Constructing Programming Languages from SpherePOP Primitives

A Graduate Text on Executable Prototypes:
Stacks, Lisp, Modules, BASIC, Shells, and Package Indexes

Flyxion
Independent Researcher

November 2026

Contents

Preface **1**

I Foundations **3**

1 The Reference Kernel **4**

1.1 Worlds, Spheres, and Four Primitives 4

1.2 Views and the Two Kinds of Collapse 6

1.3 Four Laws 6

1.4 Evidence and Acceptance 7

1.5 Derived Forms 8

1.6 Structural Vocabulary 9

1.7 What the Model Leaves Out 9

2 A Method for Reconstructing Languages **10**

2.1 Five Decisions 10

2.2 Adequacy Is Relative to the Observation 11

2.3 The Cost of Keeping Everything 11

2.4 Tests as Part of the Text 11

3 Register Transfers and the Clock Boundary **12**

3.1 Register Transfers 12

3.2 The Boundary in SpherePOP 13

3.3 Selection as Choice 16

3.4 Why This Chapter Precedes Forth 16

II Language Prototypes **18**

4 Forth: Programs as Histories of a Stack **19**

4.1 The Instruction Boundary 19

4.2 Operand Order and the Identity of a Copy 19

4.3 Control as Choice, Calls as Frames 21

4.4 Definitions, Shadowing, and When Names Are Resolved 22

4.5 Errors as Documented Refusals 22

4.6 Adequacy at the Instruction Boundary 23

5 A Lisp Core: Environments, Closures, and Tail Calls **24**

5.1 What Must Be Reconstructed 24

5.2 Environments and Mutation by Supersession 24

5.3 The Evaluator and Tail Calls 26

5.4 Macros as Ledger Rewriting 27

5.5 A Prelude Written in the Language 28

6 A Racket Layer: Modules, Provision, and Requirement **30**

6.1 From a Language to a Language of Modules 30

6.2 Declaration, Instantiation, and Lazy Loading 30

6.3 Conflicts and Staleness 33

7 Domain-Specific Languages by Module Language and by Macro **34**

7.1 Two Mechanisms for One Goal 34

7.2 A DSL by Module Language 34

7.3 A DSL by Macro 35

7.4 Why This Is the Package-Index Question in Miniature 35

8 Commodore BASIC: A Program Store and a Disk **37**

8.1 The Machine Being Modeled 37

8.2 Lines as Spheres, Editing as Supersession 37

8.3 Control Flow and the Interpreter Stacks 38

8.4 LOAD as the Central Operation 39

9 Shells as Loading Languages **42**

9.1 What a Shell Does . . . 42

9.2 Bash: Alias, Builtin, Hash, Path . . . 43

9.3 Windows Command Interpreter . . . 45

9.4 WSL and the Two Namespaces . . . 45

9.5 Ubuntu and the Command-Not-Found Hook . . . 46

9.6 BusyBox and the Multi-Call Binary . . . 46

10 Package Indexes and Search Paths **48**

10.1 Three Problems in One Word . . . 48

10.2 Flat Resolution with Backtracking . . . 48

10.3 Nested Resolution and Lexical Scope . . . 49

10.4 Uninstall, Upgrade, and Lock . . . 50

10.5 The Search Path and the Connection to Racket . . . 51

III Composition **53**

11 Composing Languages in One Ledger **54**

11.1 The Claim . . . 54

11.2 A Collision the Test Found . . . 54

11.3 Values Across Languages . . . 55

11.4 What Composition Does Not Show . . . 55

12 Comparative Structural Analysis and Open Problems **57**

12.1 A Measured Profile . . . 57

12.2 Weakening, Contraction, and Exchange . . . 58

12.3 The Cost of History . . . 58

12.4 What Has and Has Not Been Shown . . . 59

12.5 Open Problems . . . 59

A The Reference Kernel and the Search Function **61**

B Reproducing the Results **66**

Preface

A programming language can be studied by reconstructing a fragment of it in another calculus. The reconstruction begins with a choice about what the fragment does and which aspects of its execution must survive translation. For a stack language, the final stack may suffice to compare arithmetic programs. Once the language has named definitions, a reconstruction may also need to preserve which definition each call resolved to. For a language that touches its environment, the order and results of those interactions may be part of the semantics. Each prototype in this book therefore begins with an account of what a program can do, proceeds to an implementation, and ends by checking what the implementation preserves.

SpherePOP gives these reconstructions a common form. Execution produces a history from which a current state can be recovered. A stack, an environment, a program store, a search path, or a set of installed packages is a view of that history. Two programs can then be compared at the level of their results while the evidence needed to distinguish their executions is retained. The claim that one language reconstructs another acquires a concrete meaning: the translation can be exhibited and run, and the observations under which it agrees with the source language can be stated and tested.

The book has three parts. Part I fixes a small reference kernel with four primitives, states the method used for every later prototype, and works the method on a four-bit shift register, a machine small enough that every transition can be checked. Part II builds working prototypes: a Forth, a lexically scoped Lisp, a Racket-style module layer with require and provide, domain-specific languages made from module languages and macros, Commodore-style BASIC with its program store and disk `LOAD`, several shell dialects treated as loading languages, and package indexes with flat and nested resolution. Part III shows that the prototypes can share one ledger without disturbing each other, compares their structural profiles, and states open problems.

A thread runs through Part II and explains its order. Racket, BASIC's `LOAD`, the shell's `PATH`, and a package manager are four answers to one question: given a name, which of several ordered places supplies the thing it denotes, and what happens to the program once it is found? The prototypes make that shared structure explicit, because each of them implements it with the same ordered-search routine and records every probe as a sphere. A miss becomes a documented refusal and the hit becomes a committed pop.

The method extends an earlier treatment of a Forth stack [20] and uses the four primitives as specified in the SpherePOP specification [21].

Every listing in this book is cut from source files that accompany it, and every transcript is the output of a script that runs those files. The accompanying test suite contains seventy-six tests. Statements about behavior in Chapters 4 through 10 refer to the prototypes, and several details of real systems are deliberately simplified; each chapter says where. Where a statement concerns a real system, such as the order in which a shell resolves a command name, the prototype is a model of the documented rule and not a copy of any implementation.

Notation is fixed once. The four primitives are Pop, Refuse, Bind, and Collapse; a world is a pair (H, Ω) of a history and a family of live spheres; and a language fragment is written by the name of its tag prefix, such as `forth.out`.

Part I

Foundations

Chapter 1

The Reference Kernel

1.1 Worlds, Spheres, and Four Primitives

The kernel used throughout this book is a concrete model of four primitives. It is one model among several that satisfy the same laws, and it is chosen because it is small enough to read in full (Appendix A lists it) while being strict enough that a prototype cannot cheat. A *world* is a pair (H, Ω). The history H is an append-only sequence of events, and Ω is the family of spheres that are currently live. A *sphere* is an immutable record of a birth: an integer identity, a tag naming its kind, a key used for lookup, a payload, and a tuple of dependencies, which are earlier spheres that the new sphere is coupled to. A sphere is always in exactly one of three states, live, popped, or refused, and the transition out of the live state is irreversible.

Definition 1.1 (Primitives)**.** Let $w = (H, \Omega)$ be a world. Bind(τ, k, p, D) appends a birth event and adds a new live sphere with tag τ, key k, payload p, and dependencies D, provided every member of D was born earlier; it consumes nothing. Pop(s) appends a pop event and removes the live sphere s from Ω, committing it; it is admissible only if s is live. Refuse(s, r) appends a refusal event and removes the live sphere s from Ω while recording a nonempty reason r; the sphere and its payload remain in H. Collapse(ρ) evaluates a rule ρ on a read-only view of w and appends a record containing the rule's name and a digest of the value it returned; it changes no sphere.

The sentence that introduces a sphere deserves care, because earlier drafts of the SpherePOP literature used the word pop for object introduction. In this kernel, introduction is a Bind into an open scope, which is the canonical encoding of abstraction, Sphere := Bind(x, Ω). Pop is reserved for commitment of something already introduced. The distinction matters in every chapter that follows: a Forth literal, a Lisp binding, a BASIC program line, and a shell file are all introduced by Bind, and what happens to them afterward is a question of which of the other three primitives acts on them.

The primitives and their admissibility test are short enough to quote.

```
# the four primitives
```

Primitive	Admissible when	Effect on Ω	Appended to H
Bind	dependencies are born	one live sphere added	birth with payload
Pop	sphere is live	one live sphere removed	commitment
Refuse	sphere is live, reason nonempty	one live sphere removed	exclusion with reason
Collapse	always	none	rule name and digest

Table 1.1: The four primitives of the reference kernel.

```
def bind(self, tag, key=None, payload=None, deps=()):
    sid = len(self.spheres)
    self._commit(("bind", sid, tag, key, payload, tuple(deps)))
    return sid

def pop(self, sid):
    self._commit(("pop", sid))
    return self.spheres[sid].payload

def refuse(self, sid, reason):
    self._commit(("refuse", sid, reason))

def collapse(self, rule, name=None):
    value = rule(self.view)
    self._commit(("collapse", name or getattr(rule, "__name__", "rule"), digest(value)))
    return value
```

```
def _commit(self, e):
    k = e[0]
    if k == "bind":
        if e[1] != len(self.spheres):
            raise Inadmissible("sphere id out of order")
        for d in e[5]:
            if d not in self.spheres:
                raise Inadmissible("unborn dependency %r" % d)
    elif k in ("pop", "refuse"):
        if self.status.get(e[1]) != LIVE:
            raise Inadmissible("%s of non-live sphere %r" % (k, e[1]))
        if k == "refuse" and not (isinstance(e[2], str) and e[2]):
            raise Inadmissible("refusal requires a reason")
    elif k != "collapse":
        raise Inadmissible("unknown event %r" % (k,))
    self.ledger.append(e)
    if k == "bind":
        s = Sphere(e[1], e[2], e[3], e[4], e[5])
        self.spheres[s.id] = s
        self.status[s.id] = LIVE
        self._tag.setdefault(s.tag, {})[s.id] = None
        self._key.setdefault((s.tag, s.key), {})[s.id] = None
    elif k in ("pop", "refuse"):
        s = self.spheres[e[1]]
        self.status[s.id] = POPPED if k == "pop" else REFUSED
        if k == "refuse":
```

```
            self.reasons[s.id] = e[2]
        del self._tag[s.tag][s.id]
        kk = (s.tag, s.key)
        del self._key[kk][s.id]
        if not self._key[kk]:
            del self._key[kk]
```

1.2 Views and the Two Kinds of Collapse

State is never stored beside the ledger. Everything a language wants to know about its own state is obtained through a *view*, a read-only projection that offers the live spheres of a tag, the latest live sphere with a given key, counts, payloads, and the transitive dependencies of a sphere. A view is the pure part of Collapse. The kernel distinguishes two uses. Reading through a view in the middle of a step is unrecorded, because it belongs to the step that performs it. Committing an observation, such as the stack at the end of a Forth line or the variables at the end of a BASIC run, is a recorded Collapse that leaves a digest in the ledger. This is a modeling choice, and it has a cost and a benefit. Recording every read would make the ledger several times longer without adding information that replay does not already supply; recording commitments gives later checkers a place to compare the value a run reported against the value that replay reconstructs.

1.3 Four Laws

Proposition 1.2 (History grows)**.** *Every primitive appends exactly one event to H. No event is ever modified or removed.*

Proof. Each primitive calls the single commit routine, which validates the proposed event and then appends it. The ledger is only ever extended by that routine, and the sphere records are immutable. Hence the length of H increases by one with each successful primitive. □

Proposition 1.3 (Live accounting)**.** *Let $\phi(w)$ be the number of live spheres of w. Then* Bind *increases ϕ by one,* Pop *and* Refuse *each decrease it by one, and* Collapse *leaves it unchanged.*

Proof. Immediate from the table above. The indexes that answer view queries are updated by the same routine that appends the event, and each update adds or removes exactly the one sphere the event names. □

Proposition 1.4 (Replay faithfulness)**.** *Let w be a world with ledger H and let w' be obtained by committing the events of H, in order, to an empty world. Then w' has the same sphere statuses, the same refusal reasons, and the same live indexes as w. If a recorded* Collapse *names a rule available to the checker, then the rule evaluated on the replayed prefix reproduces the recorded digest.*

Proof. By induction on the length of H. The empty ledger yields the empty world. Suppose the claim holds for a prefix. The next event is admissible in w, because it was committed there, and the admissibility test depends only on the statuses and births recorded in the prefix, which agree by hypothesis. The state transition performed for an event is a function of the event and the prefix state alone. Hence the two worlds agree after the event. For a Collapse event, the view available to the rule is a function of the prefix state, which agrees, so the digest agrees. □

Proposition 1.5 (Independent events commute)**.** *If two events name disjoint spheres, then committing them in either order yields worlds that agree on every sphere status and on every live index up to the order of insertion.*

Proof. An event changes the status of exactly the spheres it names and updates the indexes entries of those spheres only. When the named sets are disjoint, the two updates touch different entries, and admissibility of each is unaffected by the other. □

Proposition 1.5 is the content of the parallel composition $\mathrm{Pop}(A) \parallel \mathrm{Refuse}(B)$ used for choice below. It does not say that the two ledgers are equal, since events appear in different orders, and the kernel does not pretend otherwise. The ledger digest of a world is a hash chain over the events in order, so reordering changes it, and a checker that wishes to identify the two runs must compare projections, not chains.

```
@classmethod
def replay(cls, events, rules=None):
    w = cls()
    for e in events:
        if e[0] == "collapse" and rules and e[1] in rules:
            if digest(rules[e[1]](w.view)) != e[2]:
                raise Inadmissible("collapse digest mismatch for %s" % e[1])
        w._commit(e)
    return w

def verify(self, rules=None, expected_chain=None):
    w2 = World.replay(self.ledger, rules)
    ok = (w2.status == self.status and w2.reasons == self.reasons
          and {t: list(d) for t, d in w2._tag.items()} == {t: list(d) for t, d in self._tag.
    items()})
    if expected_chain is not None:
        ok = ok and w2.chain() == expected_chain
    if not ok:
        raise Inadmissible("replay disagrees with indexed state")
    return True
```

1.4 Evidence and Acceptance

A run produces two things, a result and a ledger, and a consumer should be able to accept a result without trusting the process that produced it. The kernel supports this separation.

The *chain* of a world is a hash chain over its events. `verify` replays the ledger into a fresh world, compares the statuses, reasons, and indexes with those of the original, and optionally compares the chain with an expected value and recomputes recorded collapses with supplied rules. If any event of a stored ledger is altered, the chain changes and the comparison fails, and if the altered ledger is replayed without the expected chain, replay may still succeed and produce a different world, which is why the expected chain is part of the acceptance test and not an optional extra. Matching final answers do not establish validity: two different ledgers can end in the same observable, and the tests in this book exercise exactly that situation.

1.5 Derived Forms

Four small functions recur in every prototype. Supersession realizes mutation: the old sphere is refused with a reason and a new sphere with the same tag and key is bound with the old one as its dependency, so that the lineage of a mutable cell is the chain of its dependencies and no past value is lost. Choice realizes $\mathrm{Pop}(A) \parallel \mathrm{Refuse}(B)$, in which the taken branch is committed and every rejected branch is refused with a recorded reason. A documented refusal of an inadmissible proposal binds an attempt sphere and refuses it immediately, so that an error leaves a trace of what was tried and why it was turned away. Application is $\mathrm{Collapse}_{\rho}(\mathrm{Bind}(f, x))$: a coupling of function and argument, followed by a rule that reads both.

```
# derived forms
def supersede(w, old, payload, reason="superseded"):
    """Mutation as documented refusal of the old sphere plus a coupled new one."""
    s = w.spheres[old]
    w.refuse(old, reason)
    return w.bind(s.tag, s.key, payload, (old,))


def choose(w, taken, rejected, reason="not-taken"):
    """Choice(A,B) = Pop(A) parallel Refuse(B)."""
    w.pop(taken)
    for r in rejected:
        w.refuse(r, reason)


def attempt_refused(w, tag, payload, reason):
    """Documented refusal of an inadmissible proposal."""
    i = w.bind(tag, None, payload)
    w.refuse(i, reason)
    return i
```

1.6 Structural Vocabulary

The primitives suggest a reading of the structural rules of substructural logic that the later chapters use as an audit device. A Refuse of a value that was introduced but never used is a weakening, because something that entered the world is discarded without contributing to a result. A Bind that couples a new sphere to an existing one while leaving the existing one live is a contraction when two later spheres depend on the same source, since the source is used twice. A reordering of live spheres is an exchange. The vocabulary is used descriptively in this book. Which operations of a language weaken, contract, or exchange is a fact about its prototype that can be counted in the ledger, and Chapter 12 reports such counts. Whether a given fragment satisfies a precise substructural discipline is a theorem that belongs to the calculus and not to the prototype; the essays on scoped structural rules and on duplication and universality [22, 23] develop that side.

1.7 What the Model Leaves Out

Three simplifications deserve statement. First, Bind here introduces a sphere, whereas in the abstract calculus it is an endomorphism that narrows an admissible family; the dependency tuple carried by each sphere is the concrete stand-in for that narrowing, and no prototype relies on more. Second, identity is by integer, and the kernel has no notion of a scope boundary beyond the tags that languages choose; Chapter 11 shows why tags must therefore be namespaced by language. Third, nothing is collected: refused and popped spheres remain in the ledger forever, and live spheres that have become unreachable stay live unless a language refuses them explicitly. The price of keeping every execution distinguishable is memory, and Chapter 12 measures it.

Exercise 1.1. Add a gate to the world, a function that sees a proposed event and returns allow or a reason for refusal, and route every commit through it. Decide whether a refused proposal should itself be recorded and what the recorded form is.

Exercise 1.2. Prove that if a ledger is replayed with an event inserted at the front that names no existing sphere, the replayed statuses of all later events are unchanged except for the renumbering of sphere identities. State what must be assumed about how the kernel assigns identities.

Exercise 1.3. Show that Proposition 1.5 fails if two events name the same sphere, and give the smallest example.

Chapter 2

A Method for Reconstructing Languages

2.1 Five Decisions

Every prototype in this book is built by the same five decisions, taken in order. The first fixes the state of the source machine and assigns each component a sphere tag, namespaced by the language so that two languages sharing a ledger cannot collide. The second names the observables, which are the quantities an outside party may read when a step completes, and writes each as a collapse rule, a pure function of a view. The third defines each operation of the language as a composition of the four primitives, together with the condition under which it is admissible. The fourth specifies what happens when the condition fails: the operation is refused, the refusal is documented with a reason, and the interpreter cleans up the partial state it created by refusing it in turn. The fifth states and tests adequacy.

Definition 2.1 (Observation boundary)**.** An *observation boundary* of a source machine is a point at which the machine's state is considered complete and may be read: the end of a Forth word, a clock edge of a register, the end of a Lisp form, the end of a BASIC statement, the end of a shell command. A reconstruction is *adequate at a boundary* when a projection π of the SpherePOP history to the boundary yields the source machine's state at that boundary.

The central invariant of every chapter has the same form. If S_i is the source state after i completed steps, H_i is the history after the same steps, and π is the projection that reads the observables, then $\pi(H_i) = S_i$ for every i. The invariant is checked in two ways. The first compares the running prototype with an independent reference implementation of the same source semantics, written separately and as plainly as possible, over many programs; this is differential testing, and Chapter 4 does it for Forth with randomly generated programs. The second replays the recorded history in a fresh world and checks that the observable agrees, which tests the claim that nothing the language needed was kept outside the ledger.

2.2 Adequacy Is Relative to the Observation

An adequacy claim has no content until the observation is named. Two programs that produce the same stack are indistinguishable by the stack projection and distinguishable by the provenance projection, which reads, for each stack position, the operations that produced it. The first projection is the right one for deciding whether a compiler preserved arithmetic. The second is the right one for explaining a result, and it is available only because the reconstruction kept dependencies. The distinction appears again for module imports, where the value of an identifier does not say which module supplied it, and for package installs, where a version number does not say why a package is present.

2.3 The Cost of Keeping Everything

The price of the method is the length of the ledger. Chapter 12 gives a measured profile for one workload per prototype. The relevant general remark is that a prototype can be made cheaper by recording less, and that each saving has an identifiable cost in what can no longer be answered. A prototype that records only commitments cannot explain an intermediate value. A prototype that refuses unreachable spheres can answer questions about what is currently reachable and still answer questions about what once existed. The choice is part of the design and should be made explicitly.

2.4 Tests as Part of the Text

Every prototype ships with a test suite, and the tests are part of the specification. A test that asserts a refusal reason, for example that the second `ls` in a shell session performs no probes because the first result was hashed, states a semantic fact that prose could leave vague. The suites for the prototypes in this book total seventy-six tests, including exhaustive enumeration for the register, randomized differential testing for Forth, and a composition test that runs five languages in one ledger.

Exercise 2.1. Choose a small language not treated here, such as a Brainfuck variant, and carry out the five decisions on paper before writing code. For each observable, state the projection and the boundary.

Exercise 2.2. Give two source programs of a language of your choice that agree on every observable of one projection and disagree on another, and say what the second projection must read.

Chapter 3

Register Transfers and the Clock Boundary

A four-bit shift register is small enough that every possible transition can be checked, yet it poses the problem that every later reconstruction faces. The source machine must be specified, a choice must be made about what counts as an observable step, and a SpherePOP history must be shown to reproduce that step. Figure 3.1 gives the rule for selecting an update. Figure 3.2 gives the rule for applying it.

3.1 Register Transfers

A register-transfer description specifies what a machine will store at the next clock edge as a function of its current contents and its inputs. In Figure 3.1, several sources supply candidate values for R_0. The control signals determine which candidate is selected and whether the register loads at all. If every control signal is zero, the register holds its contents. The circuit assumes that at most one source-select signal is asserted. In the prototype, simultaneous assertions therefore produce an explicit control error. That is a choice about how to report a violation of the circuit's assumptions. A priority encoder would give a different machine, with a defined result for some of the same input combinations.

In Figure 3.2, write the four stored bits from left to right as $q = (q_0, q_1, q_2, q_3)$. If the serial input is b, then a clock edge gives

$$(q_0, q_1, q_2, q_3) \longmapsto (b, q_0, q_1, q_2).$$

Each stage samples its input from the same pre-edge state. For example, shifting a 0 into 1011 leaves 0101 in the register. The bit shifted out is the old $q_3 = 1$, and the serial output pin after the edge reflects the new $q_3 = 1$. Those two observations happen to agree in this example. With 1010 as the initial word and the same input, the next word is still 0101, but the bit shifted out is 0. A prototype that reports a shifted-out bit needs to say which side of the edge it samples.

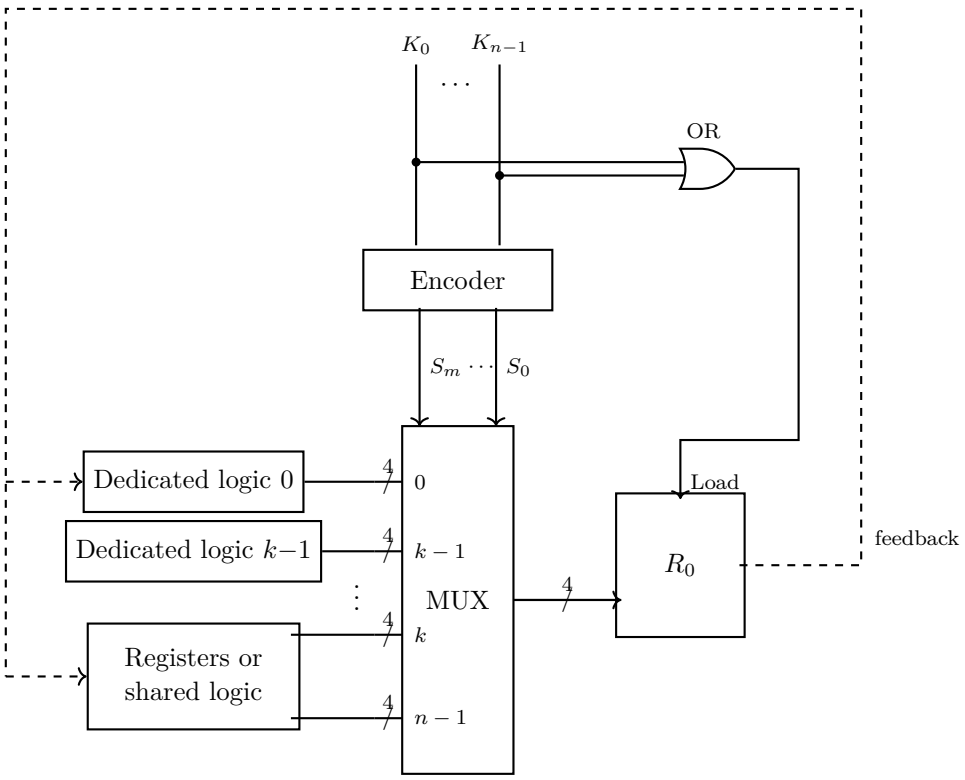


Figure 3.1: Multiplexer selection for n sources. The encoder converts the control lines $K_0, \ldots, K_{n-1}$ into select inputs for the multiplexer, and the OR of the same lines drives the load input of R_0. Slashes mark four-bit buses, and the dashed path returns the register contents to the source logic.

3.2 The Boundary in SpherePOP

For a SpherePOP reconstruction, the frontier contains four ordered references, one per stage. In the reference kernel the sampled input occurrence is introduced by a Bind into the open scope and committed at the edge by a Pop, and each destination stage is a Bind coupled to the occurrence it samples. The four old stage occurrences are popped, because the edge consumes them, and Collapse publishes the next register word at the clock boundary. The history also retains the old rightmost occurrence, in a separate sphere of kind `shifted-out`, as evidence of the bit shifted out. This is a proposed encoding for the chapter. Its central implementation requirement is that a reader observing the register at a clock boundary sees a complete word. If an implementation instead reads each stage from a frontier it has already started updating, the new input can propagate farther than one stage in a single simulated edge.

```
def edge(self, b):
    """One clock edge. Every stage samples the same pre-edge word."""
    w, v = self.w, self.w.view
    old_ids = [v.last_key("stage", k) for k in range(self.n)]
    old = [v.payload(i) for i in old_ids]
    inp = w.bind("input", None, b)
```

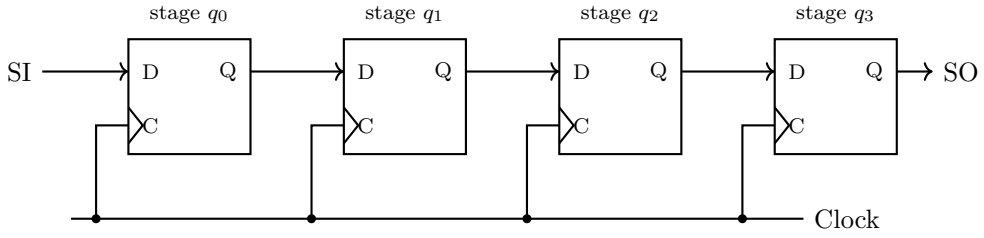


Figure 3.2: Four-bit shift register, detailed form. The upper chain carries each bit to the next stage. The lower line reaches all four clock inputs in parallel, so each stage samples its own input at the same edge.



Figure 3.3: Shift-register symbol. The block exposes the same three external connections as the detailed form.

```
    w.bind("shifted-out", None, old[-1], (old_ids[-1],))
    for i in old_ids:
        w.pop(i)
    for k in range(self.n):
        src = inp if k == 0 else old_ids[k - 1]
        w.bind("stage", k, b if k == 0 else old[k - 1], (src,))
    w.pop(inp)
    w.collapse(word_rule, "word")
    return self
```

The faulty variant makes that failure concrete. Each stage reads the frontier that earlier stages have already rewritten.

```
def edge(self, b):
    w, v = self.w, self.w.view
    prev = b
    for k in range(self.n):
        cur = v.last_key("stage", k)
        old = v.payload(cur)
        w.pop(cur)
        w.bind("stage", k, prev if k == 0 else v.payload(v.last_key("stage", k - 1)), ())
        prev = old
    w.bind("shifted-out", None, 0)
    return self
```

Proposition 3.1 (Adequacy at the clock boundary)**.** *For every register word $q \in \{0,1\}^4$ and input $b \in \{0,1\}$, the word published by* ***`ShiftRegister.edge`*** *equals (b, q_0, q_1, q_2), the sphere of kind* ***`shifted-out`*** *carries q_3, and replaying the recorded history yields the same word and the same shifted-out bit.*

Proof. The old word is read once, through a single view, before any event of the edge is committed, so q is fixed. The new stage k is bound with payload b when $k = 0$ and with payload q_{k-1} otherwise, both taken from that fixed read. No payload depends on a sphere created during the edge. The `shifted-out` sphere is bound with payload q_3. After the four pops and four binds exactly four stage spheres are live, one per key, so the view sorted by key returns (b, q_0, q_1, q_2). Replay faithfulness (Proposition 1.4) yields the second claim. □

The first exercise of the book verifies the encoding exhaustively. There are 16 possible register words and 2 possible input bits. For each of those 32 cases the test checks the next word and the old rightmost bit against the equations, then replays the recorded history and checks both observations again. Table 3.1 lists all cases as produced by the prototype.

Initial word	Next, input 0	Out	Next, input 1	Out
0000	0000	0	1000	0
0001	0000	1	1000	1
0010	0001	0	1001	0
0011	0001	1	1001	1
0100	0010	0	1010	0
0101	0010	1	1010	1
0110	0011	0	1011	0
0111	0011	1	1011	1
1000	0100	0	1100	0
1001	0100	1	1100	1
1010	0101	0	1101	0
1011	0101	1	1101	1
1100	0110	0	1110	0
1101	0110	1	1110	1
1110	0111	0	1111	0
1111	0111	1	1111	1

Table 3.1: All thirty-two transitions of the four-bit shift register, generated by the prototype. The column Out is the old q_3.

Two short counterexamples make clear what those tests protect. The initial state 1000 with input 0 must become 0100, which exposes accidental propagation through multiple stages. The initial states 1010 and 1011 with input 0 both become 0101, which exposes any attempt to recover the shifted-out bit from the next word alone. The faulty ripple variant above fails the exhaustive test on 28 of the 32 cases and turns 1000 into 0000, so the test does what it is supposed to do. The ledger distinguishes the 1010 and 1011 runs even though their published words are equal, since their chains differ.

3.3 Selection as Choice

The transfer register of Figure 3.1 is the same method applied to selection. Each source supplies a candidate sphere coupled to the current contents of R_0. When the control lines select source k, the candidate k is popped and every other candidate is refused with the reason `not-selected`, which is exactly Pop(A) $\|$ Refuse(B) generalized to n branches. When no control line is asserted the register holds, because the load signal, the OR of the controls, is zero. When more than one control line is asserted the clock is refused: an attempt sphere is bound and refused with the reason `multiple-select`, an error is raised, and the register is unchanged.

```
def clock(self, sources, controls):
    w = self.w
    if len(sources) != len(controls):
        raise ValueError("one control per source")
    if sum(controls) > 1:
        attempt_refused(w, "attempt", tuple(controls), "multiple-select")
        raise ControlError("more than one source-select asserted: %r" % (tuple(controls),))
    if not any(controls):
        return self.value()          # Load = OR(K) = 0: hold
    cands = [w.bind("candidate", k, s, (self.r,)) for k, s in enumerate(sources)]
    k = controls.index(1)
    choose(w, cands[k], [c for j, c in enumerate(cands) if j != k], "not-selected")
    new = w.bind("reg", 0, w.spheres[cands[k]].payload, (cands[k],))
    w.refuse(self.r, "loaded")
    self.r = new
    w.collapse(lambda v: v.payload(v.last_key("reg", 0)), "R0")
    return self.value()
```

With three sources there are eight control combinations. One holds, three select a single source, and four assert more than one line; the test enumerates all eight and checks that the four refused combinations leave the register untouched and are recorded in the ledger.

```
start  1011
edge 0 0101  out 1
edge 0 0010  out 1
edge 1 1001  out 0
ripple fault on 1000 with 0: 0000 (expected 0100)
transfer: select 1 of [5,9,12] -> 9
transfer: no select holds -> 9
transfer: more than one source-select asserted: (1, 1, 0)
ledger events: {'bind': 22, 'collapse': 3, 'pop': 15}
```

3.4 Why This Chapter Precedes Forth

That last example gives the Forth chapter a precise starting point. A clock edge and a Forth instruction define different observation boundaries, but both require a rule about what becomes

visible when the step completes. The register test checks that the four new stage values come from one old word. The Forth test checks that a word such as `-` reads its two operands in stack order and publishes one new stack, with enough history to recover the operands that produced it. The shift register lets the reader verify that whole method on a machine whose transition table fits on one page.

Exercise 3.1. Extend the register to eight bits with a parallel-load control, so that the next word is either the shifted word, a supplied word, or the held word. Decide whether asserting shift and load together is a control error or has a defined priority, and encode the decision as a refusal or a choice accordingly.

Exercise 3.2. Add a reset that refuses all four stage spheres with the reason `reset` and binds four zeros. Show that the history still distinguishes a register that was reset from one that was shifted to zero.

Exercise 3.3. Define the projection that reads the shifted-out bits of the last m edges, and prove that the register word together with those bits determines the word m edges earlier.

Part II

Language Prototypes

Chapter 4

Forth: Programs as Histories of a Stack

4.1 The Instruction Boundary

A Forth program is a sequence of words, and each word is a function from stacks to stacks [1, 2, 3]. The semantics is therefore the simplest of any language in this book, and for that reason it is the place where the central invariant of the method can be stated without interference. Let S_i be the stack after i completed instructions of a program, let H_i be the SpherePOP history after the same instructions, and let π be the projection that reads the live cells in order of birth and returns their payloads. The claim to be established is $\pi(H_i) = S_i$ for every i. The claim is made at the instruction boundary and at no other point. In the middle of an instruction the history may contain a cell that has been born but not yet consumed, or a consumed operand whose result has not been bound, and the projection is not required to agree with anything there. This restriction is the same one that Chapter 3 places on a register, where the observable is the word between clock edges and not the wires inside an edge.

The state components are few. A `cell` sphere holds one stack entry, the live cells in order of birth are the stack, and a cell that has been consumed is refused with the name of the word that consumed it. A `word` sphere holds a definition, a `frame` sphere marks an active call, and `loop`, `iter`, and `branch` spheres carry control. Output is a `forth.out` sphere, a variable is a `forth.var` sphere whose cell contents are superseded on store, and a constant is a `const` sphere. Every tag is distinct from the tags of the other languages in this book, for reasons given in Chapter 11.

4.2 Operand Order and the Identity of a Copy

Two decisions in the Forth prototype are easy to get wrong and are worth stating as decisions. The first is operand order. Addition is commutative, so the stack 5 is produced equally well by consuming the two operands in either order, and a prototype that pops them in the wrong order

passes every test that uses only addition and multiplication. Subtraction, division, comparison, and the stack shuffles expose the error. The prototype therefore pops the top cell first, calls it the right operand, pops the next cell, calls it the left operand, and binds the result with the dependency tuple (left, right) in that order. The provenance of the result then reads in source order, so that the program `2 3 +` explains itself as $(2 + 3)$ and the program `7 2 -` as $(7 - 2)$.

```
ok> 2 3 +
    stack [5]
    explain: (2 + 3)
ok> drop 3 dup *
    stack [9]
    explain: (3 * 3)
ok> drop : sq dup * ; 7 sq
    stack [49]
ok> drop : cube dup sq * ; 3 cube
    stack [27]
ok> : sq drop 0 ; 3 cube
    stack [27, 27]
    with late binding the same cube now yields [0]
ok> : fact dup 1 > if dup 1- recurse * then ; drop drop 10 fact
    stack [3628800]
    output 'result 3628800 \n'
ok> + + +
    refused: +: stack-underflow ; stack after abort []
```

The second decision concerns `dup`. A Forth programmer may think of `dup` as producing two references to one value, or as producing a value and an independent copy of it. Both are consistent with the stack projection, and the choice is observable only through provenance. The prototype takes the second reading with a link. `dup` binds a new cell with the same payload as the top cell and records the top cell as its dependency, so the two cells are distinct spheres and each can be consumed independently, and the history still shows that one was derived from the other. The explanation function follows that dependency through `dup`, which is why 3 dup $*$ explains as $(3{*}3)$ and not as an expression over two unrelated literals. The alternative, in which `dup` binds nothing and the top cell is simply counted twice, makes consumption ambiguous, because the first `drop` after it would have to decide which occurrence disappears. The listing shows the stack words as they appear in the source.

```
elif name == "dup":
    self.need(1, name)
    t = v.last("cell"); self.push(v.payload(t), "dup", (t,))
elif name == "?dup":
    self.need(1, name)
    t = v.last("cell")
    if v.payload(t) != 0:
        self.push(v.payload(t), "dup", (t,))
elif name == "drop":
    self.need(1, name); w.refuse(v.last("cell"), "drop")
elif name == "nip":
    self.need(2, name); w.refuse(v.live("cell")[-2], "nip")
elif name == "swap":
```

```
    self.need(2, name)
    b, bi = self.pop1(); a, ai = self.pop1()
    self.push(b, "swap", (bi,)); self.push(a, "swap", (ai,))
elif name == "over":
    self.need(2, name)
    s = v.live("cell")[-2]; self.push(v.payload(s), "over", (s,))
elif name == "rot":
    self.need(3, name)
    c, ci = self.pop1(); b, bi = self.pop1(); a, ai = self.pop1()
    self.push(b, "rot", (bi,)); self.push(c, "rot", (ci,)); self.push(a, "rot", (ai,))
elif name == "-rot":
    self.need(3, name)
    c, ci = self.pop1(); b, bi = self.pop1(); a, ai = self.pop1()
    self.push(c, "rot", (ci,)); self.push(a, "rot", (ai,)); self.push(b, "rot", (bi,))
elif name == "tuck":
    self.need(2, name)
    b, bi = self.pop1(); a, ai = self.pop1()
    self.push(b, "tuck", (bi,)); self.push(a, "tuck", (ai,)); self.push(b, "tuck", (bi,))
```

4.3 Control as Choice, Calls as Frames

Conditionals are the first place where the primitives do real work. A word such as `if` consumes a flag, then binds both branches as `branch` spheres coupled to the flag, and then applies the derived form of Chapter 1: the taken branch is popped, the other is refused with the reason that it was not taken, and the taken branch is executed. The refused branch remains in the ledger with its payload, so the history records that an alternative existed. This is a use of $\mathrm{Pop}(A) \parallel \mathrm{Refuse}(B)$ in its original sense, and it is the reason that a reader of the ledger can answer the question of which branch a given run did not take.

```
elif k == "if":
    self.need(1, "if")
    flag, fid = self.pop1()
    a = w.bind("branch", "then", it[1], (fid,))
    b = w.bind("branch", "else", it[2], (fid,))
    taken, other = (a, b) if flag != 0 else (b, a)
    choose(w, taken, [other])
    self.run_items(w.spheres[taken].payload)
```

A call binds a `frame` sphere coupled to the word being called, runs the body, and pops the frame on return. The recursion word reads the most recent live frame to find the definition it belongs to, so it works from inside nested control structures without any extra bookkeeping. The factorial transcript above computes the factorial of ten through this mechanism and ends with two live spheres, the result and one remaining cell of the original stack, while ten spheres have been refused along the way. This is the profile row for Forth in Chapter 12.

```
def call(self, name):
    w = self.w
    wid = w.view.last_key("word", name)
```

```
    if name == "recurse":
        f = w.view.last("frame")
        if f is None:
            self.fault(name, "outside-definition")
        wid = w.spheres[f].deps[0]
    if wid is not None:
        self.call_word(wid)
    elif w.view.last_key("forth.var", name) is not None:
        self.push(("addr", name), "addr")
    elif w.view.last_key("const", name) is not None:
        c = w.view.last_key("const", name)
        self.push(w.spheres[c].payload, "const", (c,))
    else:
        self.builtin(name)
```

4.4 Definitions, Shadowing, and When Names Are Resolved

Redefinition raises a question that every language answers in some way: when a word is defined in terms of another word, and the other word is later redefined, does the first word see the new definition? Forth answers that a call inside a definition is fixed when the definition is compiled, so that redefining a word does not change the behavior of definitions that already use it. The prototype supports both answers as a flag. With early binding, which is the default, the compile step resolves every call in a body to the sphere of the word it named at that moment, and the old `word` sphere is refused with the reason that it was shadowed when a new definition arrives. With late binding, calls are resolved by name when they run. The transcript shows the same program under both flags: after `sq` is redefined to discard its argument, `cube` still computes twenty-seven under early binding, and computes the redefinition's result under late binding.

An unknown word inside a definition is a compile-time refusal under early binding, with the reason that the word is unknown, and a definition that fails to compile leaves no word sphere. The same program under late binding compiles and fails when it is run. A scholar who wishes to relate this to the lambda calculus can note that early binding corresponds to closing a term over its free names at the moment of abstraction, and late binding to leaving the names free until the term is applied. The ledger records the difference without argument, since under early binding the body of `cube` contains dependencies on a specific `word` sphere and under late binding it contains only a name.

4.5 Errors as Documented Refusals

A word that cannot run, such as `+` on an empty stack, is not allowed to leave a half-built state. The prototype records the proposal as an `attempt` sphere and refuses it at once with a reason, here `stack-underflow`, then abandons the rest of the line and refuses every live cell, frame, and loop sphere with the reason `abort`. The last transcript line shows the outcome for three `+`

words applied to an empty stack: the refusal is reported and the stack after the abort is empty. Clearing the data stack after an abort matches the behavior of many Forth systems at the level this book models, and the prototype differs from them in that the refused attempt, with its reason, remains available to later inspection, as does every cell that the abort discarded.

4.6 Adequacy at the Instruction Boundary

Proposition 4.1 (Forth adequacy)**.** *Let P be a straight-line Forth program over the stack words of the prototype, and let* Ref *be the reference stack machine that applies each word to a Python list. For every prefix P_i of P that ends at an instruction boundary, $\pi(H_i) = \mathrm{Ref}(P_i)$.*

Proof. By induction on i. For $i = 0$ both sides are the empty stack. Suppose the claim holds for i and let w be the $(i + 1)$-th word. Each word of the prototype refuses the cells it consumes and binds the cells it produces, in the order the reference machine removes and appends list entries. The live cells in order of birth are therefore the list after the step, because the cells that were not consumed keep their relative order and every produced cell is born after them. For `swap`, `rot`, and the other shuffles the produced cells are bound in the order the reference machine appends them, which is the point of the operand-order decision above. Hence $\pi(H_{i+1}) = \mathrm{Ref}(P_{i+1})$. □

The proof is short because the prototype was built to make it short, and a proof of this kind never replaces testing, since it covers the prototype as specified and not as written. The test suite closes that gap by generating two hundred random programs, running each through the prototype and through an independent implementation of the reference machine, and comparing the stack after every instruction. A discrepancy at any instruction fails the test, so the invariant is checked at every boundary and not merely at the end of a program.

Exercise 4.1. Add the words `pick` and `roll`, which take an index from the stack. State the provenance dependencies of the produced cells, and extend the random tester to cover them. Decide what `explain` should return for a `roll`.

Exercise 4.2. Under early binding, a word that refers to itself must be compiled before its own definition exists. The prototype handles this with a dedicated recursion word. Explain why a plain call to the defining name cannot be resolved by the early-binding rule, and describe an alternative that resolves the name against the sphere being defined.

Exercise 4.3. Replace the integer cells with cells that carry a type tag, and make `+` refuse a pair of cells with unequal tags. Identify which observables change and which proof steps of Proposition 4.1 remain valid.

Chapter 5

A Lisp Core: Environments, Closures, and Tail Calls

5.1 What Must Be Reconstructed

A Lisp interpreter in the tradition of McCarthy and Landin [4, 5, 6] has three pieces of state that a prototype must place somewhere: the bindings that map names to values, the closures that capture those bindings, and the activations that record where a computation will resume. A conventional interpreter stores all three in host data structures, with an association list or hash table for bindings, a record for a closure, and the host call stack for activations. The SpherePOP prototype stores all three in the ledger. A binding is a `binding` sphere with the variable as its key, its value as payload, and its environment frame as a dependency. A closure is a `lambda` sphere whose payload holds the parameters and body and whose dependency is the environment in which it was created. An activation is a sphere coupled to the activation that called it. Nothing else is needed, and in particular no host recursion is needed to represent a call, which matters for the next section.

5.2 Environments and Mutation by Supersession

Evaluating a symbol looks up the latest live `binding` sphere with that key in the nearest enclosing frame and then outward. Evaluating `define` binds a new sphere in the current frame. Evaluating `set!` on an existing variable is mutation, which the kernel does not have, and the prototype realizes it by the derived form of Chapter 1: the old binding is refused with the reason that it was superseded, and a new binding is bound with the old one as its dependency. The lineage of a variable is then the chain of dependencies from the live binding back to its first value, and the function `lineage` returns it.

```
def define(self, env, name, value):
    v = self.w.view
    if v.last_key("import", (env, name)) is not None:
```

```
        self.err("define: identifier already imported: %s" % name, "already-imported")
    i = v.last_key("binding", (env, name))
    if i is not None:
        return supersede(self.w, i, value)
    return self.w.bind("binding", (env, name), value, (env,))

def set_bang(self, env, name, value):
    o = self.owner(env, name)
    if o is None:
        self.err("set!: unbound identifier: %s" % name, "unbound")
    if o[1] != "binding":
        self.err("set!: cannot mutate imported identifier: %s" % name, "immutable-import")
    return supersede(self.w, o[2], value, "set!")

def lineage(self, env, name):
    """Values of a mutable binding, oldest first, from its supersession chain."""
    o = self.owner(env, name)
    out, i = [], o[2]
    while True:
        s = self.w.spheres[i]
        out.append(s.payload)
        if s.deps and self.w.spheres[s.deps[0]].tag == "binding" and self.w.spheres[s.deps
    [0]].key == s.key:
            i = s.deps[0]
        else:
            break
    return out[::-1]

def make_closure(self, params, body, env):
    return Closure(self.w.bind("lambda", None, (params, body), (env,)))
```

The counter in the transcript shows closure capture and mutation together. The call to `make-counter` creates a frame in which `n` is bound to zero and returns a closure whose environment is that frame. Each call to the closure increments `n` by supersession and returns the new value, and after three calls the lineage of `n` in the captured frame is the list zero, one, two, three. A reader accustomed to boxes and store-passing semantics will recognize that the ledger is a store that was never overwritten, and that the lineage is the history of a location in that store.

```
> (define (make-counter) (let ((n 0)) (lambda () (set! n (+ n 1)) n)))

> (define c (make-counter))

> (c)
1
> (c)
2
> (c)
3
> (map (lambda (x) (* x x)) '(1 2 3 4))
(1 4 9 16)
lineage of n in the captured frame: [0, 1, 2, 3]
(loop 3000 0) = 3000 ; live call spheres afterward: 0
```

5.3 The Evaluator and Tail Calls

The evaluator is a function on an expression and an environment that dispatches on the form of the expression, and it follows the usual special forms: `quote`, `if`, `cond`, `define`, `set!`, `lambda`, `let`, `begin`, and `define-syntax-rule`. The listing shows the dispatch.

```
def eval(self, x, env):
    couplings = []
    try:
        value = self._loop(x, env, couplings)
    except BaseException:
        for c in reversed(couplings):
            if self.w.status[c] == "live":
                self.w.refuse(c, "abort")
        raise
    for c in reversed(couplings):
        self.w.pop(c)
    return value
```

The treatment of a procedure call is the point at which the ledger-based design pays for itself. In a host-recursive interpreter a call to a procedure in tail position still consumes a host stack frame unless the interpreter implements an explicit trampoline. In the prototype the evaluator is a loop, and a call in tail position does not recurse. It commits the activation sphere of the call it is replacing, binds a new `call` sphere coupled to the closure being applied, creates the callee's environment frame, and continues the loop with the callee's body, so that at most one activation sphere of such a chain is live at a time. The final line of the transcript reports the result: a loop that makes three thousand tail calls finishes with zero live call spheres. The corresponding non-tail recursion, such as the naive definition of Fibonacci used for the profile in Chapter 12, leaves its activations live until they return, and the live count grows with the depth of the pending computation, as it must.

```
args = [self.eval(a, env) for a in x[1:]]
if isinstance(f, Closure):
    if couplings:
        w.pop(couplings.pop())
    params, body = w.spheres[f.id].payload
    cenv = w.spheres[f.id].deps[0]
    call = w.bind("call", None, len(args), (f.id,))
    couplings.append(call)
    env = self.new_env(cenv, (call,))
    if isinstance(params, Sym):
        w.bind("binding", (env, str(params)), tuple(args), (env, call))
    else:
        if len(params) != len(args):
            self.err("arity mismatch: expected %d, given %d" % (len(params), len(args)), "
    arity")
        for p, a in zip(params, args):
            w.bind("binding", (env, str(p)), a, (env, call))
    x = self._body(body, env)
    continue
```

This property is a statement about Ω and not about the host. The host evaluator still recurses on the structure of the expression, so a deeply nested expression can exhaust the host stack, but the number of live spheres, which is the quantity the model cares about, is bounded by the number of pending non-tail calls plus the size of the data. A reader who wishes to see why this agrees with the standard definition of proper tail recursion may compare it with the space-complexity framing in the Scheme literature, where a tail call must not add to the live space that the program needs.

5.4 Macros as Ledger Rewriting

The prototype supports a pattern-based macro form, `define-syntax-rule`, with ellipsis patterns that may be nested. A macro is a sphere holding its pattern and template, and use of a macro is a rewrite performed before evaluation: the matcher binds pattern variables to the matching pieces of the form, the expander substitutes them into the template, and the result is evaluated in the use-site environment. The matcher and expander are small enough to list.

```
def match(p, f, lits, b):
    if isinstance(p, Sym):
        if p in lits:
            return f == p
        if p != "_":
            b[p] = f
        return True
    if isinstance(p, tuple):
        if not isinstance(f, tuple):
            return False
        i = 0
        while i < len(p):
            if i + 1 < len(p) and p[i + 1] == ELL:
                after = p[i + 2:]
                avail = len(f) - i - len(after)
                if avail < 0:
                    return False
                subs = []
                for it in f[i:i + avail]:
                    bb = {}
                    if not match(p[i], it, lits, bb):
                        return False
                    subs.append(bb)
                for v in pvars(p[i], lits):
                    b[v] = Seq(bb[v] for bb in subs)
                return match(after, f[i + avail:], lits, b)
            if i >= len(f) or not match(p[i], f[i], lits, b):
                return False
            i += 1
        return i == len(f)
    return p == f

def expand(t, b):
    if isinstance(t, Sym):
```

```
        return b.get(t, t)
    if isinstance(t, tuple):
        out, i = [], 0
        while i < len(t):
            if i + 1 < len(t) and t[i + 1] == ELL:
                vs = [v for v in pvars(t[i], ()) if isinstance(b.get(v), Seq)]
                n = max((len(b[v]) for v in vs), default=0)
                for k in range(n):
                    bb = dict(b)
                    for v in vs:
                        bb[v] = b[v][k]
                    out.append(expand(t[i], bb))
                i += 2
            else:
                out.append(expand(t[i], b))
                i += 1
        return tuple(out)
    return t
```

The expansion is not hygienic, in the sense of the literature on hygienic expansion [9]. A template that introduces a binding can capture a user variable of the same name, and a template that refers to a free name sees the use-site binding of that name. This is a limitation of the prototype and it is stated here because Chapter 7 builds a finite-state-machine notation on the macro form and relies on avoiding the capture by the choice of names in the template. A hygienic expander would rename template-introduced binders with fresh names, and the ledger would record those renames as spheres whose lineage ends in the macro that introduced them. The reader who adds this extension will find that the lineage function already has the shape that is needed.

5.5 A Prelude Written in the Language

The remaining list operations, `map`, `filter`, `foldl`, `foldr`, `reverse`, and `range`, are not primitives. They are defined in the language itself in a prelude, evaluated when a world is created, and they are the first programs the interpreter runs. The prelude is the reason the Fibonacci profile in Chapter 12 shows a Lisp world that is large even before the workload starts, a fact that is visible in the ledger and would be invisible in a host-state design.

Exercise 5.1. Add `call/cc` in its escape-only form. Decide what the continuation sphere depends on and which spheres must be refused when the continuation is invoked.

Exercise 5.2. Implement hygiene by renaming template-introduced binders with fresh names. Write a test in which a user variable named like the template's temporary is captured by the current expander and preserved by the new one.

Exercise 5.3. Use the lineage function to implement a stepper that reports every superseded value of a given variable and the application that caused each supersession.

Chapter 6

A Racket Layer: Modules, Provision, and Requirement

6.1 From a Language to a Language of Modules

Racket [7, 8] differs from the Lisp of the previous chapter in organizing programs as modules that are loaded on demand, and in making the language of a module a choice that the module declares. Both features are properties of how names enter a scope, and both are therefore natural subjects for a ledger that already records every name's introduction. The prototype in this chapter extends the Lisp core with the following forms. `module` declares a module with a name, an initial language, and a body. `provide` marks which definitions a module exports. `require` brings the exports of another module into the current scope. Four import modifiers, `only-in`, `except-in`, `prefix-in`, and `rename-in`, adjust the imported names. Finally `provide` accepts `all-defined-out` and `all-from-out`, which are bulk forms.

The state added by the layer is small. A `source` sphere holds the text of a module that has been indexed but not loaded, a module sphere is bound when the module is declared, an `import` sphere records one name that a scope received from a module, and a `langimport` sphere records one name that a module received from its initial language. The distinction between the last two is the mechanism of Chapter 7, and it is the reason the layer is useful for the topic of this book.

6.2 Declaration, Instantiation, and Lazy Loading

A `require` of a module that has not been declared does not fail at once. It searches the source index for a `source` sphere whose key is the module name, declares the module from its text, instantiates it by evaluating its body, and then performs the import. A module that is not in the index is a refusal with the reason that it cannot be opened. Declaration binds a `module` sphere whose dependencies are its environment and the `source` sphere it was loaded from, so the lineage of a module leads back to the text that defined it, and the ordering of events is a

readable record of what was loaded and when.

```
def _module(self, x, env):
    name, lang = str(x[1]), str(x[2])
    menv = self.w.bind("env", None, None)
    self._import_lang(menv, lang)
    self._provides.append([])
    try:
        for f in x[3:]:
            self.eval(f, menv)
        specs = list(self._provides[-1])
    finally:
        self._provides.pop()
    self._declare(name, menv, specs)
    return Sym(name)

def _require(self, x, env):
    w, v = self.w, self.w.view
    for spec in x[1:]:
        for n, e in self._spec(spec).items():
            if v.last_key("binding", (env, n)) is not None:
                self.err("require: identifier already defined: %s" % n, "import-conflicts-
    with-definition")
            prior = v.last_key("import", (env, n))
            if prior is not None:
                if w.spheres[prior].deps[0] == e:
                    continue
                self.err("require: identifier imported twice with different bindings: %s" %
    n, "conflicting-import")
            w.bind("import", (env, n), w.spheres[e].payload, (e,))
    return None
```

```
def _find(self, name):
    v = self.w.view
    m = v.last_key("module", name)
    if m is None:
        s = v.last_key("source", name)
        if s is None:
            self.err("cannot open module: %s" % name, "cannot-open-module")
        self.run(self.w.spheres[s].payload, env=self.top)
        m = v.last_key("module", name)
        if m is None:
            self.err("module %s was not declared by its source" % name, "bad-module-source")
    return m

def where(self, name, env=None):
    env = self.top if env is None else env
    o = self.owner(env, name)
    if o is None:
        return None
    e, tag, i = o
    if tag == "binding":
        return ("defined", None)
    exp = self.w.spheres[self.w.spheres[i].deps[0]]
```

```
    return ("imported" if tag == "import" else "language", exp.key[0])
```

The transcript for this chapter exercises each behavior in turn. After `(require units)` the function `c->f` is available and an internal helper that `units` does not provide is unbound. A prefixed import makes the same module's other functions available under a prefix. Defining a name that was imported is refused with the reason that the identifier is already imported, which matches the module system's rule that a module-level definition may not silently replace an import. The query `where` reports, for a name, whether it came from the module language or from an import and from which module, and the query `imports_of` reports what a module itself required. The final lines of the transcript show that the modules instantiated during the session are the base language, the macro module, the two DSLs, the traffic-light module that depends on the macro module, and the units module.

```
> (require units)

> (c->f 100)
212
> (internal-helper 2)
error: unbound identifier: internal-helper
> (require (prefix-in u: units))

> (u:km->miles 1000000)
621371
> (require game)

> (score 60 30 20)
100
> (+ 60 30 20)
110
> (require traffic)

> (run 'green '(timer timer))
red
> (light 'red 'fault)
flashing
> (define c->f 1)
error: define: identifier already imported: c->f
> (require no-such)
error: cannot open module: no-such
where c->f: ('imported', 'units') ; where score: ('imported', 'game') ; where map: ('
    language', 'base')
imports of traffic: [('define-fsm', 'fsm')]
imports of game: []
modules instantiated: ['base', 'fsm', 'game', 'saturating', 'traffic', 'units']
```

6.3 Conflicts and Staleness

Two behaviors of the layer are worth isolating because they are the same behaviors that package managers exhibit at larger scale. The first is a name conflict. If two modules provide the same name with different definitions and both are required without a prefix, the second import is refused with the reason `conflicting-import`, and the first import remains. Requiring the same definition twice, by two routes, is not a conflict and is accepted silently, and requiring a name that the scope already defines is refused with the reason `import-conflicts-with-definition`. The second is staleness. If a module is redeclared, the prototype refuses the old module sphere and its export spheres, and the importers, which hold spheres that depend on the old exports, keep the `import` spheres that they were given. Those spheres refer to the old definitions. The function `reload` redeclares a module from its current source text, and an importer sees the new definitions only when it is itself declared again and requires the module afresh. This is the same situation as an installed package whose dependency has been upgraded underneath it, and it returns in Chapter 10 with the version solver in the role of the party that decides whether the upgrade is allowed.

Exercise 6.1. Implement the `submod` form in its simplest version, a module declared inside another module, and decide what the `provide` of the inner module is allowed to refer to.

Exercise 6.2. Add cycle detection for `require`. A cycle should be a documented refusal that names the chain of modules. State whether the refused `module` sphere should remain declared.

Exercise 6.3. Write a test that declares module A, requires it from module B, replaces the source of A with a different definition, calls `reload`, and checks that B still sees the old definition until B is declared again.

Chapter 7

Domain-Specific Languages by Module Language and by Macro

7.1 Two Mechanisms for One Goal

A domain-specific language in the Racket tradition is a way of letting a program say what it means in the vocabulary of its problem, and the tradition provides two mechanisms with different costs. The first is the *module language*. A module declares its initial language, which determines the set of names that are in scope before the module body is read, so a module written in a DSL sees the DSL's versions of the standard operations. The second is the *macro*, which adds syntax to an existing language by rewriting forms before evaluation. The SpherePOP prototype implements both, and the point of this chapter is that the difference between them is visible in the ledger.

7.2 A DSL by Module Language

The module `saturating` defines arithmetic whose addition clamps at a ceiling and then provides it under the name `+`, together with the unchanged names of the base language that it wants to keep. The module `game` is declared in the `saturating` language. Its body is ordinary code that uses `+`, and because the initial import set of `game` is `saturating`, the `+` it sees is the clamping one. The transcript of the previous chapter shows the effect: the call `(score 60 30 20)` returns one hundred, while the same arithmetic in the ordinary language returns one hundred ten. The `game` module contains no import of the clamping addition and no renaming, and a reader who inspects it sees plain arithmetic.

The ledger records the mechanism precisely. Each name that `game` received from `saturating` is a `langimport` sphere, and the query `where` distinguishes it from an explicit import: `where map` reports that the name came from the language `base`, and `where score` reports that it was imported from `game`. It is a small observation, and it is the content of the claim that a module language is an initial import set. The language of a module is not a separate concept

in the prototype, it is a module whose exports are imported before the body is read. This is how Racket itself describes the relationship, and the prototype is a way of seeing that the description is not a metaphor.

7.3 A DSL by Macro

The module `fsm` provides the macro `define-fsm`, which accepts a name, an initial state, and a list of transition clauses, each naming a state, an event, and a next state, and which expands to a function that interprets the transition table. The macro uses a nested ellipsis, since each state has several event clauses. The module `traffic` requires `fsm` and uses the macro to declare a traffic light with states green, yellow, red, and flashing, and the transcript shows `run` applying a list of events and `light` applying a single event to a given state.

The notation hides the table-walking code, and it can do so only because the macro is expanded before the evaluator sees the form. The ledger evidence is the line `imports of traffic: [('define-fsm', 'fsm')]`: the module `traffic` names exactly one import from `fsm`, and what it uses it for is the macro. Comparing the two mechanisms, the module-language DSL changes the meaning of existing names throughout the module and the macro DSL adds new names without changing the old ones. The first is the right tool when the domain is a variation of an existing semantics, such as saturating arithmetic. The second is right when the domain is a new syntax, such as a table of transitions. The first can surprise a reader who does not know the module's language, and the second can surprise a reader who does not know that the macro captures names, which is the limitation noted in Chapter 5.

7.4 Why This Is the Package-Index Question in Miniature

The question of how a DSL's functions reach the programs that use them is the question that every package index answers. A module is found by name in an index of sources, loaded, instantiated once, and its exports are placed in a scope under rules that govern conflicts, renames, and prefixes. The only features that a package manager adds to this picture are those of distribution: versions, constraints between modules, and a choice among several candidate providers of one name. Chapter 10 adds them to the prototype and wires the result into the `require` form of this chapter, so that a module can be found because a package was installed and not because a source file was placed in the index by hand.

Exercise 7.1. Write a module language in which `if` requires its condition to be a boolean and refuses otherwise. Decide whether the check is a property of the language module or of the evaluator, and state what the ledger shows in each case.

Exercise 7.2. Extend `define-fsm` to accept an optional accepting-state list and to produce a predicate that reports whether a run ends in an accepting state. Show that a state name

chosen by the user can collide with a template temporary and fix the collision by renaming the temporary.

Exercise 7.3. Define a units module that provides `meters` and `feet` as functions and a conversion between them, require it with `prefix-in`, and write the call that distinguishes an import with a prefix from a name defined locally.

Chapter 8

Commodore BASIC: A Program Store and a Disk

8.1 The Machine Being Modeled

The BASIC of the Commodore home computers is an instructive target because its entire programming environment is a single interpreter with a program store, and nothing else stands between the user and the machine. A line that begins with a number is not executed, it is stored. A line that does not begin with a number is executed at once. The stored program is a sorted collection of numbered lines, `RUN` interprets it from the lowest line number, `LIST` prints it, and `LOAD` replaces it with the contents of a file read from a disk drive. Everything that distinguishes this system from a modern one follows from the fact that the program store is the same kind of thing as the program being run: text with line numbers, held in memory, edited in place.

The prototype models the documented behavior of the Commodore interpreter [17] and not a particular machine. Variable names are significant to two characters, so `COUNT` and `COUNTER` name one variable. A string variable ends with a dollar sign and an integer variable with a percent sign. Positive numbers print with a leading space and a trailing space. Error messages are the interpreter's strings, with two spaces before the word `ERROR`, and an error in a running program stops the run and reports the line number. Subroutine return addresses and loop control records live on stacks that the interpreter manages, and a `RETURN` with no pending `GOSUB` is an error.

8.2 Lines as Spheres, Editing as Supersession

The program store is the family of live `line` spheres. A line has its number as key and its text as payload. Entering a line is the operation whose three cases are the editing semantics of the interpreter: an unseen number binds a new sphere, a number already in the store supersedes the old sphere with the new text, and a number with an empty body refuses the old sphere

with the reason that it was deleted. The listing is the whole of the editor.

```
def enter(self, text):
    text = text.strip()
    m = re.match(r"(\d+)\s*(.*)$", text)
    v = self.w.view
    if not m:
        return self.direct(text)
    n, body = int(m.group(1)), m.group(2)
    old = v.last_key("line", n)
    if not body:
        if old is not None:
            self.w.refuse(old, "deleted")
    elif old is not None:
        supersede(self.w, old, body, "replaced")
    else:
        self.w.bind("line", n, body)

def put(self, key, val, line=None):
    if key.endswith("$") != isinstance(val, str):
        self.fail("type", line)
    i = self.w.view.last_key("basic.var", key)
    if i is None:
        self.w.bind("basic.var", key, val)
    else:
        supersede(self.w, i, val)
```

Because editing is supersession, the history of a program is available. The old text of line 30 is not lost when the line is replaced, it remains in the ledger as a refused sphere with a successor that depends on it, and a debugger can reconstruct every version of the program that existed during a session. The original machine had no such facility. The prototype provides it as a consequence of the representation and not as a feature, which is the reason that the program store was chosen to be spheres.

Variables follow the same rule. `put` binds a sphere for a variable that has not been assigned, and supersedes it when it has, with a type check against the suffix of the name. Arrays declared with `DIM` and the cells of `DATA` statements read by `READ` are spheres of their own tags, a data pointer is a sphere that is superseded on every read, and `PEEK` and `POKE` address a family of `mem` spheres by integer address.

8.3 Control Flow and the Interpreter Stacks

`GOSUB` binds a sphere holding the address to return to, and `RETURN` pops the most recent such sphere and resumes there. `FOR` binds a loop record with the variable, limit, step, and loop address, and `NEXT` supersedes the variable, tests the limit, and either jumps back or pops the record. A jump is itself recorded as a `jump` sphere, so the ledger of a run shows each transfer of control. The listing shows the subroutine call and return, which is the whole of the mechanism and fits in a few lines.

```
elif sig[0] == "gosub":
    w.bind("gosub", None, (li, si + 1))
    li, si = jump(n, sig[1])
elif sig[0] == "return":
    g = v.last("gosub")
    if g is None: self.fail("return", n)
    li, si = w.pop(g)
```

The transcript for the chapter lists a program, runs it, shows the directory, loads by wildcard, edits a line and runs again, and ends with the failure cases. The program computes a sum of squares with a `FOR` loop and a subroutine. After the run, the output `SUM = 55` shows the leading and trailing spaces of numeric output. The edit of line 30 to accumulate the index and not its square produces `SUM = 15` on the next run, and the old line is still in the ledger.

```
LIST
10 REM SUM OF SQUARES
20 FOR I=1 TO 5
30 S=S+I*I
40 NEXT I
50 PRINT "SUM ="; S
60 GOSUB 100
70 END
100 PRINT "DONE"
110 RETURN
RUN
SUM = 55
DONE
LOAD "$",8 then LIST
0 "DISK" 00 2A
5 "SUMSQ" PRG
LOAD "SU*",8 then LIST (first line)
10 REM SUM OF SQUARES
after editing line 30 to S=S+I and RUN
SUM = 15
LOAD "NOPE",8
?FILE NOT FOUND  ERROR
LOAD "SUMSQ"
?DEVICE NOT PRESENT  ERROR
COUNT and COUNTER share a variable: 9
```

8.4 LOAD as the Central Operation

The disk is a family of `file` spheres keyed by device number and name, each holding the lines of a saved program. `SAVE` binds a file sphere from the current store. `LOAD` is the operation that makes the program store a loading language, and its three behaviors in the prototype are the three documented behaviors of the original. Loading a name looks for a file on the device, and a trailing asterisk matches by prefix and takes the first match in sorted name order. Loading the special name `$` loads the directory itself as a program, so that `LIST` afterward prints the

disk's contents in program form, with the first line giving the disk name and identifier and the remaining lines giving file sizes and names. Loading a name that does not exist is a documented refusal, the `FILE NOT FOUND` error, and loading from a device other than the one the prototype models is the `DEVICE NOT PRESENT` error.

```
def load(self, rest, line):
    m = re.match(r'"([^"]*)"\s*(?:,\s*(\d+))?', rest)
    name, dev = m.group(1), int(m.group(2) or 1)
    if dev != self.device: self.fail("device", line)
    v = self.w.view
    for i in v.live("line"): self.w.refuse(i, "replaced by LOAD")
    self.clear("LOAD")
    if name == "$":
        for n, t in self.directory(dev): self.w.bind("line", n, t)
        return
    pat = name.replace("*", "")
    cands = sorted(v.sphere(i).key[1] for i in v.live("file") if v.sphere(i).key[0] == dev)
    hit = next((c for c in cands if (c == name if "*" not in name else c.startswith(pat))),
    None)
    if hit is None: self.fail("notfound", line)
    fid = v.last_key("file", (dev, hit))
    for n, t in v.payload(fid):
        self.w.bind("line", n, t, (fid,))
```

Two properties of the listing are worth stating. The first is that loading refuses every live line of the current program with the reason that it was replaced by the load, then clears the variables, so that a loaded program starts in a fresh state, and each line bound from a file is coupled to the file sphere it came from. The lineage of a line therefore reaches the disk, and the question of where a line of the running program came from has an answer in the ledger. The second is that the directory and the program are the same kind of object. A program that is the listing of a disk is an ordinary stored program, and nothing in the prototype treats it specially except the step that generates its lines. This is the sense in which the interpreter is a loading language: names are resolved against a device, the result is installed into the one store the interpreter has, and the stored result is then both data and code.

The comparison with the shells of the next chapter is the reason that this chapter precedes it. A shell resolves a name against a sequence of directories and installs the result into a process, and the interpreter resolves a name against a device and installs the result into a program store. The first has an ordered search and the second has a single device, but both are a lookup followed by an installation, and the prototype implements both as an ordered probe.

Exercise 8.1. Implement `VERIFY`, which compares the program in memory with a file and reports an error if they differ. Decide whether the comparison is a collapse rule, and decide what its recorded digest means.

Exercise 8.2. The interpreter stores a `GOSUB` return address as a line and statement position. Show that editing a line while a subroutine call is pending can make the return address refer

to a statement that no longer exists, and propose a behavior that is consistent with the ledger.

Exercise 8.3. Implement the `ON` statement with a computed jump and verify that its jump spheres show only the transfer that was taken. State what the refused alternatives would be if the jump targets were bound as spheres before the choice.

Chapter 9

Shells as Loading Languages

9.1 What a Shell Does

A shell is a language whose principal operation is resolving a name to something runnable and then running it. The remainder of the language, such as variables, quoting, and redirection, is secondary to this operation in the sense that a shell with no variables is still a shell and a shell that cannot resolve names is not. The resolution is ordered. A name may denote an alias, a builtin command that the shell implements itself, a cached earlier result, or a file found by searching a list of directories, and the order in which these are tried is what makes two shells different languages. The prototype separates the mechanism, which is a search over an ordered list in which every probe is a recorded event, from the policy, which is the dialect’s rule for which list to search and in what order. The search function is shared with the BASIC chapter and with the package-index chapter that follows. What is shared is a skeleton, an ordered probe with each miss refused and the hit committed, and not a semantics. The caching of a bash hash table, the wildcard matching of a BASIC load, the extension order of cmd, and the version constraints of a package index all live in the probe functions and in the code that calls the search, and the dialects differ precisely there.

```
def resolve(self, name):
    """Return (kind, detail). Kinds follow the dialect's rule order."""
    v = self.w.view
    if self.dialect == "cmd":
        if name.lower() in CMD_BUILTINS:
            return "builtin", name.lower()
        exts = [""] if os_ext(name) else PATHEXT
        if "\\" in name:
            scopes = [(name.rsplit("\\", 1)[0], e) for e in exts]
            name = name.rsplit("\\", 1)[1]
        else:
            scopes = [(d, e) for d in [self.cwd()] + self.pathlist() for e in exts]
        hit, _ = search(self.w, name, scopes, self.probe_win)
        return ("path", hit) if hit else ("missing", None)
    a = v.last_key("alias", (self.id, name))
    if a is not None:
```

```
        return "alias", v.payload(a)
    if name in BASH_BUILTINS:
        return "builtin", name
    if "/" in name:
        d, n = posixpath.split(self.abspath(name))
        out, hit = self.probe_linux(d, n)
        if out == "hit":
            return "path", hit
        return ("noexec", hit) if out.startswith("skip") else ("missing", None)
    h = v.last_key("hash", (self.id, name))
    if h is not None:
        return "hashed", v.payload(h)
    hit, tried = search(self.w, name, self.pathlist(), self.probe_linux)
    if hit:
        self.w.bind("hash", (self.id, name), hit[0])
        return "path", hit
    if any(o.startswith("skip") for _, o in tried):
        return "noexec", None
    return "missing", None
```

Each probe is a sphere. A probe that misses is refused with the reason that the file is missing, a probe that finds a file that exists but cannot be run is refused with a reason that says so, and the probe that finds an executable is popped and returned. The ledger of a command is therefore a complete record of the places that were consulted and in what order, which is exactly the evidence that an administrator wants and the real system discards.

9.2 Bash: Alias, Builtin, Hash, Path

The documented order of command lookup in a Bourne-style shell [18] is aliases, then the shell's builtins, then the table of previously found commands, then a search of the directories in the path variable. The prototype follows that order. Its first transcript shows the consequence of the table. The first run of `ls` probes six directories and finds the program in the sixth, and the second run of `ls` performs no probes at all, because the result of the first search was bound as a `hash` sphere. The command `hello`, installed in a directory that comes early in the path, is found after two probes. When the path variable is assigned, the prototype refuses every hash sphere with the reason that the path changed, and the next `hello` probes again from the front of the new path, which the transcript shows with a different first probe. This is the observable difference between a shell that caches lookups and one that does not, and it explains a familiar experience, in which a program that has been moved is still invoked from its old location until the shell is told to forget it.

```
bash> ls   (status 0)
    probe /usr/local/sbin/ls                    miss
    probe /usr/local/bin/ls                     miss
    probe /usr/sbin/ls                          miss
    probe /usr/bin/ls                           miss
    probe /sbin/ls                              miss
    probe /bin/ls                               HIT
```

```
bash> ls   (status 0)
bash> hello   (status 0)
    probe /usr/local/sbin/hello                    miss
    probe /usr/local/bin/hello                     HIT
bash> export PATH=/bin:/usr/local/bin   (status 0)
bash> hello   (status 0)
    probe /bin/hello                               miss
    probe /usr/local/bin/hello                     HIT
cmd > hello   (status 0)
    probe C:\Users\me\hello.COM                    miss
    probe C:\Users\me\hello.EXE                    miss
    probe C:\Users\me\hello.BAT                    HIT
cmd > nosuch   (status 9009)
    probe C:\Users\me\nosuch.COM                   miss
    probe C:\Users\me\nosuch.EXE                   miss
    probe C:\Users\me\nosuch.BAT                   miss
    probe C:\Users\me\nosuch.CMD                   miss
    probe C:\Windows\System32\nosuch.COM           miss
    probe C:\Windows\System32\nosuch.EXE           miss
    probe C:\Windows\System32\nosuch.BAT           miss
    probe C:\Windows\System32\nosuch.CMD           miss
    probe C:\Windows\nosuch.COM                    miss
    probe C:\Windows\nosuch.EXE                    miss
    probe C:\Windows\nosuch.BAT                    miss
    probe C:\Windows\nosuch.CMD                    miss
wsl > notepad.exe   (status 0)
    probe /usr/local/sbin/notepad.exe              miss
    probe /usr/local/bin/notepad.exe               miss
    probe /usr/sbin/notepad.exe                    miss
    probe /usr/bin/notepad.exe                     miss
    probe /sbin/notepad.exe                        miss
    probe /bin/notepad.exe                         miss
    probe /mnt/c/Windows/System32/notepad.exe      HIT
wsl > notepad   (status 127)
    probe /usr/local/sbin/notepad                  miss
    probe /usr/local/bin/notepad                   miss
    probe /usr/sbin/notepad                        miss
    probe /usr/bin/notepad                         miss
    probe /sbin/notepad                            miss
    probe /bin/notepad                             miss
    probe /mnt/c/Windows/System32/notepad          miss

type and which on bash:
    type cd   (status 0)  cd is a shell builtin
    alias ll='ls'   (status 0)
    type ll   (status 0)  ll is aliased to `ls'
    which cd   (status 1)
busybox: applets cat echo false ls true uname ; links resolving to /bin/busybox: 6 of 6
```

The commands `type` and `which` are included because they answer the same question with different authority, and the difference is a lesson in what a loading language owes its user. `type` is a builtin that runs the shell's own resolution and so reports aliases and builtins correctly, as the transcript shows for `cd` and an alias. `which` is an ordinary program that sees only the path

and the file system, and so reports nothing for a builtin. The two disagree about `cd` in the transcript, with `type` reporting a builtin and `which` returning a nonzero status and no output, because they are implemented by different parties with access to different parts of the state.

9.3 Windows Command Interpreter

The Windows command interpreter has a different order and a different rule for what a name means. Its internal commands are tried first. For an external command, the interpreter searches the current directory and then each directory of the path variable, and within each directory it tries the name with each extension listed in the extension variable, in order. The default order places `.COM` before `.EXE`, then `.BAT` and `.CMD`. The names are compared without regard to case. A name that already has an extension is tried as written.

The prototype implements this rule with the same search function, with the scopes being the pairs of directory and extension in the order the interpreter takes them. The first command in the cmd part of the transcript finds a batch file in the current directory after two misses, one for each earlier extension. The second command names something that does not exist and shows the whole search: the current directory and then two directories of the path, each with four extensions, twelve probes in all, ending in the status code that this interpreter uses for an unrecognized command. The transcript is the clearest demonstration of the sense in which a loading language is an ordered search, because the entire behavior of the command interpreter on an unknown name is visible as a list of what it tried.

9.4 WSL and the Two Namespaces

The Windows Subsystem for Linux places a Linux shell on a machine that also has a Windows file system, and its documented behavior is that the Windows path is appended to the Linux path with directory names translated to the mount point under which Windows drives appear. The prototype builds this as two namespaces in one world. A `win` namespace and a `linux` namespace each hold file spheres, the translation from a Windows directory to a mount path is a function on strings, and a WSL shell resolves names against the Linux path followed by the translated Windows path.

The transcript shows the consequence. Running `notepad.exe` probes the six Linux directories and then finds the program under the mount of the Windows system directory. Running `notepad`, the bare name, searches the same seven directories and finds nothing, with the status that bash uses for command not found. This is a modeling decision stated plainly: the prototype matches names exactly, including the extension, because the documented behavior is that a Windows executable is invoked with its extension, and the prototype does not implement any extension inference on the Linux side. Real systems have configuration that can change the details, and a reader who needs a precise statement about a particular installation should read that installation and not the prototype.

9.5 Ubuntu and the Command-Not-Found Hook

Ubuntu's shell is bash with a distribution's choice of path and one addition that is relevant to this book. When a name is not found, the shell calls a handler that consults the distribution's package index, finds a package that provides a file of that name, and suggests installing it. The prototype supports this as a hook on the resolution failure. The hook receives the name and returns text or nothing, and the failed search is already complete in the ledger, probe by probe, before the hook is consulted. In Chapter 10 the hook is connected to the package index, so that a miss in the loading language becomes a question put to the package language, and the package language's answer is a message and not an action. The separation matters, because the shell is deliberately not allowed to install anything on its own.

9.6 BusyBox and the Multi-Call Binary

A BusyBox system implements a large set of small programs as applets of one executable. The executable determines which applet to run from the name it was invoked under, and the file system presents each applet as a link to the executable. The prototype models this directly. The machine contains one file, `busybox`, and a link for each applet in the same directory. Resolving `ls` finds the link, resolves the link to the executable, and passes the name `ls` as the program's first argument, and the program dispatches on that argument. The last line of the transcript reports that all six applets of the model resolve to the single executable. Invoking `busybox` with the name of an applet as its first argument is the same dispatch without the link, and invoking it with no argument lists the applets.

The multi-call design is the cleanest example in the book of a loading language whose search is trivial and whose installation is interesting. The name does the work that a path search does elsewhere, because it has been moved from the path into the argument vector of a single program. For the purposes of the comparison in Chapter 12, a BusyBox system has the shallowest search of the shells in this chapter and the most indirection between name and code.

Exercise 9.1. Add the `hash -r` builtin to bash and a test that checks the next lookup probes again. Then add a hash table entry that goes stale because the file it names has been deleted, and determine what the shell should do when the cached file is missing.

Exercise 9.2. Implement an extension inference rule for WSL in which a bare name that is not found is retried with the Windows executable extension against the Windows part of the path only. Describe how the transcript of Section 9.4 changes and why the decision belongs to the dialect and not to the search function.

Exercise 9.3. Make cmd treat a name containing a backslash as a path and not as a name to search, and make a name that already contains an extension skip the extension list. Check

the transcript's probe counts against your implementation.

Chapter 10

Package Indexes and Search Paths

10.1 Three Problems in One Word

The phrase package index covers three distinct problems [19] that are usually solved by one tool, and the prototype separates them. The first is publication: an index holds, for each package name, a set of releases, and each release has a version, a list of constraints on other packages, and a set of artifacts such as modules. The second is resolution: given a request for some packages with constraints, choose a version of every package that is needed, so that every constraint is met. The third is lookup: given an installed set, decide where a name used in a program is found. Resolution is a computation about the future, and lookup is a computation about the present, and a package manager needs the first in order to guarantee the second.

10.2 Flat Resolution with Backtracking

The simplest resolution policy is flat. At most one version of each package is installed, so every constraint from every dependent applies to the single installed version. The prototype implements the flat solver as a depth-first search over the versions of each needed package, newest first. Each tentative choice is a `decision` sphere, coupled to the decision that required it. When a choice leads to a state in which a later constraint cannot be met, the search abandons it by refusing the decision sphere with the reason `backtrack`, and tries the next version. A failed resolution is a refused `attempt` sphere with the reason that it is unsatisfiable, together with an error.

```
def solve_flat(self, reqs, pinned=None):
    w = self.w
    pinned = pinned or {}
    decided = []

    def rec(todo, chosen, cons, parent):
        if not todo:
            return chosen
```

```
    name = todo[0]
    for ver in self.releases(name):
        if not all(satisfies(ver, s) for s, _ in cons[name]):
            continue
        deps = dict(self.release(name, ver)[0])
        ncons = {k: list(x) for k, x in cons.items()}
        bad = False
        for dn, ds in deps.items():
            ncons.setdefault(dn, []).append((ds, name))
            if dn in chosen and not satisfies(chosen[dn], ds):
                bad = True
        if bad:
            continue
        d = w.bind("decision", name, ver, tuple(parent.get(r) for r in [name] if parent.
get(r) is not None))
        decided.append(d)
        nxt = todo[1:] + [dn for dn in deps if dn not in chosen and dn not in todo and
dn != name]
        res = rec(nxt, {**chosen, name: ver}, ncons, {**parent, **{dn: d for dn in deps
}})
        if res is not None:
            return res
        w.refuse(d, "backtrack")
    return None

cons = {n: [(s, "root")] for n, s in reqs.items()}
for n, v in pinned.items():
    cons.setdefault(n, []).append(("==" + v, "installed"))
res = rec(list(cons), {}, cons, {})
if res is None:
    attempt_refused(w, "attempt", tuple(sorted(reqs.items())), "unsatisfiable")
    raise ResolutionError("no consistent set of versions for %r" % (reqs,))
return res
```

The transcript for the chapter gives the diamond example. Packages `A` and `B` each depend on `C` with constraints that overlap on exactly one release, and the newest release of `C` satisfies one and not the other. The solver first selects the newest release of `C` and finds that it conflicts, backtracks, and arrives at the older release that both accept. The report shows two backtracked decisions. The next line is a request that pins `A` to a release that requires a version of `C` that `B` cannot accept, and it is refused with the message that no consistent set exists. Each refusal is a sphere, so the ledger answers the question of what was tried, in the order in which it was tried, which the error message of a conventional solver can only summarize.

10.3 Nested Resolution and Lexical Scope

The flat policy fails on the diamond whenever the constraints are incompatible. The alternative, used by some language ecosystems, is to allow several versions of one package in one installation, by giving each package its own private copy of the packages it depends on. The prototype implements this as nested resolution. Each requirement is resolved in the context of

the package that made it, producing a `nested` sphere keyed by the path of ancestors and the name, and lookup of a name from inside a package searches the package's own scope and then each enclosing scope outward, which is the rule for resolving a variable in lexical scope.

```
def solve_nested(self, reqs, path=()):
    w, out = self.w, {}
    for name, spec in reqs.items():
        ver = next((x for x in self.releases(name) if satisfies(x, spec)), None)
        if ver is None:
            attempt_refused(w, "attempt", (name, spec), "unsatisfiable")
            raise ResolutionError("no version of %s satisfies %s" % (name, spec))
        w.bind("nested", (path, name), ver)
        out[(path, name)] = ver
        tag = "%s@%s" % (name, ver)
        if tag not in path:
            out.update(self.solve_nested(dict(self.release(name, ver)[0]), path + (tag,)))
    return out

def node_lookup(self, frm, name):
    v = self.w.view
    for k in range(len(frm), -1, -1):
        i = v.last_key("nested", (frm[:k], name))
        if i is not None:
            return v.payload(i)
    return None
```

In the transcript the request that the flat policy refused succeeds. Package `A` at its pinned release gets the release of `C` that it requires, package `B` gets a different release of `C`, and the lookup of `C` from `A` and from `B` returns two different versions. The price is visible in the same report, since the installation now contains two copies of `C`, and the question of what happens when objects created by one copy are passed to the other is the question that this policy leaves to the programmer. The comparison with flat resolution is the same as the comparison between dynamic scope with one global environment and lexical scope with nested environments. Flat resolution is simple to reason about and fails more often. Nested resolution succeeds more often and has no single answer to the question of which version is installed.

10.4 Uninstall, Upgrade, and Lock

An uninstall request is checked against the installed packages that depend on the target. If any does, the request is refused with the reason that names them, which the transcript shows for an attempt to remove `C` while `A` and `B` require it. An upgrade is a supersession: the old installed sphere is refused with the reason that it was upgraded, and the new release is bound with the old one as its dependency, so the history of a package is a lineage in the same sense as the history of a variable. A lock is a recorded collapse over the installed set. It produces a digest that identifies the exact versions, and a later check can compare the digest with the digest of the then-current set to detect drift. The lock is not a file the prototype writes, it is an event in the ledger.

10.5 The Search Path and the Connection to Racket

The third problem, lookup, is a search path. The prototype models the Racket collection search path as an ordered list of three scopes, a user scope, an installation scope, and the main distribution scope, and a module name is resolved by probing the scopes in order with the same search function used for shells. The transcript shows a module found in the user scope, which shadows a module of the same name in the installation scope, and another module found in the installation scope. Reassigning the path, which is a supersession of the `SearchPath` sphere, changes the answer for the same name without touching any package, and the transcript shows the first module found in the installation scope after the user scope is dropped from the list. Anyone who has debugged a program that imported the wrong version of a library because of a stale environment variable has met this behavior.

The connection to the module system of Chapter 6 is made by the function `sync_racket`. It reads the installed packages from the ledger, collects the module sources that each installed release carries, and adds them to the source index of a Racket world, which is the place a `require` looks when a module has not been declared. A module that was provided by a package that has since been removed is withdrawn from the index, with the reason that the package was removed. The effect is that installing a package makes its modules requirable, and removing it makes them unavailable, without any step in which a person places files in a directory.

```
def sync_racket(self, rkt):
    """Feed installed packages' modules into a Racket source index; retract removed ones."""
    v, w = self.w.view, self.w
    want = {}
    for name, ver in self.installed().items():
        for m, src in self.release(name, ver)[2]:
            want[m] = src
    for m, src in want.items():
        cur = rkt.w.view.last_key("source", m)
        if cur is None or rkt.w.view.payload(cur) != src:
            rkt.add_source(m, src)
        if rkt.w.view.last_key("owned", m) is None:
            rkt.w.bind("owned", m)
    for i in rkt.w.view.live("owned"):
        m = rkt.w.view.sphere(i).key
        if m not in want:
            s = rkt.w.view.last_key("source", m)
            if s is not None:
                rkt.w.refuse(s, "package-removed")
            rkt.w.refuse(i, "package-removed")
```

This is the sense in which the prototype demonstrates what happens in a package index in a language ecosystem. The index resolves, the installed set records, the synchronization publishes the modules of the installed set into a source index, and the language's loader finds a module by name and loads it on first use, which is the behavior of Chapter 6. Each stage is a sphere-level operation, and the history of an import can be followed from the line that required it back

to the release that provided it. The command-not-found hook of Chapter 9 uses the reverse of this path: given a missing name, the function `provider_of_file` finds the package whose release lists a file of that name in a standard directory, and the hook returns the text that proposes installing that package. The loading language, the package language, and the module language are three programs that consult one another through the ledger and share no private state.

Exercise 10.1. The flat solver chooses the newest satisfying version first. Change it to choose the oldest and describe a case in which the answer changes. Explain why the choice rule belongs to the policy and not to the search.

Exercise 10.2. Add a lock check that compares the digest of the current installed set with a stored digest and refuses the installation of anything that would change it. Decide whether the refusal reason should name the first package that differs.

Exercise 10.3. Implement a cycle in the nested solver, with package P requiring Q and Q requiring P, and observe how the path-membership test terminates the recursion. State what lookup from inside Q returns for P.

Part III

Composition

Chapter 11

Composing Languages in One Ledger

11.1 The Claim

Each prototype of Part II was tested alone, in a world of its own. A natural question is whether the prototypes are really instances of one construction or merely seven programs that share a kernel, and the test that separates the two is composition: place several of them in one world, interleave their operations, and ask whether each language still behaves as it does alone. If the prototypes use the kernel only through its four primitives and keep their state in tagged spheres, then by Proposition 1.5 operations of different languages name disjoint spheres and should commute. The claim to be tested is that a program of any one language produces the same observables, in a world that has been used by the others, as it produces in a fresh world.

The composition test does this with five languages. It runs a Racket definition and call, a BASIC program, a bash session that ends with `type cd`, and a shift register edge in one world, interleaved with a Forth program that computes a factorial and prints it, and then compares the Forth stack, output, and provenance with those of the same Forth program run alone. It also compares the other languages' observables with their expected values and verifies the replay of the shared ledger. The test passes in the final form of the code, and the section that follows is the account of how it came to pass, because the failure that preceded it is a better argument for the method than the pass.

11.2 A Collision the Test Found

The first version of the prototypes used short, natural tag names. Forth output was a sphere with the tag `out`, Forth variables had the tag `var`, and BASIC input was tagged `in`. Each language was correct alone. In the shared world these tags were claimed by more than one language, so a rule written for one language collected spheres that another language had bound, and the composition test failed where every single-language test passed.

The failure is a precise instance of the situation that the method is designed to expose, and the repair is a rule and not a patch. Every tag is namespaced by its language: `forth.out`, `forth.var`, `basic.var`, `basic.in`, `basic.out`, `lisp.out`, `shell.out`. The tags that are shared by design, such as `binding` for Lisp and Racket, which are one language in two layers, and the tag `probe` produced by the shared search function, are the exceptions that are documented as such. With the rule in place the composition test passes, and the rule is now part of the method of Chapter 2: the first decision, to place state in spheres, includes choosing a tag namespace that the language owns.

The incident also shows the value of a design in which isolation is a property to be tested and not a convention to be trusted. A host-state implementation of the same languages would have had no collision to find, because each interpreter would have kept its own variables, and it would also have had no shared substrate, and so the question that the test answers would not have been askable.

11.3 Values Across Languages

The second composition test moves a value from one language to another through the ledger. A Racket program computes the sum of the integers from one to ten by a fold over a range, and the result is bound as a Forth cell with provenance naming its origin. A Forth program then duplicates and multiplies it, and the stack holds the square of the sum, three thousand and twenty-five. The point of the test is modest and the consequence is not. The two languages share no data representation other than a payload in a sphere, and the transfer is a single `Bind`, so the interface between languages is the interface of the kernel, and every one of the prototypes can in principle be coupled to every other by the same means.

In the package chapter, the shell's command-not-found hook queries an index, the index synchronizes into a Racket loader, and the Racket loader finds modules whose DSLs are written in Racket itself. That is four layers connected only by spheres, and the composition tests of this chapter are the justification for treating that connection as sound.

11.4 What Composition Does Not Show

Composition shows that the prototypes do not interfere through the kernel. It does not show that the languages are correct, and it does not show that a program written in one can call a function written in another without a convention for what the values mean. The cell that carries a number from Racket to Forth is a number in both, and a list is not, because the Forth prototype has no list. A multi-language system would need agreement on a small set of data shapes, and the ledger would make the agreement checkable, since each crossing would be an event with a payload that a rule could inspect.

Exercise 11.1. Introduce a deliberate collision by tagging a Forth structure and a BASIC structure with the same name, and write the smallest test that detects it. Then write a test that detects a collision between two tags that differ only in case.

Exercise 11.2. Let a Racket program call a Forth word through a bridge that moves the arguments to Forth cells and the result back. Specify what the bridge refuses and why.

Exercise 11.3. Prove, from Proposition 1.5 and the disjointness of tag namespaces, that the stack observable of a Forth program is unchanged by interleaving any operations of the other prototypes. State the assumption about the shared search function that the proof needs.

Chapter 12

Comparative Structural Analysis and Open Problems

12.1 A Measured Profile

A ledger can be counted. For one workload per prototype, the table below gives the number of spheres bound, popped, and refused, the number of recorded collapses, the total number of events, and the number of spheres that remain live at the end. The workloads are the factorial of ten in Forth, eight clock edges of the four-bit register, the twelfth Fibonacci number in Lisp with its prelude loaded, the sum of the integers from one to one hundred in BASIC, four commands in bash, and the diamond installation in the package index.

Workload	Bind	Pop	Refuse	Collapse	Events	Live
Forth, factorial of 10	90	78	10	1	179	2
Shift register, 8 edges	52	40	0	8	100	12
Lisp, fib 12 (with prelude)	1450	465	0	8	1923	985
BASIC, sum 1 to 100	207	1	199	1	408	7
bash, four commands	36	1	15	0	52	20
Index, diamond install	14	0	2	0	16	12

Table 12.1: Event counts and live spheres at the end of one workload per prototype.

The table is a comparative measurement of what each language does with the primitives, and it reads cleanly once the vocabulary of Chapter 1 is applied. The Forth and BASIC workloads have many refusals because both languages consume and replace state continuously: a Forth operand is consumed when it is used, and a BASIC variable is superseded on every assignment. The BASIC row has a refusal for nearly every bind, one hundred and ninety-nine of two hundred and seven, and a single pop, because a loop that accumulates a sum supersedes the sum and the index on every pass and commits almost nothing. The shell row has fifteen refusals, one for each probe that missed. The index row has two, one for each backtracked decision. The

Lisp and register rows have no refusals at all.

A caution about reading the BASIC row is needed. Its refusals are supersessions, and a superseded value is not discarded without use: the old sum was consumed to produce the new one, and the successor depends on it. In the substructural vocabulary that is closer to a linear update than to weakening, and the large refusal count measures how often live state is replaced, not how much is thrown away. The same applies to the broader claim that mutation does not exist in the model. The ledger is immutable, but the live family Ω changes at every event, so the accurate statement is that mutation has been relocated from the history to the live set, and that what the model adds is a record of each replacement with its reason and its predecessor.

12.2 Weakening, Contraction, and Exchange

The structural vocabulary of Chapter 1 is a way of saying what each of these rows is doing. A refusal of a sphere that never contributed to a result is a weakening. The Forth `drop`, the refused branch of a conditional, the probes of a shell that missed, and the decisions of a solver that were abandoned are weakenings, and they are the natural places to look for what a language throws away. The Forth `dup` is a contraction in the sense that the new cell is coupled to the cell it copies, so one value is now used twice. Variable lookup in Lisp is a contraction as well, since a binding can be read any number of times and each read leaves the binding live. The `swap` and `rot` words are exchanges, and the prototype realizes them as the refusal of the consumed cells followed by binds of the produced cells in the permuted order.

Only refusals that discard a value without its contributing to any result count as weakening in this audit, and the supersession refusals of BASIC and of Lisp assignment are excluded from that count. The audit is not decoration. A prototype that implements a language with no weakening at all, in which every value that enters the world is used, has a different substructural character than a language in which values may be discarded, and a faithful reconstruction should reproduce that character. The test that reveals the difference is simple: count the refusals whose reason is a discard and compare with the language's specification. A Forth reconstruction in which `drop` did not refuse a cell would leave it live and would fail the live-accounting proposition, and the check is therefore a consequence of the laws of the kernel and not an extra obligation.

12.3 The Cost of History

The last column of the table is the measure of a cost that Chapter 2 introduced. The Lisp row ends with nine hundred and eighty-five live spheres, because the prelude's definitions, the environments, and the closures stay live for as long as a binding refers to them, and the prototype has no garbage collection. The ledger itself is never collected, by design, since the ledger is the evidence. The two statements are different. The set of live spheres could be reduced by a collector that refuses spheres that no live sphere depends on, with the reason that

they are unreachable, and that collector would itself be an application of the primitives and would leave a record. The ledger could be compressed by a policy that keeps only commitments, at the price that Chapter 2 states. The question for a long-running service is whether the ledger can be bounded, and the honest answer is that bounding it gives up part of what it was kept for. A checkpoint that records the digest of a prefix and then discards the prefix preserves the ability to verify that later events extend it, and loses the ability to replay or explain anything before the checkpoint. This is the same trade that event-sourced systems make with snapshots, and the prototypes here make no attempt to choose a point on it. The table suggests the order of magnitude of the difference. A short Lisp computation produces nearly two thousand events and a short Forth computation produces fewer than two hundred, and the ratio reflects that the Lisp prototype stores environments and closures as spheres and the Forth prototype stores a stack.

12.4 What Has and Has Not Been Shown

Three claims have been argued in the book and they should be separated. The first is that the four primitives are enough to implement the state of each of these languages, and the evidence is the prototypes themselves, which use only the kernel's interface. The second is that the prototypes are adequate at their observation boundaries, and the evidence is of two kinds: proofs for the register and for Forth, which are short because the prototypes were built to make them short, and differential tests against independent references for those two, exhaustive for the register and randomized for Forth. The third is that the models of the shells, BASIC, and the package index are faithful to the documented rules of the systems they represent. For these the evidence is the tests that assert specific documented behaviors, and no more than that. A statement about how a real shell or a real index behaves in every case is not supported by a prototype of this size, and where the book makes a modeling choice, such as exact name matching for a Windows executable invoked from WSL, it says so.

12.5 Open Problems

Several extensions would test the method more severely than the work here does. A hygienic macro expander would show whether the ledger's lineage is enough to implement renaming. A type system, even a simple one, would introduce a new kind of refusal, one made before execution on the basis of a static judgment. A language with first-class continuations would stress the claim that activation records are spheres, since a continuation may be resumed after the spheres it refers to have been popped. A concurrent language would require the kernel's treatment of independent events to account for interleavings. The rewriting systems of the combinatory and circle calculi that the earlier essays treat would be the natural next family of prototypes, since their central difficulty, duplication, has an explicit counterpart in the contraction of a shared sphere. Finally, the package index and search path could be extended with the one feature of real systems that the prototype omits for brevity, which is signed provenance of releases, and the ledger's hash chain suggests a design in which the provenance

of an installed module is verified by replay.

Exercise 12.1. Add a mark-and-refuse garbage collector for the Lisp prototype that refuses unreachable binding and closure spheres with the reason `unreachable`. Rerun the profile and report how the live count changes. Decide whether the collector's own events are counted in the profile.

Exercise 12.2. Choose a language not in this book and predict the shape of its profile row before implementing it, giving the expected ratios of refusals to binds. Implement it, compare, and explain any difference in terms of weakening and contraction.

Exercise 12.3. State a precise sense in which the Forth word `dup` is a contraction and the Lisp variable read is a contraction, and show that the two differ in whether the copy is a new sphere or a new use of an existing one.

Appendix A

The Reference Kernel and the Search Function

The kernel is reproduced in full from the file that all prototypes import. It is followed by the ordered search function that the shell, the package index, and the Racket collection path share.

```
"""Reference kernel: four primitives over an append-only ledger.

Pop commits a live sphere. Refuse excludes a live sphere with a recorded
reason. Bind introduces a sphere coupled to its dependencies without
consuming them. Collapse projects the world under a rule and records the
commitment. All language state in later chapters lives in a World and is
read only through views.
"""
import hashlib
from collections import Counter
from dataclasses import dataclass
from typing import Any, Tuple

LIVE, POPPED, REFUSED = "live", "popped", "refused"


class Inadmissible(Exception):
    """A primitive was proposed outside the admissible region."""


@dataclass(frozen=True)
class Sphere:
    id: int
    tag: str
    key: Any
    payload: Any
    deps: Tuple[int, ...]


def digest(value):
    return hashlib.sha256(repr(value).encode()).hexdigest()[:16]
```

```
class View:
    """Read-only projection of a world (the pure part of Collapse)."""

    def __init__(self, w):
        self._w = w

    def live(self, tag):
        return list(self._w._tag.get(tag, ()))

    def live_key(self, tag, key):
        return list(self._w._key.get((tag, key), ()))

    def last(self, tag):
        d = self._w._tag.get(tag)
        return next(reversed(d)) if d else None

    def first(self, tag):
        d = self._w._tag.get(tag)
        return next(iter(d)) if d else None

    def last_key(self, tag, key):
        d = self._w._key.get((tag, key))
        return next(reversed(d)) if d else None

    def count(self, tag):
        return len(self._w._tag.get(tag, ()))

    def sphere(self, i):
        return self._w.spheres[i]

    def payload(self, i):
        return self._w.spheres[i].payload

    def status(self, i):
        return self._w.status[i]

    def reason(self, i):
        return self._w.reasons.get(i)

    def size(self):
        return len(self._w.ledger)

    def ancestors(self, i):
        seen, stack = set(), [i]
        while stack:
            for d in self._w.spheres[stack.pop()].deps:
                if d not in seen:
                    seen.add(d)
                    stack.append(d)
        return seen
```

```
class World:
    def __init__(self):
        self.ledger = []
        self.spheres = {}
        self.status = {}
        self.reasons = {}
        self._tag = {}
        self._key = {}
        self.view = View(self)

    # the four primitives
    def bind(self, tag, key=None, payload=None, deps=()):
        sid = len(self.spheres)
        self._commit(("bind", sid, tag, key, payload, tuple(deps)))
        return sid

    def pop(self, sid):
        self._commit(("pop", sid))
        return self.spheres[sid].payload

    def refuse(self, sid, reason):
        self._commit(("refuse", sid, reason))

    def collapse(self, rule, name=None):
        value = rule(self.view)
        self._commit(("collapse", name or getattr(rule, "__name__", "rule"), digest(value)))
        return value

    # admissibility and application of events
    def _commit(self, e):
        k = e[0]
        if k == "bind":
            if e[1] != len(self.spheres):
                raise Inadmissible("sphere id out of order")
            for d in e[5]:
                if d not in self.spheres:
                    raise Inadmissible("unborn dependency %r" % d)
        elif k in ("pop", "refuse"):
            if self.status.get(e[1]) != LIVE:
                raise Inadmissible("%s of non-live sphere %r" % (k, e[1]))
            if k == "refuse" and not (isinstance(e[2], str) and e[2]):
                raise Inadmissible("refusal requires a reason")
        elif k != "collapse":
            raise Inadmissible("unknown event %r" % (k,))
        self.ledger.append(e)
        if k == "bind":
            s = Sphere(e[1], e[2], e[3], e[4], e[5])
            self.spheres[s.id] = s
            self.status[s.id] = LIVE
            self._tag.setdefault(s.tag, {})[s.id] = None
            self._key.setdefault((s.tag, s.key), {})[s.id] = None
        elif k in ("pop", "refuse"):
            s = self.spheres[e[1]]
            self.status[s.id] = POPPED if k == "pop" else REFUSED
```

```
            if k == "refuse":
                self.reasons[s.id] = e[2]
            del self._tag[s.tag][s.id]
            kk = (s.tag, s.key)
            del self._key[kk][s.id]
            if not self._key[kk]:
                del self._key[kk]

    # evidence and acceptance
    def chain(self):
        h = ""
        for e in self.ledger:
            h = hashlib.sha256((h + repr(e)).encode()).hexdigest()
        return h[:16]

    def counts(self):
        return Counter(e[0] for e in self.ledger)

    @classmethod
    def replay(cls, events, rules=None):
        w = cls()
        for e in events:
            if e[0] == "collapse" and rules and e[1] in rules:
                if digest(rules[e[1]](w.view)) != e[2]:
                    raise Inadmissible("collapse digest mismatch for %s" % e[1])
            w._commit(e)
        return w

    def verify(self, rules=None, expected_chain=None):
        w2 = World.replay(self.ledger, rules)
        ok = (w2.status == self.status and w2.reasons == self.reasons
              and {t: list(d) for t, d in w2._tag.items()} == {t: list(d) for t, d in self.
    _tag.items()})
        if expected_chain is not None:
            ok = ok and w2.chain() == expected_chain
        if not ok:
            raise Inadmissible("replay disagrees with indexed state")
        return True


# derived forms
def supersede(w, old, payload, reason="superseded"):
    """Mutation as documented refusal of the old sphere plus a coupled new one."""
    s = w.spheres[old]
    w.refuse(old, reason)
    return w.bind(s.tag, s.key, payload, (old,))


def choose(w, taken, rejected, reason="not-taken"):
    """Choice(A,B) = Pop(A) parallel Refuse(B)."""
    w.pop(taken)
    for r in rejected:
        w.refuse(r, reason)
```

```
def attempt_refused(w, tag, payload, reason):
    """Documented refusal of an inadmissible proposal."""
    i = w.bind(tag, None, payload)
    w.refuse(i, reason)
    return i


def apply_via(w, rule, fn, arg, name="subst"):
    """Application = Collapse_rule(Bind(fn, arg))."""
    c = w.bind("app", None, None, (fn, arg))
    value = w.collapse(lambda v: rule(v.payload(fn), v.payload(arg)), name)
    return c, value
```

```
"""Ordered search, the one operation shared by PATH lookup, collection paths and device
    lookup."""


def search(w, name, scopes, probe, tag="probe"):
    """Probe scopes in order. Each probe is a sphere: a miss is refused with its
    reason, the hit is popped (committed). Returns (target or None, tried)."""
    tried = []
    for sc in scopes:
        outcome, target = probe(sc, name)
        p = w.bind(tag, None, (sc, name, outcome))
        tried.append((sc, outcome))
        if outcome == "hit":
            w.pop(p)
            return target, tried
        w.refuse(p, "missing" if outcome == "missing" else outcome)
    return None, tried
```

Appendix B

Reproducing the Results

The source tree has five directories. The directory `spop` holds the kernel and the prototypes, `rkt` holds the Racket modules used in Chapters 6 and 7, `tests` holds the suites, `tools` holds the scripts that generate the transcripts and cut the listings, and `book` holds the fragments of the text. The tests are run with the standard library test runner, from the root of the tree, with the command `python3 -m unittest discover -s tests -t .`. The transcripts and the profile table are regenerated by `python3 tools/transcripts.py`, the listings are regenerated by `python3 tools/excerpts.py`, and the script `tools/build.sh` runs both and then assembles and compiles this document, so that every listing and every transcript in the text is produced from the code that the tests exercise. A listing that no longer matches the source cannot survive a rebuild, since it is cut from the source on every build.

Bibliography

[1] Charles H. Moore. Forth: A New Way to Program a Mini-Computer. *Astronomy and Astrophysics Supplement*, 15:497–511, 1974.

[2] Leo Brodie. *Starting Forth*. Prentice Hall, second edition, 1987.

[3] American National Standards Institute. *Programming Languages: Forth*. ANSI X3.215-1994, 1994.

[4] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[5] Peter J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, 1964.

[6] Harold Abelson and Gerald Jay Sussman, with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996.

[7] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010.

[8] Matthew Flatt. Composable and Compilable Macros: You Want It When. In *Proceedings of the ACM International Conference on Functional Programming*, pages 72–83, 2002.

[9] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic Macro Expansion. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 151–161, 1986.

[10] David A. Turner. A New Implementation Technique for Applicative Languages. *Software: Practice and Experience*, 9(1):31–49, 1979.

[11] Moses Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924.

[12] Alonzo Church and J. Barkley Rosser. Some Properties of Conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[13] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[14] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, 2008.

[15] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[16] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[17] Commodore Business Machines. *Commodore 64 Programmer's Reference Guide*. Howard W. Sams, 1982.

[18] Chet Ramey and Brian Fox. *Bash Reference Manual*. Free Software Foundation, current edition.

[19] Russ Cox. Surviving Software Dependencies. *Communications of the ACM*, 62(9):36–43, 2019.

[20] Flyxion. Reconstructing a Forth Stack in SpherePOP. Unpublished essay, 2026.

[21] Flyxion. The SpherePOP Specification. Unpublished essay, 2026.

[22] Flyxion. Scoped Structural Rules in Combinatory Reduction. Unpublished essay, 2026.

[23] Flyxion. Duplication and Universality in the Circle Calculus. Unpublished essay, 2026.