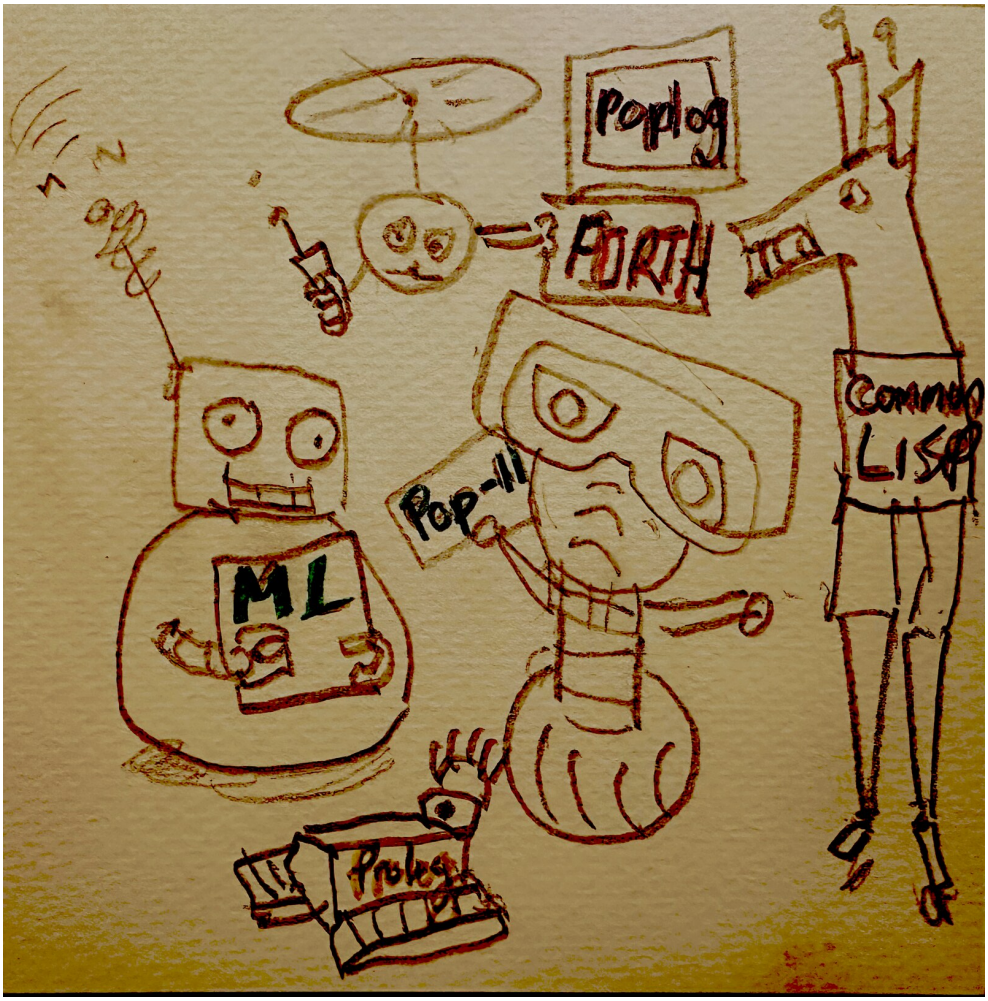
Pop-11 and the Robot Army

Poplog as the engine behind AI agents and the machines they command

David J. Kordsmeier and Claude



Cover art by David J. Kordsmeier · Edition 1.0, September 2026

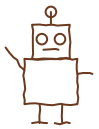
Authors. Written by David J. Kordsmeier and Claude, Anthropic's AI model, working together in Claude Code: the framing, the examples and the verification were done in one live session against the source tree, with Claude drafting and running and Kordsmeier directing, reviewing and drawing.

Five robots, one engine. The languages on the cover — Pop-11, Prolog, Common Lisp, Standard ML and Forth — are the five front-ends of the Poplog system, and this book is about using them, from a live session driven by an AI agent, to run a fleet. It accompanies the source tree at `github.com/IoTone/poplog`, a maintained fork of the Poplog system developed at the University of Sussex. Every listing in it was executed against that tree on Apple Silicon while the text was written; the outputs shown are the outputs obtained, and the full programs live at `examples/robotarmy/` in the repository.

Contents

CHAPTER 1

The premise, and why Poplog



You have a fleet. Scouts, sappers, medics, haulers — a few today, a few hundred later — each one reporting telemetry, each one waiting for orders. Above them sits an AI agent that reads the reports, decides, and issues the orders; and that agent needs a tool that can hold the whole picture of the fleet in one place, change its own behaviour while running, and never make the fleet wait on a build.

That tool, in this book, is Poplog. Not because it is the newest thing to hand but because three properties it has had since the 1980s turn out to be exactly what an agent-driven command post wants.

- **The compiler is a procedure in the running system.** Defining a new behaviour — a new order, a new triage rule — costs about fifteen microseconds and produces native machine code. An agent can extend its own tooling mid-mission.
- **The whole session is a value.** The heap — the fleet registry, every compiled procedure, every rule learned so far — can be written to a file in milliseconds and restored in a new process. A command post that survives a restart, or forks, is a file copy.
- **Five languages share it.** Pop-11 is the core; Prolog, Common Lisp, Standard ML and Forth are compiled onto the same virtual machine and live in the same heap. The chain of command is a Prolog relation; the robots' control words are Forth; the typed sensor model is ML; and one `?-` at the Pop-11 prompt reaches any of them.

Poplog is not a language. It is a language toolkit that ships with five languages already built on it, and this chapter is about the machinery that makes the three properties above true.

1.1 The two-level virtual machine

Poplog's central idea is a split that lets language implementation and machine porting proceed independently.

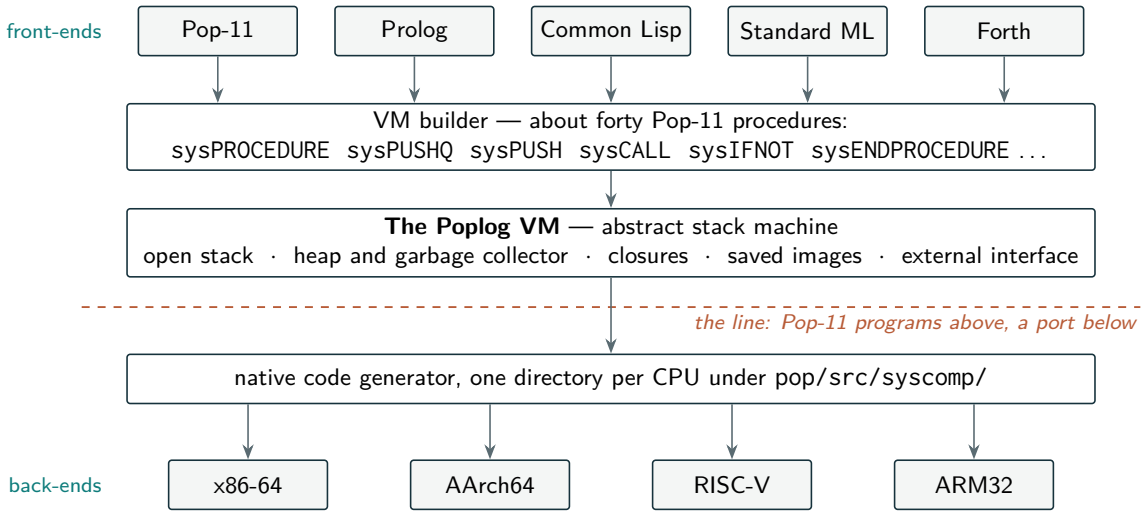
- **Above the line**, a language front-end reads source and *plants* instructions for an abstract stack machine — the Poplog VM. It does this by calling about forty ordinary Pop-11 procedures (`sysPUSHQ`, `sysCALL`, `sysPROCEDURE`, ...). A front-end is a Pop-11 program. It contains no assembly and no knowledge of any CPU.
- **Below the line**, a back-end translates planted VM code into real instructions for the host architecture. This is where a port lives: the x86-64, AArch64, RISC-V and ARM32 code generators are each one directory under `pop/src/syscomp/`.

The two halves never meet. A language hosted on the VM inherits every port for free — the Forth on the cover was written once, in Pop-11, and runs natively on Apple Silicon, on a Raspberry Pi 5 and on a StarFive RISC-V board with no per-platform work. That matters for a fleet, because the robots and the command post will not be the same kind of machine.

1.2 Incremental compilation, measured

"Incremental" here means what it says: the compiler is a procedure in the running system, and compiling a definition is an ordinary call. There is no build step, no link step, and no separate compile-time world.

```
vars t0 = systime(), k;
for k from 1 to 10000 do
```



The two-level VM (§1.1). A front-end plants abstract instructions through the VM builder; a back-end turns them into machine code. Add a language and every port gets it.

```
    pop11_compile(stringin('define gen_' sys_>< k sys_><
                           '(x); x + ' sys_>< k sys_>< ' enddefine;'));
endfor;
npr('10000 definitions compiled in ' sys_>< ((systime() - t0) * 10) sys_>< ' ms CPU');
gen_9999(1) =>
```

```
10000 definitions compiled in 150 ms CPU
** 10000
```

Fifteen microseconds per definition, each one a real, callable, garbage-collected, natively compiled procedure. When compilation is that cheap you stop treating it as an event: an agent that has just learned a better triage rule from the last hour's telemetry defines it and moves on.

For calibration, the project's benchmark suite compares the same workloads against CPython on identical hardware (`BENCHMARKS.md` in the tree has the methodology and the full matrix):

Workload	Poplog (x86-64)	Python 3.13	Python 3.10
`nfib29` — 1.35M recursive calls	14.5 ms	3.67× slower	8.56×
`intloop10M` — tight integer loop	61.5 ms	5.60×	6.26×
`lists` — 1M cons cells, allocate + walk	11.4 ms	7.20×	7.76×
`closures1M` — 100k closures, 1M calls	22.0 ms	2.13×	3.00×
`compile500` — runtime-compile ×500	66.7 ms	0.12×	0.08×

The last row is the honest one: Python wins it, because `compile()` produces bytecode while Poplog produces machine code. Poplog is doing strictly more work and still lands within an order of magnitude. On the other rows — calls, loops, allocation, closures — the native code shows.

1.3 The saved image

Because the VM owns the whole heap, the whole heap can be written to a file and read back. `syssave` does exactly that; the resulting `.psv` image contains your data *and* your compiled procedures.

```
$ popsession checkpoint post.psv          # 10,000 procedures live in the heap
  2.9M post.psv
$ popsession restore  post.psv            # a fresh process, 64 ms including spawn
```

```
$ gen_9999(1) =>
** 10000
```

This is how the shipped languages arrive: `clisp.psv`, `prolog.psv` and `pml.psv` are each an image of a Pop-11 system that has already compiled a language on top of itself. It is also what makes a command post durable: the fleet registry and every order-handling procedure survive a restart together, as one file.

> **What is *not* in the image**
> Heap state restores; operating-system state does not. Open sockets to the robots, open files, and pointers into C libraries are stale after a restore. The discipline is simply to reopen them — Chapter 7 returns to this.

1.4 Where it runs

OS	Architecture	Status
Linux	x86-64	Supported — reference platform
Linux	AArch64	Supported — Raspberry Pi 5, MediaTek Genio 720
macOS	Apple Silicon	Supported — native Mach-O, C ↔ Pop callbacks
Linux	RISC-V RV64GC	Supported — self-hosting on StarFive VisionFive
Linux	ARM32 (v6/v7)	Supported — long-standing port, Raspberry Pi 1–3
Solaris / FreeBSD	x86 / x86-64	Supported — upstream ports
Windows	x86-64	Not yet ported (WSL2 runs the Linux build)

"Supported" means it builds, self-hosts, and runs all the languages. The first four rows are benchmarked in this fork — and the Pi and the RISC-V board are the kind of hardware a robot actually carries.

1.5 Further reading

Poplog has a long paper trail, and much of it is still the best explanation of why the system is shaped the way it is.

- **Where it came from.** Aaron Sloman, *The Evolution of Poplog and Pop-11 at Sussex University* (1989), in *POP-11 Comes of Age*, ed. J. A. D. W. Anderson — the first-hand account, from POP-2 on an Elliott 4130 in Edinburgh in 1972 to the Sussex system: `cogaffarchive.org/sloman.pop11.pdf`. Wikipedia's `POP-2`, `POP-11` and `Poplog` entries are the short versions, with the Burstall and Popplestone lineage.
- **Learning the language.** The Pop-11 Primer, the standard text for experienced programmers, and the Birmingham teaching corpus (the TEACH files that `teach` serves in-system) are preserved at the Poplog Archive, `poplogarchive.getpoplog.org`; `tools/fetch-learning.sh` in this tree pulls them in (§8.2). Stephen Leach's *Hidden Gems of Pop-11* is a short tour of what the language does that others do not, and a good next read after Chapter 3.
- **The system itself.** `ref vmcode` and `ref subsystem` are the VM and subsystem specifications behind the diagram in §1.1. `hebisch/poplog` is the cleaned-up upstream source this fork descends from; `GetPoplog` maintains the archive and packaged distributions.

CHAPTER 2

Quickstart

2.1 Getting a Poplog

Three ways in, fastest first.

1. A relocatable binary (about a 2 MB download). This unpacks a ready-built `basepop11` plus libraries under `~/.local/share/pop11-skill`, and finishes with a live smoke test:

```
curl -fsSL https://raw.githubusercontent.com/IoTone/poplog/master/tools/install-skill.sh | sh
```

2. From source. Poplog bootstraps from a small seed binary, `corepop`, which is vendored for the ported platforms:

```
git clone https://github.com/IoTone/poplog && cd poplog
./configure && make all
```

`make all` builds the core system, then compiles the Prolog, Common Lisp and Standard ML front-ends and saves each as an image under `target/psv/`. See `INSTALL` for the details.

3. Nix. A flake builds and bootstraps everything from source with no manual seed or toolchain setup:

```
nix run .#pop11          # or .#prolog / .#clisp / .#pml
nix shell .#poplog       # put all five front-ends on $PATH
```

2.2 The first session

```
$ ./poplog pop11
Sussex Poplog (Version 16.019997 ...)
Copyright (c) 1982-1999 University of Sussex.  All rights reserved.

Setpop
: 6 * 7 =>
** 42
```

The prompt is `:` and the workhorse is `=>`, which prints whatever the preceding expression left, prefixed with `**`. Two more you will use constantly: `npr(x)` prints `x` and a newline, and `==>` is `=>` with fuller structure printing.

Statements end with a semicolon. An expression followed by `=>` does not need one — `=>` terminates it.

When something goes wrong you get a *mishap*: a diagnostic naming the error, the culprits, and the chain of procedures that was running.

```
: 'three' + 4 =>
;;; MISHAP - NUMBER(S) NEEDED
;;; INVOLVING:  'three' 4
;;; PRINT DOING
;;; DOING    :  + pop_setpop_compiler
```

The session survives. Fix the expression and carry on — this is the normal working rhythm, not an exceptional one.

2.3 Running files

A Pop-11 source file is just a sequence of statements, so there is no `main`:

```
./poplog basepop11 myprog.p        # compile and run, then enter the REPL
./poplog basepop11 myprog.p </dev/null   # ...or run and exit
```

Inside a session, `load 'myprog.p';` does the same thing, and `uses foo;` loads library `foo` from the library search list if it is not already loaded — this is how you reach `lib json`, `lib crypto`, `objectclass`, and the rest.

2.4 Documentation, in the system

Poplog ships its own manual — over 900 files — and it is queryable from the prompt. The three kinds matter:

- `help name` — practical, task-shaped. Start here.
- `ref name` — the exhaustive reference for a primitive or a subsystem.
- `teach name` — tutorials, many of them the original Birmingham AI course material.

```
: help matches
: ref regexp
: teach json
```

The same corpus is published as a website by a Pop-11 program in the tree (`tools/gen-docs.sh`) at `https://iotone.github.io/poplog/`, with an `llms.txt` for AI assistants.

2.5 Editing and long-lived sessions

Because the interesting object is a *running heap*, the tooling in this fork is built to talk to one rather than to a file.

- **VED**, the built-in editor: `ved myfile.p`. `ENTER l1` compiles the current line into the live system, `ENTER lmr` the marked range, `ENTER lcp` the current procedure.
- **LSP** (`tools/pop11-lsp`): diagnostics come from the real compiler with `pop_syntax_only` set, so your file is type-checked by the thing that would compile it, and nothing in it runs. A Neovim plugin ships in `editors/nvim/`.
- **VS Code** and **Zed** (`editors/vscode/`, `editors/zed/`): syntax highlighting, outline and bracket support for `.p`, `.pop11` and `.ph`, both built on the `tree-sitter-pop11` grammar. VS Code installs from the `pop11-vscode-v.vsix` on the releases page; Zed as a dev extension pointed at `editors/zed`. Either client can run the LSP server above for real-compiler diagnostics. An Emacs mode with a live listener is in `editors/emacs/`.
- **Swank** (`tools/pop11-swank`): the other half — a live session an editor can attach to. Output streams as it is produced, mishaps arrive as data with real frames, and a runaway loop can be interrupted. The Emacs package in `editors/emacs/` is the client; `teach swank` is the walkthrough.
- **MCP** (`tools/pop11-mcp`): the same idea for agents, and the way the general of the Robot Army actually gives orders. A persistent, natively compiled session exposed as four tools — `pop11_eval`, `pop11_help`, `pop11_checkpoint`, `pop11_state`. Register it with `claude mcp add -scope project pop11 - tools/pop11-mcp` and the agent has the command post in its hands.

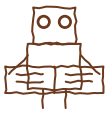
All three servers sit on one transport, `pop/lib/lib/jsonrpc.p`, itself written in Pop-11.

The working pattern

Start one session and keep it. Define helpers once; call them thereafter. Redefinition is instant and applies to every later call, so a procedure is something you *sculpt* in a live image rather than edit-compile-run. Checkpoint the image when the session has earned something worth keeping.

CHAPTER 3

Pop-11 fundamentals



Pop-11 reads like an Algol-descended imperative language and behaves like a Lisp with a stack machine underneath: infix syntax, block keywords that always close, dynamic typing, garbage collection, first-class procedures, an extensible reader. What makes it distinctive is the *open stack*, and almost everything idiomatic follows from it. The examples in this chapter are the pieces of `examples/robotarmy/fleet.p`, the core of the Robot Army.

3.1 Syntax at a glance

```
;;; comments run from three semicolons to end of line
vars charge = 100;            ;;; a global (permanent) variable
charge - 10 -> charge;        ;;; ASSIGNMENT POINTS RIGHT: value -> destination
npr('charge ' sys_>< charge); ;;; print a line; sys_>< concatenates anything
charge =>                     ;;; the REPL's printer: "** 90"
```

The assignment arrow is the first thing to internalise: `-> x` reads "into x". It points that way because assignment is a stack operation — the value is already there, and the arrow says where to put it.

3.2 Words and strings

```
'advance to ridge'   ;;; single quotes: a mutable vector of characters
"r1"                 ;;; double quotes: a WORD -- an interned symbol, not a string
```

This is the single most common beginner error, and it has a silent cousin. Words are interned: every `"r1"` is the same object. Strings are not: two `'r1'`s are two objects that merely print alike. Poplog's default hash table, `newproperty`, matches keys by *identity*, so a registry keyed by unit id must use words:

```
vars fleet = newproperty([], 64, false, true);
90 -> fleet("r1");
fleet("r1") =>            ;;; ** 90
fleet('r1') =>            ;;; ** <false>  -- a different object, silently
```

Use words as keys, or use `newmapping`, which compares with `=`. Both are right; silently missing is not.

3.3 Lists, and the quoting trap

An order in the Robot Army is a list of words, and list brackets *quote* their contents, like Lisp's `'(...)`:

```
vars order = [unit r1 advance to ridge];
order(2) =>               ;;; ** r1
```

So `[unit u advance to target]` is a list of the literal words `unit u advance to target`. To build an order from values, escape with `^` (one item) or `^^` (splice a list):

```
vars u = "r4", target = "ridge", squad = [r1 r2];
[unit ^u advance to ^target] =>     ;;; ** [unit r4 advance to ridge]
[all ^^squad hold] =>               ;;; ** [all r1 r2 hold]
```

Vectors, `{1 2 3}`, are the constant-time-subscript counterpart.

3.4 Procedures

A procedure's result is whatever it leaves behind; there is no `return` statement, and the last expression is the value. A procedure may leave *several* results, which the caller collects in reverse order:

```
define split_order(o); hd(o), tl(o) enddefine;
vars verb, rest;
split_order([advance to ridge]) -> rest -> verb;   ;;; verb = advance, rest = [to ridge]
```

Closures are made by partial application with `%. . . %`:

```
define drain(charge, n); charge - n enddefine;
vars step_cost = drain(% 2 %);      ;;; every step costs 2%
step_cost(100) =>                   ;;; ** 98
```

And every accessor has an *updater* — the procedure that runs when it appears on the left of an arrow. That is why `roster(2)` both reads and writes:

```
vars roster = [r1 r2 r3];
"r9" -> roster(2);
roster =>                           ;;; ** [r1 r9 r3]
```

3.5 The open stack

There is one user-visible stack, and the calling convention does not hide it. A procedure can leave zero, one, or a hundred results, and the number need not be known until it runs. `fleet.p` uses this for the roster: `units()` walks the registry and leaves every robot on the stack, and the *caller* chooses the container.

```
define units();
    appproperty(fleet, procedure(k, v); v endprocedure);
enddefine;

define muster();  [% units() %]  enddefine;       ;;; a list of every unit

define fit_for_duty();
    lvars r;
    [% for r in muster() do
           if rb_charge(r) >= 25 then r endif       ;;; leave it, or don't
       endfor %]
enddefine;
```

`[% . . . %]` means "run this, and make a list of everything it left". That one construct replaces most of what other languages need generators, varargs, spread operators and comprehensions for. It also makes Forth nearly free to host (§5.4): Forth's data stack *is* this stack.

> **Why this matters for performance**
> Because results are passed on the stack rather than boxed into tuples, multiple return values cost nothing, and neither does a procedure that sometimes returns nothing. The `nfib` and `closures1M` benchmark rows in Chapter 1 are measuring this calling convention.

3.6 Control structures

Every opener has a matching closer.

```
if charge < 25 then recall() elseif charge < 60 then light_duty() else deploy() endif;
unless linked then hail() endunless;
for r in muster() do ... endfor;    for i from 1 to n do ... endfor;
```

```
while cond do ... endwhile;          repeat ... quitif(done); ... endrepeat;

define readiness(charge);
    switchon charge
        case < 25 then 'return to base'
        case < 60 then 'light duty only'
        else 'fit for duty'
    endswitchon
enddefine;
readiness(20) =>                     ;;; ** return to base
```

3.7 Scope, and `dlocal`

`vars` declares a permanent (global) identifier; `lvars` declares a lexical local inside a procedure — prefer `lvars`. `constant` and `lconstant` are their immutable counterparts.

`dlocal` has no close equivalent elsewhere: it gives a variable a new value for the *dynamic extent* of a call and restores the old value on exit — whether the procedure returns, jumps out, or is unwound by a mishap. Here an order is relayed down three echelons, and the indent follows it:

```
vars echelon = '';
define relay(order, depth);
    dlocal echelon = echelon >< '> ';
    npr(echelon sys_>< order);
    if depth > 1 then relay(order, depth - 1) endif;
enddefine;
relay('advance', 3);
```

```
> advance
> > advance
> > > advance
```

After `relay` returns, `echelon` is empty again. The same mechanism is how Poplog rebinds output streams and the compiler's own state safely — it is Pop-11's answer to `try. . . finally`, and it is declarative.

3.8 The list matcher

Pop-11's pattern matcher works on lists and is built into the language with the `matches` operator: `?x` binds one item, `??x` binds a segment, `=` matches one item without binding. It is what turns an order into an action in `fleet.p`:

```
vars id, place, n;            ;;; pattern variables must be permanent (vars)

define obey(order) -> reply;
    lvars r;
    'unintelligible' -> reply;
    if order matches [unit ?id advance to ?place] then
        fleet(id) -> r;
        if r then
            place -> rb_place(r);
            rb_charge(r) - 10 -> rb_charge(r);
            'unit ' sys_>< id sys_>< ' advancing to ' sys_>< place -> reply
        else 'no such unit: ' sys_>< id -> reply endif
    elseif order matches [unit ?id recharge ?n] then
        ...
    elseif order matches [report] then
        '' sys_>< length(fit_for_duty()) sys_>< ' of ' sys_><
        length(muster()) sys_>< ' units fit for duty' -> reply
    endif;
```

```
enddefine;
```

```
[report]  ->  3 of 4 units fit for duty
[unit r1 advance to ridge]  ->  unit r1 advancing to ridge
[unit r9 advance to ridge]  ->  no such unit: r9
```

> **A trap worth naming**
> The matcher assigns through the *permanent* identifier, so pattern variables declared with `lvars` are silently not filled in — you get `<undef>`. Declare them `vars`.

The same matcher, plus a dozen lines of ranking and reassembly, is the whole of ELIZA; `examples/eliza.p` in the tree is a faithful implementation of Weizenbaum's original.

3.9 Objects: Objectclass

Objectclass is Pop-11's CLOS-like object system: classes with slots, and generic procedures with methods dispatched on argument class. A robot is one:

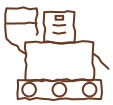
```
uses objectclass;

define :class Robot;
    slot rb_id     = "unknown";
    slot rb_kind   = "scout";      ;;; scout | sapper | medic | hauler
    slot rb_charge = 100;
    slot rb_place  = "base";
enddefine;

define :method print_instance(r:Robot);
    printf('<%p %p  %p%%  at %p>\n',
           [% rb_kind(r), rb_id(r), rb_charge(r), rb_place(r) %]);
enddefine;

define enlist(id, kind) -> r;
    newRobot() -> r;
    id -> rb_id(r);  kind -> rb_kind(r);
    r  -> fleet(id);
enddefine;
```

```
<scout r1  90%  at base>
<sapper r2  100%  at base>
```

Slots come with accessors *and* updaters for free, which is why `rb_charge(r)` works on both sides of the arrow in `obey`.

CHAPTER 4

Running the fleet



Four programs that do the fleet's real work. Each was run against the tree while this chapter was written; the transcripts are the actual output. The listings here are excerpts — the full programs are at `examples/robotarmy/` on GitHub, and each runs from the repository root.

4.1 Telemetry triage: regexps, done Poplog's way

Every robot streams telemetry, and the first job of the command post is to read it faster than it arrives. `examples/robotarmy/telemetry.p` makes one pass over a fleet log and answers three questions: how much of each level, which units are in trouble, and what was slowest.

Poplog ships a full regexp engine (`ref regexp`) — but with inverted syntax, and this catches everyone exactly once. **The escape character is `@`, not backslash, and `. * [ ] ^ $` are literal by default.** A pattern pasted in from any other engine compiles without complaint and then matches almost nothing.

PCRE	Poplog	PCRE	Poplog
`.`	`@.`	`x?`	`x@{0,1@}`
`*`	`@*`	`x{m,n}`	`x@{m,n@}`
`[abc]`	`@[abc@]`	`^ $`	`@^ @$`
`x+`	`x@{1,@}`	`(x)`	`@(x@)`

There is no alternation (`|`) and no lookaround: compile one pattern per branch. In exchange, `regexp_compile` hands back a *compiled procedure* you define once and then call at native speed.

```
uses regexp;
vars err, match_ms, err2, match_unit;
regexp_compile('@[0-9@]@{1,@} ms') -> (err, match_ms);     ;;; "1420 ms"
regexp_compile('r@[0-9@]@{1,@}')   -> (err2, match_unit);  ;;; "r7"

define triage(path) -> (levels, trouble, slowest, slow_unit);
    lvars dev, rep, line, lev, i, n, ms, unit;
    newproperty([], 8,  0, true) -> levels;    ;;; default 0: += needs no branch
    newproperty([], 32, 0, true) -> trouble;
    0 -> slowest;  false -> slow_unit;
    sysopen(path, 0, "line") -> dev;
    line_repeater(dev, inits(4096)) -> rep;    ;;; size the buffer generously
    repeat
        rep() -> line;
        quitif(line == termin);
        match_unit(1, line, false, false) -> (i, n);
        if i then consword(substring(i, n, line)) -> unit else false -> unit endif;
        for lev in [INFO WARN ERROR] do
            if issubstring(lev sys_>< '', 1, line) then
                levels(lev) + 1 -> levels(lev);
                if unit and lev /== "INFO" then trouble(unit) + 1 -> trouble(unit) endif
            endif
        endfor;
        match_ms(1, line, false, false) -> (i, n);
        if i then
            strnumber(substring(i, n - 3, line)) -> ms;
            if ms and ms > slowest then ms -> slowest; unit -> slow_unit endif
        endif;
    endrepeat;
enddefine;
```

```
$ ./poplog basepop11 examples/robotarmy/telemetry.p </dev/null
INFO: 5
WARN: 2
ERROR: 3

units by fault count:
  r7  2
  r5  1
  r3  1
  r4  1

slowest operation: 2750 ms (r5)
```

Two idioms are doing quiet work. `newproperty([], 8, 0, true)` gives a table whose *default is 0*, so incrementing an unseen unit needs no initialisation branch. And `line_repeater` returns a procedure that yields one line per call and `termin` at end of file — the standard Poplog shape for a stream, and the same shape `sys_obey_linerep` uses for the output of a shell command. Note `consword`: the unit id is interned so it can key a property.

4.2 The command post: a web service in pure Pop-11

The agent does not type at the Pop-11 prompt; it talks HTTP. `examples/robotarmy/c2.p` composes `lib http_server`, `lib json` and `fleet.p` into the command post: the roster as JSON, one unit's status, and orders as plain text. Unknown routes answer 404 and a mishap inside a handler answers 500 without taking the post down. The whole program is the handler.

```
define handler(req) -> resp;
    lvars path = req('path'), obj, r, u;
    if path = '/fleet' then
        http_json({% applist(muster(), unit_json) %}) -> resp;
    elseif path = '/status' then
        allbutfirst(5, req('query')) -> u;          ;;; "unit=r1"
        fleet(consword(u)) -> r;
        if r then http_json(unit_json(r)) -> resp else false -> resp endif;
    elseif path = '/order' then
        newmapping([], 4, false, true) -> obj;
        obey(words_of(req('body'))) -> obj('reply');
        http_json(obj) -> resp;
    elseif path = '/boom' then
        mishap(0, 'deliberate handler failure');    ;;; server answers 500
    else
        false -> resp;                              ;;; server answers 404
    endif;
enddefine;

http_serve_n(port, handler, count);
```

```
$ ./poplog basepop11 examples/robotarmy/c2.p 8099
$ curl -s localhost:8099/fleet
[{"charge":100,"kind":"scout","place":"base","id":"r1"},
 {"charge":100,"kind":"sapper","place":"base","id":"r2"}, ...]
$ curl -s -d 'unit r1 advance to ridge' localhost:8099/order
{"reply":"unit r1 advancing to ridge"}
$ curl -s 'localhost:8099/status?unit=r1'
{"charge":90,"kind":"scout","place":"ridge","id":"r1"}
$ curl -s -o /dev/null -w '%{http_code}\n' localhost:8099/boom
500
```

`lib json` maps JSON onto Pop-11 directly: an object becomes a `newmapping` property, an array a vector, a string a string of UTF-8 bytes, and `null` the distinguished word `json_null`, kept apart from `false`. Member order is not preserved — properties are hash tables — which is why the

keys above come out in the order they do. `teach json` walks through how the library itself was built, bug included.

4.3 Orders in plain words

An order arrives at `/order` as text and `obey` wants a list of words, so `c2.p` has a five-line bridge, and it is worth seeing because it is the whole of what "parsing" costs here:

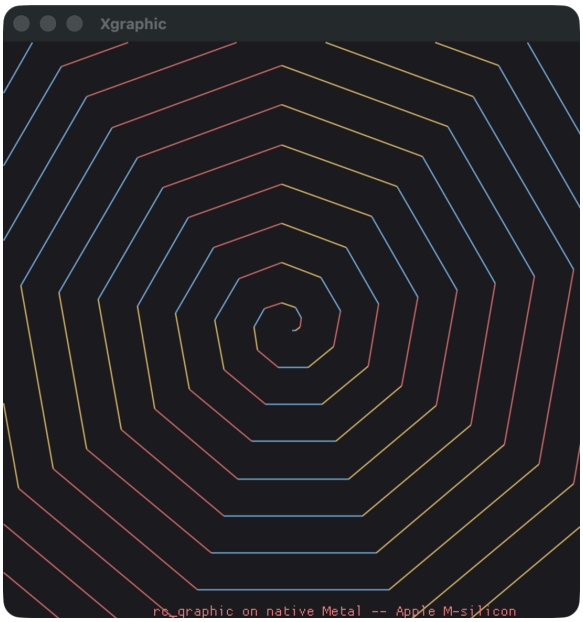
```
define words_of(s) -> l;
    lvars w;
    [% for w in str_split(str_trim(s), ` `) do
           if w /= '' then
               if strnumber(w) then strnumber(w) else consword(w) endif
           endif
       endfor %] -> l;
enddefine;
```

After that the matcher in §3.8 does the rest. Adding an order is adding an `elseif ... matches` clause — and because redefinition is instant, the agent can add one to the running post and issue it in the same breath.

4.4 The map, briefly

Poplog's graphics were historically tied to X. This fork adds an optional native backend built on Dear ImGui, chosen at build time with `./configure -experimental-graphics`: Metal + Cocoa on macOS, or SDL3 + OpenGL3 on macOS or Linux, which picks Wayland, X11, KMS/DRM or headless software rendering at run time. The classic `rc_graphic` turtle library runs on it unchanged, which is enough to draw a fleet on a map.

`examples/tenprint.p` — the 10 PRINT maze



`rc_graphic` turtle on the Metal backend

Graphics are strictly opt-in: the default build, and `nix build .#poplog`, are console-only.

4.5 The commander's image

The last example is not a program but a habit. Because compiling is microseconds and saving the heap is milliseconds, the natural unit of work is a *session* that accumulates the state of the fleet, with snapshots at the points worth returning to.

```
$ popsession start --name post                       # boot (~50 ms)
$ popsession send  --name post -f examples/robotarmy/fleet.p   # define once
$ popsession send  --name post -c "obey([report]) =>"          # ~1 ms per call
$ popsession checkpoint --name post post-0800.psv   # freeze heap + procedures
$ popsession restore    --name post post-0800.psv   # a new process, fully loaded
```

The image contains the compiled procedures, so restoring is not re-loading:

```
$ ls -lh post.psv
2.9M post.psv                # a heap holding 10,000 compiled procedures
$ time popsession restore post.psv
0.064 total                  # including process spawn
$ gen_9999(1) =>
** 10000
```

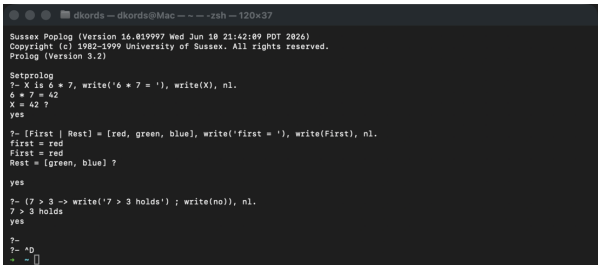
Two cautions, both from §1.3. A restore rolls the *whole* session back — anything defined after the snapshot is gone, so re-checkpoint after adding to a restored session. And handles into the operating system or into C libraries go stale across a restore; reopen them rather than trusting them.

CHAPTER 5

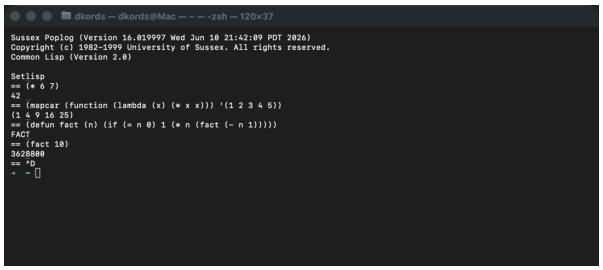
The five robots

The robots on the cover are the five front-ends of Poplog, and they are not bindings, bridges or embeddings. Each is a compiler written in Pop-11 that plants code for the same VM, and the result lives in the same heap as everything else. Switching language switches the *reader*; it does not switch worlds. In the Robot Army each language does the job it is best at: Prolog holds the chain of command, ML types the sensor model, Lisp plans, Forth runs on the robot.

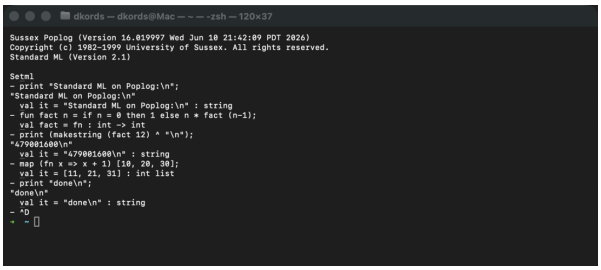
Pop-11 — the core language



Prolog — Edinburgh syntax



Common Lisp — CLTL2



Standard ML — type inference

Four front-ends, one engine, one banner — captured on the Apple Silicon build. Each image is the same `basepop11` with a different language compiled into it.

5.1 Prolog

Edinburgh-syntax Prolog, implemented as 51 Pop-11 files under `pop/plog/src/` — reader, term representation, unification, clause store and control. Because it is open Pop-11 rather than a sealed engine, extending it with new built-ins is an afternoon's work, not a fork.

```
commands(commander, sq1).      commands(sq1, r1).   commands(sq2, r3).
commands(commander, sq2).      commands(sq1, r2).   commands(sq2, r4).

% X may order Y if X commands Y directly, or commands someone who may
can_order(X, Y) :- commands(X, Y).
can_order(X, Z) :- commands(X, Y), can_order(Y, Z).
reports_to(Unit, Boss) :- commands(Boss, Unit).
obeys(Unit, From) :- can_order(From, Unit).
```

```
$ ./poplog basepop11 -target/psv/prolog.psv
?- ['examples/robotarmy/chain.pl'].
examples/robotarmy/chain.pl consulted
yes
?- can_order(commander, r4).
yes
?- reports_to(r4, Who).
Who = sq2
```

```
?- obeys(r1, sq2).
no
?- can_order(sq1, U).
U = r1   U = r2
```

The full file is `examples/robotarmy/chain.pl`. A relation, two clauses of recursion, and the question every unit must ask of every order — *is this from up my own chain?* — is answered by the engine rather than by code.

> **A small surprise**
> Poplog's Prolog is deliberately lean: list predicates such as `member/2` and `append/3` are *not* built in. Several library files under `pop/plog/lib/` define them, or you write the two clauses yourself — which, for a system whose point is that you can see and change the engine, is in keeping.

5.2 Common Lisp

Compatible with most of CLTL2. It is a genuine Common Lisp — packages, macros, `format`, the condition system — compiled to native code by the same back-end as everything else.

```
(defparameter *fleet* '((r1 scout 90) (r2 sapper 100) (r4 scout 20)))
(defun charge (u) (third u))
(defun fit-for-duty (fleet)
  (remove-if (lambda (u) (< (charge u) 25)) fleet))
(print (mapcar #'first (fit-for-duty *fleet*)))
(print (reduce #'+ (mapcar #'charge *fleet*)))
```

```
(R1 R2)
210
```

Remember that the reader upcases: `sq` and `SQ` are the same symbol, and `#`, `.`, `,`, `@` and `:` are reader macros. `help poptolisp` is a translation table for Pop-11 programmers arriving in Lisp, and `help poplisp` documents the mixed-language interface.

5.3 Standard ML

A full Standard ML with Hindley–Milner type inference, and the strongest demonstration that the VM is language-agnostic: static typing, pattern-matching function clauses and polymorphism all sit on the same dynamically typed heap.

```
datatype unit = Scout of int | Sapper of int | Medic of int;
fun charge (Scout c) = c | charge (Sapper c) = c | charge (Medic c) = c;
fun fit u = charge u >= 25;
val fleet = [Scout 90, Sapper 100, Scout 20, Medic 55];
List.filter fit fleet;
List.length (List.filter fit fleet);
```

```
  val charge = fn : unit -> int
  val fit = fn : unit -> bool
  val fleet = [Scout(90), Sapper(100), Scout(20), Medic(55)] : unit list
  val it = [Scout(90), Sapper(100), Medic(55)] : unit list
  val it = 3 : int
```

A unit that is not one of the three constructors cannot be built, and a `charge` that forgets a case is a compile error — the type-checker is where ML's surprises live, not the runtime. (Its other habit: an unannotated `fun sq x = x * x` is rejected, because `*` is overloaded and there is nothing to resolve it against.)

5.4 Forth

Those three, with Pop-11 itself, are classic Poplog. This fork adds a fifth, and it is the shortest route to understanding why hosting on the VM is cheap: **Forth's data stack is the Poplog user stack.** A Forth primitive is an ordinary open-stack Pop-11 procedure — `+` is `define f_plus(x,y); x+y enddefine` — and no data structure is interposed at all.

Colon definitions are not threaded code. `: name ... ;` is transpiled to a Pop-11 procedure and run through the incremental compiler, so a Forth word is a real native procedure; `if/else/then`, `begin/until`, `begin/while/repeat`, `do/loop`, `recurse` and `exit` map onto Pop-11 constructs at compile time.

```
variable charge   100 charge !
variable steps      0 steps !

: drain    charge @ swap - charge ! ;          \ ( n -- )  spend n% of charge
: step     steps @ 1+ steps !   2 drain ;
: low?     charge @ 25 < ;
: status   charge @ . steps @ . cr ;           \ prints: charge steps

\ walk n steps, stopping early if the battery gets low
: patrol   0 do  low? if leave then  step  loop  status ;

10 patrol
30 patrol
```

```
$ tools/forth.sh < examples/robotarmy/patrol.fth
80 10           <- ten steps, 80% left
24 38           <- asked for thirty more; stopped at 24% after 28
forth> testbench
21 passed, 0 failed   ALL OK   (0.273 ms)
```

That is `examples/robotarmy/patrol.fth`: a robot's control vocabulary in nine lines, each word a native procedure the moment it is defined.

The core covers arithmetic, about forty stack, comparison, bitwise and I/O words, native colon definitions, the control words above, counted loops with `i/j/leave`, `variable/constant/@/!`, a return stack (`>r r@ r>`), and string literals. It is younger and leaner than the mature four — no `+loop`, `create does>` or `value` yet, and it is case-sensitive. Because it is pure Pop-11 it runs on every platform Poplog does; the built-in `testbench` is 21/21 on both macOS `arm64` and RISC-V. Implementation: `pop/forth/src/forth.p`, 523 lines.

5.5 Crossing between them

All five share one image, so crossing is a change of reader, not a marshalling boundary.

From any language back to Pop-11, type `pop11`:

```
$ ./poplog basepop11 -target/psv/clisp.psv
(print (* 6 7))
42
pop11
: [a b c] <> [d] =>
** [a b c d]
```

```
$ ./poplog basepop11 +target/psv/pml.psv
val x = 6 * 7;
  val x = 42 : int
pop11
: [a b c] <> [d] =>
** [a b c d]
```

From Pop-11 into Prolog, the syntax word `?-` runs a goal inline, against the same clause store:

```
: vars order = [unit r4 advance to ridge];
: ?- ['examples/robotarmy/chain.pl'].
: ?- can_order(sq2, r4), write(yes_sq2_may_order_r4), nl.
yes_sq2_may_order_r4
yes
: ?- can_order(sq1, r4).
no
: rev(order) =>
** [ridge to advance r4 unit]
```

So the Pop-11 command post can ask the Prolog chain of command whether an order is legitimate before `obey` ever sees it.

Defining Prolog predicates in Pop-11 is a documented form (`help define_prolog`) — useful when a predicate wants a data structure or an I/O facility that is easier to reach from Pop-11:

```
define :prolog hello/0(contn);
    lvars contn;
    printf('Hello world\n');
    chain(contn);            ;;; call the continuation = succeed
enddefine;
```

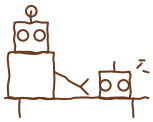
```
?- hello.
Hello world
yes
```

The continuation-passing shape is not incidental: it is how Poplog's Prolog represents "and then carry on solving", so a Pop-11 predicate that calls `chain(contn)` succeeds once, and one that calls it in a loop is a generator with backtracking.

> **Which image has which language**
>
> Each shipped image contains the languages compiled into it. `prolog.psv` has Pop-11 and Prolog; `clisp.psv` has Pop-11 and Common Lisp; `pml.psv` has Pop-11 and ML. Forth is pure library code — `uses forth` works anywhere. To get several at once, build an image that loads them, exactly as `make` builds these.

CHAPTER 6

Hosting a language of your own



Sooner or later the fleet wants its own language — a terse orders notation the agent can emit and a robot can read — and Chapter 5's front-ends are not privileged. The API they target is public, and a new language is an ordinary Pop-11 program. This chapter is the recipe; `HOSTING-LANGUAGES.md` in the tree is the longer version, and `ref vmcode` is the specification.

6.1 Plant VM code, don't emit machine code

You write a **reader**, a **parser**, and then calls to about forty **VM-builder** procedures that plant abstract instructions. The back-end turns those into native code for whatever CPU you are on.

To plant...	call
a literal constant	`sysPUSHQ(item)`
the value of a variable	`sysPUSH(word)`
a store into a variable	`sysPOP(word)`
a call of a named procedure	`sysCALL(word)`
a call of a literal procedure	`sysCALLQ(item)`
start / end of a procedure	`sysPROCEDURE(props, nargs)` ... `sysENDPROCEDURE()`
locals, dynamic locals	`sysLVARS`, `sysLOCAL`, `sysNEW_LVAR`
control flow	`sysLABEL`, `sysGOTO`, `sysIFSO`, `sysIFNOT`, `sysGO_ON`
a permanent assignment	`sysPASSIGN(item, token)`
a syntax word / macro	`sysSYNTAX`

The whole model fits in six lines. Here we build, at run time, the equivalent of `define add1(x); x fi_+ 1 enddefine`:

```
define build_add1() -> p;
    sysPROCEDURE(false, 1);      ;;; open a 1-argument procedure
        sysPUSHQ(1);             ;;;   push literal 1    -> stack: x 1
        sysCALL("fi_+");         ;;;   fast integer add  -> stack: x+1
    sysENDPROCEDURE() -> p;      ;;; p is now a real compiled procedure
enddefine;

vars add1 = build_add1();
isprocedure(add1) =>          ;;; ** <true>
pdnargs(add1)     =>          ;;; ** 1
add1(41)          =>          ;;; ** 42
```

> **Execute level**
> Planting must happen *while something is running*, not at the top level of a file being compiled — at the top level you get `sysEXECUTE: NOT AT EXECUTE LEVEL`. Wrap the plant sequence in a procedure and call it, as above.

That is the entire trick. A compiler for your language is a tree walk that emits the right `sys...` calls. What you get for free, without writing a line of assembly: native code generation on every port, the garbage collector, the open stack, closures, recursion, the incremental compiler, the external interface, the REPL, and saved images.

For the reader, you can reuse Pop-11's itemiser (`incharitem` over `proglist`, documented in `ref popcompile`) with your own character table — for a whitespace-delimited language like Forth it needs almost no configuration — or write your own, as Prolog does with its operator-precedence

reader.

6.2 Three tiers of effort

Tier	What you get	Effort
1. Embedded interpreter	A plain Pop-11 program: a dictionary and an eval loop. No VM compilation.	days
2. VM-compiling front-end	Parse, then plant — genuinely native definitions, closures, recursion, full interop. *The Poplog way.*	1–3 weeks
3. Full subsystem	Tier 2 plus registration: its own REPL, prompt, banner, file extension and saved image.	+1–3 days

The tiers stack. Build tier 1 as a throwaway to pin the semantics with no compiler in the way, replace its core with planting, then wrap the subsystem around it.

6.3 Tier 3: registering a subsystem

A subsystem is a record in `sys_subsystem_table` with a name, a file extension, a prompt, a title, search lists, and an eight-slot procedure vector in a fixed order (`ref subsystem`):

```
constant myforth_subsystem_procedures =
    {% forth_compile,     ;;; SS_COMPILER   <- the only substantial one
       identfn,           ;;; SS_RESET      (between top-level reads)
       identfn,           ;;; SS_SETUP      (on startup)
       Forth_banner,      ;;; SS_BANNER
       identfn,           ;;; SS_INITCOMP   (compile a standard prelude)
       erase,             ;;; SS_POPARG1    (handle a command-line argument)
       identfn,           ;;; SS_VEDSETUP   (editor; no-op if you skip Ved)
       identfn,           ;;; SS_XSETUP     (X; no-op if you skip graphics)
    %};
```

Only `SS_COMPILER` is real work — it is your tier-2 compiler. The rest can start as `identfn`/`erase` stubs. One macro then makes the language reachable by name from a Pop-11 prompt:

```
define global macro forth;
    "forth" -> sys_compiler_subsystem(`c`);
enddefine;
```

Finally, build an image the way the shipped languages are built — a `mkimage` invocation that loads your sources and installs `mylang.psv` — and you have `poplog mylang`, live subsystem switching, and your language on every platform Poplog runs on. The four existing subsystems under `pop/lisp/`, `pop/pml/`, `pop/plog/` and `pop/forth/` are the templates; each is a `*_subsystem_procedures` file, a compiler, and a Makefile rule.

6.4 The other starting point: extend what is there

Not every new language needs to be new. The shipped Prolog is 51 files of ordinary Pop-11, so a specialised variant has an enormous head start: reuse the term reader, unification and clause store, and replace only the *control* — a different search order, tabling, constraint propagation, a custom resolution rule. Adding built-ins and operators to the existing engine is hours; swapping the control regime is days to weeks, scaling with how much of the engine you replace.

CHAPTER 7

The C interface: orders a robot can trust

A unit in the field must not obey an order just because it arrived. The command post signs every order under a key the unit was given at enlistment, and the unit refuses anything that does not verify. The cryptography comes from OpenSSL, and reaching it is the job of Poplog's foreign-function interface — which has existed since 1987 and is unusually deep: external code is loaded into the *running* image by Poplog's own dynamic linker, declared with a syntax form, and called with argument conversion that understands Poplog's data representation. This chapter shows the path this fork actually uses in production libraries, then names the alternatives and the traps.

7.1 The mechanisms, briefly

`external_do_load`	The primitive: a mark item, a list of object files or shared libraries, and a list of symbol specifications. Procedural.
`exload`	The declarative syntax form over it — name the library, name the functions, give their types. **Start here.**
`exacc`	The syntax form that *calls* an external function or accesses external data, converting arguments and results.
`cons_access`	Builds an accessor procedure for a field of an external structure.
`lib external`, `lib newexternal`	Older higher-level layers; see `help external` and `help newexternal`.

The reference material is `ref external` (mechanisms and data representation), `ref defstruct` (structures) and `ref external_data`.

7.2 The shim discipline

The libraries in this fork — `lib crypto` over OpenSSL, `popcurl` over libcurl, `poppcre` over PCRE — all go through a small C shim rather than binding the library's own API directly. That is a deliberate rule, and it is worth stating plainly:

> **Bind a shim, not the library**
>
> Keep the `exload` surface to a handful of **non-variadic** functions with **explicit lengths** and **caller-allocated buffers**, and keep object lifecycle entirely on the C side. Variadic calls through the FFI are hazardous — notably on Apple `arm64`, where the variadic calling convention differs from the regular one — and library handle lifetimes are exactly what a garbage collector should not be guessing about.

A shim is usually fifty lines. Here are two of the seven functions behind `lib crypto`:

```
/* pop/extern/popcrypto/popcrypto_shim.c */
#include <openssl/evp.h>
#include <openssl/hmac.h>

/* digest of data[0..len) with a named algorithm; writes raw bytes to
   out, returns the digest length, or -1 */
int pcr_digest(const char *alg, const void *data, long len,
               unsigned char *out)
{
    const EVP_MD *md = EVP_get_digestbyname(alg);
    unsigned int n = 0;
```

```
    if (!md) return -1;
    if (!EVP_Digest(data, (size_t)len, out, &n, md, NULL)) return -1;
    return (int)n;
}

int pcr_random(unsigned char *buf, long n)
{
    return RAND_bytes(buf, (int)n) == 1 ? 0 : -1;
}
```

Note the shape: no allocation crosses the boundary, every buffer has an explicit length, and every failure is a negative return rather than an exception or an `errno` the caller must remember to read.

7.3 Declaring it: `exload`

```
uses crypto;   ;;; ...is, inside, this:

exload popcrypto
    #_IF sys_file_exists(sysfileok('$usepop/pop/extern/popcrypto/popcrypto.dylib'))
    ['$usepop/pop/extern/popcrypto/popcrypto.dylib']
    #_ELSE
    ['$usepop/pop/extern/popcrypto/popcrypto.so']
    #_ENDIF
(language C)
    lconstant
        pcr_digest(alg, data, len, out)             :int,
        pcr_hmac(alg, key, keylen, data, len, out)  :int,
        pcr_random(buf, n)                          :int,
        pcr_pbkdf2(alg, pass, passlen, salt, saltlen, iters, out, outlen) :int,
        pcr_version()                               :exptr,
        ;;; ...and pcr_gcm_encrypt / pcr_gcm_decrypt
    ;
endexload;
```

Three things are happening. `popcrypto` is the *mark item* — the tag under which this set of symbols is loaded and can be unloaded. `#_IF` is a compile-time conditional, so one source file picks the right shared-library extension on macOS and Linux. And each declaration gives the argument names and the *result* type: `:int` for a C `int`, `:exptr` for a pointer (Poplog wraps it in an external-pointer record; `exacc_ntstring` pulls a C string out of one). The full type vocabulary — `:byte`, `:short`, `:int`, `:long`, `:sfloat`, `:dfloat`, pointers, arrays and structures — is in `ref external`.

7.4 Calling it: `exacc`

```
lconstant OUTMAX = 64;              ;;; SHA-512 is the widest digest

define crypto_digest(alg, data) -> raw;
    lvars n, out = inits(OUTMAX);   ;;; caller allocates the buffer
    unless isstring(alg) and isstring(data) then
        mishap(alg, data, 2, 'crypto_digest: two strings needed')
    endunless;
    exacc pcr_digest(cstr(alg), data, length(data), out) -> n;
    if n < 0 then
        mishap(alg, 1, 'crypto_digest: unknown algorithm')
    endif;
    substring(1, n, out) -> raw;
enddefine;
```

```
: uses crypto;
: crypto_digest_hex('sha256', 'hello poplog') =>
** 8b05ad6e876a9ce82062d33918c82d6b9bfac5b0f1cb575bb4dc8cd70762c6ae
```

```
: crypto_version() =>
** OpenSSL 3.6.3 9 Jun 2026
```

On top of that, the signed order is a dozen lines — `examples/robotarmy/signed_orders.p`:

```
vars fleet_key = 'a shared secret, provisioned at enlistment';

;;; wire format:  <hex mac> <space> <order text>
define sign_order(text) -> wire;
    crypto_hmac_hex('sha256', fleet_key, text) <> ' ' <> text -> wire
enddefine;

define verify_order(wire) -> text;
    lvars sp = locchar(` `, 1, wire), mac, body;
    false -> text;
    if sp then
        substring(1, sp - 1, wire) -> mac;
        allbutfirst(sp, wire) -> body;
        if mac = crypto_hmac_hex('sha256', fleet_key, body) then body -> text endif
    endif
enddefine;
```

```
on the wire: 81bdc45a...581146c3 unit r1 advance to ridge
genuine:     unit r1 advance to ridge
tampered:    <false>          <- one character of the order changed
replayed:    <false>          <- a real signature on a different order
```

7.4.1 Two traps that bite immediately

Pop-11 strings are not null-terminated. They are counted byte vectors, so a string passed where C expects `char *` must be terminated explicitly. `lib crypto` does it in one line:

```
define lconstant cstr(s);
    s <> consstring(0, 1)       ;;; append a NUL
enddefine;
```

Data arguments do not need this — they are passed with an explicit length, which is the better interface anyway.

Packed fields versus full fields. Poplog integers and floats are tagged in ordinary records and vectors. Passed as direct arguments they are converted for you; passed *inside* a structure they are not, and the C side sees encoded nonsense. An integer array destined for C must therefore be built with a packed representation (`intvec` and friends), not `newarray`. Strings made with `inits(n)` are packed byte vectors, which is exactly why they serve as C buffers above.

7.5 Building and shipping the shim

The shim is built by a small script, not by the main `make`, so a Poplog without OpenSSL still builds:

```
tools/build-popcrypto.sh      # -> pop/extern/popcrypto/popcrypto.{so,dylib}
```

```
if command -v pkg-config >/dev/null 2>&1 && pkg-config --exists libcrypto; then
    flags="$(pkg-config --cflags --libs libcrypto)"
else
    flags="-lcrypto"
fi
cc -O2 -shared -fPIC -o "$lib" "$src" $flags
```

And the library fails loudly at load time if the shim is missing, rather than mysteriously at the first call:

```
unless sys_file_exists(sysfileok('$usepop/pop/extern/popcrypto/popcrypto.so'))
or    sys_file_exists(sysfileok('$usepop/pop/extern/popcrypto/popcrypto.dylib'))
then
    mishap(0, 'lib crypto: shim not built -- run tools/build-popcrypto.sh first')
endunless;
```

Both `lib json` and its FFI-based sibling `lib crypto` are validated on x86-64, AArch64 (Raspberry Pi 5) and RISC-V; `tools/test-crypto.sh` is the acceptance suite and `tools/test-remote.sh host` reruns it anywhere with a built tree. That matters more than usual for FFI code, because calling conventions are exactly what differs between ports — the RISC-V float ABI, for instance, was found and fixed by running these suites.

7.6 Images, handles, and when not to bother

External state does not survive `syssave`. Library handles, open connections and `exptr` values are stale after a restore — the right shape for a binding is to *mishap cleanly* on a stale handle rather than crash, and the fix is always to reopen. Procedures, by contrast, survive fine, so a restored image is ready to work as soon as you reconnect its resources.

Finally: the FFI is not always the answer. Poplog can stream the output of a shell command as a line repeater, and for a one-off that is often enough:

```
vars r = sys_obey_linerep('set +e; curl -s https://example.com/api | jq -r ".id"');
```

Note `set +e`: the command line runs under `errexit`, so without it the first failing command aborts the whole pipeline *silently* — empty output and no error.

The trade-off against the FFI is a process per call, and it is not subtle: on the machine this chapter was written on, 100 round trips through `sys_obey_linerep('echo hi')` took 284 ms of wall time — about 2.8 ms each — against an unmeasurable zero for 100 in-image calls. Use a shim when the call sits in a loop; use the pipe when it does not.

CHAPTER 8

Where to go next

8.1 The documentation you already have

The in-tree corpus is over 900 files and it is the primary source for everything in this book. From any prompt:

```
: help matches          ;;; practical, task-shaped
: ref  vmcode           ;;; the exhaustive reference -- and the VM spec
: teach json            ;;; tutorials, including how lib json was built
```

The same material is published at `https://iotone.github.io/poplog/`, regenerated on every push by a Pop-11 program; `llms.txt` there is for AI assistants. The handful of entries worth reading early: `help external` and `ref external` (Chapter 7), `ref vmcode` and `ref subsystem` (Chapter 6), `ref regexp`, `help json`, `help crypto`, and `teach swank`.

8.2 Four decades of teaching material

Poplog was the vehicle for a great deal of university AI teaching, and most of it survives. The tree does not vendor any of it; one script fetches it from public archives and wires it into `help`/`teach` inside the system:

```
tools/fetch-learning.sh list        # what's available
tools/fetch-learning.sh --all       # about 30 MB into learn/
```

`hidden-gems`	Small idiomatic demos of what makes Pop-11 distinctive: the open stack, updaters, dynamic lists, coroutines, `dlocal`, the GC, extensible syntax.
`bhamteach`	The Birmingham introductory-AI course — pattern-matching chatbots, grammars and parsing, story generation, recursion exercises.
`packages`	The classic package tree: SimAgent and Poprulebase, popvision, rclib, neural nets, Braitenberg vehicles.
`paradigms`	A UMASS programming-language-paradigms course taught in Pop-11 (1997–2000).
`primer`	The Pop-11 Primer — the standard text for experienced programmers.
`examples`	Curated teaching examples: Othello, Life, robolang.

Upstream licences apply per pack; each records its origin in `learn/pack/.source`.

8.3 Reading the source

Poplog is written mostly in itself, which makes it unusually approachable.

- `pop/src/` — the core: `vm_plant.p` and `vm_control.p` are the VM builder from Chapter 6; `syscomp/` holds one directory per CPU back-end.
- `pop/lib/lib/` — the library, and the best source of idiomatic Pop-11 at length. `json.p`, `crypto.p`, `http_server.p`, `jsonrpc.p`, `swank.p` and `forth.p` are all readable in an afternoon each.
- `pop/plog/`, `pop/lisp/`, `pop/pml/`, `pop/forth/` — the four front-ends, i.e. four worked answers to Chapter 6.
- `examples/` — runnable programs: the HTTP service, ELIZA, a Z-machine interpreter, graphics demos.

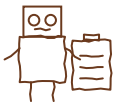
8.4 Benchmarks, ports, and the state of the project

`BENCHMARKS.md` carries the full cross-platform, cross-language matrix with its methodology; the numbers quoted in Chapter 1 are drawn from it. The `PORTING-*.md` files are the working notes from the AArch64, Apple Silicon and RISC-V ports — including the bugs, which is the part usually missing from porting documentation and the part most useful to the next porter. Windows is the one platform still marked TODO.

> **If you take one idea away**
>
> It is that a compiler cheap enough to call in a loop changes what a command post can be. Everything distinctive here — live redefinition, saved images, five languages in one heap, an editor that talks to a running process, an agent that keeps a session warm across tool calls and teaches it new orders mid-mission — falls out of fifteen microseconds per definition. The full Robot Army is at `examples/robotarmy/`; start with `fleet.p` and give it an order.

APPENDIX A

Cheat sheet



A.1 Syntax

`;;; text`	comment to end of line
`value -> x`	assignment; the arrow points at the destination
`expr =>`	print the result(s), prefixed `**`
`expr ==>`	as `=>` but prints structure in full
`npr(x)`	print `x` and a newline
`a sys_>< b`	concatenate anything into a string
`a >< b`	concatenate two strings
`'text'`	string (mutable, counted, *not* NUL-terminated)
`"word"`	word — an interned symbol
`[a b c]`	list; contents are *quoted*
`[^x ^(e) ^^xs]`	. . . with `x`, `e` evaluated and `xs` spliced
`{1 2 3}`	vector
`[% e %] {% e %}`	list/vector of everything `e` leaves on the stack
`p(% a %)`	closure: `p` partially applied to `a`

A.2 Blocks — every opener has a closer

`define f(a, b); . . . enddefine;`	`if . . . then . . . elseif . . . else . . . endif`
`define f(a) -> r; . . . enddefine;`	`unless . . . then . . . endunless`
`procedure(a); . . . endprocedure`	`for i from 1 to n do . . . endfor`
`define :class C; slot s = v; enddefine;`	`for x in list do . . . endfor`
`define :method m(x:C); . . . enddefine;`	`while c do . . . endwhile`
`section $-name => exports; . . . endsection;`	`repeat . . . quitif(c); . . . endrepeat`
`exload tag [libs] . . . endexload;`	`switchon e case < 0 then . . . endswitchon`

A.3 Declarations

`vars` permanent (global) — required for matcher pattern variables. `lvars` lexical local — the default choice inside a procedure. `constant`, `lconstant` their immutable forms. `dlocal x = v` rebind for the dynamic extent of the call, restored on *any* exit.

A.4 Mishap decoder

`DECLARING VARIABLE x`	(a warning) you used an undefined name — usually a typo or a missing `define`
`NUMBER(S) NEEDED`	arithmetic on a non-number, often the value of that undefined name
`STRING NEEDED`	a word `"x"` was passed where a string `'x'` was wanted — check the quotes
`ILLEGAL DECLARATION OF WORD`	the name is already a protected system identifier; rename
`MISPLACED EXPRESSION ITEM`	syntax: usually a missing keyword or a construct used with the wrong shape
`EXECUTING NON-PROCEDURE`	calling something that is not a procedure — often an undefined name
`CAN'T RESTORE - NOT SAME SYSTEM AND VERSION`	the `.psv` image predates the current `basepop11`; rebuild the images
`sysEXECUTE: NOT AT EXECUTE LEVEL`	VM planting attempted at compile-time top level — wrap it in a procedure

A.5 Files, processes, tables

```
;;; read a file line by line
vars dev = sysopen('/path/file', 0, "line");
vars rep = line_repeater(dev, inits(4096));   ;;; buffer size = max line length
repeat
    rep() -> line;
    quitif(line == termin);
    if issubstring('ERROR', 1, line) then ... endif;
endrepeat;

sysobey('ls /tmp > /tmp/out');                        ;;; run a command
vars r = sys_obey_linerep('set +e; grep pat file');   ;;; ...and stream it

newproperty([], 50, false, true) -> tbl;   ;;; hash table, keys matched by ==
newmapping([],  50, false, true) -> m;     ;;; ...keys matched by =
```

`line_repeater` truncates lines longer than its buffer, so size `inits(n)` generously. `sys_obey_linerep` runs its command line under `errexit`: prefix `set +e;` whenever a failure is expected.

A.6 Regular expressions

The escape character is `@`; `. * [ ] ^ $` are literal unless escaped; there is no alternation and no lookaround.

```
vars err, p;
regexp_compile('ERROR @[0-9@]@{1,@}') -> (err, p);   ;;; test err -- false = ok
p(1, 'ERROR 42 happened', false, false) -> (i, n);   ;;; -> 1  8
substring(i, n, 'ERROR 42 happened') =>              ;;; ** ERROR 42
```

Case-insensitive: `regexp_compile(s, 1, false)`, or `@i` in the pattern. `p` is a compiled procedure — define it once, reuse it.

A.7 The Robot Army examples

All at `examples/robotarmy/`; each runs from the repository root.

File	Ch.	What it shows
`fleet.p`	3, 4	`Robot` class, word-keyed registry, `muster()` on the open stack, `obey()` on the matcher
`telemetry.p`, `fleet.log`	4	One-pass log triage: `line_repeater`, default-valued properties, `@`-regexps
`c2.p`	4	The command post: `http_server` + `json` over `fleet.p`
`chain.pl`	5	The chain of command in Prolog
`patrol.fth`	5	Robot control words in Forth, with an early `leave`
`signed_orders.p`	7	HMAC-signed orders through `lib crypto`

A.8 Running things

`./poplog pop11`	Pop-11 REPL
`./poplog basepop11 -target/psv/prolog.psv`	Prolog
`./poplog basepop11 -target/psv/clisp.psv`	Common Lisp
`./poplog basepop11 +target/psv/pml.psv`	Standard ML
`tools/forth.sh`	Forth
`./poplog basepop11 prog.p`	run a Pop-11 file
`tools/pop11-lsp tools/pop11-swank tools/pop11-mcp`	editor and agent servers