not production-grade.

For a production install, the same query would return a *very* different summary: reactor running, sync active, dozens of rules firing per hour, daily briefing pipeline producing output.

The inventory query is your *baseline diagnostic*. Run it weekly; track the deltas. Big jumps in `sync_outbox` row count, or `errors` in reactor status, are early warning signs.

52.2 — Try an export

The seven formats serve different audiences:

Format	Tool to open	Audience
HTML	Browser	Casual readers, email recipients
GraphML	Gephi (free)	Academic visualization
GEXF	Gephi (native)	Same as GraphML, more features
DOT	Graphviz `dot` CLI	Programmatic typesetting
Cypher	Neo4j Desktop	Graph-database imports
Tree	Plain text terminal	Briefing inserts
Chart	Image viewer	Reports, slides

What each *shows* that the CLI doesn't:

- HTML lets you *interact* — drag nodes, zoom, click for details. The dashboard does this live; HTML is the *exportable* version.
- GraphML opens in Gephi, which has built-in community detection (Louvain), centrality metrics, and aesthetic layout algorithms (ForceAtlas2). Better visuals than Cascade's built-in renderer.
- Cypher imports the graph into Neo4j, where you get full Cypher query language — much richer than Cascade's SQL+NetworkX combination for certain queries.
- Tree is a one-entity vertical layout — great as a screenshot in a briefing.

The lesson: **export per use case, not per preference**. No format is "best"; each optimizes for a different audience.

52.3 — Run the briefing

```
$ uv run cascade briefing --output today.md
$ head -50 today.md

# Cascade Daily Digest — 2026-04-15

## Headlines
- 12 new entities ingested overnight
- 8 new high-confidence relations
- 3 claims flipped TRUE → BOTH
- Top cascade impact: Iran → crude oil → ...

## New Developments
...

## What to Watch (LLM synthesis)
The convergence of [X] and [Y] suggests...
...
```

Three questions answered:

1. Is the data-only summary useful as-is? Mostly yes for daily skim — counts, top entities, truth-flip events. The headlines + new developments sections are *what changed since yesterday* — directly useful

for an analyst's morning review.

2. Does the LLM section add insight or just paraphrase? Mixed. On busy news days it genuinely synthesizes patterns the analyst would have to read all 12 entities to spot. On quiet days it pads with generic statements ("continue to monitor..."). The cost-benefit is worth it for weekly publication; daily can feel padded.

3. Where would you edit before publishing?

- Strip the obvious filler ("continue to monitor").
- Add 2-3 sentences of *editorial framing* that the LLM can't generate (knowing your audience's bias).
- Verify each cited claim still holds (a truth-flip between briefing-generation and publication is rare but embarrassing).

The briefing is a *first draft*, not a final product. The pipeline's value is reducing daily writing from 1-2 hours to 15 minutes of editing — not eliminating the analyst.

52.4 — Sketch a 3-person team deployment

Sample answer:

Mode: Networked single-tenant (§52.1 middle tier).

Hardware: One EC2 t3.medium instance (2 vCPU, 4 GB RAM, ~$30/mo) running Cascade in a Docker container. Sufficient for ~50 articles/day plus the dashboard.

Daemons (all on the EC2):

- Reactor under systemd (`Restart=on-failure`).
- Sync daemon under systemd (push to Supabase).
- Cron entry for monitor runs (every 6 hours).
- Dashboard via nginx reverse proxy on HTTPS port 443.

Team auth: Dashboard set-password for each analyst (3 passwords). For higher-security needs, add an OIDC layer (Okta or Google) in front via nginx.

Daily briefing: Cron entry at 06:00 UTC to run `cascade briefing daily --output briefings/today.md`, then a Python script emails the file to the team's distribution list. ~$0.30/day in LLM cost ≈ $9/mo.

Monthly cost estimate:

- EC2: $30
- Supabase Pro tier: $25
- LLM (Anthropic + OpenAI mix): $50-100 (mostly ingestion + briefing)
- Domain + TLS: $1
- **Total: $106-156/month** for 3 analysts

Comparable products (managed intelligence dashboards) charge $500-2000/month per analyst — so this is 10-100× cheaper at the cost of running your own infra.

52.5 — Disaster recovery drill

Sample walkthrough for a corrupted `cascade.db`:

Source of truth depends on what was running:

- **With Supabase sync active**: pull from Supabase. `cascade sync pull --all` reconstructs the entire `cascade.db` from the Postgres mirror. **Recovery time** for 10k entities: ~10 minutes.
- **Without sync**: re-ingest from `sources` table (if recoverable) or from the URLs stored in `~/.cascade/ingested_urls.log`. **Recovery time** for 10k entities: ~1-2 hours and ~$10-50 in LLM cost.

What's lost forever vs recoverable:

What	Recoverable?
Entities, relations, claims	Yes (via re-ingest or Supabase)
Truth-flip history	Yes if Supabase synced; **lost** otherwise
Manual analyst edits	**Lost** unless the change_log was synced
Pinned items	Yes if synced; **lost** otherwise
Reactor's rule-firing log	Acceptable to lose

The lesson: **Supabase sync is your disaster-recovery insurance**. The marginal cost ($25/month for Supabase Pro) buys you full recoverability. Skipping it means each local corruption is a fresh re-ingestion run.

Procedure for production recovery:

1. Stop reactor + sync daemons.
2. Move corrupted `cascade.db` aside.
3. Initialize a fresh database.
4. Run `cascade sync pull --all`.
5. Restart daemons.
6. Verify with `cascade stats` against the pre-corruption baseline.

Total downtime: ~15-30 minutes if Supabase synced.

52.6 — Missing from the punch list (open)

Sample answer — **Backup-and-restore drill**:

The §52.9 punch list covers operational practices but says nothing about *practicing recovery*. A monthly drill:

1. Take a snapshot of `cascade.db` and Supabase to a separate location.
2. In a dedicated test environment, simulate a corruption (drop `entities` table).
3. Execute the recovery procedure from §52.5.
4. Verify the recovered DB matches the pre-corruption snapshot via `cascade stats` and spot-checks on key entities.
5. Document any deviations.

Why it matters: untested backups are *hypothetical* backups. Without a drill, the first time you find out your recovery procedure is broken is during an actual disaster.

The cost is one analyst hour per month. The benefit is *confidence* that your recovery works. For any system with operational criticality (publication deadlines, customer SLAs), this is mandatory; for hobby deployments, optional.

Other gaps worth considering:

- **Incident response runbook**: who gets paged when the reactor crashes at 3am?
- **Cost-spike alerting**: a CloudWatch/ PagerDuty alert when LLM spend exceeds daily threshold by 2×.
- **Compliance audit log**: who edited which claim and when (for journalism integrity / legal discovery).
- **Quality regression suite**: rerun the same evaluation queries weekly and alert on results that drift.

The general principle: **operational practices compound**. Each one prevents a class of failure mode. Together they're what makes "production" different from "running on a laptop."

Notes

52.1 — The inventory as routine

Production deployments often automate the inventory: a daily cron job emails the team the previous day's stats. Drift detection then catches *anomalies* — sudden drops in ingestion rate, sudden spikes in `sync_outbox` size, sudden errors in reactor.

52.2 — Multi-format export is a strength

Most knowledge-graph tools lock you into their own visualization. Cascade's commitment to *portable* exports (GraphML, GEXF, Cypher, DOT) means you can take your KB anywhere. This matters for academic citation and for escape-hatch portability ("if I stop using Cascade tomorrow, I can still use my data").

52.3 — Briefings as workflow, not output

The deeper lesson: the briefing pipeline isn't about *producing* prose; it's about *focusing attention*. An analyst reading 50 raw events from `change_log` would miss the patterns; the briefing reduces the cognitive load to "what changed; what matters; what to watch."

This is the **synthesis layer** of operational intelligence work. Cascade automates the mechanical parts (counting, ranking, citing); the analyst's value-add is the narrative overlay.

52.4 — The $100/month threshold

A 3-analyst team for $100-150/mo is *the breakeven point* against managed services. Any team smaller (1-2 people) probably doesn't need this infrastructure; any team larger (5+) is probably hitting the Cascade limits documented in §52.10 and should consider sharding or upgrading to enterprise tier.

52.5 — Sync is insurance, not luxury

The temptation is to skip Supabase sync for a "simple" deployment. The cost-benefit:

- **Skip sync**: save $25/month + complexity. Risk: one corruption = one full re-ingest (~$50 in LLM costs + 2 hours of work).
- **Use sync**: spend $25/month. Risk: a few hours of work on initial setup.

The expected-loss calculation: if you expect *one* corruption over a year, you save money *and* time with sync. The lower bound is breakeven; the upper bound is "you'll thank yourself" — production deployments *overwhelmingly* recommend sync.

52.6 — Operational maturity

The progression of operational maturity:

1. **Level 0**: It runs on my laptop.
2. **Level 1**: It runs on a server.
3. **Level 2**: It runs on a server with monitoring.
4. **Level 3**: It runs reliably; failures trigger alerts; recovery is automated.
5. **Level 4**: It runs at scale; teams can share it; multi-tenant isolation works.

Cascade today supports Level 2-3 deployments out of the box; reaching Level 4 requires the multi-tenant work Cascade itself is still maturing on. Most readers will deploy at Level 2-3; that's plenty for serious work.

Chapter 54 · Solutions

Exercise 54.1 — id at two types

```
#language axioma/hm
id: func(x) [x]
expect("id int", id(1), 1)
expect("id bool", id(true), true)
```

```
id :: a -> a
```

Commentary. `id` does not inspect `x`, so nothing pins `a`. Each `expect` is a separate instantiation. That is let-polymorphism in the smallest possible file. If you wrote two functions `id_int` and `id_bool`, you would have missed the point: one generalized binding, two uses.

`println` would refuse (`println has no known type here`). `expect` is in the catalog and types as `(String, a, a) -> ()` — the actual and expected values must agree with each other, which is why the assertion itself is typed.

Exercise 54.2 — greet with str_concat

```
#language axioma/hm
greet: func(name) [str_concat("hello, ", name)]
expect("greet", greet("Ada"), "hello, Ada")
```

```
greet :: String -> String
```

Commentary. Host Axioma would have accepted `"hello, " + name`. The island types `+` as numeric, so that spelling is a rejection (`string and array concatenation use str_concat and array_concat`). The catalog function both concatenates and *tells the type* that both sides are `String`, which is how `name` becomes `String` without an annotation.

Exercise 54.3 — integer pow

```
#language axioma/hm
func pow(b, 0) [1]
func pow(b, n) [b * pow(b, n - 1)]
expect("pow 2 10", pow(2, 10), 1024)
expect("pow 3 4", pow(3, 4), 81)
```

```
pow :: (Integer, Integer) -> Integer
```

Commentary. A multi-clause group is one function. The literal `0` grounds the second parameter at `Integer`; `*` and `-` ground the first. Recursion is monomorphic: every call to `pow` is at that one pair of types. Compare `fib` in §54.6 of the chapter — same pattern, one argument. The unguarded final clause establishes coverage; only nonnegative exponents are intended to terminate. A guarded recursive clause alone would leave a possible `none` miss.

`7 / 2` is still illegal here. Integer shrinking uses `n - 1` and `div` is not required for this body.

Exercise 54.4 — filter then map

```
#language axioma/hm
nums: [-(2), -(1), 0, 1, 2, 3]
positives: filter(func(n) [n > 0], nums)
doubled: map(func(n) [n * 2], positives)
expect("doubled positives", doubled, [2, 4, 6])
```

Commentary. `-(2)` is unary minus (Chapter 1); writing `-2` inside an array is easy to mis-read against subtraction. `filter` keeps an `Array of Integer`; `map` sends it to another. Neither function is recursive in *your* code — the catalog already recurses. That is the Chapter 10 move, now inside a checker that will refuse a heterogeneous `nums`.

Exercise 54.5 — legal twins

```
#language axioma/hm
expect("concat", str_concat("a", "b"), "ab")
expect("div", 7 div 2, 3)
expect("if", if true then 2 else 3, 2)
x: 1
y: 2
expect("two names", x + y, 3)
```

Commentary. Four host habits, four island spellings:

- `+` on strings → `str_concat`

- `/` on integers → `div` (the result stays `Integer`)
- truthiness of `1` → a real `Boolean`
- find-or-update of `x` → a new name `y`

The last is the Chapter 2/16 convention inverted. On the host, `x: 2` after `x: 1` is an update. On the island it is `re-binding x`, and `x = 2` is `reassignment`. Both are refused. Two names is not a workaround for mutation; it is the program you actually meant (two values).

Exercise 54.6 (open) — a closed kernel

One answer, `compose` at two types:

```
#language axioma/hm
compose: func(f, g) [func(x) [f(g(x))]]
inc: func(n) [n + 1]
double: func(n) [n * 2]
notb: func(b) [not b]
h: compose(double, inc)
expect("compose num", h(3), 8)
expect("compose bool", compose(notb, notb)(true), true)
```

```
compose :: (a -> b, c -> a) -> c -> b
inc     :: Integer -> Integer
double  :: Integer -> Integer
notb    :: Boolean -> Boolean
```

Another, `product` via `foldl`:

```
#language axioma/hm
product: func(xs) [foldl(func(acc, n) [acc * n], 1, xs)]
expect("product empty", product([]), 1)
expect("product", product([2, 3, 5]), 30)
```

```
product :: Array[Integer] -> Integer
```

What the type made impossible. On the host, `product(["a", "b"])` might concatenate-repeat or error at run time, depending on how you wrote `*`. On the island the `foldl` body pins `n` to `Integer`, so a string array is a type error *before* any element is visited. That is the whole reason to step onto the island: the bad call does not become a skip, and it does not become a value.

Exercise 54.7 — shared and independent variables

```
#language axioma/hm
second_same[T] :: (T, T) -> T
second_same(t) = t[2]
second_any[A, B] :: (A, B) -> B
second_any(t) = t[2]
expect("same", second_same((1, 2)), 2)
expect("independent may coincide", second_any((1, 2)), 2)
expect("independent may differ", second_any((1, "two")), "two")
```

`A` and `B` are independent choices, not a type-inequality assertion. Returning `t[1]` from `second_any` would produce arbitrary `A` where arbitrary `B` is promised. An integer-only test would miss this error; HM body checking rejects it before execution. Returning the first component from `second_same` would still satisfy its type, although it would violate the intended behavior of a function named “second”: types do not specify every property of a program.

Exercise 54.8 — a typed missing case

```
#language axioma/hm
data Choice[T] = Chosen(T) | Missing
choose[T] :: Choice of T, T -> T
choose(value, fallback) = match value with | Chosen(x) => x | Missing => fallback
expect("chosen", choose(Chosen(7), 0), 7)
expect("missing", choose(Missing, "fallback"), "fallback")
```

The matched field and fallback both have the same arbitrary T. A constant Integer cannot satisfy that promise for String callers. Removing Missing loses coverage: an ordinary match may then produce `none`, which the island excludes. A wildcard fallback also covers the missing case; a guard alone does not.

Exercise 54.9 — keep the tail type

```
#language axioma/hm
drop_one[T] :: List of T -> List of T
drop_one([]) = (`[])
drop_one([head | tail]) = tail
rest_array[T] :: List of T -> Array of T
rest_array([]) = ([])
rest_array([head, ..tail]) = tail
expect("empty tail", drop_one(`[]), `[])
expect("List tail", drop_one(`[1,2,3]), `[2,3])
expect("Array rest", rest_array(`[1,2,3]), [2,3])
```

A List cons tail preserves its List type. Dot-rest returns an Array and therefore requires the second

signature and an empty Array result in its base clause. Both pairs cover length zero and every positive length. Removing the empty clause makes an ordinary clause group potentially return none, so strict checking refuses it even if all written tests call nonempty Lists.

Exercise 54.10 — an opaque generic slot

```
#language axioma/hm
module Vault [
  data Slot[T] = Stored(T)
  put[T] :: T -> Slot of T
  put(value) = Stored(value)
  get[T] :: Slot of T -> T
  get(Stored(value)) = value
  export Slot, put, get
]
expect("Integer slot", Vault.get(Vault.put(3)), 3)
expect("Boolean slot", Vault.get(Vault.put(false)), false)
```

The exported functions preserve the parameter relationship. Stored is absent from both the module and its exported type's constructor properties, so a client uses get. Exporting `Slot/transparent` would deliberately expose Stored.

Exercise 54.11 — one promise, two implementations

```
#language axioma/hm
signature VaultAPI [
  data Slot[T]
  put[T] :: T -> Slot of T
  get[T] :: Slot of T -> T
]
Vault :: VaultAPI
module Vault [
  data Slot[T] = Stored(T)
  put(value) = Stored(value)
  get(Stored(value)) = value
]
module StampedVault :: VaultAPI [
  data Slot[T] = Packed(T, Integer)
  put(value) = Packed(value, 0)
  get(Packed(value, stamp)) = value
]
expect("first implementation", Vault.get(Vault.put(42)), 42)
expect("second implementation", StampedVault.get(StampedVault.put(true)), true)
```

The contract supplies both function arrows and the exact opaque export surface. Each module's Slot has its own static identity, so mixing their values fails before execution. A constant `0` cannot satisfy get's

arbitrary T result. Ascription checks the implementation body; passing one Integer example alone would not establish that generic promise.

Exercise 54.12 — expose an equality, keep an abstraction

```
#language axioma/hm
signature StoreAPI [
  type Key = Integer
  data Slot[T]
  put[T] :: Key, T -> Slot of T
  get[T] :: Slot of T -> T
]
module Store :: StoreAPI [
  type Key = Integer
  data Slot[T] = Stored(Key, T)
  put(key, value) = Stored(key, value)
  get(Stored(key, value)) = value
]
key :: Store.Key: 7
expect("Integer key, Boolean value", Store.get(Store.put(key, true)), true)
expect("independent payload", Store.get(Store.put(8, "hello")), "hello")
```

Every implementation's Key is Integer by the published equality. Slot remains nominal and fresh for each ascription, so agreeing about keys does not identify the payload containers. A Float equation fails exact type equality; numeric promotion at function return boundaries does not establish a type equation. Missing required type bindings also fail before execution. The equation does not export Stored or grant an alias constructor.

Exercise 54.13 — publish an initial state

```
#language axioma/hm
signature CounterAPI [
  data Count
  zero :: Count
  increment :: Count -> Count
  read :: Count -> Integer
]
module Counter :: CounterAPI [
  data Count = Counted(Integer)
  zero: Counted(0)
  increment(Counted(n)) = Counted(n + 1)
  read(Counted(n)) = n
]
expect("initial state", Counter.read(Counter.zero), 0)
expect("next state", Counter.read(Counter.increment(Counter.zero)), 1)
```

Count has a fresh abstract identity for this ascription, so another implementation's read operation cannot consume Counter.zero. A Boolean initializer disagrees with Count before execution. A shared empty Array starts with one unknown element type; sealing its first fixed collection requirement constrains that type for every alias. A later incompatible requirement fails instead of assigning a second type to the same shared value.

54.14 Constructors and mutual recursion

```
#language axioma/hm
even(n) = if n == 0 then true else odd(n-1)
odd(n) = if n == 0 then false else even(n-1)
data Flag[T] = Flagged(T)
expect("parity flags", map(Flagged,[even(10),even(11)]),
 ↪  [Flagged(true),Flagged(false)])
```

The assertion passes. A call between the definitions is an executable statement, so it ends the block; the checker does not move definitions across that boundary.

54.15 Report every failed expression

Run from the repository root:

```
#language axioma/hm
import "lib/hm/Result.ax" as R
import "demos/functional_expression/Parser.ax" as P
import "demos/functional_expression/Interpreter.ax" as I
evaluate_tokens(tokens) = R.and_then(func(expr) [I.evaluate(expr,`[])],
  R.map_err(P.describe_problem,P.parse(tokens)))
batch: `[`[P.TokInt(2)],`[P.TokName("x")],`[P.Open]]
expect("batch first error", R.traverse(evaluate_tokens,batch), R.Err("unbound variable:
 ↪  x"))
expect("batch every error", R.validate_all(evaluate_tokens,batch), R.Err(`["unbound
 ↪  variable: x","token 1: expected an expression"]))
count_results(xs :: List of Integer) = len(xs)
valid_batch: `[`[P.TokInt(2)],`[P.TokInt(3)]]
expect("count typed batch results",
  R.map_ok(count_results,R.traverse(evaluate_tokens,valid_batch)), R.Ok(2))
```

All three assertions pass. The callback returns Err values for parse and lookup failures, which validation can accumulate. A runtime error in the callback would stop evaluation instead; the library does not turn arbitrary errors into Results. The successful traversal yields an Integer List that satisfies `count_results`.

Appendix D · Where to Read Next

The book ends, but the field is enormous. This appendix points at the most worthwhile next reads, organized by what they extend.

The original — and the tradition that produced this port

Felleisen, Findler, Flatt, Krishnamurthi — *How to Design Programs* (2nd ed., MIT Press, 2018). Free online at https://htdp.org/. The book this port follows. Reading the original after this one is the natural next step: you'll see the *same* design recipe applied to *the same kinds of problems*, in a different surface language (Racket / BSL). The contrast is itself an education in what's accidental to a language and what's essential.

Abelson and Sussman — *Structure and Interpretation of Computer Programs* (2nd ed., MIT Press, 1996). Free online at https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/index.html. "SICP" is the deeper, harder cousin of HtDP. Same family (Scheme), same recipe-style thinking, but pushed all the way to a *full* meta-circular evaluator, streams, register machines, and compilers. Read it after this book if Ch.22 left you wanting more.

Friedman and Felleisen — *The Little Schemer* (4th ed., MIT Press, 1995). A small book that teaches structural recursion through a Socratic dialog. Companions: *The Seasoned Schemer* and *The Reasoned Schemer* (the latter introduces logic programming the same way Ch.21 does — facts and rules first, syntax second).

The Axioma reference

The Axioma Manual — `resources/manual/Axioma Manual.md` in this repository, also exported to HTML and PDF. The exhaustive reference for every language construct, including ones this textbook deliberately skipped:

- Conceptual graphs (CG) and Sowa's notation
- Modal logic (necessarily / possibly, S4 / S5)
- The Cascade knowledge-base integration
- Stack-based programming (Forth/Pop-11-style)
- The full MCP server interface

Treat the manual as a *lookup* — you don't read it end-to-end, you flip to it when you've forgotten exactly how some form works.

Formal foundations

Pierce — *Types and Programming Languages* (MIT Press, 2002). The standard graduate-level introduction to type systems. If you found yourself wishing the type system in Ch.19 were more powerful, this is the book that explains how to build one. Covers simply-typed lambda calculus, subtyping, parametric polymorphism, recursive types, and dependent types. Difficult but worth it.

Sipser — *Introduction to the Theory of Computation* (3rd ed., Cengage, 2012). Automata, formal languages, Turing machines, computability, NP-completeness. Pairs naturally with Ch.22 — once you've built an interpreter, you start wondering *what's the upper limit of what an interpreter can do*, and this book answers that.

Mitchell — *Concepts in Programming Languages* (Cambridge, 2002). A survey of language design decisions: type-checking strategies, scoping rules, evaluation orders, object models. After this book, you'll understand *why* Axioma made each choice it did; Mitchell shows you the landscape of choices that were available.

Logic and reasoning

Bratko — *Prolog Programming for Artificial Intelligence* (4th ed., Pearson, 2011). The classical Prolog textbook. Chapter 21 of this book ports a slice of Prolog's capabilities; Bratko shows you the rest — pattern matching with backtracking, custom unification, constraint logic programming, expert systems.

Antoniou and van Harmelen — *A Semantic Web Primer* (2nd ed., MIT Press, 2008). Where logic programming meets the real-world web. RDF, OWL, SPARQL, and the rule-based reasoners that power knowledge graphs at scale.

Belnap — *"A Useful Four-Valued Logic"* (in *Modern Uses of Multiple-Valued Logic*, Reidel, 1977). The original paper introducing B4. Short and readable; the foundation of Ch.20's paraconsistent material.

Łukasiewicz — *Selected Works* (North-Holland, 1970). The collected papers introducing many-valued logic, including L3. Of historical interest; the Wikipedia article on many-valued logic is a faster way in.

Interpreters, compilers, and language implementation

Nystrom — *Crafting Interpreters* (self-published, 2021). Free online at https://craftinginterpreters.com/. The modern successor to SICP's evaluator chapter. Builds two full interpreters (one tree-walking, one bytecode) for a toy language. Excellent companion to Ch.22 — same architecture, two orders of magnitude more code.

Appel — *Modern Compiler Implementation in ML* (Cambridge, 1998). If "tree-walking interpreter" left you wanting to build a *compiler*, Appel is the bridge. Lexer, parser, type-checker, intermediate representations, register allocation, code generation. Available in ML, Java, and C flavors.

Cooper and Torczon — *Engineering a Compiler* (3rd ed., Morgan Kaufmann, 2022). A more practical, more recent take on compiler engineering than Appel. Less theory-heavy, more focused on the choices that actually matter in production compilers.

Adjacent: math and philosophy of programming

Polya — *How to Solve It* (Princeton, 1945). The patriarch of all problem-solving books. The Design Recipe is partly Polya's "understand the problem → devise a plan → carry out the plan → look back" applied to programming. Twenty pages of Polya rewards a lifetime of programming.

Knuth — *The Art of Computer Programming* (multiple volumes, Addison-Wesley, 1968–present). The encyclopedia. You don't read TAOCP; you *consult* TAOCP when you need to understand an algorithm in real depth. Pairs well with Chapter 14 (generative recursion) and Chapter 15 (accumulators).

Hofstadter — *Gödel, Escher, Bach* (Basic Books, 1979). Not a programming book, but a book about the structures *underneath* programming — self-reference, recursion, formal systems, meaning emerging from rules. If Chapter 22's meta-circular evaluator delighted you, GEB is the next delight.

Where to look online

The Racket forum — https://forum.racket-lang.org/ — remarkable community discussion of HtDP and the Lisp/Scheme tradition more broadly. Even when the answer involves a Racket form Axioma doesn't have, the *thinking* transfers.

lambda-the-ultimate.org — long-running blog and forum covering programming-language research. Slower-paced but deeper than most language communities.

Programming Language Theory wikis — the Wikipedia articles for *operational semantics*, *denotational semantics*, *lambda calculus*, *Hindley–Milner type inference*, and *Curry–Howard correspondence* are unusually well-written and a fine free introduction to each topic.

How to pick

Don't read all of these. Pick **one** based on what you found most interesting in this book:

If Chapter NN delighted you…	…go to
14 (generative recursion)	TAOCP, Cormen *Introduction to Algorithms*
16 (mutation, state machines)	*Concurrency: State Models and Java Programs*, Magee & Kramer
17 (streams)	SICP Chapter 3, *Why Functional Programming Matters*, Hughes 1989
19 (enums, subranges)	Pierce TAPL, Cardelli's *Type Systems* survey
54 (closed Hindley–Milner island)	Pierce TAPL ch. 22 (type reconstruction); the Wikipedia article *Hindley–Milner type inference*; Damas & Milner 1982
20 (multi-valued logic)	Belnap, *Stanford Encyclopedia of Philosophy* article on "Many-Valued Logic"
21 (logic programming)	Bratko, *Concepts, Techniques, and Models of Computer Programming* (CTM)
22 (meta-circular evaluator)	Nystrom *Crafting Interpreters*, SICP §4

Most languages, most ideas, most paradigms — they're all *variations on the small set of moves you now know*. Go explore.

> *"You can know the name of a bird in all the languages of the world, but when you're finished, you'll know absolutely nothing whatever about the bird... So let's look at the bird and see what it's doing — that's what counts."* — Richard Feynman.

Appendix E · Introspection and Debugging

This appendix is the *lookup card* for Axioma's introspection and debugging machinery — the tools that let you peer inside a running program. Most of these were skipped in the main chapters to keep the pedagogical line clean; they live here so you know they exist when you need them.

The book covers many of these in passing — this appendix pulls them together in one place.

E.1 Seeing values

Form	What it does	Where it's at home
`println(x)`	Print `x` + newline	Scripts; clear and explicit
`println(x, y, z)`	Print multiple values space-separated	Multi-arg output
`print(x)`	Print without newline	Building lines incrementally
`expr .`	**Trailing-dot inspect sigil** — evaluate and print	Scripts, blocks, anywhere
`inspect(x)`	Print `name = value` form	Variable inspection

The REPL prints results automatically; scripts don't. Use `println` for explicit, named output, and the trailing `.` for quick "show me this" probes — particularly useful when you want to peek at intermediate values without rewriting a line.

The trailing dot as a tracing tool

The trailing `.` works at the top level *and* inside any `[...]` block — while-loop bodies, if-then/else branches, function bodies, lambda bodies. That makes it the most *lightweight* tracing tool Axioma has:

```
total: 0
i: 1
while (i <= 5) [
  i .                     # show me i each iteration
  total = total + i .     # show me total each iteration
  i = i + 1
]
total .                   # show me the final total
```

This prints `1, 1, 2, 3, 3, 6, 4, 10, 5, 15, 15` — every intermediate value, no `println(...)` calls cluttering the code. Delete the dots when you're done debugging, and the program runs silently again.

For deeper tracing — full call/return logging across a recursive function — see the `trace` statement below.

E.2 Knowing what something is

Form	Returns
`type(x)`	The runtime type Concept, such as `Integer`, `String`, or `Array`; compare with `type(x) == Integer`, not a String
`@x`	Same as `type(x)` — the type-of sigil, the type of the *value*
`::name`	The type the *name* was declared with; falls back to `@name` without an annotation
`declared(name)`	Strict form of `::name` — a bare name, never evaluated; `none` without an annotation
`x is Concept`	Whether `x` is an instance of `Concept` (also walks `is` chain)
`x is Type`	Type predicate (alternative form)

Examples:

```
type(42)              # Integer
type([1, 2, 3])       # Array
type({1, 2, 3})       # Set
type("hi")            # String
type(func(x) [x*x])   # Function
```

Type-of is essential for *defensive functions* that handle multiple input types, and for debugging when a value isn't what you expected.

Value type vs declared type — `@x` and `::x`

`type(x)` and `@x` answer "what kind of *value* is this?" `::x` asks a different question: "what was this *name* declared as?" The two agree until a declaration is narrower or broader than the value it holds — a union, a literal type, or `Number`:

```
x :: Number: 5
expect("at sigil reads the value", @x, Integer)
expect("at agrees with type()", @x, type(x))
expect("double-colon reads the annotation", ::x, Number)
y :: Integer | String: 5
expect("value type under a union", @y, Integer)
expect("annotation is the union", string(::y), "Integer | String")
```

Without an annotation, `::name` falls back to the value's type, and so does any operand that isn't a bare name. `declared(name)` is the strict form: it never evaluates its argument and answers `none` when nothing was declared:

```
n: 5
expect("unannotated falls back to value", ::n, @n)
expect("strict form is none", declared(n), none)
expect("non-name operand falls back", ::5, Integer)
f: func(v :: Integer | String) [string(::v)]
expect("parameter annotation", f(7), "Integer | String")
```

The full five-question table — value, declaration, arrow, inference, and signature — lives in Manual § *Type-of — `:: expr` — and `annotations(f)`.

E.3 Asking the language itself — the self-describing runtime

Since July 2026 the reference lives *in the runtime*: the language answers its own reference questions, so "check the manual" is the second resort, not the first. One row per question:

Question	Ask	Notes
What is this word or type?	`doc Integer`, `doc(word)`, `doc("if")`	full type cards for the 12 core types; a real statement, script-legal
What is this **value**?	`describe(x)`	per-type fact card (parity/digits, bytes-vs-runes, element types, …) + pointers onward
How can a program inspect a value and its binding?	`reflect(x)`	read-only descriptor; optional lesson E.23 distinguishes names, values, and snapshots
Where is anything about …?	`apropos("term")`	searches names AND documentation text
What exists at all?	`builtins()`, `concepts()`, `bindings()`, `keywords()`	sorted catalogs; 1-arg forms are membership checks (`builtins("map")`, `concepts("Integer")`, `bindings("x")`, `keywords("if")`)
What can I do with a type?	`functions(Integer)` ≡ `methods(Integer)`	curated, verified per-type builtin catalog
How do I call this?	`signature(f)`, `arity(f)`, `parameters(f)`	spec-backed for builtins, user functions too
What is this function's definition?	`source(f)`	reconstructed source String — `eval(source(f))` round-trips
What does this source parse to?	`ast("src")`	parse a runtime string *without* running it

Question	Ask	Notes
Does this compile to the VM?	`compile(src)`	callable on success; catchable Error names the boundary

A ninety-second tour (every line runs):

```
describe(42)                # Value/Type/Parity/Sign/Digits + pointers
signature(round)            # "round(x, [digits])"
first(functions(Integer), 5)   # ["abs", "add1", "bin", "bit_not", "choose"]

double: func(x) [x * 2]
source(double)              # "double: func(x) [(x * 2)]"
eval(source(double))(21)    # 42 — the definition round-trips
concepts("Integer")         # true
bindings("double")          # true

ast_string(ast("a + b * c"))   # "(a + (b * c))" — parsed, not run
apropos("shuffle")          # every name + doc line mentioning it
```

The split to remember: **`doc` documents names, `describe` inspects values, `source` returns definitions, `ast` parses, `compile` compiles.** The `'(…)` quote (next section) is the *lexical* spelling of code-as-data; `ast()` is its runtime twin.

E.4 Inspecting AST values

Quote/parse give you the AST as data. To *see* what's inside, use the rendering family:

Form	Output
`fullform(ast)`	Mathematica-style prefix form: `If(>(x, 0), *(x, 2), -(x))`
`treeform(ast)`	Vertical ASCII tree (default)
`treeform(ast, "horizontal")`	Horizontal ASCII tree
`headof(ast)`	The operator/keyword at the root: `"+"`, `"If"`, `"sqrt"`
`argsof(ast)`	Array of immediate children as strings
`hold(expr)`	Capture an AST without evaluating it (like `quote`)
`ast_type(ast)`	Node-kind label: `"InfixExpression"`, `"IfExpression"`
`ast_string(ast)`	Source-style rendering (close to how you wrote it)
`ast_eval(ast)`	Evaluate the AST and return the value

Example session:

```
a: parse("if x > 0 then x * 2 else -x")

fullform(a)
# If(>(x, 0), *(x, 2), -(x))

treeform(a)
# If
#     ├── >
#     │   ├── x
#     │   └── 0
#     ├── *
#     │   ├── x
#     │   └── 2
#     └── -
#         └── x

headof(a)              # "If"
argsof(a)              # [">(x, 0)", "*(x, 2)", "-(x)"]
ast_type(a)            # "IfExpression"
```

`fullform` and `treeform` are the *two views* of the same tree — pick whichever reads better for the expression you're looking at. `fullform` is compact (good for short ASTs and for comparing structure: `fullform(2 + 3) == fullform(+(2, 3))`). `treeform` shows the recursive shape clearly (good for deep nested expressions where the prefix form gets line- wrapped). They're both implemented as one-pass walks over the AST.

Building, matching, and rewriting code

`fullform` / `headof` / `argsof` *read* an AST. A second family **rebuilds and rewrites** it — using the same uniform `head[args]` shape that Mathematica is built on.

Form	What it does
`ast_head(ast)`	operator/functor as a String — uniform across infix/prefix/postfix/call/statement
`ast_operands(ast)`	Array of AST — the arguments, functor excluded (like `argsof`, but AST not strings)
`ast_make(h, ops)`	rebuild a node from a head + operands Array — the inverse
`ast_match(pat, subj)`	structural match of a `?x`-pattern → bindings Dictionary or `none`
`ast_substitute(tmpl, bindings)`	fill a template from bindings
`ast_rewrite(subj, rules)`	repeated bottom-up rewriting until unchanged or 10,000 passes; returns the current AST
`ast_rules(p1, t1, …)`	flat-pairs sugar for a rule list

```
ast_head('(2 + 3))                                  # "+"
ast_operands('(2 + 3))                              # [<AST: 2>, <AST: 3>]
ast_make(ast_head('(2 + 3)), ast_operands('(2 + 3)))   # <AST: (2 + 3)>  — round-trips

ast_match('(?a + ?b), '(2 + 3))             # {a: <AST: 2>, b: <AST: 3>}
ast_rewrite('(x + 0), ast_rules('(?a + 0), '(?a)))  # <AST: x>  (additive identity)
```

Patterns reuse the `?x` variable syntax; a rule's optional third element is a **guard** (Mathematica's `/;`), and `?xs__` / `?xs___` match variable-arity argument runs. That's enough to write a symbolic differentiator or algebraic simplifier as a rule list — see Chapter 22 (the meta-circular evaluator) and Manual §19.10 for the worked CAS.

These `ast_*` functions take evaluated AST values and can be passed to other functions. Their older unqualified names (`head`, `operands`, `make_expr`, `match_pattern`, `subst`, `replace_all`, `rules`) are retired with migration hints. Constructors follow the same vocabulary: `ast_make_integer(5)` builds syntax, whereas `integer(5)` produces a number. `ast_make_string` constructs syntax; `ast_string` renders it. Direct evaluator calls to `ast_rewrite` let guards use caller-local bindings; the VM refuses guarded rules.

The held form `headof` remains distinct: it inspects the source of a call rather than executing that call to obtain an AST. `parts` and `ast_children` also have different node coverage and are not interchangeable.

For a quoted `with transaction [ body ]`, `ast_head` is `"with"`; both `parts` and `ast_operands` contain the resource-kind String AST and the body BlockExpression AST, in that order. `ast_make` checks that the kind is `"transaction"` and that the body is a block before rebuilding the scope. Inspection does not start a transaction. Executing the reconstructed scope uses the same synchronous rules taught in Chapter 16 §16.15.

A quoted form also behaves as **the sequence of its operands**, so generic collection code traverses code like data:

```
len('(2 + 3))           # 2
'(add(7, 9))[1]         # <AST: 7>
[op | op <- '(a + b)]   # comprehension over a form
```

Two converters cross between code and value:

```
to_data('(2 + 3))       # 5          — evaluate a form to its value
from_data(5)            # <AST: 5>   — lift a value into an AST literal
```

`from_data` is the friction-free way to turn any value (scalar, array, set, tuple) into an AST, sidestepping the `'(…)` quote-form bracket gotchas.

E.5 Tracing function calls

```
factorial: func(n) [
  if n <= 1 then 1 else n * factorial(n - 1)
]

trace func control

factorial(3)
# → Evaluating Call Expression
#   → Applying Function
#     → Evaluating If Expression
#         condition_result: false
#         is_truthy: false
#         executing: alternative branch
#       → Evaluating Call Expression
#         → Applying Function
#           ...
#         ← Function Application: 1
#       ← Call Expression (user function): 1
#     ← If Expression: 2
#   ← Function Application: 2
# ← Call Expression (user function): 6
```

Tracing is a **statement**, not a call: `trace <domain>` switches on logging for a whole *class* of operation, and `untrace` switches it off. `trace func` covers function application and `trace control` covers `if`/`while`/`foreach`, so together they print the full call/return chain with depth-aware indentation. Name several in one statement, as above, to widen the trace; `untrace func control` mirrors it. A typo rejects the whole statement rather than half-applying it.

The other domains follow the same shape — `trace binding` (every value as it is assigned), `trace sets`, `trace comprehension`, `trace epistem`, `trace quantifiers`, `trace relations` (queries and answer counts), `trace reasoning` (rule firing), `trace concepts`, `trace stack`, and `trace all`. Add the `/verbose` refinement (`trace/verbose func`) for bound variables, condition sources, and result types; `/debug` for internal state.

To select individual function values, use `trace/functions(factorial)` with a bracket or newline/end body (optional §E.22). The call `trace(m)` remains the linear-algebra matrix trace, the sum of a matrix diagonal. It does not select a function.

Each step shows:

- → enter, ← return
- the kind of node being evaluated (`Call Expression`, `If Expression`, `Function Application`, etc.)
- intermediate values that drove the dispatch (`condition_result`, `is_truthy`, branch chosen)
- the final return value of each frame

This is the **structured-trace** equivalent of `print`-debugging: you can see exactly what the evaluator did, not just end-to-end inputs and outputs. Particularly valuable for debugging recursion (where the call

stack is the interesting part) and for *teaching* — students can read the trace output and follow Felleisen's "design recipe in motion."

To stop tracing, write `untrace` (bare, to clear every domain) or `untrace func` (to clear just that one).

Better still, don't stop it by hand. Give `trace` a bracket block and it traces only that region, restoring whatever was in force when it exits:

```
trace func [ result: factorial(3) ]
```

The domain is optional here too — `trace [ ... ]` traces every domain for the block's extent. Because the block restores rather than switching off, one nested inside an outer `trace` leaves that outer domain running; and because it opens no scope, bindings made inside it survive. That second property is the one that matters for a debugging tool: wrapping code in `trace` must never change what the code does.

The transcript outlives the block, so you can assert on it after the fact:

```
clear_trace_log()
trace binding [ total = 7 ]
trace_log()                    # → ["  total = 7"]
```

E.6 REPL commands

When you're at the `axioma>` prompt, the colon-prefixed commands are *meta-operations* — not Axioma syntax, but commands to the REPL itself.

Command	Effect
`:help`	List available REPL commands
`:exit`	Quit the REPL (also `:q`)
Restart with `--no-image`	Start a clean environment
`:save filename`	Save the session to a file
`:load filename`	Load a previously-saved session
`:source filename`	Execute a file in the current native CLI session
`:refresh`	Re-read and execute the last sourced path (see §E.20)
`:stack` (or `:s`)	Show the current interpreter stack
`:stack trace`	Toggle automatic stack tracing
`doc <name>`	Show the docstring for a builtin or user-defined name

Example:

```
axioma> doc parse
Word: parse
Definition: Built-in function: parse
Source: builtin
```

```
Type: BUILTIN
Value: builtin function: parse
Signature: parse(code, [mode])

Documentation:
Low-level parse of a source String to AST; mode "expression" (default — FIRST statement
↪  only), "statement", or "program" (ast() is the no-truncation front-end). Not a typed
↪  value reader — that is parse_as(Integer, s) / Integer.parse(s).
```

The `doc` command is your fastest path from "what does this function do?" to a usable answer.

E.7 Inspecting Concepts

A *Concept* (Russell-style class) prints itself in a one-line summary when you ask for its value directly:

```
concept Animal
Mammal extends Animal
Dog extends Mammal

println(Dog)
# concept Dog extends Mammal { properties: cardinality: ∞, doc: "concept of Dog" }
```

The output shows the parent (via `extends`), declared properties, cardinality, and any attached documentation. For deeper inspection:

Form	Purpose
`x is Concept`	Does `x` belong to this Concept (walks the chain)?
`x.slot`	Read a slot value
`x's slot`	Same, possessive form

(Note: the natural-language forms `Concept show properties` and `Concept display hierarchy` only apply to **conceptual graphs** — a separate Axioma construct that the textbook doesn't cover — not to ordinary Russell-style concepts. For the latter, use `println(Concept)` and the `is` / `is` predicates.)

E.8 Provenance and grounding (logic programs)

Chapter 21 covers the basics. The full toolkit:

Form	Purpose
`grounding("rel", args...)`	The grounding level of a stored fact (`"axiom"`, `"theorem"`, `"conjecture"`, `"datum"`, etc.)
`truth("rel", args...)`	The Belnap B4 value attached to a fact (or `"unknown"`)

Form	Purpose
`set_truth("rel", args..., "true"/"false"/"both"/"neither")`	Annotate a fact with a B4 value
`proof("rel", args...)`	Array of `(fact_string, grounding, depth)` tuples walking back to axioms
`why fact()`	**Keyword form** — print a human-readable explanation
`challenge("rel", args...)`	Mark an axiom as suspect (Q1 requirement)
`challenged("rel", args...)`	Check if a fact has been challenged
`cancel("rel", args...)`	Suppress a defeasible conclusion (canonical exception pattern)
`uncancel("rel", args...)`	Remove a cancellation marker
`canceled("rel", args...)`	Check if a fact is canceled

E.9 HiLog meta-queries

When you want to ask **about** facts, not just **for** them:

Form	Returns
`predicates_of(X)`	All facts mentioning `X` in *any* position, any arity: `[(rel, args_tuple, grounding), ...]`
`predicate_names()`	The set of all relation names with at least one fact
`?P(X)`	Per-position arity-specific meta-call (`P` is a PatternVariable). Like `predicates_of` but with a fixed arity and bound position
`?P(X, ?Y)`	2-arity facts starting with `X`, `Y` free
`?P(?X, "carol")`	2-arity facts ending with `"carol"`

The `?Name` form uses Axioma's existing pattern-variable parser (Ch.21). The evaluator intercepts a `CallExpression` whose head is a PatternVariable and routes to a meta-call handler.

E.10 Runtime fact mutation

Beyond the static fact base, Axioma supports runtime mutation:

Form	Effect
`insert("rel", args..., [grounding])`	Assert a new fact at runtime; grounding defaults to `"datum"`
`forget("rel", args...)`	Retract a fact and its metadata; already derived conclusions remain

Form	Effect
`forget_cascade("rel", args...)`	Retract the fact and its dependent conclusions; surviving support can re-derive them
`transaction_begin()`	Open an atomic mutation block
`transaction_commit()`	Persist mutations since `begin`
`transaction_rollback()`	Undo mutations since `begin`
`with transaction [ body ]`	Commit the synchronous body on success; roll back an escaping error
`cardinality(Concept, "prop", min [, max])`	Register a min/max bound on a concept property

`forget` rather than `delete` or `retract` because both of those are reserved as keywords for separate features. The transaction trio gives you atomicity for groups of mutations — useful when "this set of changes should all succeed or none of them should."

The scoped form closes its own transaction; do not call the explicit controls inside it, including through a helper function. Rollback covers journaled knowledge-base state and unification bindings, not variables, mutable collections, or I/O. A caught error value permits commit. See the optional Chapter 16 §16.15 for worked examples and the synchronous producer boundary.

E.11 The two null types — `none` and `om`

Axioma has **two** null-like values with different semantics — a distinction the book mentions only in Chapter 20 (Ω as Kleene unknown). Here's the full picture:

Value	Spelled	Meaning	Truthiness	Display
`none`	`none`	Absent value, of type `None`	falsy	`none`
`om`	`om`, `Ω`	SETL-inspired *unknown* / *undefined* — value exists but its identity is not yet determined	falsy	`Ω`

The old spelling `null` is retired and produces a syntax error.

Not interchangeable. `om == none` returns `false`. Use `none` for "this slot was never set." Use `om` for "this slot has a value but we don't know what it is yet" — database unknowns, partially-known data, K3 propagation.

The K3 unknown shorthand `om` (used throughout Chapter 20) is the same value — it just doubles as the Kleene three-valued unknown when the operator dispatcher sees it.

E.12 Two-tier knowledge — `postulate` and `axiom`

Keyword	Meaning
`axiom <name>: ...`	Foundational truth, accepted by the system
`postulate <name>: ...`	Tentative claim, may be verified or refuted
`axiom/persist <name>: ...`	With refinement — persist across sessions
`axiom/transient <name>: ...`	With refinement — session-local only
`declare/persist x = ...`	Deferred binding marked for saving; demand first, serialization limits apply
`declare/transient x = ...`	Session-local (default for scripts)

The refinement system lets you control persistence on a per-binding basis. REPL mode persists by default; script mode is transient by default.

E.13 Where to put what

A practical rule of thumb for *which tool to reach for*:

Situation	Use
One-off "what is this value right now?"	`expr .`
Production logging	`println(...)`
Tracing a function call chain	`trace func` … `untrace`
Looking at AST structure	`fullform` (compact) or `treeform` (deep)
Debugging a logic-program derivation	`why <fact>`
Auditing which facts mention `X`	`predicates_of(X)`
Probing runtime types	`type(x)`
"I forgot what this builtin does"	`doc name` — or `apropos("…")` when you forgot the name too
"What IS this value?"	`describe(x)`
"Which binding is this, and what does it hold?"	`reflect(x)` (E.23)
"What can I do with this type?"	`functions(Integer)` (alias `methods`)
"How do I call this?"	`signature(f)`
"Show me that function's definition"	`source(f)`
Stepping the global interpreter stack (REPL)	`:stack`
"What's the type of this unannotated function?"	`axioma --infer` (Ch.54); on the island, that is the checker

The `.` and `trace` cover most day-to-day debugging. Everything else is for the rare-but-important cases — metaprogramming, KB-fact auditing, AST manipulation. Keep this page nearby and reach for the right tool.

E.14 Types you didn't write — `--infer` and `axioma/hm`

`--typecheck` (Chapter 19) honors annotations you wrote. `--infer` *reads* an unannotated body. The two jobs are easy to mix up:

	`axioma --infer file.ax` on `axioma/all`	`#language axioma/hm`
Job	print an arrow, or a skip reason	type the whole file
Untypable	skip, exit 0	refuse, exit 1
`println`	"not inferred (world)"	no known type — fatal

On an island file, `--infer` **is** the run-time checker, not a second opinion. Chapter 54 is the tutorial; the Manual section *`#language axioma/hm` — the closed Hindley–Milner island* is the reference. `inferred(f)` on the host is the same lens as `--infer` for one function (it returns `none` when the body is outside the fragment).

> *"To find a fault is easy; to do better may be difficult."* — Plutarch, *Moralia*. Axioma's introspection tools won't write your code for you, but they'll show you exactly where the trouble is.

E.15 Finding current language help

The running language can describe its own vocabulary. Use local help when checking a spelling or an available operation:

```
signature(map)
describe([1, 2, 3])
oracle Float
manual "ranges"
```

`oracle` supplies local vocabulary hints; `manual` searches or reads the manual embedded in the binary. Neither needs a model call. The REPL can enable unknown-word hints with `:oracle on`; `oracle/on` is the source form. An installed binary's embedded help can lag a newer website, so record the binary revision when reproducing an example.

For a keyword or punctuation, quote its name so help inspects the spelling: `doc("for")`, `doc("end")`, `doc("|>")` and `doc(".+")`. The REPL command `:doc for` uses the same lookup. `:man` is the short **REPL command** for `:manual`; source programs use `manual("for loop")`, not `man(...)`.

Short help combines authored cards, runtime signatures, Manual excerpts and labeled library-reference notes. The notes can include argument diagnostics from different overloads; they are not a complete specification. A name can also be reserved, retired or experimental: finding it is not evidence that every intended feature is implemented. Your own binding's attached documentation takes priority over the builtin of the same name.

The online edition has a type-ahead search in its toolbar. Start with **This book**, or choose **All docs** to

include the Manual, language reference and studies. Search literal language words such as `for` or `end`, or an operator such as `|>`. Select a result to jump to the relevant section. Browser Find remains useful for moving between occurrences on the current page.

`ask` is different: it contacts a configured model provider and can send context about the user's words and definitions. It returns a suggestion, not a proof and not an automatic language extension. Use that route deliberately; local help and ordinary evaluation never need to invoke it.

E.16 Commands are values; execution is a separate operation

A command literal groups process arguments without executing them. Interpolated text remains one argument, including embedded spaces.

```
report_name: "quarterly report.txt"
job: c"printf '%s' ${report_name}"
println(job.executable)       # printf
println(job.args)             # ["printf", "%s", "quarterly report.txt"]
```

On a system providing `printf`, `os.run(job)` executes that command and returns captured `stdout`, `stderr`, and `code`. A nonzero exit status is data in `code`; failure to start or a timeout is an Error. Commands do not expand shell wildcards, variables, or redirections.

When a shell pipeline is the intended program, an explicit shell block runs it immediately. This example uses Unix shell syntax:

```
sorted_lines: [shell |
  printf 'pear\napple\n' | sort
]
println(sorted_lines.stdout)
# apple
# pear
println(sorted_lines.code)   # 0
```

This `|` belongs to the shell; Axioma's `|>` passes a value to a function. Shell blocks do not implicitly capture Axioma variables. Prefer Command arguments for text supplied as data. Filesystem operations live under `io` and working-directory/process operations under `os`; a relative path uses the process working directory, not automatically the source file's directory. Browser builds can construct command values but cannot launch local processes. Run the shell example only in a local environment.

E.17 Certified theorems and ordinary mutable data

The proof library distinguishes a formula, a proposed derivation, and a checked theorem. Certification checks the steps and the requested goal; it does not certify an arbitrary record merely because its field says "theorem".

```
import "proof"
atom_a: kpred("A", [])
atom_b: kpred("B", [])
checked: certify([
  assume(kand(atom_a, atom_b)),
  andEL(1),
  impI(1, 2)
], kimp(kand(atom_a, atom_b), atom_a), [])
println(proved(checked))        # true
println(len(hyps(checked)))     # 0
```

This proves `(A ∧ B) → A` with no undischarged assumptions. A proof that retains assumptions is conditional; those assumptions must remain visible. An invalid proof is rejected with a reason.

Certification owns a private copy of the conclusion and assumptions. Changing the original formula data afterward does not alter what was proved. Accessors return detached data, so editing an accessor result cannot rewrite the theorem either. This protection is different from making all arrays or dictionaries immutable. A relation engine's derived fact, a neural suggestion, and a sealed checked theorem also have different justifications; never substitute one for another based on a similar label.

E.18 Descriptions and premise revision

A fact can carry an explicit description with `qua`. The description is a handle for rules, not proof that a person experiences the world that way. Use person and stage arguments when representing a particular appraisal.

```
relation departed(item)
departed("estate") qua "returned"
bearable_description(X) <~~ departed(X) qua "returned"
expect("description derives", {X | X <- bearable_description(X)}, {"estate"})
forget_cascade("departed", "estate")
expect("unsupported description disappears", {X | X <- bearable_description(X)}, {})
expect("forgotten fact has no tags", framings_of("departed", "estate"), {})
```

Generic rules using `it qua "returned"` and relation-specific rules such as the example both depend on the ordinary underlying fact. `proof` follows that fact; `framings_of` inspects its handles. Cascading withdrawal removes unsupported transitive conclusions as well. A conclusion with independent support can reappear on the next query.

Plain `forget` removes a fact and its handles. A conclusion that rested on it alone is withdrawn at the next query of its relation; `forget_cascade` withdraws it at once and reports it. Reasserting the fact does not restore its old handles.

Cancellation suppresses a fact and keeps its handles for `uncancel`. A cancelled fact supports nothing, so the next query withdraws the conclusions that rest on it alone until `uncancel` brings them back. A transaction rollback restores removed metadata together with facts.

Both strict and defeasible consequences can lose their premises. A relation engine's `theorem` label records a strict derivation from those premises; it is not proof that the source is truthful or that the conclusion is permanent. An empty query is not an explicit negation. In an executable philosophy example, removing a predicted disturbance is not evidence of a psychological cure. These distinctions keep the model's results honest.

E.19 A viewpoint is not an endorsement

When reading a formalized argument, keep the source passage beside the model and identify which definitions and bridging premises the modeler added. The Epictetus study follows this approach: chapter XLII separates an offender's judgment, an observer's report of it, and the facts assumed for the case. Its executable checks also show what happens when the contrary case fact is withdrawn.

Axioma's stored `believes` and `knows` calls take an Agent entity, a relation name and its arguments. `beliefs_of` lists attributed proposition strings; `believers_of` finds their holders. Quoted content stays distinct through nested attribution. Reporting that someone believes a claim does not assert the claim itself, nor does it assign that belief to the reporter. These are explicit stored reports, not an automatic proof system for nested knowledge.

Common knowledge is stronger than everyone knowing something: it also requires everyone knowing that everyone knows, and so on. The internal finite-model checker explores all reachable worlds, so

it needs no fixed depth cutoff. It has no public `common_knowledge` builtin. A stored-belief example therefore does not demonstrate common knowledge.

E.20 Optional: edit a file and replay it in the REPL

The native CLI REPL can execute an Axioma file without losing the definitions already in your session. Save the following as `scratch.ax` in the directory where you started Axioma:

```
base = base + 1
```

Start `axioma --no-image`, then enter:

```
axioma> base: 40
40
axioma> :source "scratch.ax"
41
axioma> :refresh
42
```

Now change the file's `1` to `10`, save, and enter `:refresh` again: the result is `52`. Refresh reads the current file and repeats its work in the existing session. It does not reset `base` or undo earlier work. The same rule applies to output, mutation and other side effects. Top-level `let` scopes survive to the next prompt. Existing binding and type rules still apply.

For a path containing spaces, use `:source my examples/scratch.ax` or put matching single or double quotes around the path. Only those outer quotes are removed; there is no shell expansion or escape processing. Empty or unmatched outer quotes produce usage help. The whole remainder names one file. The REPL remembers its absolute path at source time. After `!cd`, refresh still reads that path, even if that filename also exists in another directory. Sourcing does not change cwd or the REPL's import base: a relative import in the file still uses the existing REPL context. Normal module loading rules apply; this command does not add dependency reloading.

A source attempt is remembered before the file is read. If the file is missing or unreadable, create it or fix access, then retry with `:refresh`. Fix syntax errors the same way; a parse error executes none of the file. A runtime error stops at the failing statement and preserves preceding effects, which will run again on refresh. Incorrect command arguments leave the previous target intact. If no file has been sourced, the command asks for a source path first. The target lasts for this REPL session and is not stored in a saved image.

Try it. After the transcript above, save the change from `1` to `10` and refresh twice. What is `base`? **Answer:** `62`: the two runs add `10` each to the existing `42`. To start over, explicitly initialize the value again. This is an optional workflow for Chapter 16's mutation exercises; their programs and solutions keep the same semantics.

These commands work in the native interactive and piped CLI REPL, using its evaluator and usual result display. They are not source-language syntax and are not added to the embedded session, browser or GUI handlers. Do not put colon commands in `.ax` files. `source(f)` remains the reflection builtin for rendering function/AST source; `:load` restores a saved session.

E.21 Optional exercise: construct and rewrite an expression

Build an AST for `2 + 3`, inspect its head and operands, and reconstruct it. Then simplify the quoted expression `value + 0` with one AST rewrite rule. Explain why constructing an Integer AST differs from calling `integer`.

One solution (requires the full language):

```
syntax_sum: ast_make_infix("+", 2, 3)
expect("operator", ast_head(syntax_sum), "+")
expect("operands", ast_operands(syntax_sum), [from_data(2), from_data(3)])
expect("reconstruction", ast_make(ast_head(syntax_sum), ast_operands(syntax_sum)),
 ↪  syntax_sum)
expect("rewrite", ast_rewrite('(value + 0), ast_rules('(?x + 0), '(?x))), '(value))
expect("AST constructor", type(ast_make_integer(5)) == AST, true)
expect("value constructor", type(integer(5)) == Integer, true)
```

An AST is code represented as data. `ast_make_integer(5)` constructs that code; `integer(5)` gives the numeric value itself. Neither operation executes generated code. The rewrite changes the AST without looking up a runtime `value` binding.

E.22 Optional: trace only selected functions

A recursive program may call many helpers. Start with the function whose behavior you want to explain; expand the selection when a helper matters. The following example also shows that aliases retain the selection:

```
double(x) = x * 2
saved_double: double
clear_trace_log()
trace/functions(double) [answer: saved_double(3)]
println(answer)
println(traced(double))
println(len(trace_log()))
# → double(3)
# ← double: 6
# 6
# false
# 2
```

The result is unchanged, and the selection ends with the body. To use an end-form body and see calls made by a higher-order function:

```
double(x) = x * 2
trace/functions(double)
  doubled: map(double, [2, 3])
end
println(doubled)
# → double(2)
# ← double: 4
# → double(3)
# ← double: 6
# [4, 6]
```

Select several values with `trace/functions(f, g)`. Target expressions run once before tracing begins. Nested selections add functions and restore the outer settings even when an error or early return leaves the body. `traced(f)` asks whether that exact function value is selected; `traced()` still lists whole domains. Rebinding `f` does not transfer selection to its new value. A partially applied function is a separate value.

Ordinary recursion indents its selected calls. An optimized tail call prints `↪ ... [tail call]`, followed eventually by one final outcome for the actual evaluator invocation; the optimizer stays active. A propagating error prints `!`. Previews are limited in size and do not force lazy arguments, consume generators, or run user printers. The transcript is kept until cleared.

Try it. In the first example, assign `double = func(x) [x * 3]` inside the trace body before calling `saved_double(3)`. Predict the result and selection. **Answer:** it still returns 6 and logs the original function under the label `double`. `traced(saved_double)` is true inside the block and `traced(double)` is false, since the latter now names a different value. Selecting a value is more stable than looking up its name again at each call. Chapter 7's function exercises and their solutions continue to use the same call

semantics.

This optional full-language tool requires Axioma `Function` targets and a body; it refuses builtins, a direct fresh `let` inside that body, and combined display refinements. Move a fresh declaration outside the trace block; ordinary bindings inside the body remain visible afterwards. `untrace` or `untrace func` stops selections in the current context, while leaving a nested scope restores its previous settings. The VM refuses this syntax explicitly.

E.23 Optional: inspect a binding with `reflect`

The mutation chapter distinguishes a name from the value it holds. `reflect` makes that distinction visible. It returns a read-only snapshot with five root fields: `type`, `preview`, `binding`, `function`, and `representation`. Start with an ordinary variable name and explore its fields through member completion:

```
var reflection_total :: Number = 7
reflection_before: reflect(reflection_total)
expect("value type", reflection_before.type, Integer)
expect("binding name", reflection_before.binding.name, "reflection_total")
expect("declared type", reflection_before.binding.declared_type, Number)
reflection_total = 8
expect("saved preview", reflection_before.preview, "7")
expect("new preview", reflect(reflection_total).preview, "8")
expect("same binding cell", reflection_before.binding.id,
 ↪  reflect(reflection_total).binding.id)
```

The preview is a bounded String, not a writable link to the original value. Changing the program after inspection does not change the saved descriptor. The root `type` is a read-only view of a type such as `Integer`, not the String `"Integer"`. Its binding's `declared_type` answers a separate question. These opaque views preserve type equality and named-type membership while refusing slots or actions; membership follows a named type's definition, which is not frozen by the snapshot. Direct `is` tests with a reflected structured descriptor, such as an arrow or union, refuse.

A bare name selects its lexical binding. An expression runs once and describes the resulting value, without pretending that the result has a variable name:

```
var reflection_calls = 0
reflection_next() = [rebind reflection_calls = reflection_calls + 1; 42]
reflection_result: reflect(reflection_next())
expect("ordinary expression runs once", reflection_calls, 1)
expect("result type", reflection_result.type, Integer)
expect("no invented binding", reflection_result.binding, none)
expect("quoted text stays text", reflect("reflection_calls").type, String)
expect("quoted code stays code", reflect('(reflection_next())).type, AST)
expect("inspecting code does not run it", reflection_calls, 1)
```

Name inspection can also explain why a value is not yet available. It does not force a deferred declaration or a lazy parameter, and it does not consume a generator. An unknown name still raises an error.

```
var reflection_hole :: Integer = _
expect("uninitialized binding", reflect(reflection_hole).binding.state,
 ↪  "uninitialized")
expect("no actual value type yet", reflect(reflection_hole).type, none)
expect("annotation is available", reflect(reflection_hole).binding.declared_type,
 ↪  Integer)
var reflection_demands = 0
```

```
reflection_work() = [rebind reflection_demands = reflection_demands + 1; 21]
declare reflection_delayed :: Integer = reflection_work()
expect("deferred binding", reflect(reflection_delayed).binding.state, "deferred")
expect("reflection did not demand it", reflection_demands, 0)
expect("explicit ordinary read demands it", reflection_delayed, 21)
expect("one demand", reflection_demands, 1)
```

Function metadata lives under `function`: parameter names, signature, reconstructed source, local declarations, and captured bindings. Inspection does not call the function. `complete` and `limitations` state whether the supported source analysis is exhaustive; an empty list from an incomplete analysis is not proof that no locals or captures exist. The metadata lists are immutable Tuples, and the descriptor records have type `Reflection`.

```
reflection_double: func(item) [item * 2]
reflection_function: reflect(reflection_double).function
expect("one parameter", len(reflection_function.parameters), 1)
expect("parameter name", reflection_function.parameters[1], "item")
expect("immutable parameter list", type(reflection_function.parameters), Tuple)
expect("source available", type(reflection_function.source), String)
```

Binding IDs identify cells for comparisons within this run. Assignment and a closure's capture retain the cell; a fresh shadow or a new function invocation has new cells. A function value has a separate identity from the binding that holds it. Neither kind of ID is a memory address or a persistent identifier. `binding.shadowed` lists hidden outer bindings, nearest first. `writable` concerns rebinding; an immutable binding can still hold a mutable Array.

`representation` describes implementation choices separately from the semantic type. For example, both small and large integers have type `Integer` even when their storage categories differ. Most programs should use `type`; tools using representation details should check `schema_version` and `engine` first.

Call `reflect` directly. Calling the intrinsic through an alias or higher-order function is refused because its meaning depends on the supplied name at the call site. A wrapper can call it, but observes its own parameter. Pass an already-created descriptor to a helper to preserve the caller's information. The evaluator and VM support this tool for their supported execution forms. The deferred/lazy examples use the evaluator: the VM still refuses those forms, and also refuses inspection of its synthetic recursive self binding. Unavailable source metadata stays explicit. `inspect`, `describe`, `declared`, and the AST tools retain their earlier roles.

Exercises

1. Predict `reflection_before.preview` after another assignment to `reflection_total`. How do you obtain its current preview, and which binding ID should it have?
2. Define a helper that returns the binding name of its parameter `item`. Call it with a variable named `reflection_original`. Explain which name it returns, then write a second helper that receives the caller's descriptor.
3. Why does `reflect("reflection_original")` describe a String? Why does `reflect('(reflection_original + 1))` not add one to the variable?

Solutions

The saved preview stays `"7"`. Call `reflect(reflection_total)` again for the current value; reassignment preserves its binding ID. Both wrapper styles in exercise 2 are useful, but they inspect different things:

```
reflection_name: func(item) [reflect(item).binding.name]
reflection_saved_name: func(snapshot) [snapshot.binding.name]
reflection_original: 42
expect("wrapper sees its own parameter", reflection_name(reflection_original), "item")
expect("pass the caller's snapshot",
 ↪  reflection_saved_name(reflect(reflection_original)), "reflection_original")
expect("String is a value", reflect("reflection_original").binding, none)
expect("AST is inert", reflect('(reflection_original + 1)).type, AST)
expect("original value unchanged", reflection_original, 42)
```

In exercise 3, a String is text and a quoted expression is an AST value. Neither is a request to look up or execute its contents. This is the same code-as-data boundary taught in E.4, now exposed through one structured inspection interface.

Appendix F · Cross-language translation

What this appendix is. Axioma is a *learning* language — most of its readers arrive already knowing one or more other languages. The cross-language translation feature is built for that audience: it lets you look at a Python (or JavaScript, Julia, R) snippet next to its Axioma equivalent, *inside* the language, without leaving your editor. Three operators and one builtin do all the work.

F.1 The three operators

You've already seen `[python | …]` in the polyglot interop chapter — it *executes* foreign code from inside an `.ax` file. Two arrow-form siblings sit next to it:

Form	Direction	What you get back
`[L | body]`	execute body in L	the foreign runtime's value
`[A -> B | body]`	translate body from A to B	a `String` of B's source
`[A --> B | body]`	translate, then execute	typed value (or scope leak)
`[nl -> B | description]`	symbolize a description into B	a `String` of B's source
`[nl --> B | description]`	symbolize + execute	typed value

Read the arrows as **"more arrow = more action."** Zero arrows runs. One arrow renders source. Two arrows goes all the way through.

Pure translation — the -> arrow

```
src: [axioma -> python | [n*n | n <- range(10)]]
println(src)
# [n * n for n in range(10)]
```

The body is read raw (multi-line, embedded brackets, indentation all preserved — the same reader the execute form uses). No foreign runtime is touched; what comes back is a String holding the equivalent target-language source. Useful when you want to *see* the translation, not run it — copy it into a Python file, compare two languages side by side, then test that the translation preserves the behavior you require.

Translate + execute — the `-->` arrow

```
sum_sq: [axioma --> python | sum([n*n | n <- range(1, 11)])]
# → 385  (Python computed it; marshaled back as Axioma Integer)
```

Same Axioma input, but now the translated Python actually runs in Python's runtime and the typed result lands back in Axioma. Useful when you trust Python's library for a specific operation but want the surrounding code to stay in Axioma.

A particularly nice variant is **`B == "axioma"`** — the translated source is parsed and evaluated *in the current scope*, so definitions leak into the surrounding program:

```
[python --> axioma | def double(x): return x * 2]
println(double(5))   # → 10
```

You can lift small functions, comprehensions, or whole modules out of a Python tutorial and continue writing Axioma against them without manually rewriting.

Describe what you want — the `nl` source

The source-language slot also accepts `nl` (alias `english`, `natural`), which treats the body as a *natural-language description* rather than source code. The LLM symbolizes it into idiomatic target-language code:

```
# Just show me the syntax
src: [nl -> axioma | double the value 7]
println(src)
# double: 7 * 2          (the LLM picks the Axioma form)

# Just give me the value
v: [nl --> axioma | sum of squares from 1 to 10]
println(v)
# 385                    (Axioma value after Eval)
```

This is the closest Axioma has to a "look it up in the textbook" interface from inside the language. Useful when you have an intent in mind ("I want to filter the even numbers") but don't yet know the construct Axioma uses for it.

The catch: `nl` is always LLM-required. There's no deterministic path for natural language → code, so every call hits the announce-print billing line and burns API budget. The `-->` variant also inherits the LLM's non-determinism — if the model leans on training data from another language and emits a foreign idiom (Elixir's `|>` pipe-forward, Rust's `let mut`, Haskell's `<-` in a do-block), the resulting Axioma source won't parse. Re-run, or fall back to writing the syntax by hand once you've learned it.

F.2 The builtin — when code lives in a String

The block form needs its body to be a literal in your source. For code that lives in a String variable (read from a file, pulled from an API, copy-pasted at runtime), reach for the function-call counterpart:

```
py_src: read_file("script.py")
ax_src: translate(py_src, "python", "axioma")

# Forward direction works on AST expressions too:
py: translate([n*n | n <- range(10)])
# → "[n * n for n in range(10)]"
```

Signature is `translate(code, source_lang, target_lang)` — English "from X to Y" order. Defaults: `source="axioma"`, `target="python"`, so `translate(EXPR)` is enough for the common Axioma → Python case.

F.3 What runs offline vs what calls an LLM

Translation is hybrid:

- **Axioma → Python** uses a built-in deterministic AST visitor. Within its supported subset, it handles: bindings, named functions, lambdas, infix arithmetic, list comprehensions, conditionals, `println`, calls. Output is byte-for-byte deterministic, runs entirely offline, and requires no API key. Unsupported AST forms can fall back to an LLM even in this direction; inspect the provider announcement instead of assuming the arrow alone promises offline execution.
- **Other general-program directions** — such as `python -> axioma`, `axioma -> javascript`, and forms outside the deterministic subset — routes to an LLM. The translated output is *idiomatic but non-deterministic*; you'll get slightly different phrasings between runs.

Dedicated domain translators can have their own offline rules: Chapter 53 shows SQL-to-algebra and SQL-to-calculus translation. The general-program description here does not classify those separate domain paths.

Deterministic output also does not establish semantic equivalence. Range endpoints, division, indexing and numeric types differ between languages. For example, passing a call named `range` through to Python does not prove that it enumerates the same endpoints as an Axioma range. Review generated code and compare boundary cases before using it for a calculation.

Before any LLM call goes out, Axioma prints one line to stderr so you know what's about to be billed:

```
[axioma translate] python → axioma   provider=openrouter
↪  model=google/gemini-2.5-flash-lite   endpoint=https://openrouter.ai/api/v1   (paid
↪  API)
```

Local Ollama gets `(local, no billing)`. The print is on stderr, so it doesn't pollute the `String` returned to your script. Suppress it with `AXIOMA_TRANSLATE_QUIET=1` once you've chosen a provider you're

comfortable with.

An optional subscription route preserves Axioma's own translation and reasoning code. After `make install-model` and `axioma-model auth login`, inspect `axioma-model models` and select an eligible model using the launch environment `AXIOMA_AI_PROVIDER=openai-subscription` and `AXIOMA_SUBSCRIPTION_MODEL`. This explicit choice outranks API keys and announces subscription allowance. Missing authorization or exhausted allowance does not fall back to a paid API. Response length is provider-managed; the API's output-token caps are unavailable. The remote account setup is an optional external integration, not a prerequisite for this appendix's deterministic exercises. Full setup and limitations are in `Axioma Subscription Integration.md` at the project root.

Picking a provider

The translator auto-selects the first provider whose API key is visible in the environment, in this order:

OpenRouter → Anthropic → Gemini → Grok → OpenAI → Groq → Ollama

OpenRouter sits at the top because it routes to many backend models behind one billing account — set `OPENROUTER_API_KEY_AXIOMALANG` (project-scoped, takes precedence) or `OPENROUTER_API_KEY` (generic), and the default model is `google/gemini-2.5-flash-lite`. To use a different OpenRouter slug, set `OPENROUTER_MODEL=anthropic/claude-3.5-sonnet` (or any other OpenRouter model name).

If you'd rather not put a key in your shell at all, install Ollama and pull a model — the default hardcoded is `llama2`, but most local setups want `OLLAMA_MODEL=deepseek-r1:7b` (or whatever you actually have installed). Local Ollama is the only provider labeled `(local, no billing)` in the announce line.

F.4 When to use what

You have…	You want…	Reach for
An Axioma expression	the Python equivalent (just to read)	`[axioma -> python | EXPR]`
An Axioma expression	Python to compute it and give back the value	`[axioma --> python | EXPR]`
A Python snippet	the Axioma rewrite (just to read)	`[python -> axioma | BODY]`
A Python snippet	to USE it inside Axioma	`[python --> axioma | BODY]`
Source code in a String variable	any of the above	`translate(src, "python", "axioma")`
An intent in your head, no syntax yet	the Axioma source for it	`[nl -> axioma | description]`
An intent in your head, no syntax yet	just the value	`[nl --> axioma | description]`

Two negative cases worth flagging:

- **`[python --> axioma]` with a multi-statement Python body** works (the translated Axioma source is Eval'd as a program in scope). But **`[axioma --> python]` with multi-statement bodies** doesn't — Python's `eval()` is expression-only. For statement execution and captured stdout, use a separate `[python | …]` block. Wrapping a multi-statement body in `print` does not make that body a Python expression.
- **LLM translations are non-deterministic.** For tests or reproducible builds, prefer the deterministic `axioma → python` path (or convert by hand).

F.5 Example session

```
# Forward — deterministic, no LLM
src: [axioma -> python | doubled: map(func(x) [x * 2], [1, 2, 3])]
println(src)
# doubled = list(map(lambda x: x * 2, [1, 2, 3]))

# Compute via Python's runtime
result: [axioma --> python | sorted([3, 1, 4, 1, 5, 9, 2, 6])]
println(result)
# [1, 1, 2, 3, 4, 5, 6, 9]

# Pull a Python idiom into Axioma
[python --> axioma | def square(x): return x * x]
println([square(n) | n <- range(5)])
# [0, 1, 4, 9, 16]
```

That last block — definitions from foreign code becoming first-class Axioma values you compose against — is the load-bearing trick. Translation isn't a one-way export: it's a two-way bridge that lets you stand inside Axioma and reach for any concept you already know, in whichever language taught it to you.

F.6 Further reading

The manual covers the same material in reference form, including the full provider table and edge cases: Axioma Manual § 27.1b. The in-language tests in `tests/axioma/translation/` are also readable as worked examples — they exercise every operator with byte-for-byte assertions.

Appendix G — How the Axioma Interpreter Works

You can write useful programs without knowing how an interpreter is built. This appendix offers a short route into the implementation when you become curious: what reads your program, what gives it structure, and what runs it? It follows the **Go tree-walking interpreter**; the compiler, VM and native execution paths are outside its scope.

Read this after functions and local environments (Chapters 3 and 12). Chapter 22 builds a small teaching evaluator in Axioma. Here we follow the Go implementation that actually runs the book's programs. That teaching subset is not a complete implementation of Axioma.

Source snapshot. Implementation links are pinned to reviewed revision `3c37576a6` (September 7, 2026). Paths are relative to the repository root. Function names are search targets, not promises that files will never move.

G.1 From text to a result

For an ordinary program, keep this simplified picture in mind:

```
source text
    | lexer: recognize words, numbers and punctuation
    v
tokens, with source positions
    | parser: organize tokens into nodes
    v
abstract syntax tree (AST)
    | evaluator + environments + runtime values
    v
result / printed output / diagnostic
```

The lexer is asked for tokens as parsing proceeds; the diagram does not mean the CLI first saves a complete token list. Nor is it the entire startup sequence. The CLI handles options, source-language selection and output. Macro preparation can transform a parsed program before evaluation, and static checking can report findings before the program runs. Our example uses ordinary Axioma and no macros.

The division of work matters. The parser can record `base + x` without knowing either value. The evaluator needs an environment that gives those names meaning. Parsing an expression is not the same as executing it.

G.2 A small source map

Start with these files; follow a named function rather than reading every package from beginning to end.

Place	What to look for
`cmd/axioma/main.go`	Starts the CLI. File and inline input meet in `script_runner.go`.
`engine/engine.go`	Connects parsing, statement evaluation and result handling.
`lexer/lexer.go`	Supplies positioned tokens; `token.go` defines their kinds.
`parser/parser.go`	Builds the AST; defines expression handlers and precedence.
`ast/ast.go`	Defines the Go node structures that represent source code.
`types/types.go`	Defines runtime objects: numbers, functions, errors and other values.
`evaluator/evaluator.go`	Evaluates nodes; manages environments, lookup and calls.
`evaluator/builtins.go`	Provides core builtins; neighboring files contain many helpers.

For a file, follow `runScript` into `runSource`, then the engine's `ParseSource` and `EvalStatements`. For parsing, follow the lexer's `NextToken` and the parser's `ParseProgram`. These functions connect the stages shown in the diagram.

The interactive REPL reuses parsing and evaluation through `cmd/axioma/repl_eval.go`. Its environment survives between inputs; the ordinary script path creates an environment for that run. This is why a later REPL line can use a name entered earlier.

Axioma's knowledge features extend this architecture. The evaluator has cases for Concepts, relations and queries; the environment carries more than a table of variables. `engine/knowledge.go` connects knowledge loading, while `kb/` contains knowledge-base services. These are useful landmarks when you later follow Chapters 18, 20 and 21; they are not required to understand the arithmetic path below.

G.3 Follow one program

Save this as `walk.ax`:

```
let base = 2 + 3 * 4
let add_base = func(x) [base + x]
println(add_base(5))
```

From the repository root, build the CLI with the Go toolchain declared in `go.mod`, then run the file:

```
go build -o /tmp/axioma-interpreter ./cmd/axioma
/tmp/axioma-interpreter --no-kb --no-image walk.ax
```

The output is `19`. The two flags disable persistent knowledge loading and automatic session-image restore/save. The examples here need neither.

Tokens. For the first line, these are the real token constant names and their literal text (positions omitted):

```
LET("let") IDENT("base") ASSIGN("=")
INT("2") PLUS("+") INT("3") ASTERISK("*") INT("4")
```

`ASSIGN`, for example, is a Go constant whose token-type value is `"="`. `Token` also has `Line`, `Column` and optional `File` fields; the `base` token starts at line 1, column 5. Reading just this line reaches `EOF` next. The lexer recognizes `*`; deciding that multiplication groups more tightly than addition belongs to the parser.

Structure. Inspect the arithmetic part without running it:

```
/tmp/axioma-interpreter --no-kb --no-image --ast -e '2 + 3 * 4'
```

This is the actual tree display at the reviewed revision:

```
+
    ├── 2
    └── *
        ├── 3
        └── 4
```

The display is produced by `astinspect/`, not a dump of every Go field. The operator nodes have the Go type `ast.InfixExpression`. The complete `Program` contains two declarations (`LetStatement`) and an outer call wrapped in an `ExpressionStatement`. A `FuncExpression` holds the function body; a `CallExpression` describes a call. Run `--ast walk.ax` to inspect that larger tree.

Execution. `Eval` dispatches on node kinds. For these ordinary numeric infix nodes it evaluates the operands, then applies the operator to their runtime values. Multiplication produces the Integer `12`; addition produces `14`. The first declaration binds `base` to that value. Evaluating the function expression creates a function value with a body and its defining environment; it does not yet add anything. The second declaration names that value `add_base`.

For `add_base(5)`, the evaluator finds the function, evaluates the argument, creates a call frame with local `x = 5`, and evaluates the body there. Lookup finds `x` in the call frame and `base` in the captured environment. The last body expression returns `19`. Finally, `println` prints it and returns `none`; printing and returning a value are different operations.

G.4 Why a Pratt parser groups the expression correctly

A *prefix* handler starts an expression: a literal, a name, unary minus, or a parenthesized expression. An *infix* handler continues an expression that already has a left side. The same token can have different roles: `-3` begins with a prefix use of `-`, whereas `8 - 3` uses it between operands. The parser also uses continuation handlers for calls such as `f(5)`.

In `parser.go`, `prefixParseFns` and `infixParseFns` select these handlers. The function `parseExpression` starts with a prefix handler, then continues while the next operator binds more tightly than the current threshold. This threshold is often called *binding power*. The actual loop also checks statement boundaries and Axioma-specific forms.

For `2 + 3 * 4`, the parser first has `2`. The `+` handler needs a right operand. While reading it, `*` has greater precedence (`PRODUCT`) than `+` (`SUM`), so `3 * 4` becomes one subtree. The resulting shape is `2 + (3 * 4)`. Nothing has been multiplied yet.

Grouping in `(2 + 3) * 4` makes the parenthesized sum a subtree first. Associativity answers a different question: how equal-precedence operators group. Ordinary subtraction produces `(10 - 3) - 2`; exponentiation has its own right-associative handler, `parsePowerExpression`. Consult Appendix A for operator syntax rather than treating this small example as the full precedence reference.

G.5 Values, environments and closures

An AST node describes code; a runtime value is what executing that code produces. The node `ast.IntegerLiteral` describes a number in source; the object `types.Integer` holds an evaluated integer. Runtime objects implement the Go `types.Object` interface, including `Type()` and `Inspect()`. The latter provides a display representation; it does not execute the object.

For ordinary local names, think of an environment as a map with a link to an outer environment. Lookup through `Environment.Get` checks the local store, then walks outward. Bindings also carry rules about mutability and type annotations. Fresh `let` declarations can introduce a new environment layer, which the statement runner adopts for subsequent statements. A single flat dictionary would miss this part of the implementation.

A function value stores parameters, body and `Env`. This saved environment is its **lexical context**: the context where the function was created. For an ordinary complete call, `applyFunctionCore` creates a frame linked to that saved context and binds arguments locally. It does not search the caller's unrelated locals for missing names.

Here is a closure that outlives the call which created it:

```
let make_adder = func(base) [func(x) [base + x]]
let add14 = make_adder(14)
let base = 100
println(add14(5))                    # 19
```

The inner function retains access to `make_adder`'s parameter binding. The later `let base = 100` declaration is a different binding. Capturing an environment does not mean copying and freezing all reachable values; mutation still follows the binding rules taught in Chapters 12 and 16.

Builtins connect this machinery to Go. `GetBuiltins` assembles their table. An ordinary `types.BuiltinFunction` contains an `Fn` that receives runtime objects and returns a runtime object. `evalIdentifier` tries environment lookup before builtin lookup. The call path evaluates ordinary arguments and invokes the builtin's Go function. Read the short `println` implementation in `builtins.go`. Forms that control evaluation, such as conditionals or quotation, need special handling; this description of an ordinary call does not apply to every language form. Namespaced Go packages have a separate registry in `builtins/registry.go`; Axioma library sources also live in `lib/`.

G.6 Where errors belong

Use the stage of a failure to choose where to investigate. Run these two intentional failures separately, in fresh processes:

```
2 + )
```

This is a **parser error**: `SyntaxError: unexpected token ')'`. `ParseSource` cannot construct the expression; this source is not evaluated.

```
missing_name + 1
```

This is a **runtime error**: `Undefined word: missing_name`. The structure is valid, but lookup cannot supply a value in the fresh environment.

An **optional static check** can report, for example, a conflict between an annotation and a known expression type. `--typecheck` reports findings without ordinary program execution unless `--run` is requested and the check succeeds.

The CLI normally also runs a non-fatal static warning pass; `--no-typecheck` disables that pass. Static checks do not replace runtime checks or prove that every accepted program will succeed. See Chapter 54 for the separate Hindley–Milner language mode rather than assuming that its guarantees apply to unrestricted Axioma.

Runtime errors use `types.Error`, with a message and diagnostic context. The runner stops on an uncaught error and the CLI reports it. A caught error can instead be an ordinary value. Source locations help connect the failure to the program you wrote; Go implementation details are not the whole user diagnostic. Appendix E covers the user's debugging tools.

G.7 A guided experiment

Change only the first line of `walk.ax` to this:

```
let base = (2 + 3) * 4
```

Before running it, draw the arithmetic tree and predict the final output. Use `--ast walk.ax` to check the grouping, then run the program normally. **Answer:** the `*` node now has `+` as its left child, `base` is `20`,

and the final output is `25`. The call and closure machinery did not change.

To follow arithmetic, search for `parseExpression`, then `parseInfixExpression`, then the infix-node case in `Eval`. For scope, start at `evalFuncExpression`. Follow the call through `evalCallExpression` into `applyFunctionCore`, and read `Environment.Get` for lookup. The parameter-shadowing regression checks that a parameter shadows a global without changing it. The singleton-block parser test shows a complementary method: assert the node kind before any evaluation.

This appendix's examples are also source-linked in `verification/examples.json`. Run `make test-textbook` to check them. The inventory distinguishes runnable examples, displayed trees, shell commands and exercise fragments. When exploring, keep the same discipline: predict a result, inspect the structure, execute a small program, and compare what happened with the relevant test.