Contents

1 Axioma Programming Language Manual

Version 0.9 · *Calculemus! — Leibniz* **A multi-paradigm language for mathematics, logic, knowledge, and reasoning.**

> ***Calculemus!*** is Latin for **"Let us calculate!"** — the motto of the philosopher and mathematician **Gottfried Wilhelm Leibniz** (1646–1716), co-inventor of the calculus. Leibniz dreamed of a *characteristica universalis*: a formal language in which any idea could be written down exactly, paired with a *calculus ratiocinator*, a mechanical procedure for reasoning over it. Then, he wrote, two people who disagreed would no longer need to quarrel — they could take up their pens and say *"Calculemus"* (*"let us calculate"*) and settle the matter by reckoning. Axioma is a step toward that vision: a language where logic, mathematics, and knowledge are things you compute with.

1.1 Table of Contents

1.2 1. Introduction

Axioma is a programming language for mathematical computing, formal logic, and knowledge representation. It blends styles freely — set theory, first-order logic, multi-valued logics, lambda calculus, frame-based concepts, relational logic programming, stack-based programming, and natural-language definitions live side by side and compose with each other.

1.2.1 Key features

- **Concept system** with natural-language syntax (`concept Stock`, `Stock has price: 150`).
- **Logic programming** through deterministic, set-based pattern queries — Prolog power without backtracking.
- **Six-grade epistemic grounding**: every fact carries a grade on the ladder `axiom > postulate > theorem > conjecture > hypothesis > datum`, propagated through derivation as provenance.
- **Five first-class logics**: Boolean, Kleene K3, Łukasiewicz L3, Belnap B4, Gödel G3 (intuitionistic), automatically dispatched by operand type.
- **Strict and defeasible rules** in both backward and forward directions.
- **Bilattice truth values** with paraconsistent contamination (Belnap B4).
- **SQLite-backed knowledge base** shared with Cascade.
- **Stack programming** with both a user-accessible `Stack` type and a global interpreter stack.
- **MCP server** exposing 11 tools for AI integration.
- **Tools**: `axiomadoc` (literate programming), VM mode, Wails GUI, web GUI, Jupyter kernel.

1.2.2 Influences

Axioma stands in a long line of languages and gratefully adapts the best of them, just as most languages do. For the record, the main lineages are:

- **REBOL** — the `:` value binding, the family of scalar value literals (URL, email, file, money, pair, issue, …), the get-word (`:w`), refinements (`name/ref`), and the value-returning (non-throwing) error model.
- **Forth / Pop-11** — the stack model: the global interpreter stack, the postfix sequence notation, and the stack-shuffle verbs.
- **SETL** — set-theoretic data structures, comprehensions, bags, and the `om` / `Ω` undefined value.
- **Lisp / Scheme** — the `'` quote (the `'w` word literal), homoiconicity (code as data), and the S-expression view of the AST; **Mathematica** — `fullform` for viewing the AST as a symbolic expression.
- **Prolog / Datalog** — relational facts, rules, and queries.

- **F-logic & Description Logic** — the concept system: frame slots and the concept relation rule duality (F-logic), and subsumption / satisfiability reasoning over a tableau (Description Logic / ALC).
- **Python, Haskell, Racket, Rust, Swift** — assorted surface conveniences (comprehension spellings, string-escape and byte syntax, list accessors).

1.3 2. Getting Started

1.3.1 Installation

```
git clone https://github.com/vevenhar/axiomalang.git
cd axiomalang
go build -o axioma .
./axioma                          # Start REPL
```

1.3.2 Run a script

```
./axioma scripts/examples/hello.ax        # Run a script
./axioma --no-kb script.ax                # Skip Cascade KB preload (~10× faster startup)
./axioma --vm script.ax                   # Run in VM mode
./axioma --mcp                            # Start MCP server (stdio JSON-RPC)
```

1.3.3 First steps

```
axioma> a: {1, 2, 3}
{1, 2, 3}
axioma> b: {2, 3, 4}
{2, 3, 4}
axioma> a union b
{1, 2, 3, 4}
axioma> concept Person
axioma> Person has name: "Alice"
axioma> parent("John", "Mary")
axioma> {Y | Y <- parent("John", Y)}
{"Mary"}
```

1.4 3. Language Fundamentals

1.4.1 Bindings — one canonical form

Axioma has **a single canonical value-binding operator**: `:`. It's the shortest binding form of any mainstream language (two characters of overhead), puts the name first, and composes naturally with type annotations.

```
x: 5                              # bare binding
radius: 3.14 * 2                  # expression on the RHS
a, b, c: 1, 2, 3                  # multi-assignment (parallel)
d :: Day: Mon                     # with type annotation
```

All variants compile to the same `LetStatement` AST node. The `axioma/beginner` subset uses only the bare form; multi-assign and type annotations show up later in the curriculum.

> **`=` is a find-or-update synonym of `:`.** `=` declares a name if it is absent and updates it if present — for **any** name and case — so textbook mathematics reads verbatim: `B = {2, 4, 6}`, `C = A`

`union B`, `Pi = 3.14159`. `:` stays the canonical binding (and the form to learn first); `=` is the math/textbook spelling. (`=` is no longer strict-update-only.)

Casing is context-local, not a global type tag. Both `:` and `=` admit a name of **any case** — `B: {2, 4, 6}` and `B = {2, 4, 6}` are set variables. The **right-hand side** decides concept-vs-value: a `concept { ... }` RHS names a Concept (uppercase required — `Stock: concept { ... }`; a lowercase LHS with a `concept` RHS errors), while any other value binds an ordinary variable. Uppercase still means a **logic variable** inside rules (the Prolog convention) and is the **Concept-naming** convention — it just no longer *forces* a top-level binding to be a Concept.

`A` and `E` are ordinary identifiers, not quantifier shorthands. Quantify with `forall` / `exists`, the glyphs ∀ / ∃, the backtick digraphs `` `forall `` / `` `exists ``, or `!` (unique existential). `E!` (the free-logic existence predicate) is a separate token.

There is no `let` or `:=`. Bind with `:` (canonical) or `=`. Writing `let x = value` or `x := value` is a syntax error with an inline hint pointing at `x: value`.

1.4.2 Three ways to introduce a name

`:` is the canonical *value* binding; two further keywords introduce a name **lazily**. They differ in *when* the right-hand side runs and *what* they can name:

Form	RHS evaluation	Use for
`x: expr` (or `x = expr`)	**eager** — runs at the binding	ordinary values
`declare x = expr`	**lazy** — runs on first use, then **memoized** (computed once, then cached, so later reads don't re-run it)	deferred / expensive values; `/persist` · `/transient` control session persistence
`define …`	**lazy + memoized** (value form), or a *definition*	concepts, types/schemas, and knowledge claims

```
x: 1 + 2                                  # eager - computed now
declare big = slow_query()                # lazy - runs on first read of `big`, then cached
define concept Point = { x: 0, y: 0 }     # a definition, not a value
```

`:` evaluates now; `declare` and `define` defer until the value is first needed and then reuse the cached result. Beyond that, `declare` adds the persistence refinements, while `define` is the gateway to the definition / speech-act family — `define concept` / `define axiom` / `define postulate` / `define theorem` / `define word` / `define dialect` (see §13).

1.4.3 Guarded identifiers and atoms

To use a **reserved word, a multi-word name, or punctuation** as an identifier, guard it with `$`:

```
$forall: 5                        # `$name` - a reserved word used as a plain name
$"interest rate": 0.0525          # `$"..."` - spaces / punctuation in a name
$"GDP growth %": 2.1
```

`$` disambiguates by what follows it: a **digit** keeps the money literal (`$5`, `$5.00`); a `"` opens a quoted guard; a letter opens a bare guard. The `${...}` interpolation form inside strings is untouched — guards never use it.

Symbolic set elements (atoms) use the self-denoting word literal `'name`, and **cardinality** is `len(...)`:

```
A = {'p, 'q, 'r}                  # a set of three atoms
'q in A                           # → true
len(A)                            # → 3   (the cardinality of A)
```

1.4.4 Persistence refinements

The `:` operator deliberately has **no refinement slot** — adding `/persist` to `:` would force three-token lookahead on every identifier parse, and `:` is one of the highest-frequency tokens in the language. For persistent value bindings, use the dedicated `declare` form:

```
declare/persist counter = 0     # Saved to .axioma_session.bin
declare/transient temp = 42     # Discarded at session end
declare scratch = 0             # No refinement - uses the mode default
```

Default: **REPL persists, scripts are transient.**

Note the operator: `declare` uses `=` (the `ASSIGN` token), not `:`. This keeps the canonical `:` form refinement-free and concentrates the persistence vocabulary in one place. The same `/persist` / `/transient` refinements apply to `axiom`, `postulate`, and `define`:

```
axiom/persist   gravity_constant = 9.81
postulate/transient working_hypothesis = "..."
```

Persistence-controlled bindings are written `declare/persist counter = 0` (and `declare/transient`).

See `resources/docs/claude/REFINEMENT_PERSISTENCE.md` for the full matrix.

1.4.5 Statement vs. expression syntax

Most language constructs come in **dual forms**:

Operation	Statement form	Expression form
Property assignment	`usa.gdp: 27000`	`usa[gdp -> 28000]` (returns the value)
Concept definition	`concept Stock`	`s: a Stock {}`
Fact assertion	`assert parent("a", "b")` (relation declared)	`insert("parent", "a", "b")`
Retraction	`retract [...]`	`forget("parent", "a", "b")`

Use the natural-language statement form for **concept lifecycle and properties**, the functional/expression form for **values and computation**. See the design philosophy in `CLAUDE.md`.

1.4.6 Typing the glyphs

Axioma's math notation (→ …) is always **optional** — the operators have plain word twins (`in`, `union`, `intersect`, `subset`, `superset`, `subsetneq`/`supsetneq` for the proper forms , `forall`, `exists`, `lambda`, `and`, `or`, `implies`, `!=`), so no setup is ever required. The exceptions are the description-logic pair , which stay glyph/digraph-only. When you *do* want the glyphs, pick whichever input layer fits where you're typing — all of them resolve the same names from the same catalog (the one `symbols()` prints), so a spelling learned once works everywhere:

Where you're typing	How	Example
Any source file, any editor	**backtick digraph** — pure ASCII that *lexes as the glyph*	`C `sqcap D C D` (no word form exists); `forall
Canonicalize a whole file	`axioma --glyphify file.ax` (inverse: `--asciify`)	`x in U and P` → `x  U  P`
The REPL (and wizards)	type `\in` then **Tab**	`\forall` →
VS Code / Cursor / Neovim / Helix	`\in` + accept the completion (served by `axioma-lsp`)	`\cup` →
Browser playground	`\in` then **Tab**	`\subseteq` →

Where you're typing	How	Example
Anywhere on your system	the generated espanso pack (`resources/espanso/axioma-symbols.yml`)	`:in:` →
macOS, no installs	System Settings → Keyboard → Text Replacements; or Unicode Hex Input (`2208` →)	

Names are LaTeX first (`` `cup ``, `\subseteq`), then Axioma's canonical names (`` `union ``), then word aliases — ~75 spellings. Discover them from inside the language: `symbols()` lists the whole catalog with LaTeX names and meanings; `glyph("cup")` looks one up.

Notes. The digraph leader is a **backtick**, not the Agda/Julia backslash, because `\` is Axioma's set-difference operator; a backtick outside strings and comments was previously a syntax error, so the form is collision-free. A digraph is byte-identical to its glyph at the token level, so it works in every construct, in the VM, and in the playground. `--glyphify` is token-aware (strings and comments are never touched) and verifies the rewrite lexes and parses identically before writing; conversions only happen where both spellings produce the same token, so `not`/`¬` and value literals like `true`/`false`/`om` are deliberately left alone.

1.5 4. Data Types

1.5.1 Primitives

Type	Examples	Notes
`Integer`	`42`, `-17`, `0`, `0xFF`, `0b1010_1100`, `0o755`, `1_000_000`	`int64`; hex/binary/octal prefixes; underscore separators
`Float`	`3.14159`, `-2.5`, `0.75`, `1.5e6`, `3.14e-2`, `1E+9`	IEEE 754 double; scientific notation `e`/`E` ± optional sign
`String`	`"hello"`, `"unicode:  "`, `"\u{2203}"`, `r"raw \n"`	UTF-8; escape sequences + `r"..."` raw prefix — see Strings
`Boolean`	`true`, `false`	Classical two-valued
`Byte`	`byte(0xFF)`	Single byte 0..255; distinct from `Integer`. See Binary data
`Bytes`	`b"hello"`, `b"\xff\x00"`, `bytes(0x48, 0x69)`	Immutable byte sequence

1.5.2 Dual null types

Axioma has **two distinct null-like values** with different semantics — they are **not interchangeable**.

Value	Semantics	Truthiness	Display	Use for
`none`, `null`	Absent / nil / missing	**Falsy**	`null`	Uninitialized data, missing fields
`om`, `Ω`	SETL "omega" — a value that exists but is undetermined	**Truthy**	`Ω`	Database unknowns, undefined computations

```
none == om             # false - never interchangeable
if none then ...       # never executes
if om   then ...       # executes (om is truthy)
```

See `resources/docs/claude/NULL_TYPES.md` and `SETL_DATABASE_SYSTEM.md`.

1.5.3 Multi-valued logic types

Constructed via builtins; participate automatically in operator dispatch (priority **Belnap** > **Intuit3** > **Łukasiewicz** > **Kleene** > **Boolean**). See §9.

```
half: lukasiewicz(0.5)
contradiction: belnap("both")
unknown_g3: intuit3("unknown")
```

1.5.4 Strings — escape sequences, raw form, codepoint builtins

`String` literals are written between double quotes (`"…"`) or single quotes (`'…'`). They are stored as UTF-8, so Unicode characters can appear directly in source:

```
greeting: "hello, world"
math:     " x  y. x + y = 0"
greek:    "   Ω"
```

For cases where the character can't be typed directly, escape sequences decode at parse time:

Escape	Result
`\n \t \r`	LF, TAB, CR
`\\ \" \' \0`	literal \, ", ', NUL
`\u{H...H}`	Unicode codepoint, 1–6 hex digits, full U+0000–U+10FFFF

```
"line1\nline2"           # 2 lines (LF in the middle)
"tab\there"              # 3 fields separated by TAB
"quote: \"hi\""          # quote: "hi"

"\u{2203}"               # →    (BMP - 4 hex digits)
"\u{1D54A}"              # →    (math S - 5 hex digits, supplementary plane)
"\u{1F600}"              # →    (emoji - 5 hex digits)
"\u{10FFFF}"             # →    (max valid Unicode codepoint)
```

The braced `\u{H...H}` form takes 1–6 hex digits. The 4-hex `"\uXXXX"` form is intentionally **not** supported — it only reaches the Basic Multilingual Plane and forces a second `\U00000000` escape for everything above U+FFFF. The single braced form handles the entire codepoint range.

Surrogate codepoints rejected. `\u{D800}` through `\u{DFFF}` are rejected at parse time because they cannot appear in valid UTF-8. `chr(N)` enforces the same check at runtime.

Unknown escapes are preserved verbatim. `"regex \d+"` keeps `\d` as a literal `\` followed by `d` — a back-compat hatch for strings that contain regex metacharacters. Define a proper escape only when needed; everything else passes through.

1.5.4.1 Raw strings — `r"..."` and `r'...'`

The `r` prefix bypasses escape decoding entirely. Useful for paths, regex patterns, and any genuinely-literal content:

```
winpath:  r"C:\Users\alice\new"      # literal path - no \U, \n decoding
pattern:  r"\d+\.\d+"                 # literal regex
raw_unicode: r"\u{2203}"             # → 8 chars: \, u, {, 2, 2, 0, 3, }
```

The result is byte-identical to the source between the quotes. `r"…"` and `"…"` produce the same `*ast.StringLiteral` AST node — the raw form just skips the decode pass.

1.5.4.2 Triple-quoted and interpolated strings

Both also decode escapes:

```
multi: """
line one
line two \u{2203} ${some_var}
"""

x: 42
interp: "value = ${x}, glyph = \u{2200}"
```

1.5.4.3 Codepoint builtins — `chr` and `ord`

For programmatic codepoint construction (when the literal form can't help because the value comes from runtime):

```
chr(8707)              # → "∃"
chr(0x1F600)           # → "😀"
ord("A")               # → 65
ord("∃")               # → 8707
ord(chr(N)) == N       # round-trip for any valid codepoint
```

Both accept the full Unicode range U+0000..U+10FFFF; surrogates are rejected; `ord("")` errors with "empty string has no codepoint".

Form	In Axioma
`"\n"` `"\t"` etc.	decoded
`"\u{H...H}"` (1–6 hex, braced)	decoded
`"\uXXXX"` (4-hex, no braces)	use `"\u{XXXX}"`
`"\xHH"` (byte hex)	use `b"\xHH"` for bytes
`"\N{NAME}"` (Unicode name)	(future, optional)
`r"..."` / `r'...'`	raw
`chr(N)` / `ord(s)`	

1.5.4.4 Regular expressions — native `regex_*` builtins

Five native builtins wrap Go's RE2 engine (linear-time, no catastrophic backtracking). Argument order is **`(subject, pattern)`** throughout. An invalid pattern returns a catchable `Error` value rather than crashing the host.

Builtin	Returns	Example → result
`regex_match(s, pat)`	`Boolean`	`regex_match("a1b2", "[0-9]")` → `true`
`regex_find_all(s, pat)`	`Array<String>`	`regex_find_all("a1b2c3", "[0-9]")` → `["1","2","3"]`
`regex_replace(s, pat, repl)`	`String`	`regex_replace("John Smith", "(\\w+) (\\w+)", "$2 $1")` → `"Smith John"`
`regex_split(s, pat)`	`Array<String>`	`regex_split("one  two", "\\s+")` → `["one","two"]`
`regex_captures(s, pat)`	`Array<String>`	`regex_captures("2026-06-14", "(\\d+)-(\\d+)-(\\d+)")` → `["2026-06-14","2026","06","14"]`

`regex_replace` supports `$1`/`$2` backreferences; `regex_captures` returns group 0 (the whole match) followed by each capture group, or `[]` on no match. Use a raw pattern (`r"\d+\.\d+"`) to avoid double-escaping. A typical tokenize-then-convert pipeline:

```
nums: regex_find_all("temp 38.5 hr 72 sat 0.97", "[0-9.]+")   # ["38.5", "72", "0.97"]
float(nums[1]) > 38.0                                          # → true
```

1.5.5 Binary data — `Byte` and `Bytes`

Distinct from `Integer` and `String` so the type system can dispatch byte-specific operations and so `bs[0] == 0xff` reads as a `Byte`/`Byte` comparison rather than implicit coercion. The cost is verbosity, the win is no silent UTF-8 corruption.

```
# Three construction forms
b1: byte(0xFF)                # single byte, 0..255 - errors otherwise
b2: bytes(0x48, 0x69)         # variadic - each arg 0..255 or Byte
b3: bytes([72, 105])          # from Integer array
b4: bytes("Hi")               # from String (UTF-8 byte view)
b5: b"Hi"                     # b"..." literal
b6: b"\xff\x00\x01"           # hex escapes (also \n \r \t \\ \" \0)
```

Form	Result
`byte(0xFF)`	`Byte(255)` — explicit narrowing
`bytes(...)`	`Bytes` from variadic, array, or String
`b"..."`	Literal — escape-decoded at parse time
`int(b)`	Widen `Byte → Integer`

1.5.5.1 Operations

```
# Length, indexing (1-based), slicing, concat, equality
len(b"hello")                # 5
b"hello"[1]                  # Byte(104)  - that's 'h'
b"hello"[2:4]                # b"ell"
b"AB" + b"CD"                # b"ABCD"
b"AB" == bytes(0x41, 0x42)   # true
```

1.5.5.2 Conversions (explicit + fallible)

Function	Returns	Errors when
`bytes_to_hex(bs)`	`"ff00ab"`	never
`hex_to_bytes(s)`	`Bytes`	input has odd length or non-hex chars
`bytes_to_string(bs, "utf-8")`	`String`	bytes aren't valid UTF-8
`string_to_bytes(s, "utf-8")`	`Bytes`	encoding unknown
`base64_encode(bs)` / `base64_decode(s)`	round-trip	decoder errors on bad input
`read_bytes(path)` / `write_bytes(path, bs)`	file I/O	path missing / permission

1.5.5.3 Bitwise ops — word-form infix (v3) + functional form Symbolic bitwise operators (`& | ^ << >>`) all conflict with existing Axioma syntax (`&` is address-of, `|` is comprehension separator, `^` is `POWER`). Word-form operators sidestep the conflict and match Axioma's pattern of `and` / `or` / `not` / `union` / `intersect`.

```
# Word-form infix (precedence: SUM - same as +/-, binds tighter than ==)
0xF0 band 0x0F                  # 0
0xF0 bor  0x0F                  # 255
```

```
0xFF bxor 0x0F                    # 240
1    bshl 4                       # 16
0x80 bshr 4                       # 8
0xFF band 0x0F == 0x0F            # true - (0xFF band 0x0F) == 0x0F

# On two Bytes: stays Byte for &|^, also for shifts
byte(0xF0) band byte(0x0F)        # Byte(0x00)
byte(1) bshl byte(4)              # Byte(0x10)

# Mixed Byte/Integer: widens to Integer
byte(1) bshl 4                    # Integer(16)

# Functional form (still available - useful when you need a callable)
bit_and(byte(0xF0), byte(0x0F))   # Byte(0x00)
bit_or(byte(0xF0), byte(0x0F))    # Byte(0xFF)
bit_xor(byte(0xFF), byte(0x0F))   # Byte(0xF0)
bit_not(byte(0x0F))               # Byte(0xF0)
bit_shl(byte(1), 4)               # Byte(0x10)
bit_shr(byte(0x80), 4)            # Byte(0x08)
reduce(bit_or, byte(0), bytes_to_list(bs))   # fold OR over bytes
```

Shift counts must be 0..63. Shift by a larger amount errors rather than wrapping.

Arithmetic on `Byte` widens to `Integer` (no overflow surprise — explicit `byte((a+b) % 256)` to wrap):

```
byte(200) + byte(100)              # Integer(300) - wider than 255
byte((int(byte(200)) + 100) % 256) # Byte(0x2c) = 44 - wrap explicitly
```

1.5.5.4 Integer literal prefixes

Adding bytes also brought standard hex / binary / octal literals to the language:

```
0xFF               # 255
0xFF_AB            # 65451  - underscores for readability
0b1010_1100        # 172
0o755              # 493
1_000_000          # 1000000 - underscores work in decimal too
```

These produce `Integer`, not `Byte` — wrap with `byte(0xFF)` for the byte form.

1.5.5.5 AST inspection

```
fullform(b"AB")             # "Bytes(65, 66)"
headof(b"AB")               # "Bytes"
argsof(b"AB")               # ["65", "66"]
fullform(byte(0xFF))        # "byte(255)"  - AST of the call, not the value
```

1.5.5.6 Design notes

- **Encoding-aware separation**. `Bytes` doesn't carry encoding metadata; conversion to `String` is explicit and fails on invalid input.
- **Immutable.** Operations return new `Bytes` rather than mutating in place — composes cleanly with comprehensions, rule derivation, and the VM's bytecode constant pool.
- **1-based indexing** matches `Array` / `String` / `Tuple`.
- **No infix bitwise ops in v1** — adding them would touch lexer + parser + VM. Functional form unblocks bit work now; infix sugar can come later.

- **Bit pattern matching** (Erlang's `<<v:4, len:16, payload/binary>>` — the most expressive byte primitive of any language) is provided in v2 as `pack` / `unpack` builtins using Python's `struct`-style format strings (see next subsection).

See `tests/axioma/bytes/test_bytes.ax` for a 46-assertion smoke test that runs identically under both the evaluator and `--vm`.

1.5.5.7 Binary serialization — `pack` / `unpack` Python-`struct`-style format strings. `pack` serializes values into `Bytes`; `unpack` reads `Bytes` back into a `Tuple` of typed values. The format spec is small enough to memorize:

```
bs: pack(">BBH", 1, 2, 256)        # b"\x01\x02\x01\x00" (4 bytes)
header: unpack(">BBH", bs)
println(header[1], header[2], header[3]) # 0x01 0x02 256

# TCP-header-style parse: port (u16), length (u16), flags (u16), seq (u32)
packet: b"\x04\xd2\x00\x10\x00\x20\x00\x00\x00\x01"
parts: unpack(">HHHI", packet)
println(parts[1], parts[2], parts[3], parts[4])  # 1234 16 32 1
```

Format string grammar: - **Endian prefix** (optional): `<` little, `>` big, `!` network (= big), `=` "native" (= big). Default is big-endian. - **Type codes** (each optionally preceded by a count):

Code	Size	Pack from	Unpack to	Notes
`b` / `B`	1	Integer / Byte	Integer / Byte	signed / unsigned 8-bit
`h` / `H`	2	Integer	Integer	signed / unsigned 16-bit
`i` / `I`	4	Integer	Integer	signed / unsigned 32-bit
`q` / `Q`	8	Integer	Integer	signed / unsigned 64-bit
`f`	4	Float	Float	IEEE 754 single
`d`	8	Float	Float	IEEE 754 double
`s`	N	String / Bytes	Bytes	counted byte field, pads/truncates
`c`	1	Bytes(1)	Bytes(1)	single-byte field
`x`	1	(none)	(none)	pad byte — consumes 0 values

- **Count prefix:** `4B` = four unsigned bytes (consumes 4 args). For `s`, the count is the byte-length of the field: `4s` packs/unpacks a 4-byte string field.

Out-of-range values error rather than silently wrap:

```
pack(">B", 300)            # ERROR: pack 'B': value 300 out of range 0..255
pack(">b", 200)            # ERROR: pack 'b': value 200 out of range -128..127
```

Round-trip identity:

```
unpack(">d", pack(">d", 3.141592653589793))[1]   # 3.141592653589793
unpack(">q", pack(">q", -9_000_000_000))[1]      # -9000000000
```

See `tests/axioma/bytes/test_pack_unpack.ax` for the full 48-assertion smoke test.

1.5.5.8 Slicing (v2)

`Bytes` slicing now works under `--vm` (the `OpSlice` opcode added in v2 also unlocked slicing for `String`, `Array`, and `Tuple` in the bytecode engine):

```
b"Hello World"[1:5]          # b"Hello"
b"hello"[3:]                 # b"llo"  - open end
b"hello"[:3]                 # b"hel"  - open start
```

1.5.5.9 Endian-aware read/write at offset (v3)

Offset-style protocol parsing sugar over `pack`/`unpack`. Each builtin reads or writes a single field of fixed width at a given **1-based** offset (matching Axioma's indexing convention). Reads return values; writes return a **new** `Bytes` (originals are immutable).

Builtin	Returns	Notes
`read_u16_be(bs, off)` / `read_u16_le(bs, off)`	Integer 0..65535	unsigned 16-bit
`read_i16_be` / `read_i16_le`	Integer ±32767	signed 16-bit (sign-extended)
`read_u32_be` / `read_u32_le`	Integer 0..2^32-1	unsigned 32-bit
`read_i32_be` / `read_i32_le`	Integer ±2^31	signed 32-bit
`read_u64_be` / `read_u64_le`	Integer	may wrap to negative for > int64 max
`read_i64_be` / `read_i64_le`	Integer	signed 64-bit
`read_f32_be` / `read_f32_le`	Float	IEEE 754 single
`read_f64_be` / `read_f64_le`	Float	IEEE 754 double
`write_*` (same suffix family)	Bytes	new copy with field overwritten

```
# Parse a 10-byte packet (u16 port, u16 length, u16 flags, u32 seq)
packet: b"\x04\xd2\x00\x10\x00\x20\x00\x00\x00\x01"
port: read_u16_be(packet, 1)    # 1234
length: read_u16_be(packet, 3)    # 16
flags: read_u16_be(packet, 5)    # 32
seq: read_u32_be(packet, 7)    # 1

# Build a response - start with zeros, overwrite fields
resp: bytes(0, 0, 0, 0, 0, 0, 0, 0)
resp1: write_u16_be(resp, 1, port)
resp2: write_u16_be(resp1, 3, length)
# resp is still b"\x00\x00\x00\x00\x00\x00\x00\x00" - original untouched.
```

These compose with `pack`/`unpack`: round-trip identity holds. Use them when you want to read individual fields at known offsets without computing slice ranges, or when you want to mutate a buffer in protocol-style.

See `tests/axioma/bytes/test_bitwise_endian.ax` for a 39-assertion v3 smoke test that runs identically under both engines.

1.5.6 Scalar value types & literals

Axioma has a family of **lexer-recognized scalar literals** — you write a URL, an e-mail address, a file path, a date, money, or a 2-D pair directly, and the lexer gives it a first-class type. `type()` returns a TitleCase string that agrees with the `@` sigil and the primitive-type concepts (`@v`, `type(v)`, and `v is T` all match).

Type	Literal	`type()`	Predicate	Notes
`URL`	`https://example.com/path`	`"URL"`	`is_url`	`http://` / `https://`; `read()` fetches it
`Email`	`user@example.com`	`"Email"`	—	`local@domain` shape
`File`	`%data/file.txt`	`"File"`	—	`%` + path; **relative** (see the gotcha below)
`Date`	`2026-05-02`, `1-Jan-2024`, `7/2/26`	`"Date"`	—	several spellings
`Time`	`12:34:56`	`"Time"`	—	`HH:MM:SS`
`Money`	`$123.45`	`"Money"`	`is_money`	`$` + digit
`Pair`	`100x200`	`"Pair"`	`is_pair`	2-D integer pair `XxY`
`Issue`	`#123`	`"Issue"`	—	`#` + **digit**
`Percent`	`42%`	`"Percent"`	`is_percent`	number + `%`
`Word`	`'hello`	`"Word"`	`is_word`	self-evaluating symbol
`GetWord`	`:name`	`"GetWord"`	—	reads a value without evaluating
`QuoteWord`	`#hello`	`"QuoteWord"`	—	`#` + **letter**

```
type(https://example.com)   # → "URL"
type($123.45)               # → "Money"
type(100x200)               # → "Pair"
type(42%)                   # → "Percent"

is_url(https://example.com) # → true
is_pair(100x200)            # → true
is_url($5)                  # → false
```

Sigil disambiguation — the next character decides. `$` + digit is `Money` (`$5`), otherwise `$name` / `$"…"` is a guarded identifier. `%` + alphanumeric is a `File` (`%data`), otherwise `%` is the modulo operator. `#` + digit is an `Issue` (`#123`), `#` + letter is a `QuoteWord` (`#hello`).

1.5.6.1 Arithmetic on `Pair` / `Money` / `Percent`

```
100x200 + 20x30   # → 120x230   (component-wise)
100x200 * 2       # → 200x400   (scalar scale)
$100 + $25        # → $125
$100 * 1.5        # → $150
10% * 200         # → 20        (percent of a number)
10% * $200        # → $20       (percent of money)
```

1.5.6.2 Words — `'w`, `:w`, `#w`

A word *is* a value (a symbol), distinct from a variable you look up.

```
'hello            # → 'hello    (natural word - self-evaluating)
x: 42
:x                # → 42        (get-word - reads the bound value, no call)
#tag              # → #tag      (quote-word - a literal symbol)
```

> **`:x` is the get-word; `@x` is *not*.** `@` is the type-of sigil, so `@x` returns `"Integer"` (the *type* of `x`), while `:x` returns `42` (the *value*). Use `:x` to read a value without evaluating it as a call.

1.5.6.3 `read` and file I/O

`read(source)` is the file/URL read verb — one argument, returns a `String`:

`source`	Example	Behavior
`String` path	`read("/tmp/notes.txt")`	read the file
`String` / `URL` http(s)	`read(https://example.com)`	HTTP GET → body
`%file` literal	`read(%data/notes.txt)`	read the (relative) file

```
f: "/tmp/notes.txt"
write(f, "Hello from Axioma!")   # → true
file_exists(f)                   # → true
read(f)                          # → "Hello from Axioma!"
append(f, " More.")              # → true
remove(f)                        # → true   (delete the file)
file_exists(f)                   # → false
```

`write` / `append` return a `Boolean`; non-string content is stringified. A failed `read` (missing file, network error) returns a catchable `Error` value, not a crash. For a richer path/line API, `import "builtin:io" as IO` exposes `IO.read_file`, `IO.read_lines`, `IO.join_path`, etc.

> **Naming gotchas (reserved-word collisions).** - **Existence is `file_exists(path)`, not `exists(...)`** — the bare word `exists` lexes as the existential quantifier ( ), so `exists("/p")` is a parse error. `file_exists` matches the io package's `IO.file_exists`. - **Deletion is `remove(path)`, not `delete(...)`** — `delete` is the reserved retract keyword. - **`Tag` (`<tag>`) is shadowed** by the natural-language `<…>` literal, so `type(<html>)` is `"NaturalLanguage"`, not `"Tag"` — treat `<tag>` as unavailable from source. - **Absolute `%/abs/path` does not lex** (the character after `%` must be alphanumeric); use the `String` form `read("/abs/path")` for absolute paths.

Runnable tour: `tests/axioma/rebol/rebol_data_types_showcase.ax`. Assertion tests: `tests/axioma/rebol/test_rebol` and `tests/axioma/io/test_io_functions.ax`. Full reference: `resources/docs/Data Types.md`.

1.6 5. Collections & Stacks

1.6.1 Arrays

```
xs: [1, 2, 3]
xs[0]               # 1
len(xs)             # 3
```

A **trailing comma** is allowed, and it is the way to write a **one-element array** unambiguously: `[x,]`. This matters because `[...]` is also a block body (in `if cond then [body] else [body]` the branch brackets are blocks, so a bare single-statement `[x]` *evaluates to* `x`). The comma forces the array reading in every position:

```
[5,]                              # → [5]        (a 1-element array)
["w1",]                           # → ["w1"]     (len 1 - NOT the string "w1")
[1, 2, 3,]                        # → [1, 2, 3]  (trailing comma in n-arrays too)
if c then ["a", "b"] else ["w1",] # else-branch is a 1-element array
if c then [42] else [0]           # bare [x] → the scalar 42 / 0 (block body)
```

1.6.2 Tuples

```
pair: (3, 4)
trip: ("Alice", 30, "engineer")
```

1.6.3 Sets

Sets are **unordered** collections of **unique** elements, written with braces. Duplicates collapse and order is irrelevant — `{3, 1, 2, 2}` *is* `{1, 2, 3}`, and `{1, 2, 3} == {3, 2, 1}`. The empty set is `{}` (equivalently `set()`, `emptyset`, or); `len(s)` is the cardinality.

```
{3, 1, 2, 2, 1}          # → {1, 2, 3}    (deduped, unordered)
{1, 2, 3} == {3, 2, 1}   # → true         (order-independent)
{} == emptyset           # → true         ; set() ==   → true
len({1, 2, 3})           # → 3
```

Algebra — each operation has a word form and a glyph, and yields a set:

Operation	Word	Glyph	`a · b →`
union	`union`		`{1, 2, 3, 4, 5, 6, 7}`
intersection	`intersect`		`{3, 4, 5}`
difference	`difference`	`\`	`{1, 2}`
symmetric difference	`symdiff`	(also)	`{1, 2, 6, 7}`

```
a: {1, 2, 3, 4, 5}
b: {3, 4, 5, 6, 7}
a union b        #     → {1, 2, 3, 4, 5, 6, 7}
a intersect b    #     → {3, 4, 5}
a difference b   # \   → {1, 2}
a symdiff b      #     → {1, 2, 6, 7}   (in exactly one of the two)
```

Relations & membership — each returns a `Boolean`:

Test	Word	Glyph	True when
membership	`in`		the element is in the set
non-membership	`notin`		the element is not in the set
subset	`subset`		every element of the left is in the right
superset	`superset`		the right is a subset of the left
proper subset	`subsetneq`	( )	subset **and** not equal
proper superset	`supsetneq`	( )	superset **and** not equal
equality	`==`	—	same elements (order-independent)

```
2 in a               # → true    ;  9 notin a → true
{1, 2} subset a      # → true    ;  a superset {1, 2} → true
{1, 2} subsetneq a   # → true (proper) ;  a subsetneq a → false
2  a   and   {1, 2}  a    # → true    (glyphs work everywhere the words do)
```

Powerset & Cartesian product:

```
powerset({1, 2})                      # → all 4 subsets: {}, {1}, {2}, {1, 2}
{(x, y) | x <- {1, 2}, y <- {3, 4}}   # Cartesian product via a comprehension →
                                      #   {(1, 3), (1, 4), (2, 3), (2, 4)}
```

See also. Set **comprehensions** and **quantifiers** — `{x | x <- s, p(x)}`, `forall` / `exists` over a set — are covered in §7. The **Ellipsis sets** subsection just below covers textbook and **infinite** sets (`{2, 4, 6,`

`...}`) and the built-in number sets . Draw set relationships as a diagram with `venn(...)` — see §23.

1.6.3.1 Ellipsis sets — textbook `{2, 4, ..., 100}` / `{2, 4, 6, ...}` The `...` ellipsis writes an arithmetic progression the way a textbook does. A bounded form materializes a set; an open form (no upper bound) is a **lazy infinite set**. The step is inferred from the leading terms — a third term, if given, must confirm it.

```
{2, 4, ..., 100}        # → {2, 4, …, 100} - a 50-element set
{1, ..., 10}            # step defaults to 1 → {1..10}
{10, 8, ..., 2}         # descending (step -2)

B: {2, 4, 6, ...}       # infinite (lazy)
first(B)                # → 2    ;  tenth(B) → 20  ;  nth(B, 50) → 100
first(B, 5)             # → {2, 4, 6, 8, 10}
100 in B                # → true ;   7 in B → false
first((x | x <- B, x > 10), 3)       # → [12, 14, 16]   (a filtered lazy stream)
```

Integer progressions only (non-integer or non-linear → a clear error pointing at an explicit generator like `{x | x <- range(...)}`); bounded sets cap at 1,000,000 elements. For an *ordered* infinite sequence use this form or `infinite_set("naturals")` — a bare `naturals` / generator enumerates unordered.

Pulling elements. `first(s)` is the first term; the ordinals `second(s)` … `tenth(s)` and the general `nth(s, k)` give the k-th — on a finite collection or an infinite set (`tenth({2,4,6,...})` → 20). `last(s)` errors on an infinite set (no last element). Membership uses `in`: ellipsis and named sets test it exactly and instantly, while a *custom* function-defined set (`infinite_set(func(n) [...])`) does a bounded generate-and-check (up to 100k terms, with an early exit once an ascending sequence passes the target) — so `9 in infinite_set(func(n) [n * n])` is `true` without ever running away.

The standard number sets. The chain is built in. The identifiers `rationals` / `reals` / `complexes` and the glyphs / / are membership sets:

```
3 in reals               # true   (an integer is real…)
2.5 in reals             # true   ; 2.5 in rationals → false  (a decimal reads as a real)
rational(1, 2) in rationals   # true
complex(0, 1) in complexes    # true ; complex(0, 1) in reals → false (nonzero imaginary part)
first(rationals, 5)      # {0, 1, -1, 1/2, -1/2}   (  is countable → enumerable)
first(reals, 5)          # ERROR:  is uncountable - use a membership test instead
```

Membership is type-faithful: an `Integer` belongs to all four; a `RationalNumber` to / / ; a `Float` reads as a **real, not a rational** (decimals approximate reals — assert rationality with a `rational(p, q)` literal); a `ComplexNumber` belongs to , and to only when its imaginary part is 0. is countable, so `first(rationals, n)` enumerates it (Calkin–Wilf order, signs interleaved); and are uncountable and so are membership-only (enumeration is a clean error). `infinite_set("rationals" | "reals" | "complexes")` builds the same sets.

1.6.4 Bags (multisets)

A **Bag** is an unordered collection in which the same value may appear multiple times. Bags sit between Sets (no duplicates) and Arrays (ordered duplicates):

	Order	Duplicates	Multiplicity
Array	yes	yes	positional
Set	no	no	n/a
Bag	**no**	**yes**	**counted**

```
words: bag(["the", "quick", "the", "fox", "the"])
count(words, "the")          # 3 - multiplicity of an element
len(words)                   # 5 - total Σ multiplicities
"the" in words               # true
distinct(words)              # {the, quick, fox} - underlying support set
```

Operators. Standard multiset algebra is wired infix:

```
b1   b2        # union - max counts per element
b1 + b2        # additive sum - counts add (independent observations)
b1   b2        # intersection - min counts
b1 - b2        # difference - clamped at zero
b1 == b2       # multiset equality
```

and + are genuinely different: `bag([a,a])  bag([a]) == bag([a,a])` but `bag([a,a]) + bag([a]) == bag([a,a,a])`.

Comprehension iteration is over distinct elements (the underlying Set support), since `{...}` cannot carry multiplicities. Use `to_array(b)` to iterate every occurrence.

As slot values. Bags can be slot values on Concepts; the `unify` machinery uses **additive sum** as the merge policy — two partial observations of the same entity combine their evidence.

Typical use cases: word-frequency counts, inventory, voting tallies, evidence merging. Bags are first-class in SETL, Z, B, VDM-SL, Smalltalk, Python's Counter, C++ `std::multiset`, and Guava `Multiset`; they fill the same role in Axioma.

Tests: tests/axioma/collections/test_bag_basic.ax, test_bag_operators.ax, test_bag_use_cases.ax, test_bag_slot_value.ax.

1.6.5 Dictionaries (`{key: value}` literals)

```
person: {name: "Anna", age: 24}              # Unquoted keys
obj: {"first-name": "Bob", "age": 30}     # Quoted keys
nested: {user: {address: {city: "NYC"}}}

person.name          # "Anna" (dot access)
person["name"]       # "Anna" (bracket access)
dict()               # Empty dictionary  ({} is the empty SET )

type(person)         # "Dictionary"
person is Dictionary # true
```

A **dictionary** is a string-keyed `{key: value}` map. Build one with the `{...}` literal or, for the empty case, `dict()`. `type()` and `@` report `"Dictionary"`, and `x is Dictionary` classifies it.

> **Naming note.** This type was previously named `ObjectMap` (June 2026 standardized it to `Dictionary` and **fully retired** the old name — `ObjectMap` is now an undefined word, not an alias). `Dict` is accepted as a short alias in type annotations (`x :: Dict`) and `is` checks, matching the `dict()` constructor, but `type()`/`@` always report the canonical `Dictionary`.

1.6.6 Stacks

First-class stack data type. See §18.

```
s: a Stack
s push 1
s push 2
s pop                # 2
```

1.7 6. Operators & Control Flow

1.7.1 Arithmetic

```
2 + 3       2 - 3       2 * 3       6 / 3       7 % 2       2 ^ 10
100 // 7    10.0 // 3.0                                     # INT_DIV (floor)
```

Each binary operator has a symbolic form, a keyword form, and a prefix-builtin form. They produce identical AST and identical results — pick by readability:

Operation	Symbolic	Keyword (infix)	Prefix builtin
Quotient (floor / trunc)	`//`	`div`	`quotient(a, b)`
Remainder	`%`	`mod` / `modulo`	`remainder(a, b)`
Both at once	—	—	`divmod(a, b) → (q, r)`

```
100 // 7      # → 14   (symbolic)
100 div 7     # → 14   (keyword)
quotient(100, 7)   # → 14   (prefix)

100 % 7       # → 2    (symbolic)
100 mod 7     # → 2    (keyword)
100 modulo 7  # → 2    (keyword longhand)
remainder(100, 7)  # → 2    (prefix)

divmod(100, 7)     # → (14, 2)   (combined - one division, both halves)
```

`/` vs `//` for floats. `/` preserves operand kind (int/int → int truncated; float/float → float real value); `//` always returns an integer-valued result via Go-native truncation (ints) or `math.Floor` (floats). The two differ for *all* float cases and for negative-result float cases in particular:

```
100.0 / 7.0   # → 14.2857…   (real value)
100.0 // 7.0  # → 14         (floor)
-7.0 / 3.0    # → -2.333…    (real value)
-7.0 // 3.0   # → -3         (floor toward -∞, NOT -2)
```

`mod` / `modulo` / `div` are **soft keywords**: lexed as plain identifiers and recognized as infix operators only when they sit between two expressions on the same line. Variables, hash keys, and function names with these spellings keep working (`mod: 99`, `h.div`).

1.7.2 Comparison

```
==   !=   <   >   <=   >=
```

1.7.3 Logical (auto-dispatched on operand type)

```
and    or    not    implies    iff
```

When operands are multi-valued logic instances (Belnap, Intuit3, Łukasiewicz, Kleene), the operators dispatch automatically. See §9.

1.7.3.1 `therefore` / `∴` — the conclusion connective

`p therefore q` (glyph `p ∴ q`) draws a conclusion, marking it with `[logical]` grounding at full strength — distinct from the truth-functional `implies`. The glyph and the word are byte-identical:

```
p  q                  # prints:  p →[logical] q  (strength: 1.00)
p therefore q          # identical to the glyph form
```

Type directly, with the backtick digraph `` `therefore `` / `` `qed `` / `` `thus ``, or via the REPL `\therefore`+Tab expansion.

1.7.4 Set operations

```
union    intersect    difference    symdiff    in       notin
subset   superset    subsetneq     supsetneq           #       word twins
                    \                                # glyph forms
                                                      # subset family (neq = proper/strict)
```

1.7.5 Conditional

```
if x > 0 then "positive" else "non-positive"

if x > 10 then [
    println("big")
] else if x > 0 then [
    println("small")
] else [
    println("non-positive")
]
```

Bracketed if-bodies are blocks, not arrays. Inside `then [...]` and `else [...]`, a single-expression body returns the expression's value, not a single-element array — `if cond then [42] else [99]` returns 42, not `[42]`. Comma-separated `[1, 2, 3]` and empty `[]` still parse as array literals everywhere. The same contextual rule applies to `match … | pat => [x]` arms.

1.7.6 Loops

```
# Pre-test: while
n: 1
while n <= 5 [
  println(n)
  n = n + 1
]

# Counted
repeat 5 [println("hi")]
repeat i <- [1..5] [println(i)]

# Pre-test (block-form condition)
n: 0
repeat [n < 3] [n = n + 1]

# Post-test (Pascal-style - body always runs at least once)
n: 0
repeat [
  n = n + 1
] until n >= 3

# Concept-extent / foreach
foreach d in Day [println(d.name)]
foreach v in [10, 20, 30] [println(v)]
```

`break` and `continue` work inside any loop form. `repeat`-style loops without `until` are pre-test (the count or condition is checked before each iteration); the `until` form is post-test (body runs at least once, the exit test fires after each iteration when `cond` becomes truthy).

1.7.7 Operator precedence (high → low)

1. Function application: `f(x)`
2. `^` (exponent), `not`, `&` (address), `*` (deref)
3. unary `-` (negation)
4. `*`, `/`, `%`, `//`, `mod`, `modulo`, `div`
5. `+`, `-`
6. Set: `union`, `intersect`, `difference`
7. Comparison: `<`, `>`, `<=`, `>=`, `==`, `!=`, `subset`, `in`, `is`
8. `and`
9. `or`
10. `implies`, `iff`

Associativity. `^` / `**` / `.^` (exponentiation) are **right-associative**, matching standard math convention: `2 ^ 3 ^ 2` is `2 ^ (3 ^ 2)` = `512`, not `(2 ^ 3) ^ 2` = `64`. Every other infix operator (`+`, `-`, `*`, `/`, `%`, …) is left-associative.

Unary minus binds looser than `^`. Negation sits *below* exponentiation in the table above, so `-2 ^ 2` parses as `-(2 ^ 2)` = `-4`, not `(-2) ^ 2` = `4` — again the math-textbook reading. Parenthesize the base (`(-2) ^ 2`) to negate first. It still binds tighter than `*`/`/`, so `-2 * 3` is `(-2) * 3`.

1.8 7. Set Theory & Comprehensions

1.8.1 Comprehensions

```
{x * 2 | x <- {1, 2, 3, 4, 5}}                  # {2, 4, 6, 8, 10}
{x | x <- range(1, 10), x mod 2 == 0}           # {2, 4, 6, 8, 10}
{(x, y) | x <- {1, 2}, y <- {3, 4}}             # {(1,3), (1,4), (2,3), (2,4)}
```

The generator clause is `target <- source` — or, in the set-builder slot described below, `target in source` / `target  source`.

1.8.2 British/ISO colon separator

Math texts write set-builder two ways: `{x | P(x)}` (American) or `{x : P(x)}` (British/ISO). Set comprehensions accept both separators — the two forms are identical down to the AST:

```
{x : x <- range(1, 10), x mod 2 == 0}           # same set as the pipe form
{2 * k : k <- range(1, 5)}                      # {2, 4, 6, 8, 10}  - the {2k : k = 1..5} style
{(x, y) : x <- {1, 2}, y <- {3, 4}}             # multi-generator works too
```

Hash literals are unaffected: `{k: v, ...}` stays a hash — the colon is read as a comprehension separator only when a generator clause (`target <- source`, or a gated `target  source`) provably follows it. The colon form applies to **set** comprehensions only; dict comprehensions keep `|` (their head already uses `:`), and list comprehensions keep `|` (inside `[...]` a colon means a block binding).

1.8.3 ISO membership generators — `{x : x  U, P(x)}`

Real ISO/British texts put membership in the binder slot. `in` / is accepted as a generator alias for `<-` under one rule: **the clause `x in S` is a generator iff `x` is not already bound in the comprehension and occurs in the head; otherwise it keeps its membership-filter meaning.** This makes the textbook form executable verbatim while leaving every membership test untouched:

```
{x : x  {1, 2, 3, 4}, x > 1}                    # {2, 3, 4}  - full ISO form
{x | x in range(1, 10), x mod 2 == 0}            # word form, pipe separator
[x * 2 | x in [1, 2, 3]]                         # lists too → [2, 4, 6]
{(x, y) : x  {1, 2}, y  {8, 9}}                 # multi-generator Cartesian
{a + b : (a, b)  pairs}                          # tuple-destructure target

# Membership FILTERS keep their meaning (the gate):
{x | x <- s, x in t}                             # x bound → filter (s  t idiom)
{c | c <- cs, g in c.allies}                     # g is outer/global → filter
{x | x <- s, x  t}                              #  never converts
[x for x in xs if x in t]                        # Python if-clause: always a filter
```

Works in all four comprehension flavors (set / list / dict / lazy), with both spellings (`in`,) and both separators (`|`, `:`). One reinterpretation to know about: the hash `{x: x in s}` — the *same* name as key and membership subject — now reads as a comprehension; write `{x: (x in s)}` (parenthesized value) or quote the key to keep the hash.

1.8.4 Relational comprehensions (Prolog-style)

```
{X | X <- parent(X, _)}                          # All parents
{(X, Y) | parent(X, Y)}                          # All parent-child pairs
{X | X <- parent(X, _), age(X, A), A >= 18}      # With filter
```

1.8.5 Concept-extent comprehensions

When the iterable is a bare concept name, comprehension iterates the concept's auto-maintained `Instances` map:

```
usa: a Country {}
china: a Country {}
{X | X <- Country}                               # All countries
```

1.8.6 Tag-filter comprehensions

Filter by epistemic grounding tag:

```
{X @theorem    | X <- derived_relation(X, _)}    # Theorems only
{X @axiom      | X is Country}                  # Axioms only
{X @conjecture | X <- flies(X)}                  # Defeasibly derived
```

Accepts `@axiom`, `@postulate`, `@theorem`, `@conjecture`, `@hypothesis`, `@datum`, `@canceled`, `@all`, `@*`.

1.8.7 Dict comprehensions

Produce a hash by emitting a key/value pair per iteration. Pipe form and pipe-less form are both supported:

```
xs: [1, 2, 3, 4, 5]

{x: x * x | x <- xs}                             # {1:1, 2:4, 3:9, 4:16, 5:25}
{x: x * x | x <- xs, x > 2}                      # {3:9, 4:16, 5:25}
{x: x * x for x in xs if x > 2}                  # pipe-less form, same result
```

1.8.8 Walrus bindings (compute once, reuse)

Bind a name mid-clause to avoid recomputing. The canonical form is `:`, the same operator used for ordinary value binding:

```
{y | x <- xs, y: f(x), y > 0}                    # bind y once, filter on it
```

Bindings are **iteration-local** — they do not leak out of the comprehension (unlike Python's walrus, which escapes into the enclosing function scope).

> A comprehension binding clause is written `y: expr`; there is no `y := expr` or `let y = expr` form.

1.8.9 Multi-filter folding

Multiple bare-expression filters in one comprehension are folded with `and`:

```
{x | x <- 1..10, x > 3, x < 8, x % 2 == 1}          # {5, 7}
```

1.8.10 Keyword aliases & the pipe-less form

Axioma also accepts `for x in iter` and `if cond` as aliases for `<-` and the bare filter inside the standard pipe form, plus a fully pipe-less form for all three comprehension flavors (so a comprehension copied from Python pastes in unchanged):

```
{x * 2 | for x in xs, if x > 2}                    # keywords inside pipe form
[x * 2 for x in xs if x > 2]                       # pipe-less list comp
{x * 2 for x in xs if x > 2}                       # pipe-less set comp
{x: x * x for x in xs if x > 2}                    # pipe-less dict comp
```

The first generator may use either `for x in iter` or `x <- iter`. Subsequent clauses are equally flexible.

> **Tests:** tests/axioma/comprehensions/test_python_superset.ax

1.8.11 Lazy generator expressions

Comprehensions wrapped in **parens** produce a lazy `Generator` instead of materializing. The source is pulled one element at a time on demand.

```
g: (x * 2 | x <- [1, 2, 3, 4, 5])      # Axioma pipe form
g: (x * 2 for x in [1, 2, 3, 4, 5])    # pipe-less form
g: (x + 1 | x <- xs, x > 15)           # with filter
g: (x + 1 for x in xs if x > 15)       # Python with filter
```

Driver builtins:

Builtin	Effect
`first(gen, n)`	Pull the first `n` elements (the **memorable** spelling — the same verb works on arrays, strings, sets, and infinite sets).
`first(gen)`	Pull one element (the first).
`force(gen)`	Pull all remaining elements into an `Array`.
`gen_take(n, gen)`	Pull first `n` elements — the explicit, lower-level alias of `first(gen, n)`.
`gen_drop(n, gen)`	Advance past `n` elements (mutates gen in place, returns it).
`gen_next(gen)`	Pull one element. Returns `Ω` when exhausted.

Prefer **`first(gen, n)`** — it's the same "first n" verb you already use for arrays and infinite sets. The `gen_*` prefix exists only because `take` is a reserved keyword (natural-language selective import) and `drop` is the Stack op.

```
g: (x | x <- [1, 2, 3, 4, 5])
first(g, 3)           # [1, 2, 3]   (same as gen_take(3, g))
force(g)              # [4, 5]      - remainder after partial take
```

Phase 2b — full multi-clause surface. Lazy comprehensions now accept the same clause vocabulary as eager list/set/dict comprehensions:

```
# Multi-generator (Cartesian, streamed - inner is re-eval'd each outer step)
pairs: (x * y | x <- [1, 2, 3], y <- [10, 20])
py_pairs: (x + y for x in [1, 2] for y in [100, 200])

# Walrus binding (`:` - the canonical form)
g: (s | x <- xs, s: x * x)

# Tuple destructure in first OR subsequent generators
dest: (n + 1 | (n, _label) <- tagged)
lab: (s + ":" + name | (s, name) <- pairs)

# Concept-extent source - bare concept name OR is form
gdps: (c.gdp | c <- Country)
gdps2: (c.gdp | c is Country)

# Relational predicate source - a fact-store query as the iterable
kids: (c | c <- parent("John", c))           # single generator
gkids: ((p, g) | p <- parent("John", p), g <- parent(p, g))  # chained
```

Streaming semantics. Multi-generator lazy form materializes inner sources once per outer combination (so the inner can reference outer-bound names — same as the eager path). The outermost source can still stream from a true `Generator` or `InfiniteSet`. Inner positions cannot use `InfiniteSet` — they'd re-materialize infinitely on each outer advance. `gen_take(N, ...)` bounds total pulls, naturally cutting the outer loop short when N is reached.

Relational sources. When a generator's iterable is a relational predicate call (`x <- parent(x, _)`), the lazy paths fall back to the relational machinery — eager evaluation of such a call fails, so this is detected by probe. The relation extent is bounded, so it is resolved once per entry; downstream pulls stay lazy. Works in single-generator, chained, inner, and filtered positions.

Limitations. `OrderBy` and `Having` clauses are accepted by the parser inside lazy form but silently discarded — both are intrinsically eager (sorting requires full materialization). For an ordered lazy stream, `force(gen)` then sort, or sort eagerly first.

> **Tests:** tests/axioma/comprehensions/test_lazy_generators.ax (Phase 2a — single gen), tests/axioma/comprehensions/test_lazy_generators_phase2b.ax (Phase 2b — multi-clause), tests/axioma/comprehensions/test_lazy_relational_source.ax (relational sources)

1.8.12 ORDER BY clause

List comprehensions accept an `orderby` clause that sorts the result. Sets and dicts ignore `orderby` (they're unordered by nature). Direction defaults to `asc`; add `desc` to reverse:

```
[x | x <- xs, orderby x]                    # asc by default
[x | x <- xs, orderby x desc]               # descending
[p.name | p <- people, orderby p.age]      # sort by computed key
[x for x in xs orderby x desc]             # pipe-less form works too
```

The sort key is evaluated in the iteration environment (where the loop variable refers to the source element), so referring to fields of the source element works as expected.

1.8.13 Aggregation: `group_by`, `items`, `keys`, `values`

Four builtins fill the SQL-style aggregation gap. `group_by(fn, coll)` partitions a collection into a hash; `items(hash)` exposes it as `(key, value)` pairs; `keys`/`values` return the parts individually.

```
orders: [
  {country: "US", amount: 100},
  {country: "UK", amount: 50},
  {country: "US", amount: 200},
]
by_country: group_by(func(o) [o.country], orders)
# {"US": [{...}, {...}], "UK": [{...}]}

# Iterate the hash with tuple destructure + ORDER BY:
pairs: items(by_country)
counts: [(p[1], len(p[2])) | p <- pairs]
sorted: [pair | pair <- counts, orderby pair[2] desc]
# [("US", 2), ("UK", 1)]
```

1.8.14 Tuple destructuring in <-

In any generator (first or subsequent), the iteration target can be a tuple pattern instead of a single variable:

```
pairs: [(1, "a"), (2, "b"), (3, "c")]

# First-generator tuple destructure (canonical form)
[n + len(s) | (n, s) <- pairs]            # list comp
{n: s     | (n, s) <- pairs}              # dict comp
{n * 2    | (n, s) <- pairs}              # set comp

# Idiomatic with items() over a hash
by_country: group_by(func(o) [o.country], orders)
{k: len(g) | (k, g) <- items(by_country)}
```

Source elements must be Tuples or Arrays of the matching arity. Mismatched arity is a hard error.

1.8.15 Hash destructure in <-

First-generator targets can also be a **hash pattern** that binds named fields directly. Two surface forms — implicit binding (var name = key name) and explicit rename (`key: var`):

```
people: [
  {name: "Alice", age: 30, dept: "Eng"},
  {name: "Bob",   age: 25, dept: "Sales"},
  {name: "Carol", age: 35, dept: "Eng"}
]

# Implicit binding
[name | {name, age} <- people, age >= 30]
# → ["Alice", "Carol"]

# Explicit rename
[(n, a) | {name: n, age: a} <- people]
# → [("Alice", 30), ("Bob", 25), ("Carol", 35)]

# Mixed implicit + rename in same pattern
[name | {name, age: a, dept} <- people, a > 27 and dept == "Eng"]

# pipe-less form
[name for {name, age} in people if age > 27]
{n for {name: n} in people}
```

```
{name: age for {name, age} in people}

# Lazy form
g: (name | {name, age} <- people, age > 25)
force(g)   # → ["Alice", "Carol"]

# Combined with orderby + limit
[name | {name, age} <- people, orderby age desc, limit 2]
# → ["Carol", "Alice"]
```

Skip-on-missing-key. When a pattern key is absent from a source hash, that row is silently dropped — destructure acts as filter+extract in one step:

```
mixed: [{name: "Alice", age: 30}, {name: "Bob"}, {name: "Carol", age: 35}]
[n | {name: n, age: a} <- mixed]
# → ["Alice", "Carol"]   (Bob has no `age` key - dropped)
```

Wrong-type elements (anything not an `ObjectMap` hash) surface a runtime error.

Scope. Hash destructure is currently supported only in the **first generator** position. Subsequent generators (after a `,` in pipe form) use the single-var or tuple-pattern forms — `{...}` in that position would be ambiguous with set/hash filter expressions.

Tests: tests/axioma/comprehensions/test_hash_destructure.ax

1.8.16 Multi-column ORDER BY

`orderby` accepts a comma-separated list of sort keys, each with its own direction. Sort is lexicographic across columns:

```
[p.name | p <- people, orderby p.dept, p.age desc]
# Sorts by dept ascending; within each dept, by age descending.
```

1.8.17 HAVING clause — terminal filter

`having EXPR` is a terminal filter that runs in the iteration environment (so it can reference the loop variable). Distinct from `, EXPR` filters only in position and intent — HAVING reads as "row filter after aggregate computation" by convention:

```
[o.amount | o <- orders, having o.amount > 70]

# Combined with orderby (having runs at iteration time, not after sort):
[o.amount | o <- orders, orderby o.amount desc, having o.amount > 70]
```

1.8.18 LIMIT / OFFSET clauses

SQL-style row caps. `limit N` keeps at most N output rows; `offset N` skips the first N post-filter rows. Both apply to all four comprehension flavors (list, set, dict, lazy) and to both surface forms (pipe and pipe-less). They are terminal-ish: once seen, only the other of the pair may follow.

```
# Basic
[x | x <- xs, limit 3]                    # first 3 rows
[x | x <- xs, offset 5]                   # skip first 5
[x | x <- xs, offset 5, limit 3]          # paging: skip 5, then take 3

# Order is free - these are equivalent
[x | x <- xs, limit 3, offset 5]

# Combined with orderby (SQL convention - limit/offset run AFTER sort)
```

```
[x | x <- unsorted, orderby x desc, limit 5]

# Combined with orderby + having
[t.amount | t <- txns, orderby t.amount desc, having t.amount > 50, limit 2]

# pipe-less form (no commas between clauses)
[x * 3 for x in xs if x > 2 limit 4]
[x for x in xs offset 6 limit 2]
[x for x in unsorted orderby x desc limit 2]

# Lazy generator - limit caps total Next() pulls, offset skips post-filter rows
g: (x * 2 | x <- big_source, x > 100, offset 10, limit 5)
gen_take(3, g)   # yields up to 3 elements (lazy stops emitting at limit anyway)
```

Soft keywords. `limit` and `offset` are NOT reserved at the lexer level — they remain ordinary identifiers usable as variable names, property names, and DSL slot names (`bindings.limit`, `{limit:number}`). The parser recognizes them as clause keywords only when they appear as the first token of a comprehension clause.

Caveat. Inside a comprehension clause list, you cannot use `limit` or `offset` as the LHS of a filter expression. `[x | x <- xs, limit > 5]` parses `limit` as the LIMIT clause keyword and chokes on `> 5`. Parenthesize to disambiguate: `[x | x <- xs, (limit > 5)]`.

Validation. Both clauses evaluate to non-negative integers. Negative or non-integer values surface as runtime errors. `limit 0` yields the empty result; `offset N` with N greater than the source length yields the empty result.

Sets/dicts. Limit/offset DO apply (cap output cardinality), but the slice is taken in Go-map iteration order — non-deterministic. Use list comprehensions for deterministic paging.

> **Tests:** tests/axioma/comprehensions/test_limit_offset.ax
>
> **Tests:** tests/axioma/comprehensions/test_tier15_first_gen_multi_having.ax
>
> **Tests:** tests/axioma/comprehensions/test_tier1_orderby_groupby.ax

1.8.19 Range generators

Any numeric range expression is a valid generator source. Both half-open and inclusive variants are accepted:

```
[x * x | x <- 1..10]            # inclusive 1..10  → [1, 4, 9, 16, …, 100]
[x | x <- 1..<10, x % 2 == 0]   # half-open       → [2, 4, 6, 8]
{x | x <- 1..100, prime?(x)}    # set of primes   100
(x | x <- 1..)                  # NB: infinite - use infinite_set("naturals") instead
```

Use a finite range when you know the upper bound; use `infinite_set("naturals" | "primes" | "evens" | "odds" | "fibonacci" | "integers")` inside a **lazy** comprehension when you want a genuinely unbounded source pulled on demand.

1.8.20 Intensional-class generators (Russell iota)

Define an intensional class once with `the <Concept> where <pred>` and use it as a generator source. Membership is computed by combining the base concept's extent with the predicate restrictor:

```
adults: the Person where age >= 18
all_adult_names: [p.name | p <- adults]
```

This is the Russell-faithful form of "the things that satisfy P." Useful when the same restrictor predicate is consulted from several comprehensions — define it once, reuse by name.

1.8.21 The same question, many ways

Comprehensions, quantifiers, higher-order builtins, and the membership operator all answer overlapping questions. The cleanest illustration is the existential question — "is there an X in S satisfying P?" — which has **18 working surface forms** in Axioma:

```
persons: {mike, alice}

# Quantifier family - three spellings, same AST
exists x in persons | x.name == "Mike"                              # keyword
`exists x in persons | x.name == "Mike"                             # backtick digraph (ASCII)
       x in persons | x.name == "Mike"                             # Unicode glyph

# Set-comprehension family - "the matching set is non-empty"
len({x | x <- persons, x.name == "Mike"}) > 0                       # Axioma pipe
{x | x <- persons, x.name == "Mike"} != {}                          # Axioma pipe
not ({x | x <- persons, x.name == "Mike"} == {})                    # De Morgan
len({p for p in persons if p.name == "Mike"}) > 0                   # Python
{p for p in persons if p.name == "Mike"} != {}                      # Python

# List-comprehension family - "indicator sum > 0"
sum([1 | p <- persons, p.name == "Mike"]) > 0                       # Axioma pipe
sum([1 for p in persons if p.name == "Mike"]) > 0                   # Python
len([1 for p in persons if p.name == "Mike"]) > 0                   # Python

# Higher-order - reductions over the collection
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, reuse it
{x | x <- persons, n: x.name, n == "Mike"} != {}

# Lazy + force - produce a generator, then materialize
g: (x | x <- persons, x.name == "Mike")
len(force(g)) > 0

# Lazy + gen_take(1) - SHORT-CIRCUITED existence (asymptotically cheaper)
len(gen_take(1, (x | x <- persons, x.name == "Mike"))) > 0

# 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 so many. Each form aligns with a different intellectual tradition: math (`{x | …}`), Python (`for x in xs if …`), SQL (`orderby/having/limit/offset`), Haskell (walrus), Prolog (`<-` over relational predicates), F-logic (concept extents, `@tag` filters), and indicator-sum probability. The right form is the one your *reader* will find most natural. Working showcase: tests/axioma/showcase/method.ax.

Short-circuit note. Quantifier forms (`exists / `) and `gen_take(1, lazy)` stop at the first match. Eager comprehension forms scan the full collection. For large collections the difference matters — pick the short-

circuiting form when the predicate is cheap and the collection is large.

1.9 8. First-Order Logic

1.9.1 Quantifiers

```
a: {1, 2, 3, 4, 5}
forall x in a: x > 0          # true
forall x in a: x > 3          # false
exists x in a: x > 3          # true
exists x in a: x > 10         # false
```

1.9.2 Bounded Range & Counting Quantifiers

Axioma supports advanced bounded range quantifiers (which operate over integer intervals without requiring an explicit set domain) and comparison-based counting quantifiers (which assert that a specific number of elements satisfy a predicate):

1.9.2.1 Bounded Range Quantifiers

- **Double-sided range**: evaluates the predicate over the range `[lower..upper]` (for `<=`) or `[lower..<upper]` (for `<`).

  ```
  forall 1 <= x <= 10: x > 0       # true
  exists 1 < x < 5: x == 3         # true
  ```

- **Single-sided range**: defaults to a lower bound of `0` and evaluates up to the upper bound.

  ```
  forall x < 5: x >= 0            # true (evaluates x = 0, 1, 2, 3, 4)
  exists x <= 0: x == 0           # true (evaluates x = 0)
  ```

1.9.2.2 Arbitrary Counting Quantifiers Checks whether exactly, at least, or at most a specified number of elements in a set satisfy a predicate. Uses standard comparison operators (`==`, `>=`, `<=`, `>`, `<`) directly following the `exists` keyword:

```
domain: {1, 2, 3, 4, 5}
exists >= 3 x in domain: x > 2    # true (witnesses: 3, 4, 5)
exists <= 1 x in domain: x > 4    # true (witness: 5)
exists == 2 x in domain: x % 2 == 0 # true (witnesses: 2, 4)
exists > 1 x in domain: x < 3      # true (witnesses: 1, 2)
```

1.9.3 With predicates

```
nums: {1, 2, 3, 4, 5, 6}
forall x in nums: positive(x)
exists x in nums: even(x)
forall x in nums: x < 10 and x > 0
```

1.9.4 Combined

```
a: {2, 4, 6}
b: {1, 2, 3, 4, 5, 6}
forall x in a: even(x) and (x in b)
exists x in b: odd(x) and (x > 3)
```

1.9.5 Symbolic-mode quantifiers — textbook syntax

The bounded forms above (`forall x in a: ...`) iterate a domain and return a Boolean. Axioma also accepts **bare textbook-style quantifiers** with no `in <domain>` clause — these produce a `Formula` value, a symbolic carrier you can pass to the resolution/CNF/satisfiability machinery in `fol/`, `sol/`, and `logic/`.

```
# Bare textbook form - no marker between variable and body
f1:  x P(x)              # →  x. P(x)        (type: formula)
f2:  x F(x)              # →  x. F(x)

# Dot form
f3:  x. P(x)             # identical to f1
f4:  x. F(x)

# ASCII keyword equivalents
forall x P(x)
exists x F(x)
`forall x P(x)           # backtick digraph
`exists x F(x)

# Complex bodies - quantifier binds widely
 x P(x) → Q(x)           # →  x. (P(x) → Q(x))
 x P(x)  Q(x)            # →  x. (P(x)  Q(x))
 x ¬P(x)  Q(x)           # →  x. ((¬P(x))  Q(x))

# Nested quantifiers
 x  y P(x, y)
 x  y P(x, y) → R(x, y)
```

Mode summary:

Quantifier form	Mode	Returns
`x F(x)` / `x F(x)`	symbolic	`*types.Formula`
`x. F(x)` / `x. F(x)`	symbolic	`*types.Formula`
`x in S | F(x)`	bounded	`Boolean`
`!x in S | F(x)`	bounded uniqueness	`Boolean`
`!x F(x)`	symbolic uniqueness	`*types.Formula`

Tier 1 limitation: the connectives `¬` `→` on Formula operands still dispatch to Boolean semantics — building compound formulas like `¬ x P(x)` `x ¬P(x)` at the symbolic level requires Tier 2 Formula-aware connectives. For now, keep connectives *inside* the quantifier body.

1.9.6 Other Tier 1 textbook additions

```
# Biconditional - both Unicode forms lex to IFF
true  false              # false  (U+2194, modern textbook)
true  false              # false  (U+27FA, was lexed but never wired)

# Truth constants
                         # false  (falsum, U+22A5)
                         # true   (verum, U+22A4)

# Proper subset distinct from subset
{1,2}  {1,2,3}           # true  (proper subset)
```

```
{1,2,3}  {1,2,3}           # false (equal, not proper)
{1,2}  {1,2}               # true  (regular subset allows equality)

# Set difference - Enderton/Halmos notation
{1,2,3,4,5} \ {2,4}          # {1, 3, 5}

# Uniqueness quantifier
 !x in {1,2,3} | x == 2      # true (exactly one)
```

1.9.7 Formula type + predicate

```
f:  x P(x)
@f                           # "formula"
formula?(f)                  # true
boolean?(f)                  # false
```

See tests/axioma/logic/test_textbook_parity_tier1.ax for full coverage.

1.10 9. Multi-Valued Logics

Axioma has **five first-class logic kinds**, each with its own truth-value type. Operators (`and`, `or`, `not`, `implies`) automatically dispatch based on operand types. The dispatch priority is **Belnap > Intuit3 > Łukasiewicz > Kleene > Boolean**.

1.10.1 Boolean

Classical two-valued. Default for `true`/`false`.

```
true and false        # false
true implies false    # false
not true              # false
```

1.10.2 Kleene K3 (three-valued)

`true` / `false` / `unknown` — represented by `om` / Ω.

```
om and true           # om   (unknown)
om or  true           # true
not om                # om
```

1.10.3 Łukasiewicz L3 (real-valued)

Continuous truth values in `[0, 1]`.

```
half: lukasiewicz(0.5)
qrtr: lukasiewicz(0.25)
half and qrtr         # min(0.5, 0.25) = 0.25
half implies qrtr     # min(1, 1 - 0.5 + 0.25) = 0.75
```

1.10.4 Belnap B4 (paraconsistent four-valued)

`T`, `F`, `Both`, `Neither` — supports contradictory information.

```
p: belnap("both")           #    - contradictory
q: belnap("true")
p and q                          # Both - contamination propagates
truth("parent", "John", "Mary")  # Query stored Belnap value (relation named by string)
```

Display glyphs: , , , ? .

1.10.5 Gödel G3 (intuitionistic three-valued)

`true` / `false` / `unknown` — but with **intuitionistic** semantics:

```
p: intuit3("unknown")
not p                           # false  (G3 collapses U to F; K3 keeps U)
p implies p                     # true   (reflexive - always)
g3_lem(intuit3("unknown"))      # unknown - LEM is NOT a tautology
g3_dne(intuit3("unknown"))      # unknown - double-negation elimination FAILS
```

Display glyphs: , , ? .

1.10.6 First-class MVL values

Every three/four-valued logic kind has a **constructor** that produces a typed, introspectable value — `belnap()` (B4), `lukasiewicz()` (Ł3), `intuit3()` (G3), and **`kleene()`** (K3). `type(x)` and the `@x` type-of sigil return the proper TitleCase name, and each value is `is`-checkable against a seeded primitive Concept:

```
ku: kleene("unknown")           # also kleene(om), kleene("u"), kleene("?")
type(ku)                        # → "Kleene"      (@ku is the same)
ku is Kleene                    # → true
ku == om                        # → true   (coerces to its canonical Boolean/Om form)

kt: kleene("true")
kt and ku                       # → ?     (a Kleene value - glyphs   /   / ? )
not ku                          # → ?
ku implies ku                   # → ?     (K3 - contrast G3's reflexive →  )

type(belnap("both"))            # → "Belnap"
type(lukasiewicz(0.5))          # → "Lukasiewicz"
type(intuit3("unknown"))        # → "Intuit3"
```

> **Note:** Kleene's `unknown` is still represented by `om` (Ω) at the operator level — the existing `om and true` tables are unchanged. `kleene(...)` is the *typed* form: it normalizes to `om` for evaluation, runs the same K3 tables, then re-wraps logic results back into a `Kleene` so they stay introspectable (mirroring how `belnap`/`lukasiewicz`/`intuit3` operators return their own type).

1.10.7 Truth tables

```
Kleene       show truth_table for and
Lukasiewicz  show truth_table for implies
Belnap       show truth_table for or
Intuit3      show truth_table for implies
```

1.11 10. Modal, Temporal, Epistemic & Deontic Logic

1.11.1 Modal operators

```
necessarily(p)        #  p - p holds in all accessible worlds
possibly(p)           #  p - p holds in some accessible world
```

1.11.2 Temporal logic

```
always(p)             # G p - p holds at all future times
eventually(p)         # F p - p holds at some future time
next(p)               # X p - p holds at the next time step
until(p, q)           # p U q - p until q
```

1.11.3 Epistemic logic

```
knows("alice", p)              # K_alice p - Alice knows p
believes("bob", p)             # B_bob p
common_knowledge(p, agents)    # CK p among agents
```

1.11.4 Deontic logic

```
obligatory(action)
permitted(action)
forbidden(action)
```

The full modal-logic system is documented in `resources/docs/claude/MODAL_LOGIC.md`.

1.12 11. Fuzzy Logic & Higher-Order Logic (SOL)

1.12.1 Fuzzy logic

Continuous truth in `[0, 1]` with fuzzy set membership:

```
tall: fuzzy_set("tall", lambda h => sigmoid(h - 180))
membership(tall, 175)          # 0.38
membership(tall, 190)          # 0.88
```

1.12.2 Second-Order Logic (SOL)

Quantification over predicates and functions:

```
forall_pred P: forall x: P(x) implies P(x)        # Trivially true
exists_pred P: forall x in domain: P(x)           #  P.  x. P(x)
```

See `sol/sol.go` for implementation details.

1.13 12. Lambda Calculus & Higher-Order Functions

1.13.1 Function definitions

```
inc: lambda x => x + 1
add: lambda (x, y) => x + y
multiply: func(a, b, c) [a * b * c]
```

1.13.2 Anonymous functions

```
(lambda x => x * 2)(5)        # 10
```

1.13.3 Closures

```
makeAdder: lambda n => lambda x => x + n
addTen: makeAdder(10)
addTen(5)                    # 15
```

1.13.4 Currying & partial application

```
multiply: lambda x => lambda y => x * y
double: multiply(2)
triple: multiply(3)
```

1.13.5 Function composition

```
compose: lambda (f, g) => lambda x => f(g(x))
addOneSquared: compose(lambda x => x * x, lambda x => x + 1)
addOneSquared(3)             # 16
```

1.13.6 Map / Filter / Reduce

```
nums: {1, 2, 3, 4, 5}

map(lambda x => x * x, nums)                      # {1, 4, 9, 16, 25}
filter(lambda x => x > 2, nums)                   # {3, 4, 5}
reduce(lambda (acc, x) => acc + x, 0, nums)       # 15
sum(nums)                                         # 15  (works on arrays/sets/tuples)
Σ(nums)                                           # 15  (Unicode alias for sum)
```

1.14 13. The Concept System

A **concept** is Axioma's central unit of knowledge representation — far more than an object-oriented class or record. Axioma's *types are* concepts, but a concept is at once:

- a **type** — the built-in primitives (`Integer`, `Float`, `Set`, `Stack`, …) are themselves concepts, so `5 is Integer` and `x :: Day` run the same machinery as any concept you declare;
- a **frame** — a named thing whose **slots** carry values *and* metadata (inverse, transitive, cumulative, cardinality), with an auto-maintained **extent** of its instances you can iterate like a set (`{x | x <- Country}`), and a concept relation rule duality;
- a **description-logic concept** — composable with ¬, ordered by subsumption / , with role restrictions (`R.C`, `R.C`), `Thing` / `Nothing` (/), defined concepts, and partitions — all decided by a real ALC **tableau reasoner** (`satisfiable`, subsumption);
- a **classifier** — the `is` copula tests Russell's predication (, `rex is Dog`) and class inclusion (, `Dog is Animal`); a `defines` predicate or a `boundary` turns membership into a rule, and every classification carries an **epistemic grounding** and may be defeasible;
- a **designed, inspectable idea** — the concept-formation layer (`purpose`, `examples` / `counterexamples`, `formed_by`, `default_grounding`) lets you state *why* a concept exists and `check` that it does its job.

The surface is natural language throughout — `concept Stock`, `Stock has price`, `rex: a Dog`, `rex is Dog`, `Dog extends Animal`. The rest of this section works through each of these dimensions; together they make concepts the substrate of Axioma's knowledge representation and its cognitive paradigm.

1.14.1 Defining concepts

Axioma has two canonical concept-declaration surfaces:

There is a **single canonical creation surface** — the keyword-first `concept X` form. It absorbs four feature slots that were previously distributed across multiple competing forms:

- **bare**: `concept X`
- **with postfix doc string**: `concept X "doc"`
- **with prefix refinement**: `concept/persist X`, `concept/transient X`, `concept/system X`
- **with refinement + doc**: `concept/persist X "doc"`
- **with slot-defaults block**: `concept X { slot: default, ... }`
- **with refinement + block**: `concept/persist X { ... }`
- **namespaced**: `concept FinKB.Stock`, `concept/persist FinKB.Bond "doc"`

```
concept Stock                                    # bare concept creation
concept Stock "A financial instrument"           # with postfix doc
concept Stock {                                  # with initial slot defaults
  price: 0
  ticker: ""
}
concept/persist Stock                            # force KB persistence
concept/transient Scratch                        # never persist
concept/system InternalRegistry                  # mark as system-internal
concept/persist Stock "force-persist regardless of mode"

Stock extends Asset                              # specialize Stock as a subclass of Asset
apple is Stock                                  # classify apple as a Stock (predication)
```

The keyword-first form pairs naturally with the rest of Axioma's speech-act family (`define`, `axiom`, `postulate`). `is` remains the canonical operator for instance classification (`apple is Stock`) and Boolean type queries (`x is Stock` in expression position).

Doc strings are also accepted inside the block form:

```
concept TreasurySec { doc: "A marketable US Treasury security" }
concept TreasurySec "A marketable US Treasury security"
TBill extends TreasurySec "Treasury bill, max 365 days"
TreasurySec has issuer: "US Treasury"
TBill has max_maturity_days: 365
```

Name-inference form — an anonymous `concept { ... }` literal on the RHS of a `:` binding picks up its name from the LHS identifier. The resulting Concept is fully named and behaves identically to `concept X { ... }`:

```
Widget: concept { price: 0, ticker: "" }      # name inferred from LHS
Gadget: concept { color: "red", count: 5 }
Foo: concept FooBar { rate: 0.05 }            # explicit RHS name wins; Foo aliases FooBar
```

> **Casing rule (enforced):** the inferred name must start with an uppercase letter — the same `isValidConceptName` check every other concept-creation surface uses. `tc: concept { ... }` errors with `concept name 'tc' should start with an uppercase letter`. Use `TC: concept { ... }`. The bare-RHS form is also the only form that infers a name; anonymous `concept { ... }` literals nested inside a call argument or array element error cleanly at eval time.

Creating concepts — the canonical surface. A concept is created with the keyword-first `concept` form. There is no `create`-prefixed, postfix, or `is Concept` creation form:

```
concept Stock                       # bare
concept Stock "doc"                 # with a doc string
concept/persist Stock "doc"         # with a /persist | /transient | /system refinement
concept Stock { price: 0 }          # block form with slot defaults
concept FinKB.Stock "doc"           # namespaced
```

```
x: a Stock                          # instance creation, via the indefinite article
Stock extends Asset                 # class inclusion
Stock has price: 150                # property (auto-creates Stock on first verb use)
```

`is` is a **predication** operator, never a creation one — using it to create a concept would conflate two speech acts (constitutive declaration vs. descriptive predication, the distinction Russell draws on the copula). So `aapl is Stock` classifies an instance, `X is Concept` (in expression position) asks "is this a registered concept?", and a statement-level `Stock is Concept` is a parser error pointing at `concept Stock`.

Define-family form (`define concept`) — fills the typed-define slot left open in the existing family of `define axiom` / `define postulate` / `define theorem` / `define word` / `define dialect`:

```
define concept Stock = {                        # `=` assignment
  price: 0,
  ticker: ""
}

define concept Bond: { yield: 0.05 }            # `:` assignment
define concept Scratch = {}                     # empty body - bare creation
define/persist concept Pinned = { x: 1 }        # refinement: force-persist
define/transient concept Tmp = { x: 1 }         # refinement: skip persistence
```

Syntactically parallel to `define dialect`: uppercase name, `{ ... }` body parsed as a hash literal. The implementation synthesizes an internal `ConceptCreation` AST and delegates to the canonical creation pipeline, so every formation-layer feature (boundary capture, examples auto-classification, `formed_by` cross-map, default-grounding cross-map, KB persistence) carries through transparently. Behaves identically to the equivalent `concept Stock { ... }` on every dimension except syntax.

Mapping to the rest of the family:

Surface	Speech act	Body shape	Examples in this manual
`define axiom`	KB claim	bracketed expression	"axiom" section
`define postulate`	provisional claim	bracketed expression	"postulate" section
`define theorem`	derived claim	bracketed expression	"theorem" section
`define word`	Lojban/NSM word	block of slots `[...]`	"words" section
`define dialect`	DSL registration	array or `{cases:..., vocabulary:...}`	"dialects" section
`define concept`	concept declaration	hash literal `{ slot: value, ... }`	this section

v1 limitations: - **Inheritance**: no `extends` / `is` slot inline. Follow up with `Stock is Asset` on the next line, or use `concept Stock extends Asset { ... }` if you need it co-located. - **Defeasible boundary**: the `~` suffix on hash-literal keys is not parsed (`define concept Sage = { boundary~: ... }` will fail). Use the canonical `concept Sage { boundary~: ... }` block form, or follow up with the statement-level `Sage defines~ { ... }` rule. - **Namespaced names**: `define concept FinancialKB.Stock = ...` is not supported (the `define` parser produces a flat `[]*Identifier` symbol list). Use the prefix `concept FinancialKB.Stock { ... }` for namespaced concepts.

Tests: tests/axioma/concepts/test_define_concept.ax (happy paths) and the three matching `_reject` files (lowercase, `/unpersist`, non-hash body).

Concept introspection at every level — what `is` checks depends on syntactic position:

- `<Name> is Concept` at statement level (TitleCase LHS) → **parser error** (use `concept X [doc] [refinement]` to create a concept)
- `<Name> is <Parent>` → declares a specialization (class-inclusion)
- `<instance> is <Concept>` → classifies the instance (membership) — **canonical**

- `<expr> is <C>` in expression position → Boolean predication query — **canonical** (including `X is Concept` as a "is this a registered concept?" query)

1.14.2 Inheritance

```
concept Animal
Dog extends Animal
Dog has bark: lambda => println("Woof!")
```

1.14.3 Concept formation layer (Phase 1)

Three slot names on a concept carry first-class **concept-design** semantics. They turn `concept Foo { ... }` from a plain class declaration into a contract that pairs prose intent with executable regression tests over the extent.

```
concept CFP_Choice                                  # base concept for entities
positive_a: a CFP_Choice {}
positive_b: a CFP_Choice {}
negative_a: a CFP_Choice {}

concept DecisionFatigue
df1: a DecisionFatigue {}
df2: a DecisionFatigue {}

DecisionFatigue has purpose: "Explain degraded choices after repeated decisions"
DecisionFatigue has examples: [df1, df2]
DecisionFatigue has counterexamples: [negative_a]

check DecisionFatigue   # →    (contract holds)
```

The four reserved slot names:

Slot	Semantics
`purpose:`	Prose statement of why the concept exists. Pure metadata, queryable as `Concept.purpose`.
`examples:`	Array of `ConcreteEntity`s that MUST be classified positively.
`counterexamples:`	Array of `ConcreteEntity`s that MUST NOT be classified positively.
`formed_by:`	Closed enum naming the creation mode. Validated at creation time (Phase 2a).

The `formed_by:` enum accepts five string values. As of Phase 2b-1 the cross-map is automatically derived into a `default_grounding` slot at concept-creation time, and as of Phase 2b-2 the derived grounding is *actively applied* to stored is-facts at instance-creation and classification time:

`formed_by:`	Meaning	Default grounding
`"abstraction"`	pattern extracted from multiple observed cases	conjecture (inductive)
`"combination"`	concept synthesized from existing concepts	theorem (derivable)
`"distinction"`	concept split out of a broader one	theorem (derivable from parent)
`"stipulation"`	concept defined by fiat for a purpose	axiom (definitional)

formed_by:	Meaning	Default grounding
"metaphor"	concept formed by mapping one domain onto another	hypothesis (cross-domain)

```
concept Monoid {
  purpose: "Algebraic structure with associative binary op and identity"
  formed_by: "stipulation"
}

concept Smartphone {
  purpose: "Phone + computer + camera + internet, unified"
  formed_by: "combination"
}
```

Invalid values are rejected at creation time:

```
ERROR: concept 'X': formed_by = "stipulashun" is not a recognized creation mode.
Use one of: abstraction, combination, distinction, stipulation, metaphor
```

The contract is verified by **`check Concept`** (the existing automated-reasoning `check` keyword, specialized for Concept targets). It returns a Belnap B4 value:

Result	Meaning
`T` ( )	Every example classifies positive; no counterexample classifies positive
`F` ( )	At least one example missing from the extent, or at least one counterexample wrongly classified
`Both` ( )	The SAME entity appears in both lists (paraconsistent declaration)
`Neither` (?)	No examples and no counterexamples — contract vacuous

`check` on a non-Concept target falls through to the legacy `AutomatedReasoningObject` wrapper (`check 42` still returns the generic consistency-check string).

Phase 2b-2 adds the `boundary:` slot and the active application of `default_grounding`. A `boundary:` value is a *predicate*, not an open expression — it captures the AST before the property eval loop runs and registers it as a `defines` classifier scoped to the concept:

```
concept P_Person
P_Person has age: 0
P_Person has name: ""

concept P_Adult {
  formed_by: "stipulation"      # → default_grounding: "axiom"
  boundary: age >= 18 and is P_Person
}

alice: a P_Person {name: "Alice", age: 30}
println(alice is P_Adult)                          # → true (auto-classified)
println(grounding("isa", alice, P_Adult))            # → "axiom"
```

The stored is-fact inherits `axiom` grounding (not the strict-`defines` default of `theorem`) because Stipulation-formed concepts cross-map to `axiom`. The same active-grounding flow fires at object-instantiation time:

```
concept P_Stock { formed_by: "stipulation" }         # default_grounding: "axiom"
aapl: a P_Stock {}
println(grounding("isa", aapl, P_Stock))             # → "axiom"
```

The instance-creation is-fact is **gated**: a concept without `formed_by:` (or explicit `default_grounding:`) keeps the pre-2b-2 behavior — the instance is registered in `concept.Instances` for comprehension iteration, but no synthetic is-fact is stored. This preserves the legacy corpus verbatim.

Phase 3 closes the loop on the Phase 1 contract by making `examples:` load-bearing. When the concept has opted into the formation layer (`default_grounding` set, either via `formed_by:` cross-map or explicitly), the `examples:` slot is treated two different ways depending on whether a `boundary:` predicate is present:

- **No boundary** — examples become the *extent declaration*. Each entity in the list is force-classified as a member of the concept, at the inherited `default_grounding`. Useful for small enumerated concepts that are easier to *list* than to *describe* with a rule:

  ```
  concept VIP {
    formed_by: "stipulation"        # → default_grounding: "axiom"
    examples: [alice, carol]        # auto-classified as VIPs
  }
  println(alice is VIP)                          # → true
  println(grounding("isa", alice, VIP))            # → "axiom"
  println(check(VIP))                             # →   (trivially)
  ```

- **With boundary** — the boundary owns membership; examples become *test cases* for the boundary. `check Concept` verifies the boundary classifies every listed example and rejects every counterexample:

  ```
  concept Adult {
    formed_by: "stipulation"
    boundary: age >= 18 and is Person
    examples: [alice, carol]        # both   18 → boundary agrees
    counterexamples: [bobby]        # < 18 → boundary correctly rejects
  }
  println(check(Adult))             # →   (boundary agrees with examples)

  concept AdultBad {
    formed_by: "stipulation"
    boundary: age >= 18 and is Person
    examples: [dave]                # dave is 8 - boundary disagrees!
  }
  println(check(AdultBad))          # →   (example missing from extent)
  ```

Counterexamples are never auto-classified — there's no opponent rule to push against. They still participate in `check`'s negative verification (no counterexample may be in the extent).

Phase 4 adds the **defeasible boundary** form: `boundary~:`. The tilde parallels the statement-level `defines~` and marks the membership rule as defeasible — classifications derived from it cap at conjecture-grade, *even on a stipulation-formed (axiom) concept*. This expresses a coherent two-level epistemic stance: the concept itself is definitional, but specific memberships derived from an uncertain rule remain tentative.

```
concept Sage {
  formed_by: "stipulation"          # concept is axiom-grade
  boundary~: age >= 65 and wisdom >= 70 and is Person
}

alice: a Person {age: 70, wisdom: 80}
println(alice is Sage)             # → true
println(grounding("isa", alice, Sage))   # → "conjecture" (cap fires)
```

The cap rule: defeasibility lowers grounding to `conjecture` when the concept's `default_grounding` is stronger (axiom, postulate, theorem). When already weaker (hypothesis, datum), the value passes through unchanged. Declaring both `boundary:` and `boundary~:` on the same concept is rejected at creation time.

See tests/axioma/concepts/test_concept_formation_phase1.ax for the contract test matrix, tests/axioma/concepts/test_con
for the boundary + active-grounding tests, tests/axioma/concepts/test_concept_formation_phase3_examples.ax
for the examples-auto-classification tests, and tests/axioma/concepts/test_concept_formation_phase4_defeasible_boundary
for the defeasible-boundary tests.

Phase 5 surfaces the formation layer as queryable data via the `concepts_formed_by(mode_string)` builtin. Returns an array of every concept whose `formed_by:` slot equals the given mode, validating the mode against the same enum used at creation time:

```
concept Monoid  { formed_by: "stipulation" }
concept Group   { formed_by: "stipulation" }
concept Penguin { formed_by: "distinction" }

len(concepts_formed_by("stipulation"))   # → 2
stips: concepts_formed_by("stipulation")
println(stips[1].formed_by)              # → "stipulation"

concepts_formed_by("stipulashun")        # → ERROR (typo rejected,
                                         #   same enum as Phase 2a)
```

The return value is a plain `Array<Concept>`, so it composes with comprehensions, `len`, and the rest of the array vocabulary. Useful for KB audits and for tooling that wants to render formation-layer choices.

1.14.4 Instances

```
usa: a Country {}
china: a Country {}
alice: a Person {name: "Alice", age: 30}
```

1.14.5 Property access

```
alice.name                    # Dot
alice's name                  # Possessive
alice[name -> "Bob"]          # ErgoAI frame-form (returns "Bob")
```

1.14.6 `is` predicate

```
alice is Person              # true
{X | X is Country}           # All instances of Country
{X | X is Country, X.gdp > 20000}   # With filter
```

1.14.7 Cardinality constraints

```
cardinality(Country, "capital", 1, 1)    # Exactly one capital per country
```

1.14.8 Concept introspection

```
Stock show properties
Concept display hierarchy
```

1.14.9 KM-style surface syntax

Four ergonomic forms adopted from Peter Clark & Bruce Porter's KM 2.0 (*The Knowledge Machine*). Each composes with the existing concept system above and adds no new semantic machinery — they are surface forms over the canonical `ObjectInstantiation` AST node, concept-level `has`, and the existing `it` keyword.

> Instances are created with the natural-language `a` / `an` / `entity` forms (`a Stock {}`, `an Item {}`, `entity Gadget {}`) — all three produce the same AST and runtime value. There is no `object` keyword.

Indefinite-article instance literals — `a Concept {props}` and `an Concept {props}` are the canonical instance-creation expressions. The property block is optional. The article is a *soft* keyword: it only triggers when followed by a capitalized identifier (Axioma's concept convention), so variables named `a` or `an` continue to work.

```
mycar: a Car {make: "Toyota", price: 26000}
cat: an Animal {species: "Felis catus"}
cheap: a Car {}                    # empty-property form
dyn: (a Car {price: 99999}).price  # usable mid-expression
fleet: [a Car {price: 10000}, a Car {price: 20000}]
```

`it` anaphora — every fresh instance binds `it` in the surrounding scope, so subsequent statements can refer back to it without naming. Method-self semantics (when `it` is bound inside an object action) take precedence; the global binding only fires between consecutive top-level statements.

```
a Car {make: "Toyota", price: 26000}
println(it.make)            # Toyota
mycar: it               # capture by name
a Car {make: "Honda"}       # rebinds it
println(it.make)            # Honda
println(mycar.make)         # Toyota (unchanged)
```

`every Concept has prop: val` — universal-slot quantifier; sugar for the existing concept-level `has`. Useful when emphasizing intent in literate-style scripts.

```
every Car has wheels: 4
every Car has make: ""
c: a Car {make: "Toyota"}
println(c.wheels)           # 4 (inherited default)
```

`what is X?` / `what is X's Y?` — interrogative queries. Evaluates the operand, pretty-prints `<source> is <value>`, and returns the value (so it can be assigned). The trailing `?` is required. Slot access via either possessive (`'s`) or dot notation is accepted.

```
mycar: a Car {make: "Toyota", price: 26000}
what is mycar?              # → mycar is a Car {make: "Toyota", price: 26000}
what is mycar's price?      # → mycar's price is 26000
what is mycar.make?         # → mycar.make is "Toyota"
```

> KM influence note: Axioma already implements most of KM's deep semantic machinery (frames, inheritance, situations via hypothetical contexts, defaults via defeasible rules, reification, persistence) and goes beyond it on truth representation (B4 bilattice, six-level epistemic grounding, five logic kinds with automatic dispatch). The natural-language surface forms above bring KM's ergonomics in alongside that deeper machinery. See tests/axioma/km/ for the complete test set.

1.14.10 Coreference merging — `unify x with y`

Collapses two named entities into one canonical entity with the union of their slot values. Solves entity resolution — when "Acme Inc." and "Acme Industries" turn out to be the same company, `unify` merges them in place. Composes with B4 paraconsistent truth, six-level grounding, and atomic transactions.

```
concept CelestialBody
morning_star: a CelestialBody {visible_at: "dawn"}
evening_star: a CelestialBody {visible_at: "dusk"}

unify morning_star with evening_star          # Russell's classical case

println(morning_star == evening_star)         # → true
```

Defeasible: `unify~ x with y` records the merge with grounding `conjecture` (cancellable later).

B4 paraconsistent slot merging: conflicting single-valued slots keep the canonical's value and record a `Both` truth marker. Multi-valued slots get set-unioned.

Transaction-safe: `unify` inside `transaction_begin()` / `transaction_rollback()` is reversible — both entities' pre-merge slots and identity are restored.

1.14.11 Intensional class descriptions — `the Concept where <pred>`

A Russell-style definite description with restrictor (KM §18.2). `the Stock where price > 1000` denotes the anonymous class of Stocks whose price exceeds 1000. Composes with `is`, comprehensions, and named classes.

```
concept Stock
Stock has price: 0
luxury: a Stock {price: 80000}

big: the Stock where price > 1000
println(luxury is big)                        # → true

# Inline membership test
println(luxury is (the Stock where price > 50000))   # → true

# Comprehension over the implicit extent
bigs: {X | X is (the Stock where price > 1000)}
```

Bare slot names in the predicate resolve to `it.<slot>` (same convention as `defines` predicate bodies). The intensional class is **transient** — it's not registered in the KB, so it adds no permanent vocabulary; use it for one-off queries or as a slot value when you want to denote a class without naming it.

1.14.12 Partitions and subsumption

`Concept partition Member1, Member2, …` declares the listed subclasses as **mutually exclusive**. An instance can be a member of at most one. The disjointness is enforced by auto-classification — when a `defines` predicate would classify an instance into a partition member that conflicts with an existing membership, the new classification gets the Belnap `both` truth value (paraconsistent — no crash).

```
concept Animal
Bird extends Animal
Mammal extends Animal
Fish extends Animal

Animal partition Bird, Mammal, Fish

robin: a Bird {}
println(robin is Mammal)        # → false (disjoint)
```

`A subsumes B` is the class-class subsumption infix (Russell class- inclusion, KM §18.3) — true when A is a superclass of B.

```
Animal subsumes Bird             # → true
Concept subsumes Bird            # → true
Bird subsumes Mammal             # → false
```

`partitions_of(Concept)` returns the declared partition tuples on a concept.

1.14.12.1 Querying a partition's live extent Four builtins make the partition laws — disjointness and exhaustiveness — queryable over a concept's actual instances, returning **witness instances** (not just a verdict) when a law fails:

```
concept Thing
UpToUs extends Thing
NotUpToUs extends Thing
Thing partition UpToUs, NotUpToUs
opinion: an UpToUs {}
weather: a NotUpToUs {}

is_partitioned(Thing)            # → true   (disjoint AND exhaustive over the extent)
partition_overlap(Thing)         # → {}     (instances in >1 part - empty   disjoint)
partition_gap(Thing)             # → {}     (extent instances in 0 parts - empty   total)
partition_member(opinion, Thing) # → UpToUs (the part it falls into, or `none`)

mystery: a Thing {}              # a bare Thing - in neither part
is_partitioned(Thing)            # → false  (exhaustiveness now fails)
mystery in partition_gap(Thing)  # → true   (the gap WITNESS)
```

This models Epictetus' dichotomy of control as a real partition rather than two hand-rolled relations. Each verdict is **open-world** — a statement about the facts seen so far; an empty extent is vacuously partitioned.

1.14.13 Description Logic — concept algebra ¬ + satisfiable

Concepts compose into **compound concept expressions** with the DL operators, and a built-in tableau reasoner answers subsumption, equivalence, and satisfiability questions about them. (and) and (or) and ¬ (not) build a first-class `ConceptExpr` value; , , and `satisfiable(...)` decide.

```
concept Person
concept Student
concept Employee
Student extends Person
Employee extends Person

ws: Student   Employee           # a compound concept (intersection)
@ws                              # → "ConceptExpr"

Student   Person                 # → true   (subsumption: left is the SUBconcept)
Person   Student                 # → false
(Student   Employee)   Person    # → true   (tableau derives it from `extends`)
Student   (Student   Employee)   # → true

Student   Employee               # → false  (equivalence = mutual subsumption)
(Student   Employee)   (Employee   Student)   # → true

satisfiable(Student   Employee)  # → true
satisfiable(Student   ¬Student)  # → false  (the complement clashes)
```

vs `subsumes` — converse readings. The glyph reads in the standard DL direction (`C D` means C is the *more specific* subconcept — `Student Person` is true). The English word `subsumes` reads the other

way (`A subsumes B` means A is the *more general* superclass — `Person subsumes Student` is true). They are converses of the same relation; pick whichever reads naturally.

Operator	Glyph	ASCII digraph	Word form	Result
conjunction	`C  D`	`C `sqcap D`	`C and/concept D`	`ConceptExpr`
disjunction	`C  D`	`C `sqcup D`	`C or/concept D`	`ConceptExpr`
complement	`¬C`	—	`not C`	`ConceptExpr`
subsumption	`C  D`	`C `sqsubseteq D`	`D subsumes C`	`true/false`
equivalence	`C  D`	—	—	`true/false`

`¬` / `not` is type-dispatched: applied to a concept it builds the complement; applied to a boolean it is ordinary logical negation, unchanged. `satisfiable` reasons over the TBox assembled from `extends` declarations; it is for concepts — boolean/propositional formulas use the separate `is_satisfiable`.

The reasoner is the ALC tableau in `reasoner/`.

1.14.13.1 Role restrictions, partitions, and extensional membership A **role restriction** quantifies a concept over a binary relation (a *role*). Use the COLON form `role: Concept` (some filler is a `Concept`) and `role: Concept` (every filler is a `Concept`); the role is a native relation.

```
concept Doctor
concept Cardiologist
Cardiologist extends Doctor
relation hasChild(x, y)

g: hasChild: Doctor                                    # a ConceptExpr (prints hasChild.Doctor)
satisfiable(( hasChild: Doctor)  ( hasChild: ¬Doctor))  # → false  (a filler can't be C and ¬C)
( hasChild: Cardiologist)  ( hasChild: Doctor)          # → true   (reasons through the role)
(¬( hasChild: Doctor))  ( hasChild: ¬Doctor)            # → true   (quantifier duality)
```

A declared **partition** now becomes real logic: `C partition A, B, …` makes the members pairwise disjoint and jointly exhaustive of C.

```
concept Animal
concept Cat
concept Dog
Cat extends Animal
Dog extends Animal
Animal partition Cat, Dog

satisfiable(Cat  Dog)      # → false   (disjoint)
(Cat  Dog)  Animal        # → true    (covering)
```

, , and `satisfiable` answer **open-world** (over declared axioms). **Extensional membership** `x is <ConceptExpr>` answers **closed-world**, over the instances and relation facts actually present:

```
felix: a Cat {}
felix is (Cat  ¬Dog)                   # → true
{p | p <- Animal, p is ¬Dog}            # compound/role concepts in a comprehension FILTER

dora: a Doctor {}
mary: a Cat {}                         # placeholder unrelated entity
hasChild(felix, dora)                   # role fillers should be ENTITIES
felix is ( hasChild: Doctor)           # → true   (some child is a Doctor)
felix is ( hasChild: Doctor)           # → true   (every child is a Doctor)
```

(Role fillers must be **entities** to carry concept membership — a bare string filler is not an instance of any concept.)

1.14.13.2 Defined concepts, `/`, and more spellings A **defined concept** gives a concept a necessary-and-sufficient body with `concept X ≡ <expr>` (the word `equivalent` works too). The definition is a genuine axiom in both directions, so subsumptions that follow from it decide, and membership is read off the body:

```
concept Person
concept Doctor
Doctor extends Person            # so a Doctor is a Person
relation hasChild(x, y)
concept Parent ≡ Person ⊓ (hasChild: Person)

Parent ⊑ Person                  # → true  (from the definition)
alice: a Person {}
dora:  a Doctor {}
hasChild(alice, dora)
alice is Parent                  # → true  (a Person with a Person child)
```

`Thing` (⊤) and `Nothing` (⊥) are built in — the universal and bottom concepts, available without declaration (you may still `concept Thing "..."`; it is an idempotent alias). Every concept satisfies `C ⊑ Thing` and `Nothing ⊑ C`; `satisfiable(Thing)` is true and `satisfiable(Nothing)` is false; extensionally every individual is a `Thing` and none is a `Nothing`. A partition of `Thing` therefore says the universe is exactly covered:

```
concept UpToUs
concept NotUpToUs
UpToUs extends Thing
NotUpToUs extends Thing
Thing partition UpToUs, NotUpToUs
(UpToUs ⊔ NotUpToUs) ≡ Thing     # → true  (the dichotomy is total)
```

Role restrictions have three interchangeable spellings — the COLON form above, the textbook **DOT** form, and the OWL **Manchester** words (role on the left):

```
(∃ hasChild.Doctor) ≡ (∃ hasChild: Doctor)        # DOT
(hasChild some Doctor) ≡ (∃ hasChild: Doctor)     # Manchester
(hasChild only Doctor) ≡ (∀ hasChild: Doctor)     # Manchester
```

(The DOT form never disturbs the symbolic quantifier `∀x. flies(x)`, and `some`/`only` stay ordinary words outside this position — `some S is P` categorical syntax and an `only:` binding are untouched.)

A compound concept can also drive a comprehension **source** for the `/` cases:

```
{x | x <- (Person ⊓ ¬Doctor)}    # the persons who aren't doctors
```

A bare `¬C` / `∃R.C` / `∀R.C` / `Thing` source has no finite extent without a domain universe, so it is a clean error pointing you at FILTER position (`{x | x <- C, x is <expr>}`); those universe-bearing sources are deferred.

The older `dl_kb(...)` family of builtins remains for explicit string-keyed knowledge bases. DL operators evaluate in the tree-walking interpreter (not under `--vm`, which does not compile concept declarations).

1.14.14 Enumerated types — `Concept enumerates A, B, C` (Pascal/Ada/Inform-7)

`X enumerates A, B, C, …` declares X as an **enum concept** whose extent is sealed and ordered. Each member becomes a `ConcreteEntity` instance of X with `ord` (0-based declaration index) and `name` properties. Members are bound bare (`Mon` resolves directly) and also as concept-qualified (`Day.Mon`).

```
Day enumerates Mon, Tue, Wed, Thu, Fri, Sat, Sun

println(Mon)              # Mon (inspect renders the bare name)
println(Mon.ord)          # 0
println(Day.Mon == Mon)   # true (shared identity)
println(succ(Wed))        # Thu
println(pred(Fri))        # Thu
println(Day.first)        # Mon
println(Day.last)         # Sun
println(len(Day))         # 7

# Ordered comparison uses ord:
println(Mon < Fri)        # true

# Iteration walks members in declaration order:
foreach d in Day [
  println(d.name)
]

# Strict cross-enum (Pascal/Ada-style):
Color enumerates Red, Green, Blue
# Mon < Red             # ERROR: cannot compare Day and Color
```

Built-ins: `succ`, `pred`, `len`, `Day.first`, `Day.last`. Cross-enum comparison errors with a clean message (Pascal/Ada strict). Re-declaring a member name across two enums collides at declaration time — qualify with `Day.Mon` to disambiguate. Tests under `tests/axioma/types/`.

1.14.15 Integer subrange types — `Concept ranges A..B` (Pascal/Ada)

`X ranges Low..High` declares X as an integer subrange type. Used in `::` annotations, the bound is runtime-checked; with the `--typecheck` static pre-pass (§26), literal violations are caught before any code runs.

```
Digit ranges 0..9
Percent ranges 0..<100              # half-open

d :: Digit : 5                  # OK
d :: Digit : 12                 # type error
p :: 0..<100 : 50               # inline range annotation
p :: 0..<100 : 100              # type error

# Reassignment honors the binding-time snapshot:
r :: Digit : 3
r = 7                                # OK
r = 99                               # type error
```

The `by` step is rejected in `ranges` declarations (membership becomes ambiguous). Inverted bounds (`5..0`) and empty half-open (`5..<5`) also error at declaration time.

1.14.16 Enum subrange — `Sub extends Enum in From..To` (Pascal/Ada)

`Workday is Day in Mon..Fri` declares a subtype of an existing enum restricted to a contiguous slice. Every Workday IS a Day (widening is automatic); a Day with ord outside the slice cannot be a Workday (tightening is runtime-checked). Member identity is shared — `Mon` is still `Day.Mon`.

```
Day enumerates Mon, Tue, Wed, Thu, Fri, Sat, Sun
Workday extends Day in Mon..Fri
Weekend extends Day in Sat..Sun
```

```
d :: Workday : Wed              # OK
d :: Workday : Sat              # type error: Sat outside Workday
any :: Day : Wed                # widening - every Workday is a Day

# Iteration visits only the slice:
foreach d in Workday [
  println(d.name)                   # Mon Tue Wed Thu Fri
]
println(len(Workday))               # 5
```

Endpoints must belong to the named parent enum (cross-enum endpoints error). Inverted bounds, integer endpoints with an enum parent, and non-enum parents all error at declaration time.

1.14.17 Slot-metadata helpers — inverse, transitive, find-or-create

Three small builtins covering common KR patterns. All ship as configuration calls; no new keywords.

```
# Inverse slots - auto-maintain bidirectional relations
concept Car
concept Engine
Car has parts: none
Engine has part_of: none
inverse_slot("parts", "part_of")

car: a Car {}
engine: a Engine {}
car.parts: engine             # auto: engine.part_of = car

# Transitive slots - walk a chain in one builtin call
concept Person
Person has parent: none
transitive_slot("parent")

alice: a Person {}; bob: a Person {}; carol: a Person {}
alice.parent: bob
bob.parent: carol
ancestors: transitive_closure(alice, "parent")   # [bob, carol]

# find_or_create - definite description with reification
concept Country
Country has name: ""
usa1: find_or_create(Country, {name: "USA"})
usa2: find_or_create(Country, {name: "USA"})    # same instance as usa1
```

`find_or_create` is the canonical KB ingestion primitive — it makes "if this entity already exists, give me it; else make it" a one-liner. Inverse-slot propagation is one-step (the propagation guard prevents recursive inverse-of-inverse loops). Cycle-safe entity rendering is automatic — `ConcreteEntity.Inspect` detects already-visiting entities and emits a short reference.

1.14.18 Auto-classification via `defines`

A fourth concept-verb that attaches a **membership predicate** to a class. Any instance whose slots satisfy the predicate is automatically classified; slot mutations re-evaluate and promote/demote in real time. Faithful to KM §17, with Axioma additions (B4 truth, six-level grounding, defeasibility).

```
concept Person
Person has age: 0

Adult defines { age >= 18 and is Person }
Senior defines { age >= 65 and is Adult }
Sage defines~ { age >= 80 }      # defeasible - grounding=conjecture

alice: a Person {age: 30}
alice is Adult                   # true (auto-classified)
alice is Senior                  # false

alice.age: 70
alice is Senior                  # true (re-classified on slot mutation)

alice.age: 5
alice is Adult                   # false (auto-demoted)
```

Distinction from `has`: `Concept has prop: val` is *unidirectional* (`is(x, C) → prop(x, val)` — every member has this property). `Concept defines { body }` is *bidirectional* (`is(x, C)  body(x)` — membership iff predicate). See KM §17 for the canonical Mexican/Square contrast.

The three logical roles, separately surfaced:

Role	Direction	Surface
Slot template (structural)	—	`Adult has age: 0`
Implication (one-way)	`→`	`is(X, Adult) ==> voting(X, true)`
Equivalence (two-way)		`Adult defines { age >= 18 and is Person }`

Predicate body conventions:

- Bare slot names (`age`, `price`) implicitly refer to `it.<slot>`.
- `is Concept` in prefix position desugars to `it is Concept`.
- `and` / `or` / `not` and comparison operators work normally; Belnap values propagate through.

Grounding & truth integration:

- Strict `defines` → grounding=**theorem**; `defines~` → **conjecture**.
- `grounding("isa", instance, Concept)` reports the grounding.
- `proof("isa", instance, Concept)` reports the derivation chain.
- Demoted classifications get `truth("false")` (Belnap), preserving provenance for audit. Re-promotion re-fires the predicate cleanly.
- Belnap `Both` results from predicate body propagate to the is fact's truth without crashing (paraconsistent semantics).

Auto-create: if the named concept doesn't exist when `defines` is evaluated, it's created automatically (extending root `Concept`). No need for a separate `concept Adult` line.

1.14.19 Value constraints — `constrain()`

Write-time validation on slot writes. Where `defines` *classifies* instances based on their slot state, `constrain` *gatekeeps* the slots themselves: register a predicate that every subsequent write must satisfy, and bad writes are silently rejected (the slot retains its prior value) with a warning to stderr.

```
concept Customer
Customer has age: 0
```

```
Customer has email: ""

constrain(Customer, "age",   lambda v => v >= 0 and v <= 150)
constrain(Customer, "email", lambda v => contains(v, "@"))

alice: a Customer {age: 30, email: "alice@example.com"}   # accepted
bad: a Customer {age: -5}                                 # WARN; age stays at default 0

alice.age: 200            # WARN; alice.age stays at 30
alice.age: 45             # accepted

# Multiple constraints AND
constrain(Customer, "age", lambda v => v != 13)
teen: a Customer {age: 13}                                  # rejected (second constraint)

# Subclass inheritance - every constraint declared on Customer applies to VIP
VIP extends Customer
v: a VIP {age: -1}                                          # rejected
```

Predicate contract:

- Takes exactly one argument — the value being written.
- Returns a Boolean-ish value (truthy = accept, falsy = reject).
- May reference other slots only via captured closure variables; cross- slot constraints (predicates referencing `self.other_slot`) are better expressed as `defines`, which already sees the full entity.

Enforcement sites: every slot-write path is hooked —

- `a Concept {slot: val, ...}` — the per-property initialization loop.
- `entity.slot: val` — direct assignment.
- `entity[slot -> val]` — ErgoAI frame-attribute set.

Distinction from neighbouring features:

Feature	Fires when	On failure
`cardinality` (B.8)	Write to single-valued slot already populated with different value	Last-write-wins + violation marker
`defines` (KM §17)	Slot mutation triggers re-classification	Promote/demote membership in defined class
`constrain` (this)	Any slot write	Skip write + warning to stderr

Introspection:

```
constraints_of(Customer, "age")   # → 2 (count of predicates on this slot, walking ancestors)
constraints_of(Customer)          # → {age: 2, email: 1} (all constrained slots)
```

KM mapping: closes the gap with KM 2.0 §12 (Value Constraints) — the `must-be-a` / `must-be` facets — without committing to a specific type-system facet. The predicate body has the full expression language available, so `must-be Integer where v >= 0 and v <= 150` is just `lambda v => v >= 0 and v <= 150` when the type check is sugar for an `instanceof` call.

Test: tests/axioma/concepts/test_value_constraints.ax.

1.14.20 English paraphrases for `why` — `paraphrase()`

KM §19.3. Register an English template against a concept; when `why X is Concept` fires, render the template instead of the default structured proof. Closes the gap between Axioma's structured explanation

engine and the regulator-/user-friendly English output that compliance domains require.

```
concept Person
Person has age: 0
Person has name: ""

Adult defines { age >= 18 and is Person }
paraphrase(Adult, "{it.name} qualifies as an Adult because their age ({it.age}) is at least 18.")

alice: a Person {name: "Alice", age: 30}
why alice is Adult
# → "Alice qualifies as an Adult because their age (30) is at least 18."
```

Placeholder grammar:

Form	Resolves to
`{it}`	The instance — its `name` slot if present, otherwise `Inspect()`
`{it.slot}`	Value of the named slot on the instance
`{unknown}`	Left intact (debugging aid)

Inheritance: a subclass without its own paraphrase inherits its parent's. Most-specific wins; the parent-chain walk mirrors how `constrain` constraints are inherited.

Fallback: if no paraphrase is registered for the concept (or any ancestor), `why` runs the existing structured-proof rendering. The feature is purely additive — adopt it for the classes where English output matters.

Composition:

- `defines` + `paraphrase` → auto-classification rendered in English
- `unify` + `paraphrase` → the canonical entity's slot values are resolved before placeholder substitution
- Subclass + `paraphrase` on parent → subclass inherits the parent's template until it registers its own

Test: tests/axioma/concepts/test_paraphrase.ax.

1.14.21 Defined instances — `Concept identified by ...`

KM §17.3 — automatic coreference by identity key. Where `defines` auto-*classifies* instances by their slot state, defined instances auto-*merge* instances by a declared identity key. Declare which slot(s) uniquely identify an individual, and any two instances of the concept with matching key values are automatically unified — reactively, on every slot write, with no explicit `unify` call.

The **recommended canonical surface** is the `has/identity` slot refinement — identity-ness is slot metadata, so it belongs on the slot declaration rather than in a separate statement that re-names the slot. It parallels Axioma's existing keyword refinements (`declare/persist`, `axiom/transient`, `is/same`).

```
concept Customer
Customer has/identity ssn: ""        # ssn is the identity key
Customer has name: ""

c1: a Customer {ssn: "123-45-6789", name: "Alice Smith"}
c2: a Customer {ssn: "123-45-6789", name: "A. Smith"}
println(c1 == c2)        # → true (auto-unified)
println(c1.name)         # → A. Smith (newest write wins)

# Composite key - mark each participating slot; all /identity slots
# on a concept jointly form ONE key (declaration order preserved)
```

```
concept Order
Order has/identity customer_id: ""
Order has/identity order_number: ""
Order has total: 0
```

Alternative surfaces (all valid, all lower to the same registry, nothing deprecated):

```
Customer identified by ssn                        # statement form
Order identified by customer_id, order_number     # composite, explicit grouping
define_equivalence(Customer, "ssn")               # functional builtin
```

It is the **automatic dual** of the two manual coreference primitives:

Primitive	Trigger	Effect
`find_or_create(C, {...})`	explicit call	return existing match or make fresh
`unify x with y`	explicit call	merge two named entities
`has/identity` slot refinement	every slot write	auto-merge any two C instances with matching key

Semantics:

- Each `has/identity` slot is appended to the concept's key in declaration order; multiple such statements accumulate into one composite key (re-declaration is deduped/idempotent).
- An *incomplete* key (key slot missing, or holding `""`) never matches — a partially-populated instance is not a dedup candidate yet. A later write that *completes* the key triggers the merge.
- The triggering entity wins (newest write wins on slot conflict); the pre-existing match redirects via `Canonical`.
- Reuses the full `unify` machinery — B4 paraconsistent slot-merge, transaction rollback, `strength`/`proof`/`why`.
- The merged-away loser is removed from the concept extent, so `{X | X <- Concept}` counts distinct identities, not raw records.
- Subclasses inherit the parent's key.

Introspection: `equivalence_key_of(Concept)` returns the key slots as an array of strings (parent chain walked), `[]` if none.

KM mapping: closes KM 2.0 §17.3 (Defined Instances — Testing for Equivalence), the equivalence half of KM's automatic-classification duality. The membership half (§17.2 Defined Classes) is Axioma's `defines`.

Test: tests/axioma/concepts/test_defined_instances.ax.

1.14.22 `inspect / see` — identity-passing evaluate-and-display

A prefix directive that evaluates an expression, prints `<source> = <value>` to stdout, and returns the value unchanged. Modelled on Elixir's `IO.inspect` / Julia's `@show` — the prevailing non-imperative idiom for inline inspection.

```
c: a Car {price: 150}
inspect c.price                # prints "c.price = 150"
inspect c is Car              # prints "(c is Car) = true"

# Identity-pass: composes inside expressions
n: inspect expensive()    # binds n; the value also printed
```

Lowest-precedence prefix, so `inspect fred is Person` consumes the whole `is` expression without parens.

`see` is a true alias — same token, same semantics. `see c.price` is identical to `inspect c.price`; use whichever reads better in context. The lexer maps both `inspect` and `see` to the `INSPECT` token, so there's one implementation.

The conformance harness (tests/km-conformance/) uses `inspect` instead of `println` boilerplate, making the Axioma side structurally parallel to KM's `;;CHECK` + form, and the harness compares the two ordered value lists positionally.

Test: tests/axioma/concepts/test_inspect.ax.

1.14.23 Cumulative slots — `Concept has slot/cumulative: val`

KM-conformant **additive slot inheritance**. By default Axioma *overrides* an inherited slot when a subclass redeclares it. The `has/cumulative` slot refinement opts a declaration into KM's semantics: the newly declared value is **unioned** with the inherited value into a set. Override (default) and accumulate (opt-in) coexist.

```
concept Animal
Animal has legs: 4
Dog extends Animal
Dog has legs/cumulative: 3        # union with inherited 4
println((a Dog).legs)             # → {3, 4}

Cat extends Animal
Cat has legs: 3                   # plain decl → override
println((a Cat).legs)             # → 3
```

The refinement attaches to the **slot name** (`slot/cumulative`), parallel to `has/identity` but on the slot. Semantics:

- The union is computed at concept-declaration time, so `is` must precede the cumulative `has` (the canonical order).
- A cumulative slot is always set-valued; set operands are flattened so repeated cumulative declarations down a taxonomy accumulate into one flat set; duplicates drop.
- Slots not declared `cumulative` keep the default override behaviour.

KM mapping: closes the slot-inheritance gap found by the KM-conformance harness (tests/km-conformance/ case 06) — KM unions slot values across a taxonomy, and `has/cumulative` lets Axioma reproduce that while keeping override as its default.

Test: tests/axioma/concepts/test_cumulative_slots.ax.

1.15 14. Logic Programming (Prolog-Like)

Axioma provides Prolog-like relational programming through **set-based deterministic queries** — no backtracking, complete result sets, mathematical foundations.

1.15.1 Facts

Declare a relation with `relation`, then assert facts with `assert` (or the graded `axiom` / `postulate` — see §15):

```
relation parent(x, y)
relation age(x, n)
relation works(x, role)
```

```
assert parent("John", "Mary")
assert parent("Mary", "Alice")
assert parent("John", "Bob")
assert age("Alice", 25)
assert works("John", "Engineer")
```

> **Note:** an undeclared bare call like `parent("John", "Mary")` at statement position is read as a *function call* and errors with `Undefined word: parent`. Declare the relation first (after which a bare `parent(...)` call adds a fact), or prefix with `assert`. As a shorthand, a bare header whose arguments are all **uppercase-initial** — `pair(X, Y)` — is itself read as a relation declaration (the optional-`relation` form).

1.15.2 Pattern queries

```
{X | X <- parent(X, _)}                          # All parents
{Y | Y <- parent("John", Y)}                     # John's children
{(X, Y) | parent(X, Y)}                          # All parent-child pairs
```

1.15.3 Filtered queries

```
{X | X <- parent(X, _), age(X, A), A >= 18}      # Adult parents
```

1.15.4 Cross-relation unification (set ops)

```
parents: {X | X <- parent(X, _)}
workers: {X | X <- works(X, _)}
working_pars: parents   workers
```

1.15.5 Variable chain unification

```
# Equivalent to Prolog: grandparent(X,Z) :- parent(X,Y), parent(Y,Z)
johns_kids: {Y | Y <- parent("John", Y)}
johns_grands: {Z | Y <- johns_kids, Z <- parent(Y, Z)}
```

1.15.6 Complex term matching

```
relation person(x, attr1, attr2)
relation location(x, attr1, attr2)
assert person("John", ("age", 30), ("job", "engineer"))
assert location("John", ("city", "NYC"), ("country", "USA"))

{X | X <- person(X, _, ("job", "engineer"))}      # All engineers
{X | X <- location(X, ("city", "NYC"), _)}        # NYC residents
```

1.15.7 Relations as first-class values

A relation name resolves to a first-class `Relation` value (not its fact-adder builtin), so it is type-introspectable and **iterable over its extent** — symmetric with how a concept name iterates its instances:

```
relation parent(x, y)
assert parent("John", "Mary")
assert parent("John", "Bob")

type(parent)                        # → "Relation"   (@parent is the same)
for (x, y) in parent [ ... ]        # iterate facts directly - no comprehension
{(X, Y) | (X, Y) <- parent}         # bare relation as a comprehension source
{x | x <- node}                     # unary relation → bare elements
```

```
parent("Alice", "Carol")          # STILL callable - adds a fact
```

A predicate's **rules** are introspectable too. `rules_of(pred)` returns an `Array` of `RuleClause` values, completing the F-logic *concept → relation → rule* arc (concepts iterate instances, relations iterate facts, rules are inspectable as data):

```
relation edge(x, y)
path(X, Y) :- edge(X, Y)
path(X, Y) :- edge(X, Z) and path(Z, Y)

rs: rules_of("path")              # Array of 2 RuleClause
type(rs[1])                       # → "RuleClause"   (rs[1] is RuleClause → true)
rs[1].head                        # → "path(X, Y)"        (.head_name / .name → "path")
rs[1].body                        # → "edge(X, Y)"
rs[1].defeasible                  # → false               (.strict → true)
rs[1].direction                   # → "backward"          ("forward" for ==> / ~~>)
{c.head_name | c <- rules_of("path")}
```

> **Note:** because a relation name is now a *value*, the KB builtins that take a relation (`insert`, `forget`, `truth`, `set_truth`, `grounding`, `cancel`, `challenge`, …) expect the relation's **name as a string** — `grounding("parent", "John", "Mary")`, not `grounding(parent, …)`. See §16 and §17.

1.15.8 HiLog meta-queries

Reflective queries over predicate names and arities:

```
predicates_of("alice")            # All facts mentioning alice
predicate_names()                 # All predicate names with  1 fact

?P("tweety")                      # Prolog-style meta-call: which predicates mention tweety?
?P("alice", ?Y)                   # Pattern with wildcard in 2nd position
?P(?X, "carol")                   # Strict arity, per-position unification
```

1.16 15. Knowledge Base, Axioms & Postulates

1.16.1 Declaring knowledge — the six grounding grades

Every stored fact carries an **epistemic grounding** — *how strongly it is held* — on this ordered ladder:

`axiom > postulate > theorem > conjecture > hypothesis > datum`

You declare a fact at a grade with the matching keyword; derivation and runtime `insert` produce the rest. Read a fact's grade back with `grounding(rel, args…)`.

Grade	How you express it	Meaning
`axiom`	`axiom fact`	foundational truth — the highest grade
`postulate`	`postulate fact`	a tentative claim that may be refuted
`theorem`	*derived* by a **strict** rule (§16), or `insert(rel, …, "theorem")`	proven from axioms/postulates
`conjecture`	*derived* by a **defeasible** rule (§16), or `insert(rel, …, "conjecture")`	plausibly inferred; may be defeated

Grade	How you express it	Meaning
`hypothesis`	`hypothesis fact`, or `insert(rel, …, "hypothesis")`	a working assumption
`datum`	plain `assert fact`, or `insert(rel, …)` with no grade	a bare fact — the floor of the ladder

```
axiom      parent("Adam", "Cain")     # foundational
postulate  married("Cain", "Awan")    # tentative - may be refuted
hypothesis visited("Cain", "Nod")     # a working assumption
assert     lives_in("Cain", "Nod")    # plain fact → grounding "datum"

grounding("parent", "Adam", "Cain")   # → axiom

# theorem / conjecture come from derivation (§16); or set any grade at runtime:
insert("parent", "Seth", "Enos", "theorem")
```

Grounding **propagates** through derivation — a strict rule over axioms yields a **theorem**, a defeasible rule yields a **conjecture** (the conclusion is graded `min(body_groundings)`). See §16 for the mechanics, and `challenge` (below) / `cancel` · `force_cancel` (§17) for revising graded facts.

1.16.2 Persistence control

```
axiom/persist     gravity = 9.81          # Saved to cascade.db
axiom/transient   demo_axiom = 42         # Session-only
postulate/persist working = "..."
postulate/transient debug = "..."
```

Default: **REPL persists, scripts are transient.**

See `resources/docs/claude/POSTULATE_AXIOM_SYSTEM.md` and `REFINEMENT_PERSISTENCE.md`.

1.16.3 The shared SQLite KB

Axioma and Cascade share a single SQLite database (`cascade.db`) using the same schema. Both processes can read/write concurrently.

```
./axioma --no-kb script.ax        # Skip Cascade KB preload (~10× faster)
```

1.17 16. Rules, Derivation & Epistemic Grounding

1.17.1 Rule forms

Operator	Direction	Strictness	Produces
`<=`	Backward	Strict	**Theorem**
`<~~`	Backward	Defeasible	**Conjecture**
`==>`	Forward	Strict	**Theorem**
`~~>`	Forward	Defeasible	**Conjecture**

1.17.2 Strict rules

```
grandparent(X, Z) <= parent(X, Y) and parent(Y, Z)
parent(P, Q) and parent(Q, R) ==> grandforward(P, R)
```

1.17.3 Defeasible rules

A **defeasible** rule states a *default* — an inference that holds **unless overridden**. Where a *strict* rule's conclusion is a **theorem** that always follows, a defeasible rule's conclusion is a **conjecture**: a plausible default that can be *defeated* by a stricter rule, by conflicting evidence, or by an explicit `cancel`. This is how Axioma expresses "typically / normally / by default" reasoning (non-monotonic logic) — *birds typically fly*, even though penguins don't.

```
flies(X)  <~~ bird(X)     # backward: flies(X) holds by default when bird(X) does
bird(X)   ~~> flies(X)    # forward:  the same default, written premise-first
```

Both forms express the **same** rule and produce **conjecture**-grade conclusions; they differ only in reading direction — `<~~` writes the conclusion first, `~~>` the premise first (see the rule-forms table above). Their strict twins `<=` (backward) and `==>` (forward) instead produce **theorems**:

	Strict (→ theorem)	Defeasible (→ conjecture)
backward (conclusion ← body)	`<=`	`<~~`
forward (body → conclusion)	`==>`	`~~>`

A defeasible conclusion is **overridden** when a strict rule proves the contrary, or by a `cancel` / `force_cancel` directive (§17) — and grounding-aware `cancel` refuses to defeat a theorem. So `flies("tweety")` is a conjecture you can retract, while a theorem stands until you revise its premises.

1.17.4 Frame-logic rule bodies

Bodies can mix `is`, function calls, attribute paths, and conjunctions:

```
TradeLink extends Concept
TradeLink has source
TradeLink has target

{tl | tl is TradeLink, tl.target.gdp_billion > tl.source.gdp_billion * 2}
```

1.17.5 Epistemic grounding (six ordered grades)

```
axiom > postulate > theorem > conjecture > hypothesis > datum
```

- **axiom** — declared with `axiom`. Foundational.
- **postulate** — declared with `postulate`. Tentative.
- **theorem** — derived by a **strict** rule from axioms/postulates.
- **conjecture** — derived by a **defeasible** rule.
- **hypothesis** — user-marked working assumption.
- **datum** — a plain `asserted` fact, or a runtime `insert(...)` without explicit grounding — the floor of the ladder.

Strict rules over axioms produce theorems (provenance is tracked, not collapsed). Defeasible rules produce conjectures. Grounding propagates via `min(body_groundings)`.

The introspection builtin is **`grounding`**, and the floor grade is **`datum`**.

1.17.6 Grounding & Kind as first-class values

`grounding(...)` and `truth_kind(...)` return typed values, not bare strings. A `Grounding` is **ordered** (the ladder above); a `Kind` is a **flat** five-member set (`logical` / `empirical` / `transcendental` / `motive` / `metalogical`). Both coerce against a plain String, so existing `== "axiom"` comparisons keep working.

```
relation edge(x, y)
axiom edge("a", "b")
path(X, Y) :- edge(X, Y)

g: grounding("edge", "a", "b")              # → axiom   (a Grounding value)
type(g)                                      # → "Grounding"   (g is Grounding → true)
grounding("edge", "a", "b") >= "conjecture"  # axiom >= conjecture → true  (ordered)
grounding("edge", "a", "b") >= grounding("path", "a", "b")   # axiom >= theorem → true

relation color(x, y)
axiom/empirical color("apple", "red")
k: truth_kind("color", "apple", "red")       # → empirical   (a Kind value)
type(k)                                      # → "Kind"
k == "empirical"                             # → true  (String coercion - no ordering on Kind)
```

The possessive accessors agree with the builtins: `fact's grounding` and `fact's kind` return the same `Grounding` / `Kind` value types.

1.17.7 Bilattice truth propagation

Every stored fact carries a Belnap B4 value (`T`, `F`, `Both`, `Neither`) that propagates through Horn-clause bodies via lattice meet. Paraconsistent contamination (`Both`) survives derivation.

```
set_truth("parent", "John", "Mary", "true")
truth("parent", "John", "Mary")          # Returns Belnap value
```

1.17.8 Provenance & introspection

The relation is named by **string** (a relation name is now a first-class value — §14):

```
grounding("parent", "John", "Mary")       # a Grounding value: axiom / theorem / conjecture / datum / .
proof("grandparent", "John", "Alice")    # Walks derivation chain back to axioms
why grandparent("John", "Alice")         # Prose explanation (keyword form - takes the conclusion call)
rules_of("path")                          # The predicate's rules as first-class RuleClause values
```

1.17.9 Challenging axioms

Axioms must be challengeable (Q1 requirement):

```
challenge("parent", "John", "Mary")       # Marks suspect
challenged("parent", "John", "Mary")      # Returns true/false
```

1.17.10 The axiological (value) axis

Orthogonal to epistemic grounding, a fact can carry an explicit **value judgment** relative to an authority. The axis has three *judged* stances plus a distinct fourth "never judged" state:

```
epictetus: agent("epictetus", "Stoic - virtue is the only good")
value_good("courage", "any_agent", epictetus)
value_bad("cowardice", "any_agent", epictetus)
value_indifferent("wealth", "any_agent", epictetus)   # the Stoic adiaphoron

value_kind_of("courage", "any_agent")     # → "good"
value_kind_of("wealth", "any_agent")      # → "indifferent"   (a JUDGMENT)
value_kind_of("has_blue_eyes", "any_agent")  # → "neutral"    (SILENCE - never valued)
facts_by_value_kind("indifferent")        # enumerate the judged-indifferent facts
```

`value_indifferent` makes "positively judged neither-good-nor-bad" a first-class stance, distinct from a fact that was simply never valued (`neutral`) — the difference between the Stoic *adiaphora* and mere silence.

1.17.11 Aspectual facts — `qua`

`qua` (Latin "in the capacity of") makes the *same* fact viewable under a **handle**, and rules can key on the handle — so one fact can carry two value-laden framings with divergent consequences (intensionality made executable):

```
relation departed(x)
departed("the_estate") qua "lost"
departed("the_estate") qua "given_back"

framings_of("departed", "the_estate")            # → {"lost", "given_back"}
framed_qua("departed", "the_estate", "lost")     # → true

disturbs(it)  <~~ it qua "lost"                  # `it` binds to every fact tagged "lost"
at_peace(it)  <~~ it qua "given_back"
# the SAME fact is now both disturbing AND at peace - Enchiridion 11
```

`it qua "h"` binds `it` to every fact tagged `h`; the relation-anchored form `rel(X) qua "h"` instead binds `X` to the argument. `qua` stays an ordinary identifier outside this position (a same-line soft keyword).

1.18 17. Mutation, Transactions & Conflict Resolution

1.18.1 Runtime mutation

The relation is named by **string** (relation names are first-class values now — §14):

```
insert("parent", "Eve", "Seth")          # Defaults to grounding "datum"
insert("parent", "Eve", "Seth", "axiom") # Explicit grounding
forget("parent", "Eve", "Seth")          # Retract from all groundings
```

> **Note:** `delete` and `retract` are reserved keywords. Use **`forget`** for the function-call form. The statement form `retract [...]` is a separate existing feature.

1.18.2 Atomic transactions

```
transaction_begin()
insert("parent", "X", "Y")
set_truth("parent", "X", "Y", "true")
transaction_commit()                     # OR transaction_rollback() - undoes ops AND metadata
```

1.18.3 Defeasible-conflict resolution

```
relation bird(x)
flies(X) <~~ bird(X)
assert bird("tweety")                    # Tweety conjecturally flies
cancel("flies", "tweety")                # Tweety is a penguin
canceled("flies", "tweety")              # true
uncancel("flies", "tweety")              # Restore
```

Canceled facts retain their provenance but are filtered out of comprehensions.

1.18.3.1 Grounding-aware cancellation

`cancel` is **grounding-aware**: it refuses to defeat a derived **theorem** (a strict consequence), returning a non-fatal `"refused: …"` string that points you at revising a premise (`challenge`) or overriding with `force_cancel`. Defeasible conclusions (conjectures) and base posits (axioms / postulates / data) stay directly cancelable.

```
tranquil(P) <== free(P)                  # strict rule → derives a theorem
assert free("sage")
cancel("tranquil", "sage")               # → "refused: … is a theorem …"  (NOT canceled)
force_cancel("tranquil", "sage")         # → "canceled: tranquil(\"sage\")"  (explicit override)
```

1.18.3.2 `forget_cascade` — justification-based retraction (a TMS)

Plain `forget` retracts a premise but **orphans** the conclusions already derived from it. `forget_cascade` walks the derivation chains forward, withdraws the premise *and* every materialized fact that transitively depends on it, then lets lazy re-derivation rebuild whatever still has independent support:

```
disturbed(P) <~~ judges_bad(P)
assert judges_bad("novice")
{p | p <- disturbed(p)}                   # → {"novice"}
forget_cascade("judges_bad", "novice")
{p | p <- disturbed(p)}                   # → {}   (the conclusion lifts with its premise)
```

A conclusion that also follows from a surviving rule re-derives automatically on the next query.

1.19 18. Stack-Based Programming

Axioma implements a first-class stack model: - a **first-class `Stack` value** — construct it with `s: a Stack` (or the function form `stack_new()`); both build a real `Stack` (`@s` → `"Stack"`) - a **global interpreter stack**, inspectable from the REPL with `:stack`

A `Stack` value has **two equivalent surfaces** — use whichever reads best:

Natural-language — a message verb *mutates* (statement position) and a possessive *reads* (expression position):

```
s: a Stack
s push 5
s push 10
s's top            # → 10        (peek;  s's depth → 2,  s's empty → false)
s's items          # → [10, 5]   (top-to-bottom)
s pop              # remove the top - value discarded in statement position
v: pop(s)          # …so capture a popped value with the pop() function
```

Mutating verbs: `push pop peek drop clear dup swap over rot nip tuck`. Possessive reads: `top`/`peek`, `bottom`/`base`, `depth`/`size`/ `height`/`length`, `empty`, `items`/`elements`/`contents`. `top` on an empty stack returns `none` (check `s's empty` first).

Function form (Forth / Pop-11) — every operation is a function taking the stack as its first argument: `push(s, x)`, `pop(s)`, `peek(s)`, `dup(s)`, … These are the **same** operations as the verbs above (kept as aliases) and are the form used in the tables below. (The natural-language surface is evaluator-only; under `--vm`, use the function form.)

1.19.1 Core operations

Op	Effect	Description
`stack_new()`	`→ s`	Create a new empty stack
`push(s, x)`	`… → … x`	Push `x` onto `s`

Op	Effect	Description
`pop(s)`	`… x → …`	Remove and return the top
`peek(s)`	`… x → … x`	Return the top without removing it
`depth(s)` / `stacklength(s)`	`→ n`	Current number of items
`clear(s)` / `erase(s)`	`… →`	Empty the stack

1.19.2 Stack-shuffle operations

All take the stack as the first argument and mutate it in place.

Op	Effect	Description
`dup(s)`	`a → a a`	Duplicate top
`swap(s)`	`a b → b a`	Swap top two
`rot(s)`	`a b c → b c a`	Rotate top three
`over(s)`	`a b → a b a`	Copy second to top
`drop(s)`	`a →`	Discard top
`nip(s)`	`a b → b`	Drop second
`tuck(s)`	`a b → b a b`	Copy top below second
`pick(s, i)`	`→ … x`	Copy the element at index `i` (0 = top) to the top
`roll(s, i)`	`→ … x`	Move the element at index `i` to the top

1.19.3 Bulk & depth operations

Op	Description
`dupnum(s, n)`	Duplicate top `n` times
`erasenum(s, n)`	Drop top `n` items

1.19.4 Array conversion

Op	Description
`stack_to_array(s)`	Snapshot the stack as an array, top first
`array_to_stack(arr)`	Build a new stack from an array

1.19.5 Example

```
s: stack_new()
push(s, 1)
push(s, 5)
dup(s)                 # 1 5 5
swap(s)                # 1 5 5  (top two swapped)
println(pop(s))        # 5
println(depth(s))      # 2
println(stack_to_array(s))   # [5, 1]   (top first)
```

1.19.6 REPL inspection

The global interpreter stack is inspected with REPL commands:

```
:stack          # Show stack contents
:s              # Alias for :stack
:stack trace    # Show stack with type information
:stack depth    # Show stack depth
:stack clear    # Clear the stack
```

See `tests/axioma/stack/` for comprehensive examples.

1.20 19. Three Notations, One Tree

Most languages pick one notation and bury the others under syntactic surcharge: Lisp commits to prefix, ML to infix, Forth to postfix. Axioma treats all three as **surface forms of the same AST node**. The unifying claim is concrete: for any expression you can write in two notations, `fullform` returns the same string.

```
fullform(2 + 3)        # "+(2, 3)"   - infix
fullform(+(2, 3))      # "+(2, 3)"   - Julia-style prefix
fullform(seq_of(2, 3, +))  # "Sequence(2, 3, +)" - postfix sequence (a different node)
```

The first two are identical because both parse to the same `InfixExpression`. The third is a `SequenceExpression` whose stack-reduction semantics happen to produce the same value.

1.20.1 19.1 Prefix: operators are functions

Every binary operator can be called like an ordinary function. The parser recognizes `op(args)` where `op` is one of:

```
+   -   *   /   %   #   ^   **   .*   ./   .+   .-   .^
==  !=  <   >   <=  >=
```

```
+(2, 3)             # 5
*(4, 5)             # 20
<(3, 5)             # true
+(1, 2, 3, 4)       # 10 - left-folded across all args
```

The variadic form (`+(1, 2, 3, 4)`) is a left-fold via the same `evalInfixExpression` dispatch the infix form uses, so multi-valued logic dispatch, set operations, and lattice operators all behave identically. See §9.

1.20.2 19.2 Operators as values

A bare operator name evaluates to a synthetic two-argument function. You can bind it, pass it, and compose it like any other callable:

```
plus: +
times: *
lt: <

plus(2, 3)                          # 5
reduce(+, 0, [1, 2, 3, 4])          # 10
reduce(*, 1, [1, 2, 3, 4])          # 24
reduce(-, 20, [1, 2, 3])            # 14
```

The synthetic function has parameters `_op_a, _op_b` and body `_op_a OP _op_b` — the same dispatch path the infix form uses. **Limitation:** the synthetic is binary. `(-)(5)` returns a partial-application lambda, not the unary negation; for unary minus on a value use `-(x)` directly.

1.20.3 19.3 Postfix: comma sequences

A comma-separated chain whose elements include at least one operator or stack word is parsed as a `SequenceExpression` and evaluated by stack reduction:

```
2, 3, +                         # → 5
2, 3, +, 4, *                   # → 20  (push 2, push 3, +, push 4, *)
10, 4, +, 6, 2, -, *            # → 56
```

Sequence semantics: walk the elements left-to-right; numeric/string/array literals push onto a parse-time stack; an operator pops the top two and pushes the result; a stack-shuffle word rewrites the stack (see below). The whole sequence reduces to the single remaining stack value.

A sequence is the right-hand side of an ordinary expression — so it composes:

```
r1: 2, 3, +                     # r1 = 5
r2: 2, 3, +, 4, *               # r2 = 20
r3: 10, 4, +, 6, 2, -, *        # r3 = 56

println(2, 3, +)                    # prints 5
```

Postfix sequences live **inside** the expression grammar rather than seizing the whole program, so they coexist peacefully with infix and prefix in the same line.

1.20.3.1 Multi-bind still works

The parser only collapses a chain to a sequence when at least one element is an operator or stack word. The classic multi-bind pattern is unchanged:

```
a, b: 5, 6                      # a = 5, b = 6 - not a sequence
```

1.20.4 19.4 Stack-shuffle words inside sequences

Inside a `SequenceExpression`, the stack-shuffle vocabulary is in scope and operates on the local parse-time stack rather than the global interpreter stack from §18:

Word	Effect	Stack effect
`dup`	Duplicate top	`a → a a`
`swap`	Swap top two	`a b → b a`
`rot`	Rotate top three	`a b c → b c a`
`over`	Copy second to top	`a b → a b a`
`drop`	Discard top	`a →`
`nip`	Drop second	`a b → b`
`tuck`	Insert top under second	`a b → b a b`

```
20, 4, swap, /                  # 4 / 20? no - swap before /: 4, 20, /  → 0
20, 4, /, 2, +                  # 20 / 4 + 2 → 7
1, 2, 3, rot, -, +              # rot: 2,3,1 → 3-1=2 → 2+2 = 4
```

These names shadow the same-spelled global-stack operations only *inside a sequence*; outside sequences the global-stack semantics from §18 apply.

1.20.5 19.5 The . (dot) inspect sigil

A trailing `.` prints the value of the preceding expression — a compact "print-and-go" — and works in two positions:

Statement-trailing (after any top-level expression):

```
2 + 3 .                          # prints 5
x: 42 .                       # prints 42
fullform(2 + 3) .                # prints "+(2, 3)"
```

Sequence-internal (pops the current top of the parse-time stack and prints it — the `.` is consuming, not duplicating):

```
2, 3, +, .                       # prints 5 (stack empty after)
20, 4, /, 2, +, .                # prints 7 (stack empty after)
2, 3, +, ., 10, 5, -, .          # prints 5, then prints 5 (two checkpoints)
```

The dot is parsed as the existing `PERIOD` token wrapped in a `ScopedStatement` for the trailing form, and as a stack-print step for the sequence-internal form. It is interchangeable with `inspect` / `see` in spirit but more compact.

1.20.6 19.6 Tracing sequence reduction

The existing `trace` keyword accepts a `sequence` (or `seq`) category that prints a step-by-step view of stack reduction:

```
trace sequence
20, 4, /, 2, +, .

# push 20          →  [20]
# push 4           →  [20, 4]
# /                →  [5]
# push 2           →  [5, 2]
# +                →  [7]
# .  (inspect)     →  [7]
```

Bare `trace` is equivalent to `trace all` — every category enables, including `sequence`. The same `untrace` / `untrace sequence` disable mirror.

1.20.7 19.7 AST inspection — `fullform`, `treeform`, `headof`, `argsof`, `hold`, `seq_of`

The `fullform` / `treeform` / `headof` / `hold` family, spelled in Axioma's call-with-commas idiom. Every builtin in this group has **hold semantics**: the argument's AST reaches the builtin unevaluated, so `fullform(2 + 3)` prints the tree, not `5`.

Builtin	Returns
`fullform(expr)`	Canonical `head(arg, ...)` string
`treeform(expr)`	ASCII box-drawing tree
`headof(expr)`	Operator / function / kind of the top node
`argsof(expr)`	Array of `fullform`-rendered child strings
`hold(expr)`	A `*AST` value wrapping the unevaluated tree (rebindable, re-inspectable)
`seq_of(elts...)`	A held `SequenceExpression` whose elements are the literal arguments (the postfix counterpart of `hold`)

```
fullform(2 + 3)                    # "+(2, 3)"
fullform(2 + 3 * 4)                # "+(2, *(3, 4))"
fullform(f(x, y + 1))              # "f(x, +(y, 1))"

treeform(2 + 3 * 4)
# +
#   2
#   *
#       3
#       4

headof(2 + 3)                      # "+"
argsof(2 + 3 * 4)                  # ["2", "*(3, 4)"]

h: hold(a + b * c)
fullform(h)                        # "+(a, *(b, c))" - auto-unwraps the held AST

s: seq_of(2, 3, +, 4, *)
fullform(s)                        # "Sequence(2, 3, +, 4, *)"
```

Auto-unwrap of held AST values. When the argument to `fullform` / `treeform` / `headof` / `argsof` is an identifier whose value is a `*AST` (produced by `hold` or `seq_of`), the builtin walks through the binding and inspects the held node. This is what makes `h: hold(2+3); fullform(h)` render `+(2, 3)` rather than the identifier `"h"`.

1.20.8 19.8 Why "one tree" is the load-bearing claim

The point of the three notations is not stylistic preference. It is that the **evaluator never branches on notation** — `2 + 3`, `+(2, 3)`, and `reduce(+, 0, [2, 3])` all converge on the same `evalInfixExpression` dispatch. Adding a new operator (or fixing a multi-valued logic semantics bug) touches one path, and every notation that surfaces that operator inherits the fix.

Concretely, the identity is testable:

```
verify("identity prefix/infix",
       fullform(2 + 3), fullform(+(2, 3)))   # both "+(2, 3)"
```

See `tests/axioma/notation/` for the full assertion battery (operator-prefix, postfix sequence, dot inspect, stack words, sequence trace, fullform, operator-as-value, three notations) and `tests/axioma/showcase/11_three_notations` for a single-file walkthrough.

1.20.9 19.9 Homoiconicity — building code as data

The inspection family above (`fullform` / `headof` / `argsof`) lets you *read* code as data. The construction family closes the loop: you can *build* AST values from scratch in Axioma source, manipulate them, and `ast_eval` them back into computation. That's metaprogramming — programs that produce programs.

Constructors. Each `make_*` returns an AST value. Args may be other AST values OR raw Axioma values (auto-lifted via the same machinery quasiquote uses):

Builtin	Returns AST of
`make_integer(n)`	`IntegerLiteral`
`make_float(x)`	`FloatLiteral`
`make_string(s)`	`StringLiteral`
`make_boolean(b)`	`Boolean`
`make_identifier(s)`	`Identifier`

Builtin	Returns AST of
`make_infix(op, l, r)`	`InfixExpression`
`make_prefix(op, x)`	`PrefixExpression`
`make_call(f, args)`	`CallExpression`
`make_if(c, t, e?)`	`IfExpression`
`make_lambda([params], body)`	`LambdaExpression`
`make_array(...)`	`ArrayLiteral` (variadic or single `Array`)
`make_tuple(...)`	`TupleLiteral`
`make_set(...)`	`SetLiteral`
`make_sequence(...)`	`SequenceExpression` (postfix sequence)

Round-trip identity. Constructed and quoted ASTs compare equal structurally:

```
make_infix("+", 2, 3) == hold(2 + 3)       # true - same canonical form
ast_eval(make_infix("+", 2, 3))            # 5
make_identifier("x") == hold(x)            # true
quote(x * y) == make_infix("*", make_identifier("x"), make_identifier("y"))   # true
```

Recursive traversal via `parts(ast)`. Unlike `argsof` (which returns strings), `parts` returns `Array` of AST values — so you can walk a tree without re-parsing at each level:

```
h: hold((a + b) * c)
outer: parts(h)         # [AST(a + b), AST(c)]
inner: parts(outer[1])  # [AST(a), AST(b)]
ast_string(inner[1])         # "a" - use ast_string for inline display
                              # (fullform has hold semantics; bind first or use ast_string)
```

`is_ast(x)` predicates the type — useful in pattern-matching code:

```
is_ast(hold(2 + 3))     # true
is_ast(42)              # false
```

The dispatch loop. Recursive AST walks dispatch on `ast_kind`:

```
walk: func(expr) [
    k: ast_kind(expr)
    if k == "integer_literal" then ...
    else if k == "identifier" then ...
    else if k == "infix_expression" then ...
    else expr
]
```

1.20.9.1 The killer demo: symbolic differentiation in 25 lines

Below is a complete textbook differentiator written entirely in Axioma. No interpreter extension required:

```
diff_infix: func(expr, var_name, dx) [
    op: headof(expr)
    lhs: parts(expr)[1]
    rhs: parts(expr)[2]
    if op == "+" then make_infix("+", dx(lhs, var_name), dx(rhs, var_name))
    else if op == "-" then make_infix("-", dx(lhs, var_name), dx(rhs, var_name))
    else if op == "*" then make_infix("+",
        make_infix("*", dx(lhs, var_name), rhs),
        make_infix("*", lhs, dx(rhs, var_name)))
    else expr
]
```

```
dx: func(expr, var_name) [
    k: ast_kind(expr)
    if k == "integer_literal" then make_integer(0)
    else if k == "identifier" then (
        if headof(expr) == var_name then make_integer(1) else make_integer(0)
    )
    else if k == "infix_expression" then diff_infix(expr, var_name, dx)
    else expr
]

d: dx(quote(x * x + 3 * x + 5), "x")
println(fullform(d))   # +(+(+(*(1,x), *(x,1)), +(*(0,x), *(3,1))), 0)
                       # - unsimplified; equivalent to 2x + 3
```

A `simplify(expr)` pass (fold `0*x → 0`, `1*x → x`, `x+0 → x`) is another function in the same style; together they make a small CAS. `substitute(expr, var, value)` for evaluating at a point is five lines around `make_integer`. Everything you'd want from a computer algebra system is achievable in user-space code.

1.20.9.2 Definition-time macros Beyond runtime AST construction, Axioma has Julia/Elixir-style **macros** — `macro name(params) body` definitions that expand at the call site. Macro arguments arrive as **unevaluated AST**, the body must return an AST (typically built via `quasiquote(...)` + `unquote(...)`), and the expansion is then evaluated in the calling context.

```
macro double(x) quasiquote(unquote(x) * 2)
double(21)                                  # → 42

# Inspect the expansion without running it:
ast_string(macroexpand(double(21)))         # → "(21 * 2)"

# Multi-arg, conditional, nested all work:
macro unless(cond, body) quasiquote(if not unquote(cond) then unquote(body) else none)
macro addAndDouble(a, b) quasiquote((unquote(a) + unquote(b)) * 2)
addAndDouble(3, 4)                          # → 14

double(double(3))                           # → 12 (nested expansion)
```

Hygiene via `gensym`:

```
gensym()                # → G__1   (fresh unique symbol)
gensym("tmp")           # → tmp__2
```

Use inside macros that introduce temporaries (`__r: gensym("r")`-style) to avoid capturing user identifiers. Like Julia (and unlike Elixir or Clojure-syntax-rules), Axioma macros are **non-hygienic by default** — you opt into hygiene with `gensym`.

Implementation: parser at `parser/parser.go:9866`, expansion engine at `evaluator/macro_expansion.go:187`, `macroexpand` builtin at `evaluator/evaluator.go:25913`. Test corpus: `tests/axioma/metaprogramming/` (10+ files).

1.20.9.3 The `head` / `operands` / `make_expr` normal-form algebra Mathematica unifies every expression under one shape — `head[args]`. Axioma renders its three surface notations (infix, prefix, postfix) and call syntax to that same normal form, and a small algebra reads and rebuilds any quoted expression **uniformly**, regardless of which syntax produced it.

Builtin	Returns
`head(ast)`	the operator / functor as a **String** — infix `"+"`, a call's functor name, a statement keyword (`":"` for a binding, `"if"`, a rule operator `"<~~"`), or an atom's kind (`"Integer"`, `"Symbol"`)
`operands(ast)`	an **Array of AST** — the arguments with the functor **excluded** (the uniform companion to the older `parts`, which *includes* the functor for calls)
`make_expr(head, operandsArray)`	rebuilds the node from a head String + an operands Array — the inverse of the two above

```
head('(2 + 3))                                  # → "+"
operands('(2 + 3))                              # → [<AST: 2>, <AST: 3>]
make_expr(head('(2 + 3)), operands('(2 + 3)))   # → <AST: (2 + 3)>   (round-trips)
head('(add(7, 9)))                              # → "add"   (a call's functor)
head('(if x then 1 else 2))                     # → "if"
```

`make_expr(head(e), operands(e)) == e` holds for infix / prefix / postfix / call / binding / rule / if. Fixity is recovered from a curated operator table; an unknown head builds a `CallExpression`.

1.20.9.4 Pattern rewriting — `match_pattern` / `subst` / `replace_all` / `rules` The algebra above is the substrate for **term rewriting** — Mathematica's `expr /. rule` and the heart of a computer-algebra system. Patterns reuse Axioma's existing `?x` variable syntax; no new lexer.

Builtin	Does
`match_pattern(pattern, subject)`	structural match → a **Dictionary** of bindings, or `none`
`subst(template, bindings)`	instantiate a template from a bindings Dictionary
`replace_all(subject, rules)`	bottom-up rewrite to a fixpoint
`rules(p1, t1, p2, t2, …)`	flat-pairs sugar for a rule list (the keyword `rule` is reserved)

```
match_pattern('(?a + ?b), '(2 + 3))                 # → {a: <AST: 2>, b: <AST: 3>}
subst('(?a * ?a), {a: '(7)})                        # → <AST: (7 * 7)>
replace_all('(x + 0), rules('(?a + 0), '(?a)))      # → <AST: x>   (additive identity)
```

A nonlinear pattern (`?x + ?x`) enforces same-subtree consistency. Each rule may carry a **guard** — a third element, a quoted predicate over the `?vars` (Mathematica's `/;`); the rule fires only when the guard holds, and guards see the builtins **and the caller's definitions**. Pattern variables in head position (`?f(?x)`) bind the functor, and **sequence patterns** match variable-arity argument runs — `?xs__` (one or more) and `?xs___` (zero or more), in call-argument position. Together these are enough to write algebraic simplification and symbolic differentiation as a *rule list* rather than a hand-written dispatch.

1.20.9.5 A form is a collection — the "Julia-Plus" tier A quoted expression also behaves as **the sequence of its operands** for the universal collection operations, so generic code traverses code the same way it traverses data:

```
len('(2 + 3))                  # → 2          (was an error before)
'(add(7, 9))[1]                # → <AST: 7>   (1-indexed into the operands)
[op | op <- '(a + b + c)]      # comprehension over a form's operands
foreach op in '(2 + 3) [ println(op) ]
map(func(x) [x], '(2 + 3))     # map / filter accept a form
```

1.20.9.6 The code data bridge — `to_data` / `from_data` Two converters cross the line between a quoted AST and an ordinary value:

```
to_data('(2 + 3))                  # → 5            (evaluate a form to its value)
from_data(5)                       # → <AST: 5>     (lift a value into an AST literal)
to_data(from_data(v)) == v         # round-trips for scalars, arrays, sets, tuples
```

`from_data` understands the set-theory core (Set and Tuple values lift, not only scalars and arrays). The bridge is the *frictionless* path between the two worlds: where the `'(…)` quote-forms carry bracket-overloading gotchas (`'[…]` is a **block** quote, `'([…])` is an array-literal quote), `from_data(theValue)` is unambiguous.

1.20.9.7 Where Axioma sits in the homoiconicity taxonomy Per Wikipedia's taxonomy of homoiconic languages, Axioma is in the **"weaker tier"** alongside **Julia, Elixir, and Nim** — full toolkit (quote, quasiquote, AST construction, AST evaluation, macros, macroexpand, hygiene primitive) but source code is parsed rather than being a literal data structure.

Tier	Languages	Defining property
Purest (S-expressions)	Lisp, Scheme, Clojure, Racket	source = uniform list literal
Weaker (data structures for code)	**Julia, Elixir, Nim, Axioma**	AST as value + macros
Other recognized	Mathematica, REBOL, Red, Tcl, Io, Prolog	various
eval-on-string only	Python, Ruby, JavaScript	no in-language AST manipulation

The article explicitly notes "no consensus exists on a precise definition" of homoiconicity — under a lenient definition Axioma qualifies fully; under the strict "source IS data" definition it doesn't. Reaching the strict tier would require abandoning the multi-syntax three-notations design, which is intentional (see §19.1–§19.4).

The `head` / `operands` / `make_expr` algebra and the `match_pattern` / `replace_all` rewriter above add a facility usually associated with **Mathematica** — a uniform `head[args]` normal form plus structural pattern matching and term rewriting — onto the reified-AST base. That strengthens the practical tier (think "Julia-Plus") without changing where Axioma sits in the strict sense: a quoted form is still a dedicated `AST` value, not source-as-list, so `'[1, 2, 3] == [1, 2, 3]` remains false. The algebra unifies the **operations** over code-as-data; it does not unify the **representation**. The full design study of what strict representational homoiconicity would cost lives at `resources/docs/claude/HOMOICONICITY_FULL_SPEC.md` (assessed as not worth a kernel rewrite for the current mission).

See `resources/docs/claude/HOMOICONICITY.md` for the full user reference and `docs/code-as-data.md` for the comparison tables and cookbook patterns.

1.20.9.8 Limits

Limit	Workaround
`fullform(inline_call(...))` captures the call literally (hold semantics)	Bind to local first: `r: call(...); fullform(r)`. Or use `ast_string(call(...))` which doesn't hold.
`ast_eval` doesn't see local bindings	Use `quasiquote` to splice values *into* the AST before evaluating: `ast_eval(quasiquote(unquote(x) + 1))`
Macros aren't hygienic by default (Julia-style)	Use `gensym("prefix")` to generate fresh symbols inside macro bodies
`quasiquote` not yet compiled in `--vm`	Run quasiquote-heavy / macro-heavy scripts under the tree-walker
No reader macros — can't extend the parser itself	Out of scope for v1

Limit	Workaround
AST nodes are heterogeneous, not Lisp-uniform	Trade-off against rich surface syntax; use `ast_kind` + `headof` + `parts` to dispatch

See `tests/axioma/notation/test_homoiconic.ax` for the 46-assertion smoke test, and `demo_differentiate.ax` for the full symbolic-differentiation example.

1.21 20. Russell's Three Meanings of "is"

Axioma distinguishes the three classical meanings of the copula **"is"**:

Meaning	Syntax	Semantics
Identity	`x is same as y` / `x is/identical y`	Ontological identity
Predication	`sky is blue` / `sky is/property blue`	Property attribution
Existence	`there is x` / `exists x`	Existential claim

```
"morning star" is same as "evening star"     # Identity
sky is blue                                  # Predication
there is x in {1, 2, 3}: x > 2               # Existence
```

Refinement operators: `is/same`, `is/identical`, `is/property`.

1.22 21. Pointers & References

1.22.1 C-style pointers

```
x: 42
p: &x                       # Take address
*p                                # 42 - dereference
*p = 100                          # Write through pointer
x                                 # 100

arr: [10, 20, 30]
p2: &arr[1]                 # Pointer to array element
*p2                               # 20

alice: a Person {age: 30}
p3: &alice.age            # Pointer to property
*p3 = 31
alice.age                         # 31
```

Pointers work in both interpreted and VM modes.

1.22.2 Deep copy

`copy(value)` returns an independent deep copy:

```
a: [10, 20, 30]
b: copy(a)                        # deep copy - mutating b leaves a untouched
```

Note — `@` is type-of, not a reference get-word. `@x` returns the **type** of `x` (`@42` → `"Integer"`), identical to `type(x)`. Plain `x` already yields the value, and the address/dereference operators `&x` / `*p` (above) are Axioma's actual reference mechanism.

1.23 22. Mathematical Constants & Built-ins

1.23.1 Mathematical constants

Constant	Value	Description
`pi`	3.141592653589793	
`e`	2.718281828459045	Euler's number
`phi`	1.618033988749895	Golden ratio
`sqrt2`	1.4142135623730951	√2
`sqrt3`	1.7320508075688772	√3
`ln2`	0.6931471805599453	ln 2
`ln10`	2.302585092994046	ln 10

1.23.2 Set constants

Constant	Description
`emptyset`	`{}` (glyph)
`naturals`	`{1, 2, 3, ..., 100}` (finite; glyph) — lazy ∞ form: `infinite_set("naturals")`
`integers`	`{-50, ..., 50}` (finite; glyph) — lazy ∞ form: `infinite_set("integers")`
`rationals`	— lazy infinite set, countable/enumerable (glyph)
`reals`	— lazy infinite set, membership-only (glyph)
`complexes`	— lazy infinite set, membership-only (glyph)
`universe`	Universal set for demos

1.23.3 Mathematical functions

Function	Description
`abs(x)`	Absolute value
`sqrt(x)`	Square root
`floor(x)`	Floor
`ceil(x)`	Ceiling
`pow(b, e)`	Exponentiation
`quotient(a, b)`	Floor / truncated integer division (same as `a // b` / `a div b`)
`remainder(a, b)`	Remainder of integer division (same as `a % b` / `a mod b`)
`divmod(a, b)`	Combined: returns `(quotient, remainder)` tuple in one call
`signum(x)`	Sign as `-1` / `0` / `1`, **preserving type** (Int→Int, Float→Float, Rational→Int). A non-number errors. (`sign(x)` always returns an Integer.)
`square(x)`	`x * x`, preserving numeric type
`add1(x)` / `sub1(x)`	`x + 1` / `x - 1` (Lisp `1+` / `1-`)

Function	Description
`isqrt(n)`	Integer floor square root of a non-negative integer (exact, big-int aware)
`numerator(r)` / `denominator(r)`	Rational accessors; an integer `n` is `n/1` (so `denominator(5)` → `1`)
`to_number(s)`	Parse a string to a number — Integer if integral, else Float; a number passes through

1.23.4 Predicates

```
even(4)           # true
odd(3)            # true
positive(5)       # true
negative(-3)      # true
```

1.23.4.1 Scheme/Lisp-style predicates (`?` suffix) Lisp/Scheme spell the test marker `?` rather than Common Lisp's `p`. A trailing `?` is part of the identifier (`zero?` is one word), so these read naturally. The sign predicates treat a non-number as `false` (matching `even`/`positive`).

```
# sign / parity (over Integer, Float, Rational)
zero?(0)          # true       plus?(5)      # true   (strictly positive)
positive?(5)      # true       minus?(-1/3)  # true   (strictly negative)
negative?(-5)     # true       even?(4)      # true        odd?(3)  # true

# type predicates (join number?/integer?/string?/boolean?/null?/pair?/array?…)
list?([1, 2, 3])   # true       procedure?(func(x) [x])  # true
empty?([])         # true   - also "" / set() / {} / dict() ; an infinite set is never empty
symbol?('hello)    # true   - a lit-word is a symbol ; symbol_name('hello) → "hello"

# float domain  (`inf` / `nan` are builtins producing ±ϖ / NaN floats)
nan?(nan)          # true        infinite?(inf)      # true        finite?(3.14)  # true

# equality:  eq? / eql? / eqv?  are shallow identity (collections by reference);
#            equal?  is deep, type-strict (the `==` engine)
eq?(1, 1)          # true        eq?([1,2], [1,2])     # false  (distinct objects)
equal?([1,2], [1,2])  # true     equal?(1, 1.0)        # false  (type-strict)
```

1.23.5 Higher-order functions

```
map(fn, set)           # Transform
filter(pred, set)      # Select
reduce(fn, init, set)  # Aggregate (left fold)
sum(coll)              # Numeric sum over array/set/tuple (Σ is Unicode alias)

# Scheme/Lisp folds & combinators
foldl(fn, init, coll)  # left fold, fn(acc, elem) - like reduce (also fold_left)
foldr(fn, init, coll)  # right fold, fn(elem, acc) (also fold_right)
append_map(fn, coll)   # map then concatenate the per-element collections (flatmap / mapcat)
for_each(fn, coll)     # apply fn for side effects; returns null
constantly(x)          # → a function that ignores its args and returns x
negate(pred)           # → a predicate that logically negates pred (  set `complement`)
```

1.23.6 List & string helpers

```
butlast([1, 2, 3, 4])              # → [1, 2, 3]   (all but the last)
dedupe([1, 2, 2, 3, 1])           # → [1, 2, 3]   order-preserving (remove_duplicates / unique)
chars("abc")                       # → ["a", "b", "c"]   (string → char array)
char_alphabetic?("a")              # → true   (also char_numeric? / char_whitespace?)
```

1.23.7 Knowledge-base builtins

`rel` is the relation **name as a string** (`"parent"`, not the bare value `parent`); `args` are the fact's arguments.

Builtin	Description
`insert(rel, args, [grounding])`	Assert a new fact (default grounding `datum`)
`forget(rel, args)`	Retract a fact
`set_truth(rel, args, value)`	Attach Belnap B4 value
`truth(rel, args)`	Query Belnap value
`grounding(rel, args)`	Query grounding → a `Grounding` value (ordered)
`truth_kind(rel, args)`	Query truth-kind → a `Kind` value (flat)
`proof(rel, args)`	Walk derivation chain
`rules_of(pred)`	A predicate's rules as `RuleClause` values
`challenge(rel, args)`	Mark axiom as suspect
`challenged(rel, args)`	Check if challenged
`cancel(rel, args)`	Suppress a derived fact
`uncancel(rel, args)`	Remove cancellation
`canceled(rel, args)`	Check cancellation
`transaction_begin/commit/rollback`	Atomic mutation
`predicates_of(X)`	All facts mentioning X
`predicate_names()`	All known predicate names
`cardinality(C, "prop", min, max)`	Register cardinality

1.23.8 MVL constructors

```
intuit3("true"/"false"/"unknown")
belnap("true"/"false"/"both"/"neither")
lukasiewicz(0.0..1.0)
```

1.24 23. Venn Diagrams & Visualization

```
a: {1, 2, 3, 4}
b: {3, 4, 5, 6}
venn(a, b)                        # 2-set diagram

c: {3, 4, 5}
venn(a, b, c)                     # 3-set diagram with all intersections
```

Diagrams auto-pop in the default image viewer and are saved to `diagrams/venn_diagram_TIMESTAMP.png`.

Run `./scripts/clean_diagrams.sh` to clean accumulated PNGs.

1.24.1 The `*form` family — rendering expressions & relations

A family of introspection builtins renders a value to text. `fullform` / `treeform` show an expression's AST; `tableform` / `graphform` render a **relation's extent** (rule-derived facts included, since they walk the same fact store as comprehensions). All take the relation as a held bare identifier or a string.

Function	Renders
`fullform(expr)`	AST as a symbolic string
`treeform(expr)`	AST as an ASCII tree
`tableform(rel)`	relation extent as an ASCII table
`graphform(rel, [fmt])`	relation as a graph — `"ascii"` (default) / `"dot"` / `"cg"` / `"png"` / `"svg"`

```
relation edge(x, y)
assert edge("a", "b")
assert edge("b", "c")

println(tableform(edge))
# relation: edge  (2 facts)
#
#   x    y
#
#  "a"  "b"
#  "b"  "c"
#

println(graphform(edge))            # default ASCII adjacency
# graph: edge  (2 edges)
#   "a" → "b"
#   "b" → "c"

graphform(edge, "dot")              # Graphviz DOT - paste into `dot -Tsvg`
graphform(edge, "cg")               # Sowa conceptual-graph linear notation [a]→(edge)→[b]
graphform(edge, "png")              # force-directed PNG → visualizations/graph_<ts>.png
graphform(edge, "svg")              # scalable SVG (preferred for web / VS Code)
```

Headers come from the declared slot names; output tuples are sorted deterministically. Higher-arity relations fall back to positional notation (binary is the natural graph case).

1.25 24. Comments & Literate Programming

1.25.1 Basic comments

```
# Single-line comment
x: 5  # Inline comment
```

1.25.2 Markdown documentation blocks

`/**md ... */` blocks are extracted by `axiomadoc` for HTML/Markdown/PDF rendering:

```
/**md
# Function: calculateDistance

## Purpose
Euclidean distance between two points.

## Math
$d = \sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}$
```

```
## Cross-references
- {@link otherFunction}
- {@concept Point}
*/
calculateDistance: func(x1, y1, x2, y2) [
    sqrt((x2 - x1) ^ 2 + (y2 - y1) ^ 2)
]
```

1.25.3 Generating documentation

```
./axiomadoc generate -input . -output docs -format html  -template default
./axiomadoc generate -input . -output docs -format html  -template academic
./axiomadoc generate -input . -output docs -format markdown
./axiomadoc serve    -port 8080 -watch -input .         # Live-reload server
./axiomadoc validate -input . -check-links -run-examples
```

1.26 25. Interactive Features (REPL)

1.26.1 Line editing

- ↑/↓ — command history (persistent across sessions)
- ←/→ / Home/End — cursor movement
- Tab — keyword completion

1.26.2 Special commands

```
help               # Language help
doc <topic>        # Topic documentation
:stack             # Inspect interpreter stack
:s                 # Alias for :stack
:stack trace       # Step-by-step stack trace
exit               # Quit
```

1.26.3 Detecting interactivity — `is_interactive()` / `isatty()`

A script can ask whether it is attached to a terminal, so the same source can prompt a human interactively yet stay silent (or take defaults) when driven from a pipe, a test harness, or CI. Both return a `Boolean`:

```
if is_interactive() then
  name: input("Your name? ")
else
  name: "anonymous"             # piped / non-interactive run
```

`isatty()` is the lower-level alias (true iff stdin is a TTY); `is_interactive()` is the same check phrased for the common prompt-or-default idiom. Under a pipe both are `false`.

1.26.4 Error handling

```
axioma> x:
Parser errors:
  expected expression after `:`, got EOF

axioma> 5 / 0
ERROR: division by zero
```

```
axioma> unknown_function(5)
ERROR: identifier not found: unknown_function
```

1.27 26. CLI Flags, MCP Server & Tooling

1.27.1 CLI flags

Flag	Effect
(none)	Start REPL
`<file.ax>`	Run script
`--vm <file>`	Run in VM mode
`--mcp [start|stop|restart|status]`	MCP server
`--no-kb`	Skip Cascade KB preload (~10× faster startup)
`--verbose`	Verbose output
`--language <subset>`	Restrict surface to a subset: `axioma/all` (default), `axioma/core`, `axioma/functional`, `axioma/knowledge`, `axioma/knowledge-core`, `axioma/beginner`
`--mode <name>[,<name>…]`	Educational mode: `functional`, `logic`, `stack`, `mathematical`, `linguistic`, `imperative`, `beginner`
`--glyphify <file>` / `--asciify <file>`	Token-aware in-place canonicalizer between word operators and Unicode glyphs (`in` ∈, `and` ∧, `forall` ∀; digraphs `` `cup `` → ∪). Strings/comments untouched; verifies the rewrite lexes+parses identically before writing (see §3 "Typing the glyphs")
`--typecheck` (alias `--check`)	Static pre-pass: walk annotated code, report type errors, exit non-zero on violations
`--strict`	With `--typecheck`, also flag undefined-identifier references
`--run`	With `--typecheck`, execute the script if (and only if) the check is clean
`--learn`	Start the interactive learning wizard (single-shot Q&A)
`--learn-list` / `--learn-task <n>` / `--learn-path <dir>`	Wizard navigation + custom lessons
`--recipe`	Start the HtDP-style Design Recipe wizard (six stages per task)
`--recipe-list` / `--recipe-task <n>` / `--recipe-stage <n>` / `--recipe-path <dir>`	Recipe wizard navigation + custom recipes
`--annotate` / `-a <file>`	Generate a literate walkthrough: `<file>.annotated.md` (pure markdown) + `<file>.annotated.ax` (executable literate source). Groups statements into blocks (relation / fact / rule / func / …) and explains each from a pedagogical pattern library
`--annotate-html`	Also emit a self-contained `<file>.annotated.html` (embedded CSS, light/dark, inline SVG diagrams for binary relations)
`--annotate-llm`	Fill gaps the canned patterns don't cover via the configured AI provider (off by default — `-a` alone runs offline at $0)

Flag	Effect
`--annotate-terse` / `--annotate-verbose`	One sentence per block / 4–6 sentences with analogies
`--annotate-no-run`	Skip the auto-executed `## Output` snapshot embedded at the bottom

1.27.2 Literate annotation: `--annotate`

`axioma --annotate path.ax` parses the script, groups consecutive statements of the same kind into blocks, and emits a step-by-step walkthrough — markdown for reading plus a still-executable literate `.ax`. Binary relations with 2 facts get an inline diagram (mermaid in the markdown, inline SVG in the HTML). It runs fully offline by default; `--annotate-llm` adds an AI fallback for unrecognized constructs.

```
axioma -a prolog-like.ax                          # MD + AX, offline
axioma --annotate --annotate-html --annotate-verbose script.ax
```

1.27.3 Educational subset: `axioma/beginner`

`--language axioma/beginner` (or the `#language axioma/beginner` source pragma) restricts the surface to one canonical form per operation — `x: value` (or the textbook `x = value`), `func`, `if/then/else`, `while`, `println`, `concept Foo [doc] [refinement] [block]`, `Day enumerates …` — and rejects the alternatives (`lambda`, `fn`, `\x ->`, `match`, `repeat`, `repeat…until`, `foreach`, `is`, `print`, `printf`, `display`, `ranges`) with educational guidance pointing at the canonical equivalent. Setting this subset also auto-activates `--mode beginner` for the behavioral overlay (no metaprogramming / FFI / proofs).

Concept declaration in beginner mode. The single canonical creation surface is `concept Foo` — bare, with optional postfix doc string (`concept Foo "doc"`), prefix refinement (`concept/persist Foo`), or slot-defaults block (`concept Foo { slot: default }`). The non-canonical `Foo is Concept` / `Foo exists` / `Foo create` / `create Foo` shapes error uniformly — beginners get the same hint as expert users. `is` is canonical for instance classification (`apple is Stock`) and Boolean type queries (`x is Stock` in expression position), including `X is Concept` as a "is this a registered concept?" query.

Binding forms in beginner mode. Both live binding forms are accepted (`x: 5` and the textbook `x = 5` find-or-update), with `x: 5` taught as the canonical form in the *HtDP-in-Axioma* textbook (`x := 5` is a syntax error). This relaxation trades strict canonicalisation for compatibility — the textbook teaches one preferred form while existing tests continue to work unmodified.

1.27.4 Static `--typecheck` pre-pass

The checker walks the AST in two passes: pass 1 gathers concept / enum / subrange / function-signature declarations (so forward references work); pass 2 walks every annotated binding, every assignment to an annotated binding, and every call site with literal arguments, emitting errors when both sides are statically known and disagree. Only annotated code is checked — unannotated code stays dynamic.

Eight violation classes covered: annotation/literal mismatch, subrange over/under bounds, inline subrange, cross-enum literal, reassignment violation, call-site arg type, arity mismatch, and (with `--strict`) undefined-identifier references.

```
axioma --typecheck script.ax          # check only
axioma --typecheck --run script.ax    # check then run if clean
axioma --check --strict script.ax     # alias + undefined-ref detection
```

1.27.5 Learning wizards

`--learn` is the single-shot Q&A wizard from `tools/learn/*.json` (12 built-in tasks). `--recipe` is the multi-stage HtDP-style wizard from `tools/recipes/*.json` (5 built-in recipes), walking the student through data

→ signature → examples → template → body → tests. Both wizards inherit `--mode` and `--language` from the parent invocation — `axioma --learn --language axioma/beginner` enforces the beginner subset on student code.

1.27.6 MCP server

Start the server (stdio JSON-RPC) for Claude Desktop / Code / Cascade integration:

```
./axioma --mcp              # Start
./axioma --mcp status       # Check
./axioma --mcp stop         # Stop
```

11 tools exposed:

Tool	Purpose
`parse_axioma`	Parse to AST
`validate_syntax`	Syntax check
`analyze_code`	Semantic analysis
`translate_code`	AST-aware translation (Go, Python, Axioma)
`symbolize_nl`	Natural language → Axioma
`execute`	Run code, return result + output
`query_kb`	Query the persistent KB
`inspect_env`	Inspect session environment
`run_file`	Execute `.ax` files
`decompose_claim`	Break claim into propositions
`evaluate_b4`	Belnap B4 / K3 / L3 evaluation
`resolve_entities`	Match entities against KB

KB location: `~/.axioma/axioma.kb` (SQLite). Logs: `~/.axioma/mcp.log`. PID: `~/.axioma/mcp.pid`.

1.27.7 Claude Desktop config

```
{
  "axioma": {
    "command": "/path/to/axioma",
    "args": ["--mcp"]
  }
}
```

1.27.8 Other front-ends

- **Web GUI** — `web-gui/start.sh` (Vite/React + Go REST API)
- **Wails GUI** — `wails-gui/` (desktop)
- **Jupyter kernel** — `jupyter/install_kernel.sh`
- **VS Code extension** — `vscode-axioma/`

1.27.9 Testing — the `expect` builtin

`expect(label, actual, expected)` is a real assertion: it passes iff `actual == expected` (regular value equality — cross-type numerics, deep array/set/map, MVL coercion). It prints `go test`-style markers — `--- PASS: <label> (<duration>)` on a match, `--- FAIL: <label> (<duration>)` plus the actual/expected values on a mismatch; the parenthesized duration is the time spent evaluating `actual`/`expected` (adaptive `µs`/`ms`/`s`), so a slow assertion stands out and can flag an optimization regression the way `go test` shows per-test durations. A mismatch increments a run-level counter and **continues** (accumulate-and-continue); at end-of-run the CLI prints a Go-style summary to stderr — `PASS <script> (N assertions)` or `FAIL`

`<script> (N of M failed)` — and exits **non-zero** if any expectation failed. (The summary prints only for scripts that ran 1 `expect()`, so ordinary scripts stay silent.) This is what lets the parallel test runner (which keys on exit code) actually detect wrong answers — unlike the older `if cond then println(" ") else println(" ")` idiom, which exits 0 even when every check is wrong.

```
expect("two plus two", 2 + 2, 4)         # --- PASS: two plus two (0µs)
expect("oops", 100 // 7, 13)             # --- FAIL: oops (0µs) (actual 14, expected 13) → exit 1
expect("slow path", slowfib(30), 832040) # --- PASS: slow path (1.42s)  ← flags slow code
```

Because it's a function call (not a keyword), a local `expect: func(...)` definition shadows it — so adding it was zero-regression for the files that previously hand-rolled their own `expect`.

1.28 27. Examples

1.28.1 Set theory

```
numbers: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
evens: {2, 4, 6, 8, 10}
primes: {2, 3, 5, 7}

evens union primes              # {2, 3, 4, 5, 6, 7, 8, 10}
evens intersect primes          # {2}
numbers difference evens        # {1, 3, 5, 7, 9}
primes subset numbers           # true
venn(evens, primes)
```

1.28.2 Concept system + frame queries

```
Country extends Concept
Country has gdp_billion
Country has population_million

usa: a Country {}
usa.gdp_billion: 27000
usa.population_million: 332

china: a Country {}
china.gdp_billion: 18000
china.population_million: 1410

big_economies: {C | C is Country, C.gdp_billion > 20000}
```

1.28.3 Logic programming with rules

```
relation parent(x, y)
assert parent("Adam", "Cain")
assert parent("Adam", "Abel")
assert parent("Cain", "Enoch")
assert parent("Enoch", "Irad")

# Strict rule: ancestors
ancestor(X, Y) <= parent(X, Y)
ancestor(X, Z) <= parent(X, Y) and ancestor(Y, Z)

# Query
```

```
{Y | Y <- ancestor("Adam", Y)}
# {"Abel", "Cain", "Enoch", "Irad"}

# Provenance
proof("ancestor", "Adam", "Irad")
why ancestor("Adam", "Irad")
```

1.28.4 Defeasible reasoning

```
relation bird(x)
assert bird("tweety")
assert bird("polly")
assert bird("opus")        # A penguin

flies(X) <~~ bird(X)                              # Birds typically fly
cancel("flies", "opus")                            # But opus doesn't

{X @conjecture | X <- flies(X)}                    # {"polly", "tweety"}
```

1.28.5 Multi-valued logic

```
# Belnap B4 - paraconsistent reasoning under contradictions
relation parent(x, y)
assert parent("X", "Y")
set_truth("parent", "X", "Y", "both")              # Contradictory source
truth("parent", "X", "Y")                          #

# Gödel G3 - intuitionistic
p: intuit3("unknown")
g3_lem(p)                                          # unknown - LEM not valid
g3_dne(p)                                          # unknown - DNE fails
p implies p                                        # true - reflexive

# Łukasiewicz L3 - fuzzy-like truth
half: lukasiewicz(0.5)
half implies lukasiewicz(0.7)                      # 1.0 (truthier consequent)
```

1.28.6 Functional programming

```
data: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

# Sum of squares of odd numbers
result: reduce(
    lambda (acc, x) => acc + x,
    0,
    map(lambda x => x * x,
        filter(lambda x => odd(x), data)
    )
)
# 1 + 9 + 25 + 49 + 81 = 165
```

1.28.7 Stack-based RPN

```
3 4 + 2 *          # ((3 + 4) * 2) = 14
:s                 # [14]
```

```
5 dup *             # 5² = 25
2 swap              # [2, 25]
```

1.28.8 Atomic mutation

```
relation parent(x, y)
transaction_begin()
insert("parent", "Eve",  "Seth")
insert("parent", "Seth", "Enos")
set_truth("parent", "Eve", "Seth", "true")
# Oops, mistake
transaction_rollback()               # Undoes all three operations
```

1.29 28. Embedding Other Languages

Axioma can call out to peer languages from inside an `.ax` script. The mechanism is built around a single block form and a curated set of "shim" modules — Axioma names that quietly route to a peer's standard library. The goal is **adoption, not performance**: developers coming from Python (or Julia, R, JS) can keep familiar call sites unchanged while the rest of the program is written in Axioma, then migrate piece by piece as native equivalents land.

1.29.1 27.1 The `[<dialect> | … ]` block

```
mean: [python |
xs = [1, 2, 3, 4, 5]
print(sum(xs) / len(xs))
]
println(mean)            # 3.0
```

- The opening `[python` is a registered dialect tag. Recognized tags: `python` (alias `py`); the secondary subprocess runners `julia`, `r`, `js`, and `lisp` (Common Lisp via sbcl; aliases `cl` / `commonlisp`); plus the in-process/translation tags (`axioma`, `sql`, `model`, `solver`, `sympy`/`cas`, `nl`). The secondary runners are exec-only (see §27.1c).
- After the `|` the parser switches to a raw-source reader: every character up to the matching `]` is sent verbatim to the foreign runtime. Indentation, embedded brackets, multi-line `def`s, and string literals all survive intact.
- The block is an **expression** — its value is the foreign runtime's captured stdout as an Axioma `String`.
- Both the tree-walker and the bytecode VM execute the block; switch with `--vm` and behavior is identical.

1.29.1.1 Mode refinement: `[python/eval | … ]` The default form (`[python | … ]`) is "exec mode": the body is run as a Python script, and whatever it prints to stdout becomes an Axioma `String`. To get a typed value back without writing `print(...)`, use the `eval` mode refinement — the body is treated as a single Python expression and the result is JSON-decoded into a typed Axioma value:

```
n: [python/eval | 2 + 3]                  # 5         (float)
arr: [python/eval | [x*x for x in range(5)]]
                                                    # [0,1,4,9,16] (array)
person: [python/eval | {"name": "ada", "age": 36}]
                                                    # ObjectMap
flag: [python/eval | 3 > 2]                  # true      (boolean)
```

Trade-offs:

Form	Body kind	Returns	Use when
`[python | … ]`	full script	`String`	side effects, multiple statements
`[python/eval | …]`	single expression	typed value	you want the result, not its print

`eval` mode rejects multi-statement bodies (Python's `eval()` only accepts an expression). For a multi-statement body that should still return a value, fall back to exec and `print(json.dumps(...))` your result.

1.29.1.2 Sharing scope with Python Free identifiers in the block body that resolve to a JSON-marshallable Axioma binding are auto-injected as Python locals before the body runs. The result is that the natural form just works:

```
xs: [1, 2, 3, 4, 5]
factor: 10
mean: [python/eval | sum(xs) / len(xs)]              # 3
scaled: [python/eval | [n * factor for n in xs]]
                                                        # [10,20,30,40,50]
```

Capture rules:

- A name is captured if it appears in the body, isn't a Python keyword or common builtin, isn't a `for X in …` loop target, and resolves to an Axioma value of a marshallable type (Integer, Float, Boolean, String, Array of marshallable, Tuple, Null).
- Names of types we can't safely cross the wire (`Set`, `ObjectMap`, `Concept`, `Reference`, MVL values, lambdas) are silently skipped — the body sees nothing for that name and Python errors normally if it tries to use it.
- Reassigning a captured name inside the body is fine and has no effect on the Axioma scope. Capture is one-way: Axioma → Python.
- Identifiers inside Python comments and string literals don't trigger capture.

For names you specifically don't want captured (e.g. an Axioma `len` that would shadow Python's builtin if it weren't already on the reserved list), the simplest workaround today is to alias the value through a name that doesn't collide.

1.29.2 27.1c Secondary runners: Julia / R / Node.js / Common Lisp

Beyond Python, four foreign runtimes run as per-call subprocesses. Each spawns the host once per block and returns its captured stdout as an Axioma `String` — **exec mode** for Julia/R/Node (no `/eval` typed-value form yet; they lack a uniform JSON round-trip). **Common Lisp also offers an `/eval` mode** (`[lisp/eval | … ]`) that returns the body's value as a typed object — see "Common Lisp specifics" below. In exec mode the body must *print* its result.

Tag(s)	Runtime	Binary (override env)
`julia`	Julia	`julia`
`r`	R	`Rscript`
`js`	Node.js	`node`
`lisp` (aliases `cl`, `commonlisp`)	Common Lisp	`sbcl` (`AXIOMA_LISP`)

```
[julia | println(sum(1:10)) ]                          # → "55"
[lisp  | (format t "~a" (+ 1 2)) ]                      # → "3"
[cl    | (princ (reduce (function *) '(1 2 3 4 5))) ] # → "120"  (factorial)
```

Common Lisp specifics.

- The body runs via `sbcl --noinform --no-sysinit --no-userinit --non-interactive --eval <body>` — hermetic (no init files), no herald, debugger disabled, clean non-zero exit on an unhandled

condition (the condition text comes back as a catchable Axioma `Error`). Print with `format` / `princ` / `print`; a bare `(+ 1 2)` returns `""` because `--eval` does not echo a form's value — for a typed value back, use `/eval` mode (next bullet).

- **Eval mode** — `[lisp/eval | … ]` (aliases `[cl/eval | … ]`, `[commonlisp/eval | … ]`) returns the body's **value** as a typed Axioma object instead of captured stdout: `[lisp/eval | (+ 1 4 9) ]` → `14` (Integer), `(/ 10 4)` → `2.5` (Float), `(list 1 2 3)` → `[1, 2, 3]` (Array), `(> 5 2)` → `true`, `nil` → `none`. CL has no stdlib JSON, so the runner wraps the body in a small self-contained JSON encoder (integers / ratios / floats / strings / symbols / `T` / `NIL` / proper-lists / vectors round-trip; an integer-valued float collapses to `Integer`, matching `[python/eval | … ]`). The body's own stdout is discarded in eval mode — only the value returns. An unknown refinement (`[lisp/foo | … ]`) is rejected without spawning sbcl.
- `AXIOMA_LISP` overrides the binary, but it must be **SBCL-compatible** — the runner passes SBCL's flags. Use it to pin a specific sbcl (e.g. a Roswell-managed one) or an alternate path, *not* to switch to clisp/ecl/ccl (whose flags differ). Unset → `sbcl` on `PATH`.
- **Body-capture caveats (Lisp-aware reader).** The Lisp dialects use a Common-Lisp-aware raw reader so the quote `'` (the QUOTE operator, normally unpaired as in `'(1 2 3)`) is **not** mistaken for a string delimiter — `[lisp | (mapcar #'1+ '(1 2 3)) ]` works. Double-quoted `"…"` strings, `;` line comments, and `#\x` character literals are skipped, so a `]` inside any of them is safe. The residual limits (rare): a bare `]` inside a `|multi-escape symbol|` or a `#| … |#` block comment will still close the block early — avoid those in inline bodies. There is no escaping; the body is passed byte-for-byte.
- **VM parity:** full. Both `[lisp | … ]` (exec) and `[lisp/eval | … ]` compile to `OpLangBlock` and run identically under `--vm` (the lang tag carries the mode).

1.29.3 27.1b Cross-language translation — `[A -> B | … ]` / `[A --> B | … ]`

Where `[<lang> | body]` *executes* foreign code, the **arrow forms** *translate* code between languages. Two new operators slot in next to the existing pipe so the surface stays uniform:

Form	Semantics	Returns
`[L | body]`	execute body in L (existing)	foreign runtime value (String for exec, typed for /eval)
`[A -> B | body]`	translate body from A to B	`String` of B's source
`[A --> B | body]`	translate, then execute in B	typed value from B's runtime
`[nl -> B | description]`	symbolize natural-language description into B	`String` of B's source
`[nl --> B | description]`	symbolize, then execute in B	typed value from B's runtime

Every form reads the body **raw** via the same bracket-counting reader the execute form uses — multi-line bodies, embedded brackets, and arbitrary indentation work without quoting. Body language is named on the left of the arrow, so the lexer doesn't have to disambiguate.

```
# Pure execution - body is Python, runs in Python (existing)
[python | print("hi")]

# Pure translation - body is Axioma, get Python source as a String
src: [axioma -> python |
    pi: 3.141592653589793
    area: func(r) [pi * r * r]
    primes: [n | n <- range(2, 50), all([n % k != 0 | k <- range(2, n)])]
]
println(src)
# pi = 3.141592653589793
# def area(r):
```

```
#     return pi * r * r
# primes = [n for n in range(2, 50) if all(n % k != 0 for k in range(2, n))]

# Translate + execute - body is Axioma, run as Python, typed value back
sum_sq: [axioma --> python | sum([n*n | n <- range(1, 11)])]
# → 385  (Python computed; marshaled to Axioma Integer)

# Reverse - body is Python source, get Axioma equivalent as a String
ax_src: [python -> axioma |
    def fibonacci(n):
        if n < 2:
            return n
        return fibonacci(n - 1) + fibonacci(n - 2)
]
# ax_src: "fibonacci: func(n) [ if n < 2 then [return n]; fibonacci(n-1) + fibonacci(n-2) ]"

# Reverse + execute - pull Python code into Axioma scope
[python --> axioma | def double(x): return x * 2]
println(double(5))   # → 10

# Cross-language to JavaScript
[axioma -> javascript | nums: [n*n | n <- range(10)]]

# Natural-language source - describe intent, get Axioma code (always LLM)
src: [nl -> axioma | double the value 7]
# src: "double: 7 * 2"  (or similar - LLM output is non-deterministic)

# Symbolize + evaluate - describe intent, get the value
v: [nl --> axioma | sum of squares from 1 to 10]
# v: 385  (or [1, 4, …, 100] - depends on how the LLM reads "sum of")
```

`nl` source — describe an intent, get code or a value. The `nl` (alias `english`, `natural`) source language treats the body as a *natural-language description*, not source code. The LLM is invoked with a symbolize-style prompt and returns idiomatic target-language code. Always LLM-required (no deterministic path exists for natural-language input), and the announce-print billing line always fires. The `-->` variant Eval's the LLM output in scope, which is powerful but inherits the LLM's non-determinism — if the model emits syntactically-foreign tokens (like a pipe-forward `|>` from Elixir's training data), the resulting parse error surfaces with the translated source attached for debuggability.

Operator mnemonic:

- Zero arrows (`|`): pure execution in the named language.
- One arrow (`->`): pure translation — returns a String of target source, no execution, no side effects.
- Two arrows (`-->`): translate + execute. For `B == "axioma"` the translated source is parsed and `Eval`'d in the **current** environment, so definitions in the body leak into the surrounding scope (this is what makes `[python --> axioma | def double(x): …]` followed by `double(5)` work). For foreign B, the translated source runs through the FFI in `/eval` mode and the typed value comes back.

Composable / String-input form: the `translate()` builtin. When the source code lives in a String variable rather than inline (read from a file, pulled from an API, etc.), use the builtin counterpart:

```
py_src: read_file("script.py")
ax_src: translate(py_src, "python", "axioma")    # (code, source, target)

py: translate([n*n | n <- range(10)])             # defaults: axioma → python
# → "[n * n for n in range(10)]"
```

Argument order is `translate(code, source_lang, target_lang)` — English "from X to Y" order, matches the legacy LLM-only translate builtin. Defaults are `source="axioma"`, `target="python"`. When the first argument is an Axioma expression (an AST node, not a String), the AST is passed alongside so the deterministic emitter takes the fast path.

1.29.3.1 Engine dispatch — deterministic emitter then LLM fallback The translation engine prefers a deterministic AST emitter when it can:

- **Axioma → Python** has a built-in visitor that handles the common learner constructs (binding statement, function, lambda, infix arithmetic, comparison, conditional, list literal, list comprehension, call, return, identifiers, literals). Output is byte-for-byte deterministic and runs entirely offline — no LLM, no network, no API key.
- **Reverse direction** (any foreign source → Axioma) and **Axioma → any non-Python target** route through the LLM (see provider table below). Output is idiomatic but non-deterministic and requires an API key.
- **Unmapped constructs** (Axioma → Python forms outside the deterministic subset — relational rules, modal logic, MVL values, etc.) fall back to the LLM automatically.

The deterministic emitter is the reason `[axioma -> python | [n*n | n <- range(10)]]` works without configuring any provider — and why the deterministic forward direction is exercised in `tests/axioma/translation/test_translate_builtin.ax` (35 byte-for-byte assertions).

1.29.3.2 LLM providers and billing transparency When the LLM path is taken, **every call prints one line to stderr before the network request goes out**, naming the provider, model, endpoint, and whether it's a paid API. The line is meant to prevent "surprise on the credit card":

```
[axioma translate] python → axioma   provider=openrouter   model=google/gemini-2.5-flash-lite   endpoint
```

Local Ollama gets `(local, no billing)`. The print is on stderr so it doesn't pollute the translated source returned to scripts. Suppress with `AXIOMA_TRANSLATE_QUIET=1` once you've confirmed your provider choice.

Provider auto-selection walks the following priority, picking the first one whose API key is set in the environment:

Provider	Env var(s)	Default model	Endpoint	Notes
OpenRouter	`OPENROUTER_API_KEY` (project-scoped) or `OPENROUTER_API_KEY`	`google/gemini-2.5-flash-lite`	`openrouter.ai/api/v1`	Routes to many backends; project-scoped key takes precedence
Anthropic	`ANTHROPIC_API_KEY`	`claude-3-opus-20240229`	`api.anthropic.com/v1`	Claude family
Gemini	`GEMINI_API_KEY`	`gemini-2.5-flash`	`generativelanguage.googleapis.com/v1beta`	Google direct
Grok (xAI)	`XAI_API_KEY` or `GROK_API_KEY`	`grok-4`	`api.x.ai/v1`	Distinct from Groq
OpenAI	`OPENAI_API_KEY`	`gpt-4o-mini`	`api.openai.com/v1`	
Groq	`GROQ_API_KEY`	`llama-3.1-70b-versatile`	`api.groq.com/openai/v1`	Inference vendor, not xAI
Ollama	none (local)	`llama2`	`localhost:11434`	Override via `OLLAMA_MODEL=…`

OpenRouter is first because it routes to many backend models behind one billing surface — the recommended default. Override the OpenRouter model with `OPENROUTER_MODEL=anthropic/claude-3.5-sonnet` (or any other OpenRouter slug) without recompiling.

1.29.3.3 Multi-line bodies and """…""" are unnecessary

A common first instinct is to wrap foreign source in a triple-quoted string for the reverse direction:

```
# DON'T - body is read raw, no string quoting needed:
[axioma <- python | """
def f(x):
    return x * 2
"""]

# DO - bracket-counting raw reader handles multi-line directly:
[python -> axioma |
def f(x):
    return x * 2
]
```

The raw reader closes on the matching outer `]` and counts inner brackets correctly, so `[1, 2, 3]` and nested comprehensions inside the body work as long as their brackets balance.

1.29.3.4 MCP integration

The same translation engine is exposed via the `translate_code` tool on the MCP server (`axioma --mcp`). Clients call it with `code`, `source_lang`, `target_lang` and get back the same engine output — deterministic when applicable, LLM-backed otherwise — plus the parsed AST as JSON when the source is Axioma. One engine, two surfaces.

1.29.4 27.1c The [sql | …] block — embedded SQL

Where `[python | …]` calls out to a foreign runtime, `[sql | …]` **compiles** SQL down to native Axioma. The block is an expression; its value is the result of running the compiled comprehension/ transaction. There is no foreign process, no marshaling — SQL is treated as another *surface* over Axioma's relational substrate, sitting beside the pipe-form and Prolog-form comprehensions covered in Chapter 7.

```
[sql | CREATE TABLE teacher (instructor VARCHAR, student VARCHAR)]
[sql | INSERT INTO teacher VALUES ('Socrates', 'Plato')]
[sql | INSERT INTO teacher VALUES ('Plato', 'Aristotle')]

[sql | SELECT student FROM teacher WHERE instructor = 'Socrates']
# → {"Plato"}
```

The same data is reachable via Axioma's native idioms:

```
# Set comprehension
{Y | teacher("Socrates", Y)}
# → {"Plato"}

# Prolog-form
{Y | Y <- teacher("Socrates", Y)}
# → {"Plato"}
```

All three surfaces produce identical results because they all lower to the same comprehension over the `_relation_` store.

1.29.4.1 DDL — CREATE TABLE, DROP TABLE, TRUNCATE

```
# CREATE TABLE - declares a new relation
[sql | CREATE TABLE emp (name VARCHAR, dept_id INTEGER, salary INTEGER)]

# Parameterized types: VARCHAR(N), DECIMAL(P, S), etc. The size
# arguments are accepted at parse time for SQL compatibility but
```

```
# discarded at emission - Axioma's relations are positionally typed,
# not nominally, so size caps are not enforced.
[sql | CREATE TABLE prices (sku VARCHAR(32), amount DECIMAL(10, 2))]

# IF NOT EXISTS - idempotent CREATE; Axioma's `relation X(...)` is
# already idempotent, so this is an emission no-op accepted for SQL
# surface compatibility.
[sql | CREATE TABLE IF NOT EXISTS emp (name VARCHAR, dept_id INTEGER)]

# DROP TABLE - removes the relation's schema, all stored facts at
# every grounding tier, and parallel metadata in one sweep.
[sql | DROP TABLE emp]

# IF EXISTS - silent no-op if missing (drop_relation is already a
# silent no-op for unknown relations).
[sql | DROP TABLE IF EXISTS doesnt_exist]

# TRUNCATE TABLE - clear the extent, preserve the schema. Faster
# than DELETE without a WHERE because no per-row predicate runs.
# The TABLE keyword is optional (Postgres-style).
[sql | TRUNCATE TABLE emp]
[sql | TRUNCATE emp]            # equivalent
```

1.29.4.2 DML — `INSERT`, `UPDATE`, `DELETE`

```
# INSERT … VALUES
[sql | INSERT INTO emp VALUES ('Alice', 1, 90000)]

# INSERT … SELECT (copy rows from another relation)
[sql | INSERT INTO emp_backup SELECT * FROM emp WHERE dept_id = 1]

# UPDATE with arithmetic RHS
[sql | UPDATE emp SET salary = salary * 1.10 WHERE dept_id = 1]

# DELETE
[sql | DELETE FROM emp WHERE salary < 50000]
```

All three DML statements wrap the read+write in a transaction so a partial failure rolls back cleanly.

1.29.4.3 Queries — `SELECT` with the full join family

```
# Single-table SELECT with WHERE
[sql | SELECT name, salary FROM emp WHERE dept_id = 1]

# DISTINCT
[sql | SELECT DISTINCT dept_id FROM emp]

# INNER JOIN
[sql | SELECT emp.name, dept.dname FROM emp
       INNER JOIN dept ON emp.dept_id = dept.id]

# LEFT [OUTER] JOIN - preserves unmatched left rows, NULL-pads right
[sql | SELECT emp.name, dept.dname FROM emp
       LEFT JOIN dept ON emp.dept_id = dept.id]
```

```
# RIGHT [OUTER] JOIN - preserves unmatched right rows
[sql | SELECT emp.name, dept.dname FROM emp
       RIGHT OUTER JOIN dept ON emp.dept_id = dept.id]

# FULL [OUTER] JOIN - preserves both
[sql | SELECT emp.name, dept.dname FROM emp
       FULL OUTER JOIN dept ON emp.dept_id = dept.id]

# Comma-FROM (pre-SQL-92 implicit cross-join - the constraints
# come from WHERE)
[sql | SELECT emp.name, dept.dname FROM emp, dept
       WHERE emp.dept_id = dept.id]

# Three-table comma-FROM
[sql | SELECT emp.name, dept.dname, region.rname
       FROM emp, dept, region
       WHERE emp.dept_id = dept.id AND emp.dept_id = region.id]
```

NULL padding for OUTER joins follows the active refinement (`/b4` default → `belnap("neither")`, displayed `?` ; `/k3` → `kleene("unknown")`).

1.29.4.4 Aggregates — `GROUP BY`, `HAVING`, `COUNT/SUM/AVG/MIN/MAX`

```
# Single-column GROUP BY with COUNT
[sql | SELECT dept_id, COUNT(*) FROM emp GROUP BY dept_id]

# SUM with WHERE
[sql | SELECT dept_id, SUM(salary) FROM emp
       WHERE salary > 50000 GROUP BY dept_id]

# Multi-column GROUP BY - each distinct (col1, col2, ...) tuple
# gets its own group; result rows flatten to (col1, col2, ..., agg)
[sql | SELECT region, product, SUM(qty) FROM sales
       GROUP BY region, product]

# HAVING - filter post-aggregation
[sql | SELECT dept_id, SUM(salary) FROM emp
       GROUP BY dept_id HAVING SUM(salary) > 200000]

# Single-column scalar aggregate (no GROUP BY)
[sql | SELECT COUNT(*) FROM emp WHERE dept_id = 1]
# → 3
```

Phase 5 caveat: single aggregate per SELECT, and HAVING currently requires single-column GROUP BY.

1.29.4.5 Expressions — `CAST`, `CASE`, `LIKE`, `BETWEEN`, `IN`, `EXISTS`

```
# CAST - type conversion (CAST(expr AS type) and Postgres :: shorthand)
[sql | SELECT CAST(score AS FLOAT) FROM grades WHERE student = 'Alice']
[sql | SELECT score :: FLOAT FROM grades WHERE student = 'Alice']

# Chained postfix casts
[sql | SELECT score :: INT :: TEXT FROM grades]

# Searched CASE - independent predicates per branch
[sql | SELECT name,
```

```
              CASE WHEN salary > 100000 THEN 'high'
                   WHEN salary > 50000  THEN 'mid'
                   ELSE 'low'
              END
       FROM emp]

# Simple CASE - subject compared by equality
[sql | SELECT name,
              CASE grade WHEN 'A' THEN 'excellent'
                         WHEN 'B' THEN 'good'
                         ELSE 'fair'
              END
       FROM grades]

# LIKE - SQL wildcards (% any-sequence, _ single-char)
[sql | SELECT name FROM emp WHERE name LIKE 'A%']

# BETWEEN / NOT BETWEEN - inclusive range
[sql | SELECT name, salary FROM emp WHERE salary BETWEEN 60000 AND 90000]

# IN (literal list)
[sql | SELECT name FROM emp WHERE dept_id IN (1, 2, 3)]

# IN (SELECT …) subquery
[sql | SELECT name FROM emp
       WHERE dept_id IN (SELECT id FROM dept WHERE dname = 'Eng')]

# EXISTS - non-empty subquery test
[sql | SELECT name FROM emp e WHERE EXISTS
         (SELECT 1 FROM dept d WHERE d.id = e.dept_id)]
```

1.29.4.6 Set operations — `UNION`, `INTERSECT`, `EXCEPT`

```
# UNION - set union (deduplicates)
[sql | SELECT name FROM emp_jan
       UNION
       SELECT name FROM emp_feb]

# UNION ALL - bag union (preserves duplicates)
[sql | SELECT name FROM emp_jan
       UNION ALL
       SELECT name FROM emp_feb]

# INTERSECT - set intersection
[sql | SELECT name FROM emp_jan INTERSECT SELECT name FROM emp_feb]

# EXCEPT - set difference
[sql | SELECT name FROM emp_jan EXCEPT SELECT name FROM emp_feb]
```

1.29.4.7 Common Table Expressions — `WITH`

```
# Single CTE
[sql | WITH high_paid(name, salary) AS
         (SELECT name, salary FROM emp WHERE salary > 100000)
       SELECT name FROM high_paid ORDER BY salary DESC]
```

```
# Chained CTEs - each later CTE can reference earlier ones
[sql | WITH
         eng(name, salary)        AS (SELECT name, salary FROM emp
                                      WHERE dept_id = 1),
         high(name, salary)       AS (SELECT name, salary FROM eng
                                      WHERE salary > 100000),
         greeted(greet)           AS (SELECT 'Hi ' + name FROM high)
       SELECT greet FROM greeted]
```

1.29.4.8 Refinement modes — `/bag`, `/k3`, `/strict`, `/distinct`, `/explain` The block accepts SQL-shaping refinements that pre-configure how the compiled comprehension treats duplicates, NULLs, and shape strictness:

Refinement	Effect
`[sql/bag | …]`	Duplicate-preserving (bag) semantics
`[sql/distinct | …]`	Force DISTINCT projection (default)
`[sql/k3 | …]`	SQL NULL lowers to Kleene K3 `unknown`
`[sql/strict | …]`	Strict shape checking on projection arity
`[sql/explain | …]`	Return the **compiled Axioma source** as a String — useful for teaching and debugging

```
src: [sql/explain | SELECT name FROM emp WHERE dept_id = 1]
println(src)
# {V1 | emp(V1, 1, V3)}
```

1.29.4.9 Lineage surface — `[sql -> algebra]` / `[sql -> calculus]` For pedagogical traceability there are two "translation" forms that emit the *intermediate-representation* equivalents of the SQL:

```
[sql -> algebra | SELECT name FROM emp WHERE dept_id = 1]
# → " _{name}( _{dept_id = 1}(emp))"

[sql -> calculus | SELECT name FROM emp WHERE dept_id = 1]
# → "{ t.name | t   emp   t.dept_id = 1 }"
```

These don't run the query — they're string outputs of the relational- algebra and tuple-calculus forms, useful for the textbook chapter on database theory.

1.29.4.10 Identifying what's not yet covered Phase 5 explicitly defers a few large features to follow-on landings: **window functions** (`OVER`, `PARTITION BY`, `ROW_NUMBER`), **correlated subqueries** (subqueries that reference outer-query columns), **`CREATE TABLE` constraints** (`PRIMARY KEY`, `FOREIGN KEY`, `NOT NULL`, `DEFAULT`, `REFERENCES`, `CHECK`), **`ALTER TABLE`**, **multiple aggregates per SELECT**, **chained outer joins**, **non-equijoin OUTER JOIN predicates**, and **HAVING with multi- column GROUP BY**. None of them prevent the working examples above from running.

1.29.5 27.4b Three quietly important language fixes

Three small changes to the core language that came out of the comparison-tooling work and benefit every Axioma program, not just Python interop.

Short-circuit `and` / `or`. Previously both operands were evaluated even when the first determined the result. The natural guard pattern

```
if i <= len(a) and a[i] > 0 then …
```

errored with "index out of bounds" because `a[i]` ran when `i` was out of range. Now `and` and `or` short-circuit Boolean operands the way every other modern language does: the right side is only evaluated when the left doesn't already settle the question. The multi-valued logics (Belnap, Łukasiewicz, etc.) are unaffected — they still need both inputs for their lattice operations.

`[] in then / else` is an empty array. Previously parsed as an empty block (which evaluates to `null`), silently breaking natural recursive base cases:

```
to_list: func(t) [
  if t == none then [] else [t.value] + to_list(t.tail)
]
```

Pre-fix this stack-overflowed because the base case returned `null` and `null + [x]` fails type-check (which the recursion never gets to anyway because the recursion never bottoms out on a `null` it keeps trying to concatenate). Same fix simultaneously enables `then [x] + ys` and other forms where the bracketed clause is followed by an infix operator — both now extend correctly past the closing bracket.

`for` as alias for `foreach`. Python/JS/Java/Rust developers expect `for x in xs [body]`. Axioma already had `foreach x in xs [body]` doing exactly this; `for` is now a lexer-level alias that produces the same AST. Both forms are first-class and interchangeable — pick whichever reads better. Destructuring works under both names: `for [k, v] in pairs [...]`.

`py.eval / py.exec` auto-capture. The `[python | …]` block form already auto-injected free Axioma identifiers into the Python scope. The shim-style `py.eval(string)` did not — the string shipped verbatim. Now the two paths agree:

```
xs: [1, 2, 3, 4, 5]
py.eval("sum(xs)")              # 15 - xs is auto-captured
py.exec("print(len(xs))")       # 5
```

For multi-statement bodies under `py.eval`, the runtime lifts the prelude through `exec` + `json.dumps` internally so the typed-result contract is preserved.

1.29.6 27.5a Comparison-friendly tools

Three small additions make side-by-side Axioma/Python work feel natural rather than ad-hoc.

1.29.6.1 `bench(label, fn)` — measure one call

```
xs: [3, 1, 4, 1, 5, 9, 2, 6]

ax: bench("axioma sort", func() [my_sort(xs)])
py: bench("python sort", func() [py.call("sorted", xs)])

println(ax.label, ax.elapsed_ms, "ms vs", py.label, py.elapsed_ms, "ms")
println("agree?", ax.result == py.result)
```

`bench` returns an `ObjectMap` with three keys: `label`, `elapsed_ms` (Float, sub-millisecond precision), and `result`. Use it whenever you want to compare two implementations on more than just behavior.

1.29.6.2 `:compare` (REPL) — typed-value diff with timing

```
:compare 2 + 3 // 2 + 3
  axioma  5   [233µs]
  python  5   [368µs]
    values agree

:compare [1, 2, 3] // [3, 2, 1]
  axioma  [1, 2, 3]   [17µs]
```

```
python  [3, 2, 1]   [54µs]
  values differ
```

Both sides are evaluated as **typed expressions** (the Python side goes through `py.eval`, so it returns Axioma typed values, not captured stdout). Equality is structural — arrays and object-maps compare element-wise; integers and floats coerce freely.

1.29.6.3 `lib/cs/data_structures.ax` — pure-Axioma classics A small library of canonical data structures so you have something real to compare against rather than building each from scratch:

Structure	Constructor	Key operations
LinkedList	`list_create()`	`list_push`, `list_pop`, `list_size`, `list_from_array`, `list_to_array`
Binary Search Tree	`bst_create()`	`bst_insert`, `bst_contains`, `bst_inorder`, `bst_size`, `bst_from_array`
Stack	`stack_create()`	`stack_push`, `stack_pop`, `stack_peek`, `stack_size`
MinHeap	`heap_create()`	`heap_push`, `heap_pop`, `heap_peek`, `heap_size`, `heap_from_array`, `heap_drain_sorted`

Style: functional/persistent for the recursive structures (LinkedList, BST), array-backed with mutation for the index-heavy ones (Stack, MinHeap). Each instance is an `ObjectMap`; persistent operations take the instance and return an updated copy.

```
import "lib/cs/data_structures.ax"

h: heap_from_array([3, 1, 4, 1, 5, 9, 2, 6])
ax_sorted: heap_drain_sorted(h)
py_sorted: [python/eval | sorted([3, 1, 4, 1, 5, 9, 2, 6])]
println("agree?", ax_sorted == py_sorted)
```

HashMap is intentionally absent — Axioma's native `ObjectMap` is already a hash table and reads as one (`m.key`, `m["key"]`, `len(m.keys)`). Wrapping it in a `hashmap_*` API would just add ceremony.

1.29.7 27.5 Walkthrough: lists and loops, three ways

Same algorithm, three styles. The point is not that any one is best — it's that mixing is cheap, so you can adopt incrementally.

Goal: given a list of numbers, compute the mean and a rough standard deviation, then print the values that are more than one above the mean.

1.29.7.1 Style 1 — pure Axioma

```
xs: [4, 8, 15, 16, 23, 42]
n: len(xs)
mean: sum(xs) / n
var: sum([(x - mean) * (x - mean) | x <- xs]) / n
sigma: math.sqrt(var)
outliers: [x | x <- xs, x > mean + sigma]
println("mean=", mean, "sigma=", sigma, "outliers=", outliers)
```

`math.sqrt` is a shim — it routes to Python's `math.sqrt` under the hood, but the call site is just Axioma. If a native `math.sqrt` lands later, the line doesn't change.

1.29.7.2 Style 2 — inline Python block

```
xs: [4, 8, 15, 16, 23, 42]
report: [python |
import statistics
m = statistics.mean(xs)
s = statistics.stdev(xs)
outliers = [x for x in xs if x > m + s]
print(f"mean={m} sigma={s} outliers={outliers}")
]
println(report)
```

The Axioma `xs` is auto-captured into the Python scope. The whole Python body runs as one subprocess (or one round-trip in the persistent worker), and the captured `print(...)` text comes back as an Axioma `String`.

1.29.7.3 Style 3 — typed-result block (`/eval`)

```
xs: [4, 8, 15, 16, 23, 42]
stats: [python/eval |
{"mean": __import__('statistics').mean(xs),
 "sigma": __import__('statistics').stdev(xs)}
]
outliers: [x | x <- xs, x > stats.mean + stats.sigma]
println("from python:", stats, " outliers:", outliers)
```

Now the Python side returns a typed `ObjectMap`, and the outliers filter is back in Axioma where it composes with the rest of the program. This is usually the right balance: borrow Python's library for the hard part, keep the surrounding logic native.

1.29.7.4 Comparing two styles in the REPL

```
:compare math.sqrt(2) // statistics.stdev([1,2,3,4,5])
```

Both halves run, and the REPL prints them side-by-side with a when they match. Useful when porting a Python snippet — write the Axioma version, paste the original on the right of `//`, watch them agree (or not).

1.29.7.5 Picking the style

Situation	Style
The library exists in Axioma	Pure Axioma
You need a Python stdlib function for one expression	Shim (e.g. `math.sqrt`, `statistics.mean`)
Multi-line Python you don't want to translate yet	`[python | …]` exec block
You want a typed value back from Python	`[python/eval | …]`
You want to see the Python equivalent of your Axioma	`[axioma -> python | …]` (returns String)
You want Axioma computed via Python's runtime	`[axioma --> python | …]` (typed value back)
You're learning from a Python snippet	`[python -> axioma | …]` (returns String of Axioma source)
Pull Python code into Axioma scope	`[python --> axioma | …]` (definitions leak into scope)

Situation	Style
Code comes from a String variable	`translate(src, "python", "axioma")` builtin
You know what you want but not the Axioma syntax	`[nl -> axioma | description]` (LLM symbolize, returns String)
You want the value but don't care about the code	`[nl --> axioma | description]` (LLM symbolize + Eval)
You're porting and want a safety net	`:compare` while you write the Axioma version

Performance notes:

- The persistent worker is **on by default** (lazy-spawned on first Python use, so it costs nothing if no `[python | …]` block runs). Same script reuses one long-lived subprocess at ~0.5–2ms per call.
- Pass `--no-python-worker` to disable. Each block then spawns its own fresh `python3 -c` process (~30-50 ms). Use this when you need hard isolation between unrelated blocks or when running in a restricted environment that disallows long-lived subprocesses.

1.29.7.6 Globals persist across blocks (worker mode)

This is the most important behavior to know about the default. With the worker on:

```
_setup: [python | x = 41 ]
r: [python/eval | x + 1 ]      # → 42 - x carries over
```

The two blocks share Python's namespace. That matches REPL semantics and is what you want most of the time. But if you paste two unrelated snippets into the same script and they happen to use the same variable names, they'll see each other.

Three options when you need isolation:

1. `py.reset()` — clears worker globals between blocks without leaving worker mode:

   ```
   r1: [python | x = "first" ]
   py.reset()
   r2: [python | print('x' in dir()) ]    # False
   ```

2. **Use a unique prefix** for variables in self-contained snippets, or wrap the snippet in a function so its locals don't escape.

3. `--no-python-worker` — fall back to per-call subprocess for the whole run. Slower, but every block starts from a fresh namespace.

1.29.8 27.6 Narrowing the boundary (worker-mode features)

The persistent Python worker is on by default. Four extra channels become available that make the boundary feel less like a foreign- function call and more like a peer (none of these work with `--no-python-worker` — the per-call subprocess can't talk back over stdio):

1.29.8.1 Unified namespaces

```
py.eval("math.sqrt(2)")          # typed Float
py.exec("print('hi')")           # captured stdout (String)
py.call("math.sqrt", 2)          # function-style, typed result
py.import_("numpy")              # make numpy available downstream
py.reset()                       # clear worker globals
julia.exec("println(2+3)")       # same shape for julia / r / js
```

One obvious entry point per dialect, instead of "block exec / block eval / shim — pick the right one." All four dialects (`py`, `julia`, `r`, `js`) expose `.exec`; `py` additionally exposes `.eval`, `.call`, `.import_`, and `.reset` (clear globals between blocks).

1.29.8.2 Reverse calls — Python invokes Axioma

```
double: func(x) [x * 2]
format: func(x, y) ["x=" + x + ", y=" + y]

[python | print(axioma.call("double", 21)) ]      # 42
[python | print(axioma.call("format", 10, 20)) ]  # x=10, y=20
```

Inside any Python block, `axioma.call(name, *args)` resolves `name` in the surrounding Axioma scope, marshals args, calls the function, and returns the typed result. Python errors and Axioma errors propagate across the boundary as `RuntimeError`.

1.29.8.3 Object handles — large values, no deep copy For values too big to want to marshal whole, the `axioma.*` namespace exposes lazy access:

```
big: [...]  # imagine a 1M-element array

[python/eval | axioma.len("big")]              # length only, no copy
[python/eval | axioma.at("big", 0)]            # single element
[python/eval | axioma.array("big")[100:110]]   # slice via proxy
```

`axioma.array(name)` returns a Python proxy whose `__len__`, `__getitem__`, and `__iter__` round-trip per access. The underlying Axioma array is never deep-copied; you pay a per-element fetch cost but avoid the upfront marshalling time and the memory pressure.

1.29.8.4 Streaming progress — `axioma.emit` Long-running blocks can report intermediate values:

```
[python |
import time
for i in range(5):
    axioma.emit(f"step {i+1}/5")
    time.sleep(1.0)
print("done")
]
```

Each `axioma.emit(value)` writes a one-way stream message that the Axioma side prints live (default formatter prefixes with `[emit]`). The block's own return value is unaffected — it's still the captured stdout (or, for `/eval`, the typed expression result). Stream and return are independent channels.

1.29.8.5 What's still missing

- One-way scope only. Axioma values flow into Python; mutations inside Python don't propagate back. Use the block's return or `axioma.call` to write a setter explicitly.
- Reverse calls require worker mode. Per-call subprocess (`python3 -c`) can't talk back over stdio.
- Eval mode requires single-expression bodies. Multi-statement bodies with auto-capture get lifted through exec internally; bare multi-statement /eval errors with `SyntaxError`.

Disambiguation rule: `[expr | var <- iterable, …]` is still a list comprehension. The lang-block form is taken only when the first token is a registered dialect identifier and the next token is `|`.

1.29.9 27.2 Python-stdlib shims ( )

A shim is a thin Axioma binding whose only job is to make a familiar call site keep working. At seed time the interpreter registers a catalog of Python stdlib helpers as if they were native Axioma modules:

```
math.sqrt(2)              # 1.4142135623730951
math.pi                   # 3.141592653589793
statistics.mean([1,2,3])  # 2
```

```
random.uniform(0, 1)      # 0.6394267984578837
json.dumps([1,2,3])       # "[1, 2, 3]"
```

Under the hood each call marshals its arguments to a Python literal, splices them into a one-line `eval` expression, and round-trips through the `python_interop` FFI subprocess. The Axioma source never has to know.

Shims have a four-phase lifecycle:

1. **Route** — the shim ships, calls go out to Python.
2. **Native exists** — an Axioma implementation lands; the shim still routes to Python so the caller is undisturbed.
3. **Alias** — the shim is rewritten as a thin alias to the native.
4. **Delete** — the shim is removed; caller code already worked.

To skip shim seeding (for stricter scripts or to reclaim startup time when Python isn't on `PATH`):

```
axioma --no-shims script.ax
```

1.29.10 27.3 REPL affordances

```
:hints on             # surface translation tips inline as you type
:hints status         # show on/off + catalog size
:hints list           # print the full hint catalog
:hint range           # one-shot lookup
:compare sum([1,2,3]) // sum([1,2,3])
                      # run the Axioma side and the Python side, compare
```

`:compare` is the side-by-side tool: type an Axioma expression on the left of `//` and a Python expression on the right; both run against the live REPL environment and a ✓ / ✗ flag tells you whether they produced identical printed values.

1.29.11 27.4 When to use what

Use	Tool
Quick port of a Python snippet you already trust	`[python | … ]`
Calling a single stdlib helper inline	`math.sqrt(x)` (shim)
Tutorial / onboarding ("look how similar this is")	`:compare`
Catching pasted Python in a fresh REPL	`:hints on`

The tests under `tests/axioma/lang_blocks/` exercise both the block form and the shim catalog.

1.30 29. Errors as First-Class Values

Errors in Axioma are **catchable, inspectable values**, not just halts — a value model with no stack-unwinding exception. **Failure stays distinct from error:** an empty query result or `none` is not an error; only a genuine fault (division by zero, undefined word, type mismatch, …) produces an `Error` value.

1.30.1 Four failure policies, one expression

The same expression can be wrapped four ways, depending on what you want when it faults:

Form	On success	On error	Reach for it when…
`EXPR`	value	**halts / propagates**	errors should propagate

Form	On success	On error	Reach for it when…
`try EXPR`	value	the **Error value** (inspectable)	you want to inspect or classify it
`EXPR otherwise D`	value	`D`	you have a specific fallback
`attempt EXPR`	value	**`none`**	you just want "nothing" on failure

`try` binds **tightly**, like a unary operator: it grabs a single primary — a parenthesized group, a call, or member access — so a trailing operator applies to the **caught result**. Thus `try(risky()) is Error` `(try(risky())) is Error` → `true`, matching the bind-it-first idiom `e: try(risky()); e is Error`. To make a whole *binary* computation optional, parenthesize it: `try (a / b)`. `attempt` and the `otherwise` operator still bind their operand **greedily** to the end of the expression — parenthesize to scope those tighter.

1.30.2 `try` — catch to a value

```
e: try(10 / 0)            # caught - does NOT halt; e IS the error value
error?(e)                  # → true
type(e)                    # → "Error"     (e is Error → true - seeded primitive Concept)
e.message                  # → "division by zero"
e.hint                     # → recovery hint   (also .kind / .line / .column / .source / .file)

try(2 + 3)                 # → 5   (success passes through untouched)
```

1.30.3 Constructing & re-raising

```
b: error("boom", "fix it")          # build an inert error VALUE (not raised)
b.message                            # → "boom"   ;  b.hint → "fix it"
try(raise(b))                        # raise() re-arms propagation; try catches it back
```

1.30.4 `otherwise` / `or else` — the fallback railway

`LEFT otherwise RIGHT` (also spelled `LEFT or else RIGHT`) returns `LEFT`'s value, or — if `LEFT` faults — falls back to `RIGHT`. It is itself a catch point, so no `try` wrapper is needed. `RIGHT` is evaluated **lazily** (only on failure), and the operator is the loosest real operator, so both sides form fully before it combines them.

```
parse_int("xx") otherwise 0          # → 0    (catches the parse fault directly)
(10 / 0) otherwise 99                # → 99
lookup(k) or else "not found"        # two-word spelling, identical semantics
a() otherwise b() otherwise c()      # left-assoc chain: first success wins
```

`otherwise` is a **soft keyword** — `otherwise: 5` is still a valid binding; it reads as the operator only at the infix slot between two same-line expressions.

1.30.5 `attempt` — swallow to `none`

```
n: attempt parse_int(user_input)     # an Integer, or `none` if it didn't parse
(attempt (1 / 0)) == none            # → true
(attempt (1 / 0)) == om              # → false   (the dual-null distinction is load-bearing)
```

A failed computation has **no result** (`none` — Frege's "no Bedeutung"), not an *undetermined* one (`om` — SETL's Ω). Use `attempt` for the none-result form and `otherwise` for a non-`none` default; they are alternatives, not partners.

1.30.6 Deep recursion is a catchable error

Unbounded or very deep recursion returns a clean, **catchable** `Error` instead of crashing the host with a stack overflow. The guard bounds `Eval` re-entrancy depth, so it covers every body shape:

```
spin: func(n) [if n <= 0 then 0 else spin(n - 1)]
e: try(spin(100000))         # caught - interpreter stays usable afterward
e is Error                    # → true   ("recursion limit exceeded")
recursion_limit()             # → the live ceiling (50000 native; 350 under wasm)
```

`recursion_limit()` reports the current ceiling. The default is target-specific — the browser/playground stack is far tighter than a native goroutine stack — and the VM is already safe (it recurses on an explicit capped frame stack, returning a clean `"stack overflow"`).

1.30.7 Error kinds as sub-Concepts

A caught error classifies into a **kind**, so you can branch on *why* it failed. Six built-in kinds are seeded as sub-Concepts of `Error`:

Concept	Matches errors whose message says…
`DivByZero`	`… by zero` (division / modulo / divmod / quotient / remainder)
`TypeError`	`type mismatch …` / `unknown operator …`
`NameError`	`Undefined word …` / `identifier not found`
`IndexError`	`index out of bounds` / `out of range`
`ArityError`	`wrong number of arguments`
`CallError`	`not callable` / `not a function`

```
e: try(10 / 0)
e is DivByZero        # → true
e is Error            # → true    (the generic Error concept still matches any error)
e.kind                # → "DivByZero"   (agrees with the `is` test)

if      e is NameError then println("typo in a name")
else if e is TypeError then println("incompatible types")
else                        println(e.message)

# stamp a kind authoritatively on a user error (survives raise + re-catch):
u: error("bad input", "expected a number", "TypeError")
u is TypeError        # → true
```

The kind is **derived on demand** from the error's message by a central matcher, so the hundreds of existing error sites need no per-site tagging; unrecognized messages classify as generic `Error` only, never misattributed.

> **VM parity:** `try` / `otherwise` / `attempt` are evaluator-only — under `--vm` they report a clean `compilation not implemented`. The value-level builtins `error()` / `raise` / `error?` *do* work under `--vm`. Porting the whole errors-as-values floor to the VM is a single later phase.

1.31 30. The Cognitive Kernel

After procedural, object-oriented, functional, and logical, Axioma adds a **cognitive** layer whose defining primitive is `understand`. *Computing as the mind does* — the mind, in one word, **models**; the knowledge base *is* that world-model, and to `understand(X)` is to model X into the model of everything. The kernel adds **abduction** — Peirce's third inference mode — so deduction (strict Horn `<==`), induction (defeasible `<~~`), and abduction (`abduce`) all finally have a home.

Builtin	Role	Returns
`abduce("rel", a…)`	inference to an explanation (Peirce)	`Array` of `(explanation, "strict"/"defeasible")`
`examine(Concept)` / `examine("rel", a…)`	the gate (System 2): meaning · warrant · contract · suspicion	`ObjectMap` verdict
`understand(…)`	the engine: represent → abduce + predict → examine → graded verdict	`ObjectMap`

```
relation bird(x)
relation flies(x)
flies(X) <~~ bird(X)
assert bird("tweety")

abduce("flies", "tweety")
# → [("bird("tweety")", "defeasible")]     the rule body whose head derives it

concept Phlogiston { formed_by: "stipulation" }
examine(Phlogiston)
# → {meaningful: false, pseudo: true, contract: ? ,
#    verdict: "pseudo-concept - an empty word (no boundary, examples, or instances)", ...}

understand("flies", "tweety")
# → {represents, examination, explanations, predicts, coherent, verdict}
```

All three builtins are **shadowable** (a user binding of the same name wins). Each also has a **natural-language surface** that self-displays, like `why`:

```
understand Phlogiston          # → println(understand(Phlogiston))   (TitleCase-concept arg)
examine "gravitates"           # string arg → fact mode
abduce flies("tweety")         # fact-call form - the sibling of `why <conclusion>`
```

The NL prefixes are *soft*: they fire only on a literal / TitleCase-concept argument (or, for `abduce`, a fact-call); bindings (`understand: …`) and ordinary calls (`understand(x)`) fall through untouched.

> **Rule heads auto-register as iterable relations.** `H(X) :- B` makes `H` iterable (`for y in H`), queryable, and introspectable (`@H == "Relation"`) with no separate `relation H` declaration — *you can iterate whatever you can derive.*

1.31.1 `[ model | … ]` — the advisory lifecycle lens

The **authoring** counterpart to the kernel. `[ model | body ]` runs `body` as native Axioma in the current environment **and** prints an *advisory* panel placing your code on the epistemic lifecycle of a knowledge model — **represent → ground → infer → prove → assess-truth → apply** — naming the machinery you have not reached for yet. It is **advisory, not gating**: your code runs exactly as written, and the block returns the body's last expression.

```
result: [ model |
    relation mortal(x)
    axiom mortal("socrates")        # represent + ground (as an axiom)
    human(X) :- mortal(X)           # infer (strict backward rule)
    {X | X <- human(X)}             # query → {"socrates"}
]
# panel → stderr:
#   model · epistemic lifecycle
#    represented
```

```
#    grounded
#    inferred
#    proved        → why <conclusion>  ·  proof(rel, args…)
#    truth-valued  → set_truth(rel, args…, "both")  ·  truth(rel, args…)
#    applied       → check / examine(Concept) (→ B4)  ·  understand(…) runs the whole arc
#   3/6 phases present · advisory only - your code runs exactly as written.
```

- `[ model/report | … ]` runs the body but **returns** the classification as a dot-accessible `ObjectMap` (`{represented, grounded, inferred, proved, truth_valued, applied, present, total, value}`) instead of printing — so code can branch on the verdict (`if rep.grounded then …`).
- Suppress the panel with `AXIOMA_MODEL_QUIET=1`. The only accepted refinement is `/report`. Evaluator-only, like every language block.

1.32 31. Proof Assistant

Axioma hosts a small, **auditable natural-deduction proof checker** as an importable library — `lib/proof`. You write a proof as a list of numbered steps, hand it to `certify` against a goal, and get back a **sealed `CheckedTheorem`**: a value that exists *only* because the proof actually checks. A bad proof never crashes — it comes back as a rejection that says why. It is the classical first-order calculus **with equality** (→ ¬ =, classical RAA / excluded middle), and every certification is re-validated by a **second, independent checker written in Go** whose theorem value has unexported fields — so no Axioma literal can forge one.

```
import "proof"

atomA: kpred("A", [])
t:     certify([ assume(atomA), impI(1, 1) ], kimp(atomA, atomA), [])

proved(t)        # → true
showTheorem(t)   # → "  (A → A)"
```

`import "proof"` resolves the bundled library from anywhere — no `AXIOMA_PATH`, no `lib/` checkout, and identically in the browser playground (the library is compiled into the binary). `import "proof.Core"` and `import "lib/proof/Core.ax"` name the same module; in a source checkout an on-disk `lib/proof/Core.ax` overrides the bundled copy so development edits are live.

A proof has three parts:

- **Formulas are data**, built with `k`-prefixed constructors — `kand` / `kor` / `knot` / `kimp` / `kiff` / `kall` / `kex` / `kpred` / `kv` / `keq` / `kin`, and the value `kFalse` ( ). The prefix is required: `and` / `or` / `not` / `forall` / `in` / `eq` are Axioma keywords, so the object-language connectives can't reuse them.
- **A proof is a list of steps**, each citing earlier steps by their **1-based line number** — a Fitch/Lemmon derivation, written as data.
- **`certify(steps, goal, catalog)` is the trust boundary.** It runs the proof, checks the last line equals `goal`, collects undischarged assumptions as Γ, and returns `Γ  goal` sealed — or `{tag: "rejected", reason}`.

Walk a proof of `(A  B) → A`:

```
atomA: kpred("A", [])
atomB: kpred("B", [])

t: certify([
  assume(kand(atomA, atomB)),   # 1.  A  B          (open assumption)
  andEL(1),                     # 2.  A              ( -elim left, from line 1)
  impI(1, 2)                    # 3.  (A  B) → A     (discharge line 1)
```

```
], kimp(kand(atomA, atomB), atomA), [])

proved(t)        # → true
len(hyps(t))     # → 0           (CLOSED - the assumption was discharged)
showTheorem(t)   # → "  ((A   B) → A)"
```

The step builders cover the whole calculus: `assume` / `byAxiom` / `refl`; `mp` / `impI`; `andI` / `andEL` / `andER`; `orIL` / `orIR` / `orE`; `notE` / `notI` / `raa` / `falseE`; `ui` / `exI` / `allI` (with the eigenvariable proviso); and `eqSubst` / `iffI` / `iffEL` / `iffER`. Inspect a result with `proved`, `conclusion`, `hyps`, `whyRejected`, and `showTheorem`.

A proof that leaves an assumption undischarged certifies as a **conditional** theorem with that assumption in Γ (`A  A`, not a dishonest `A`). A non-proof returns a legible reason, never a crash:

```
bad: certify([ assume(keq(kv("a"), kv("b"))) ], keq(kv("a"), kv("c")), [])
proved(bad)       # → false
whyRejected(bad)  # → "concludes a = b, not the goal a = c"
```

And `proved` recognises a real theorem by its Go type, so a hand-written look-alike hash is rejected — `certify` is the *only* thing that mints one.

A companion module, `lib/proof/lemmas.ax`, proves eleven canonical theorems once at import and exports them as ready-made sealed values (identity, / commutativity, excluded middle, double-negation elimination, K, hypothetical syllogism, contraposition, = reflexivity/symmetry/transitivity) plus a `provenLemmas` catalog you can cite as axioms (each is a proven closed theorem = a derived rule):

```
import "proof"
import "proof.lemmas"
showTheorem(lemDne)     # → "  (¬¬A → A)"
showTheorem(lemContra)  # → "  ((A → B) → (¬B → ¬A))"
```

Scope. The kernel proves concrete *instances* over named atoms/terms (not schemata), is classical, and its trusted base is the whole Go interpreter — ideal for pedagogy, exploration, and integration with the grounding ladder, but not a high-assurance substitute for Coq/Lean. The mitigation (export a proof and re-check it with an independent verifier) is built. Full guide and reference: `lib/proof/README.md`; architecture, the trust-boundary argument, and the A0–A6 roadmap: `resources/docs/claude/PROOF_ASSISTANT_DESIGN.md`. Worked tests: `tests/axioma/proof/test_lib_proof_v1.ax`, `test_lib_proof_lemmas_v1.ax`.

1.33 32. Reference

1.33.1 Keyword index

Keyword	Group
`:` (bind), `lambda`, `func`, `fn`, `define`	Bindings & functions (bind a value with `:`)
`if`, `then`, `else`	Control flow
`true`, `false`, `none`, `null`, `om`, `Ω`	Literals
`and`, `or`, `not`, `implies`, `iff`	Logic
`forall`, `exists`, `in`	Quantifiers
`union`, `intersect`, `difference`, `subset`	Set ops
`concept`, `has`, `extends`, `delete`, `is`, `is`	Concept system (creation uses `concept` only; `exists` is reserved for the existential quantifier)
`axiom`, `postulate`	Knowledge tiers
`insert`, `forget`, `retract`, `cancel`, `uncancel`	Mutation (relation named by string)
`transaction_begin/commit/rollback`	Transactions
`<=` (`:-`), `<~~`, `==>`, `~~>`	Rules (strict / defeasible × backward / forward)

Keyword	Group
`grounding`, `truth_kind`, `why`, `proof`, `rules_of`, `challenge`, `challenged`, `canceled`	Provenance & introspection
`try`, `attempt`, `otherwise`, `or else`, `error`, `raise`, `error?`	Errors as values (§29)
`understand`, `examine`, `abduce`, `[ model | … ]`	Cognitive kernel (§30)
`kleene`, `belnap`, `lukasiewicz`, `intuit3`	Multi-valued-logic value constructors
`necessarily`, `possibly`	Modal
`always`, `eventually`, `next`, `until`	Temporal
`knows`, `believes`, `common_knowledge`	Epistemic
`obligatory`, `permitted`, `forbidden`	Deontic
`dup`, `swap`, `rot`, `over`, `drop`, `nip`, `tuck`, `stacklength`, `erase`	Stack
`is/same`, `is/identical`, `is/property`	Russell's copula
`venn`, `fullform`, `treeform`, `tableform`, `graphform`	Visualization & the `*form` family

1.33.2 Refinement table

Form	Effect
`declare/persist x = v`	Save to `.axioma_session.bin` (use `=`, not `:`)
`declare/transient x = v`	Discard at session end
`axiom/persist`	Save to `cascade.db`
`axiom/transient`	Session-only axiom
`postulate/persist`	Save to `cascade.db`
`postulate/transient`	Session-only postulate

1.33.3 Tag-filter values for comprehensions

`@axiom`, `@postulate`, `@theorem`, `@conjecture`, `@hypothesis`, `@datum`, `@canceled`, `@all`, `@*`

1.33.4 File locations

- **Knowledge base** — `cascade.db` (project) or `~/.axioma/axioma.kb` (MCP server)
- **Session state** — `.axioma_session.json`
- **Diagrams** — `diagrams/venn_diagram_TIMESTAMP.png`
- **Logs** — `~/.axioma/mcp.log`
- **PID** — `~/.axioma/mcp.pid`
- **Examples** — `tests/axioma/**/*.ax`, `scripts/examples/`

1.33.5 Error messages

Error	Cause	Fix
`identifier not found: X`	Undefined variable	Define or check spelling
`Parser errors: ...`	Syntax error	Check parentheses, brackets, operators
`wrong number of arguments`	Arity mismatch	Check function signature
`type mismatch`	Incompatible types	Convert or check operand types
`division by zero`	`n / 0`	Check denominator
`cardinality violation: <C>.<p>`	Slot over-assignment	Last-write-wins; check `_cardviolation_*`

1.33.6 See also

- `Axioma.md` — feature index and quick reference
- `Axioma Elements.md` — high-level element-group map
- `CLAUDE.md` — design philosophy and contributor guidelines
- `docs/f-logic-unification.md` — unified frame-logic / bilattice / G3 design doc
- `resources/docs/claude/` — deep-dives on persistence, postulate/axiom, null types, SETL features, modal logic

---