Raku++ Internals

How a hand-written C++ implementation of Raku works — from source text to native code

The Raku++ project

Contents

Preface

This is a book about the inside of a compiler.

Raku++ is a hand-written implementation of the Raku programming language, written in C++17, with no third-party dependencies. It lexes, parses, interprets, and — in one of its four run modes — transpiles Raku to C++ and compiles it to a native binary. It carries its own regular-expression and grammar engine, its own Unicode subsystem built from the pinned UCD tables, its own arbitrary-precision arithmetic, a foreign-function interface that loads libffi at run time rather than linking it, a C ABI for native extension modules, and a concurrency runtime with a global interpreter lock that can be switched off.

None of that is unusual for a language implementation. What is unusual is that all of it fits in about fifty thousand lines of source that one person can read, and that almost every non-obvious decision in it was made against a measurement. This book is an attempt to write down both: the mechanisms, and the reasons.

Who this is for

You should be comfortable reading C++ and comfortable reading Raku, but you do not need to be an expert in either, and you certainly do not need to have implemented a language before. Where a piece of compiler folklore is load-bearing — precedence climbing, Thompson construction, packrat memoisation, copy-on-write — it is explained where it is used rather than assumed.

Three kinds of reader were in mind:

- Someone who wants to **work on Raku++** and needs a map before they can usefully change anything.

- Someone building **their own interpreter or compiler**, who wants a worked example of a dynamic language implemented in a static one — including the parts that went wrong.
- Someone who writes **Raku** and wants to know what the machine underneath is actually doing when they write `given/when`, or a grammar, or `is native`.

What this book is not

It is not a manual for the language, and not a manual for the `rakupp` command. Those exist: the guide in `docs/guide/` covers using Raku++, and `docs/guide/REFERENCE.md` is the exhaustive per-routine lookup sheet. Nor is it a specification of Raku; the reference implementation, Rakudo, and the Roast test suite hold that role, and this book is careful to say *where Raku++ diverges from them* rather than pretending it does not.

It is also not a tour of every function. The source is the normative reference, and it moves. What is written down here is the **shape** — the data structures that everything else is built on, the invariants they rely on, and the reasons a particular design was chosen over the obvious alternative.

How to read it

The nine parts are ordered the way a program flows through the system: source text, then the tree, then the values, then execution, then the specialised engines, then the back ends, then the boundaries with the outside world. Read straight through and it is a narrative. Read a single part and it should still stand on its own; cross-references say where the rest of a thread lives.

If you are here for one thing in particular:

You want	Start at
the overall map	Chapter 1, then Chapter 24
how source becomes a tree	Part II
what a runtime value *is*	Chapter 7
how a Raku call actually happens	Chapters 13 to 15
regexes and grammars	Part V

You want	Start at
the native compiler	Part VII
calling C, or being called from it	Chapters 31 and 32

Conventions

Source excerpts are tagged with the **file** they come from:

```
// src/Value.h
static Value integer(long long x) { Value v; v.t = VT::Int; v.i = x; return v; }
```

They are lightly trimmed for the page — an ellipsis, a dropped comment, an elided error string — but are otherwise verbatim. **File names, not line numbers, are the anchors.** The code moves; grep for the function or for the quoted line.

Raku code is shown as Raku:

```
my @primes = (2..*).grep(*.is-prime);
say @primes.head(5);          # (2 3 5 7 11)
```

Measured numbers carry their conditions. Unless a chapter says otherwise, they were taken on arm64 macOS with Apple clang, using the project's benchmark policy: interleaved A/B runs, a discarded warm-up, medians or minima of several repetitions, and a *control* kernel that the change under test cannot affect. A number without a control is an anecdote, and this book tries not to print anecdotes.

Sections headed **What went wrong** are not decoration. Several of the designs here are the second or third attempt, and the discarded attempts are usually more instructive than the surviving one.

A word about honesty

Raku is a very large language, and Raku++ does not implement all of it. About ninety per cent of the declared Roast suite passes. The method resolution order is a depth-first walk rather than C3 linearisation. Multiple dispatch resolves ties by declaration order instead of raising an ambiguity error. Modules publish their whole environment to the global scope rather than only their exports. Macros, `RakuAST`, and slangs are not implemented at all.

Every one of those is stated in the chapter where it belongs, under a heading that says so. A book about compiler internals that only described the parts that work would be a brochure.

Part I

Ground

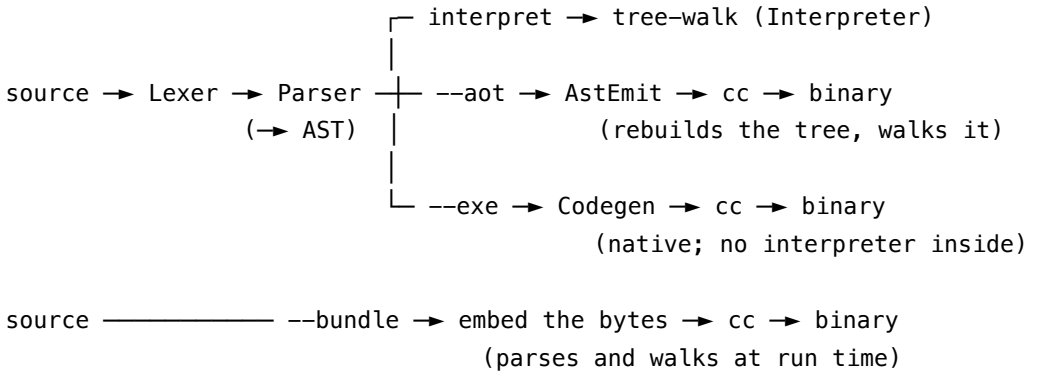
Chapter 1

What Raku++ Is

Raku++ is a Raku implementation written from scratch in C++17. It has no third-party dependencies: no parser generator, no regex library, no bignum library, no ICU, no libffi at link time, no garbage collector. The binary it produces is one file, and the same source builds it on macOS, Linux, the BSDs, Windows, and — compiled to WebAssembly — inside a browser tab.

That constraint is not asceticism for its own sake. It has a specific consequence which shapes almost every chapter of this book: **when something is slow or wrong, the code responsible is in the tree**, and it can be changed. A good deal of what follows is the story of doing exactly that.

1.1 The pipeline



A single front end feeds four back ends. The front end is a hand-written character-level lexer and a recursive-descent parser with a precedence-climbing core for expressions; it produces a `Program`, which is a vector of statement nodes. Everything downstream consumes that one artifact.

Everything except the command-line driver is compiled into a static library, **`librakupp_rt.a`** — "the runtime". The `rakupp` executable links against it, and so does every binary the compiling modes produce. That is why a feature implemented once behaves identically whether a program is interpreted or natively compiled: there is one implementation of `Value` semantics, one set of built-ins, one method dispatcher, one regex engine, one Unicode subsystem.

1.2 The four modes, in one paragraph each

Interpret (the default) lexes, parses, and tree-walks. Startup is about two milliseconds; throughput pays the per-node dispatch cost. This is the mode the language is developed against, because it is the only one that can run every construct.

`--bundle` embeds the source bytes in a generated C++ stub and links it against the runtime. The result is standalone, but at run time it still lexes, parses and tree-walks — it is the interpreter in a box. The win is distribution, not speed.

`--aot` parses at build time and emits C++ that *rebuilds the identical AST* at startup, then interprets it. Parse errors surface at build time rather than run time; execution speed is the same as bundling, because it is the same tree walk. It emits one builder function per AST node, so the generated code grows with the program.

`--exe` parses at build time and transpiles the tree into C++ that implements the program directly: native control flow, native calls, with the runtime called only for value semantics. With `-O` it additionally runs its own optimizer before the C++ compiler sees anything. Anything it cannot transpile throws a `CodegenError`, which the driver catches and answers by bundling the whole program instead — so `--exe` never refuses a program.

Chapter 24 works one small program through all four.

1.3 What the design is optimised for

Three decisions dominate everything else, and each gets a full chapter later.

One value type. Every Raku value at runtime is the same C++ struct, `Value`, carrying a one-byte tag and a field for every kind of payload. Not a union, not a `std::variant`, not a class hierarchy — a *fat struct*. Chapter 7 argues the case, which is stronger than it first looks, and prices the memory it costs.

The C++ stack is the Raku stack. There is no bytecode and no separate operand stack. `eval(Expr*)` returns a `Value`; `exec(Stmt*)` runs a statement. This makes the implementation small and makes native recursion depth the Raku recursion budget, which is why the entry point runs the program on a thread with a very large stack.

Compile time runs nothing. The parser executes no user code. `BEGIN` becomes a tagged block that the interpreter schedules after the parse; `constant` is not folded; `use` does not load anything during parsing. There is exactly one parse-time side effect — registering a user-declared operator in the parser's own tables — and Chapter 5 is about why that one is enough to support custom operators with real precedence in a single forward pass, and why it is also the reason macros and slangs are not supported.

1.4 What it deliberately is not

It is not the reference implementation. Rakudo is, and Raku++ is measured against Rakudo and against the Roast suite continuously rather than aspirationally. The current position is roughly ninety per cent of declared Roast assertions.

It has no JIT and no garbage collector. Lifetime is `shared_ptr` reference counting, which means a reference cycle leaks; the interpreter breaks the specific cycle that a self-closured nested sub would create, and otherwise the process is short-lived enough for this to be a real but tolerable limitation.

It does not implement compile-time metaprogramming. `macro`, `quasi`, `RakuAST`, and swapping the grammar for a lexical scope are all absent, for the structural reason given above. What *is* supported is everything that can be done by adding an entry to a table during a single forward pass: all six custom-operator categories with precedence and associativity traits, plus the whole runtime meta-object surface (`augment`, `.^add_method`, `does`/`but` mixins, `&routine.wrap`).

1.5 The ecosystem around the compiler

Several things in this book are not in the `rakupp` binary but are load-bearing for it, and appear where they are relevant:

Thing	What it is
Raku.js	the same runtime compiled to WebAssembly — the interpreter in a browser
the Roast harness	`tools/run-roast.raku`, written in Raku, run by rakupp against the spec suite
`perf-guard`	`tools/perf-guard.raku`, the release gate that fails a regression instead of eyeballing one
the showcase interpreters	JavaScript, Perl 5, Python 3 and Lisp interpreters *written in Raku*, used as beyond-Roast tests
`Rakupp::JSON`	the first native extension module, and the proof the extension ABI works

The showcase programs deserve a note here because they recur. An interpreter for another language, written in Raku and run by Raku++, exercises grammars, recursion, string handling, closures and dispatch simultaneously and at a scale no unit test reaches. Each of them, when first written, produced a list of genuine bugs in Raku++; several fixes in later chapters were found that way.

1.6 A map of this book

Part	Covers
I	this chapter, and the layout of the source
II	lexer, parser, user-defined operators, the AST
III	`Value`, strings, interning, numbers, containers

Part	Covers
IV	the tree walk, calls, control flow, dispatch, objects, laziness
V	the regex engine, the grammar engine, longest-token matching
VI	Unicode: graphemes, normalization, collation
VII	the four run modes, the C++ code generator, `-O`, serialization
VIII	modules, `use nqp`, NativeCall, the extension ABI, concurrency
IX	tooling built on the AST, and how any of this is proved

Chapter 2

The Shape of the Source

Before the mechanisms, the map. This chapter is the one to come back to when a later chapter names a file and you want to know what else lives near it.

2.1 The numbers

`src/` holds about **147,000 lines** of C++. That figure is misleading on its own, because **79,400 of them are generated Unicode tables** — character names, properties, collation weights, normalization data — emitted from the pinned UCD and UCA 17.0 files in `tools/ucd/`. Nobody reads those, and nobody edits them.

The hand-written implementation is about **67,700 lines**, and it is very unevenly distributed:

File	Lines	What it is
`Interpreter.cpp`	20,787	the tree walk, calls, dispatch, modules, concurrency
`Builtins.cpp`	9,887	named built-ins, `Test`, the head of the method chain

File	Lines	What it is
`Parser.cpp`	7,210	statements, expressions, declarations, interpolation
`MethodCallPart2.cpp`	3,549	the method chain, continued
`Regex.cpp`	2,785	the regex and grammar engine
`MethodCallPart3.cpp`	2,652	the method chain, continued
`Codegen.cpp`	2,586	the `--exe` transpiler
`Lexer.cpp`	2,351	tokenizer
`MethodCallTail.cpp`	2,277	the method chain, the end of it
`Interpreter.h`	1,363	the interpreter's own interface, plus the `rt*` helpers
`main.cpp`	1,281	the CLI and the four compile drivers
`Value.cpp`	1,093	coercions, comparison, `gist`, `flatten`

Two shapes stand out and both are deliberate.

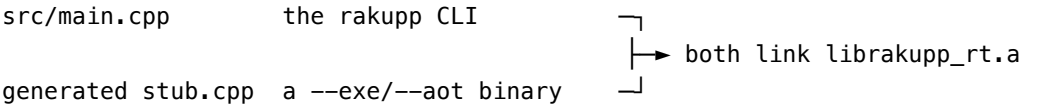
`Interpreter.cpp` is enormous. It is the tree walk, and the tree walk touches everything: scopes, calls, operators, assignment, control flow, module loading, the FFI marshaller, the concurrency runtime. Splitting it by topic would mostly move `#includes` around, because the pieces share the interpreter's private state rather than a clean interface.

The method dispatcher is split across four files for one reason. It used to be a single 9,138-line function, `methodCallInner`, and that stopped being compilable in a reasonable time. It is now four ordered *segments* — `Builtins.cpp` holds the

head, then `MethodCallPart2`, `MethodCallPart3`, `MethodCallTail` — each returning `std::optional<Value>`, where `nullopt` means "not handled here, try the next segment".

The critical property, stated in the source and worth repeating: **these are segments, not categories.** The chain is order-sensitive. Later arms deliberately catch what earlier ones decline. An arm belongs where its priority is, not where it reads nicely. `MethodCallSegment.h` exists so that the four files share one include prologue and cannot drift apart.

2.2 The library boundary



Everything except `main.cpp` and `Repl.cpp` compiles into `librakupp_rt.a`. The REPL lives in the executable rather than the library on purpose: nothing about an interactive session should be linked into the standalone binaries the compiling modes produce.

2.3 What each file is for

Front end

File	Role
`Token.h`	the token struct: kind, text, position, `spaceBefore`
`Lexer.{h,cpp}`	source text to a flat `vector<Token>`
`Ast.h`	every node type, and the `NK` tag enum
`Parser.{h,cpp}`	tokens to a `Program`; the live user-operator tables
`Pod.{h,cpp}`	the `$=pod` DOM

Values

File	Role
`Value.{h,cpp}`	the fat tagged struct, `CowStr`, `ClassInfo`, `Callable`
`IStr.h`	the interned-string field
`MethodName.h`	`MName`, the packed method name used by the dispatch chain
`BigInt.{h,cpp}`	arbitrary-precision integers, base 10 ^ 9
`IntOps.h`	portable overflow-checked arithmetic and bit intrinsics

Execution

File	Role
`Interpreter.{h,cpp}`	the tree walk and nearly everything it reaches
`Builtins.cpp`	the built-in routine table and the method chain's head
`MethodCall*.cpp`	the rest of the method chain
`BuiltinsShared.h`	helpers the split forced out of file scope
`Runtime.{h,cpp}`	the shared entry points, and the big-stack thread
`IOSpec.cpp`	`IO::Spec::*` path algorithms

Engines

File	Role
`Regex.{h,cpp}`	regex compilation, the matcher, `GrammarMatcher`

File	Role
`LtmNfa.{h,cpp}`	the declarative-prefix NFA for longest-token matching
`Unicode.{h,cpp}`	normalization, grapheme segmentation, collation, properties
`unicode_*_gen.cpp`	the generated tables those read
`unicode_names.cpp`	character names — the single largest file in the tree

Back ends and tooling

File	Role
`Codegen.{h,cpp}`	`--exe`: AST to C++
`AstEmit.cpp`	`--aot`: C++ that rebuilds the AST
`AstSerial.{h,cpp}`	the binary AST format behind the precompiled parse
`AstDump.cpp`	`--dump-ast`
`Lint.{h,cpp}`	`--lint`, static analysis over the parsed tree
`Highlight.{h,cpp}`	`--highlight`, parse-aware syntax colouring
`Profiler.{h,cpp}`	`--profile`, the routine-level wall-time profiler
`Repl.{h,cpp}`	the interactive session

Boundaries

File	Role
`Ffi.{h,cpp}`	the libffi backend: loader, ABI probe, type registry
`rakupp_ext.h`	the C ABI extension modules compile against

File	Role
`ExtApi.cpp`	the host side of that ABI
`Platform.h`	the Windows/POSIX split, in one place

2.4 Reading conventions in the source

Three habits recur, and knowing them saves a lot of confusion.

Comments explain *why*, and often carry the measurement. A comment in this tree is rarely a restatement of the code. It is much more often the reason the obvious version was rejected, sometimes with a number attached:

```
// src/Value.h — the CowStr rationale, trimmed
// Value is copied by value everywhere ... so holding a bare std::string
// meant a long string was memcpy'd on each of those. The cost is
// O(length) per OPERATION, which makes any pure-Raku tokenizer O(n^2):
// JSON::Fast spent 13.9 s on a 421 KB document that Rakudo parses in
// 50 ms, and the profile was all copying, not parsing.
```

When this book explains a decision, it is usually expanding a comment like that one.

A "decided once" field is a fact about the syntax, not a cached result. Several node types carry a small mutable field that starts at a sentinel and is written on first evaluation:

```
// src/Ast.h
template <typename T> struct DecidedOnce {
    std::atomic<T> v;
    operator T() const { return v.load(std::memory_order_relaxed); }
    DecidedOnce& operator=(T x) {
        v.store(x, std::memory_order_relaxed); return *this;
    }
};
```

The atomic is not for synchronisation. It is there because a node is shared between threads, every writer computes the same idempotent answer, and a plain field would make that a data race that ThreadSanitizer correctly reports. Relaxed atomics make it defined at plain-load cost on the architectures that matter. Chapter 18 is entirely about what these fields hold and, more importantly, what they must never hold.

`rt*` functions are the compiled backend's vocabulary. Anything named `rtAdd`, `rtIndexRef`, `rtAttrGet`, `rtCallB` is a runtime entry point that `Codegen` emits calls to. They are declared in `Interpreter.h` and are the contract between the transpiler and the runtime — which is why they are `inline` where the fast path matters. Chapters 25 to 27 are about them.

2.5 Building it

```
cmake -S . -B build
cmake --build build -j
```

`CMakeLists.txt` builds `librakupp_rt` from all of `src/` except `main.cpp`, then the `rakupp` executable. The glob is `CONFIGURE_DEPENDS`, but CMake caches it, so re-run the configure step after adding a source file.

The compiler matters more than usual here. Clang produces a binary between 1.2 and 2 times faster than GCC's on this codebase — the method dispatch chain and the tree walk are both inlining-sensitive in ways GCC handles less well — so Clang is what ships and GCC is kept as a portability gate. Link-time optimisation and `-mcpu=native` were both measured and both did nothing.

2.6 The test surface

Where	What
Roast	the Raku specification suite, run by `tools/run-roast.raku`
`t/run.raku`	the local suite: examples and showcases, byte-compared to golden output
`t/regression/`	one file per fixed bug
`t/stress/`	concurrency and memory stress, also run under TSan and ASan
`tools/perf-guard.raku`	the performance gate, compared against a recorded baseline

Where	What
the showcase interpreters	JavaScript, Perl, Python and Lisp, written in Raku

The release checklist in `docs/dev/RELEASING.md` gates on all of them. Chapter 35 is about why the performance gate is there and what happens when it is skipped.

Part II

The Front End

Chapter 3

The Lexer

Raku's grammar is too context-sensitive for a generated tokenizer, so the lexer is hand-written: a loop that skips whitespace and comments, dispatches on the current character to a handler, and appends one or a few tokens. The output is a flat `std::vector<Token>`.

What makes it interesting is not the loop. It is the set of decisions Raku forces the lexer to make that most languages never have to.

3.1 The token

```
// src/Token.h
enum class Tok {
    End, IntLit, NumLit, StrLit, VersionLit, StrInterp, RegexLit,
    SubstLit, QwList, Ident, Var, LParen, RParen, LBrace, RBrace,
    LBracket, RBracket, Semicolon, Comma, FatArrow, Op
};
struct Token {
    Tok kind = Tok::End;
    std::string text;    // identifier / operator spelling / raw literal
    std::string text2;   // s/// replacement; regex adverb flags
    long long ival; double nval;
    int line, col;
    bool spaceBefore = false;
    bool flag = false;   // S/// — the non-mutating substitution
};
```

Two absences are as informative as the contents.

There is no keyword kind. `if`, `sub`, `class`, `given` all arrive as `Tok::Ident`. The parser decides, from position, whether a given identifier is acting as a keyword. That is why Raku has no reserved-word list and why `my $if = 1` is legal.

There is no operator identity. Every operator arrives as `Tok::Op` carrying only its spelling. The lexer knows nothing about precedence, associativity, or which operators exist beyond a fixed vocabulary of spellings. All of that is the parser's problem, which is precisely what makes user-declared operators possible (Chapter 5).

3.2 `spaceBefore`: whitespace is significant

Raku is one of the few languages where a space changes the parse. `f()` is a call; `f ()` is `f` applied to a parenthesised list. `%h<k>` is a subscript; `%h <k>` is not. So every token records whether whitespace preceded it:

```
// src/Lexer.cpp — the tokenize loop
size_t before = pos_;
skipWhitespaceAndComments();
bool spaced = (pos_ > before && pos_ != atomDropEnd_) || before == 0;
// … lex the token …
t.spaceBefore = spaced;
```

The flag is true exactly when skipping advanced the cursor. The parser then leans on it constantly: to tell a postcircumfix from a list, a call from an application, a postfix operator from a fresh term.

3.3 Regex or division?

A bare `/` starts a regex in term position and divides in operator position. The lexer decides by looking at the *previous* token:

```
// src/Lexer.cpp — regexContext(), the Tok::Op case
case Tok::Op:
    return pv.text != "++" && pv.text != "--" &&
           pv.text != ">"  && pv.text != "<"  &&
           pv.text != ">>" && pv.text != "<<" &&
           pv.text != ">=" && pv.text != "<=";
```

After most operators a term is expected, so `/` opens a regex. After a value — a number, a variable, a closing bracket — or after a postfix `++`/`--`, division follows.

After an identifier it is a regex only when the word is in a small set of keywords that take an expression (`if`, `while`, `say`, `grep`, `split`, and friends). `//` and `/=` are matched earlier so they lex as operators.

This is a heuristic in the honest sense: it implements Raku's actual rule for the cases the rule covers, and a sufficiently perverse program can still be surprised.

3.4 Quote forms and heredocs

`tryQuoteForm` handles the whole `q`/`qq`/`Q` family plus `m`, `rx`, `s`, `tr` and `qw`, with adverbs (`:w`, `:to`, `:!c`, ...) and **arbitrary delimiters** — any bracket pair, or any non-bracket character. Whether the result interpolates is decided here and encoded in the token kind: `qq` produces `Tok::StrInterp`, `q` produces `Tok::StrLit`. Adverbs that flip a `Q` into interpolating emit a feature sentinel the parser reads later.

Heredocs cannot be captured inline, because the body is on *later* lines. The lexer emits an empty-bodied token immediately and defers:

```
// src/Lexer.cpp — tryQuoteForm, on a :to adverb
heredocMarker_ = raw;                // the terminator, e.g. END
heredocInterp_ = (w == "qq");
out = make(heredocInterp_ ? Tok::StrInterp : Tok::StrLit, "");
```

The pending heredoc — marker, token index, interpolation flag — is queued. When the lexer reaches the newline that ends the current line, `processHeredocs` reads lines until the marker, dedents them, and **back-patches the token's text**. Because interpolation was already fixed by the token kind, the patched body needs no further decision.

Several heredocs can be opened on one line; the queue keeps them in order, each with its own interpolation-feature string.

3.5 Identifiers, sigils, twigils, Unicode

`lexIdentOrVar` reads a sigil (`$ @ % &`), an optional twigil (`*` dynamic, `.`/`!` attribute, `^` placeholder, `?` compile-time), then the name — which may contain `-` or `'`, may be package-qualified with `::`, may be all digits (`$0`, `$^1`), and may be a Unicode identifier. The whole thing becomes one `Tok::Var`.

The rule about `-` and `'` inside a name is small and was a genuine bug, so it lives in the header where everyone who needs it can see it:

```
// src/Lexer.h
inline bool rakuIdentStart(char c) {
    return std::isalpha((unsigned char)c) || c == '_';
}
inline bool rakuIdentJoins(char sep, char next) {
    return (sep == '-' || sep == '\'') && rakuIdentStart(next);
}
```

A – continues a name only when a *letter* follows, never a digit. So `elems-1` is `elems - 1` and `$x-1` is `$x - 1`.

That rule has to be answered in two places: the lexer, and the string interpolation scanner in the parser. They disagreed. The interpolation scanner tested `isalnum`, glued the digit into the name, and handed `"$x-1"` to the expression parser — which re-split it correctly and then interpolated the *arithmetic result*. So `my $x = 5; say "$x-1"` printed `4` where Rakudo prints `5-1`. Six sites implemented the rule three different ways; the fix was to write it once, in a header, and delete the other five.

Superscripts are folded here too: a run like x³ emits a synthetic tight `**` operator and an `IntLit 3`, so it parses as `x ** 3`.

3.6 Operators are a fixed, hand-ordered vocabulary

```
// src/Lexer.cpp — lexOperator
for (const char* op : ops) {
    std::string s(op);
    bool ok = true;
    for (size_t k = 0; k < s.size(); k++)
        if (peek(k) != s[k]) { ok = false; break; }
    if (ok) { /* consume s, return Tok::Op */ }
}
// unmatched: a single-character Op
```

This is not a longest-match scanner. It is a manually ordered table where longer entries are placed before their prefixes, and the first full match wins. Adding a built-in operator means inserting it at the right position — a real maintenance hazard, and named as one.

The fall-through matters more than the table: anything unmatched becomes a **single-character `Tok::Op`**. That is why a novel user-defined operator still arrives as a token the parser can pick up, even though the lexer has never heard of it.

The vocabulary also covers Unicode aliases (⁉ for <=), hyper metaoperators (»op«), and the set operators ((elem), ∈) — but it is *static*. It tracks no user declarations whatsoever.

3.7 Rule bodies are not tokenized

One construct is deliberately opaque. A `token`/`rule`/`regex NAME { … }` declaration is not lexed as Raku:

```
// src/Lexer.h
bool tryRuleDecl(std::vector<Token>& out, bool spaced);
```

`tryRuleDecl` captures the balanced-brace body **raw** and emits it as a single `Tok::RegexLit`. Regex syntax is a different sub-language — `<[a..z]>`, `**`, `%%`, `<?{ … }>` mean nothing to the Raku tokenizer — so keeping it out of the token stream avoids teaching the lexer two grammars. The regex engine re-lexes the captured text later, in Chapter 19.

The same applies to a bare `/…/` literal and the `s///` family: the pattern text travels in `Token::text` and the replacement in `Token::text2`, both unparsed.

3.8 What else the lexer carries out

Beyond tokens, `Lexer` produces three side channels the parser and the runtime need:

Channel	Contents
`finishData()`	the text after `=finish`, for `$=finish`
`podData()`	rendered `=begin pod` content, for `--doc`
`declPod_` / `leadPod_`	`#=` and `#|` declarator documentation, keyed by line

Declarator pod is keyed by line because that is how it attaches: a `#|` run immediately above a declaration documents it, a `#=` run immediately below or beside it does. The parser resolves that association with `leadingPodFor(line)` and `trailingPodFor(line)`, and records which lines a *parameter* has already claimed — otherwise the parameter documentation of a multi-line `sub MAIN` signature would be picked up a second time as the routine's own usage text.

3.9 Errors

A construct that runs off the end of the file gets a dedicated diagnostic:

```
// src/Lexer.h
[[noreturn]] void runawayQuote(const char* construct,
                               const char* finalDelim, int startLine) const;
[[noreturn]] void runawayTerm(const std::string& close,
                              const std::string& open, int startLine) const;
```

Two spellings, because Rakudo has two: the bare quote forms name the *construct*, the bracketed `q`/`rx`/comment forms name the *terminator*. Both quote the line the construct *started* on, which is the only coordinate still meaningful once the scan has eaten the rest of the file.

An `atEof` flag rides along on the resulting error. Only the REPL reads it, to tell “give me a continuation line” from “this is a syntax error” — which is how a half-typed string literal at an interactive prompt asks for more input instead of failing (Chapter 34).

Chapter 4

The Parser

`Parser::parseProgram()` walks the token vector once, left to right, and produces a `Program`. It is recursive descent for statements and precedence climbing for expressions — the classic pairing, chosen because it is the one that survives contact with an irregular grammar.

It is a **single forward pass**, and it executes nothing.

4.1 Statement dispatch

`parseStatementImpl` is a keyword ladder that only engages when the leading token is a `Tok::Ident`. It matches the identifier's *text* and delegates:

Leading word	Goes to
`if` / `unless`	`parseIf`
`while` / `until`	`parseWhile`
`for`	`parseFor`
`loop` / `repeat`	the C-style and post-condition loops
`sub` / `multi` / `proto` / `method` / `submethod`	`parseSub`
`class` / `role` / `grammar` / `module` / `package`	`parseClass`
`my` / `our` / `state` / `has` / `constant`	strip the scope word, re-dispatch
`enum`, `subset`	`parseEnum`, `parseSubset`
`given` / `when` / `default`	the topicalisers
`use` / `no` / `need`	`UseStmt`

Leading word	Goes to
a phaser name	a `Block` tagged with the phaser

Everything not caught is a bare expression statement:

```
// src/Parser.cpp — the parseStatementImpl fallthrough
auto es = std::make_unique<ExprStmt>();
es->e = parseExpression();
return applyModifiers(std::move(es));
```

Because keywords are matched by text and only in leading position, an `if` anywhere else is an ordinary identifier.

4.2 Statement modifiers

A trailing `if`, `unless`, `while`, `until`, `for`, `given` or `with` turns the statement into the corresponding control node and sets a `modifier` flag. `applyModifiers` recurses, so modifiers chain:

```
// src/Parser.cpp — applyModifiers, the `for` case, abridged
if (isIdent("for")) { advance();
    auto fs = std::make_unique<ForStmt>();
    fs->list = parseExpression();
    fs->modifier = true;      // no implicit block
    fs->body = wrapStmt(std::move(s));
    return applyModifiers(std::move(fs));
}
```

The flag is not cosmetic. `$x for @a` has no implicit block, so a `my` declaration inside `$x` is *not* scoped away when the loop ends — unlike `for @a { … }`. Every looping and topicalising node carries the same flag for the same reason.

4.3 Block or hash?

`{ … }` is Raku's classic ambiguity. The parser uses a small `looksHash` heuristic: it is a hash if the first token is a pair key followed immediately by `=>`, or a colon-pair (`:name`, `:$v`, `:!flag`), or a `%`-variable followed by `,` or `}`. The empty case is decided by **position**:

```
// src/Parser.cpp — a statement-leading '{' is a BLOCK unless looksHash
// src/Parser.cpp — parsePrimary, expression position:
if (a.kind == Tok::RBrace) isHash = true;    // {} in value context: a Hash
```

So a bare `{}` statement is an empty block, and `{}` where a value is expected is an empty hash. `{ $_ * 2 }` — first token a `$`-variable, no `=>` — is a block.

There is a second, subtler brace rule. A `}` that closed a **block** at the end of a line ends the statement, so an infix on the next line starts a new statement rather than continuing the expression:

```
($r, $g, $b) = (...).map: { ... }      # the statement ends here
%(r => $r, ...)                        # a new statement, not `} % (...)`
```

A subscript's `}` does not count, because `%h{'a'}` followed by a newline and `+ 3` really is a continuation. That is why the parser records the token *index* of the last block-closing brace, `lastBlockClose_`, rather than testing the token kind.

4.4 Expressions: the Pratt core

```
// src/Parser.cpp
ExprPtr Parser::parseExpr(int minbp) {
    ExprPtr lhs = parsePrefix();
    for (;;) {
        InfixInfo in = classifyInfix(cur());
        if (!in.valid || in.lbp < minbp) break;
        advance();
        int nextMin = in.rightAssoc ? in.lbp : in.lbp + 1;
        ExprPtr rhs = parseExpr(nextMin);
        // ... build a Binary / Assign / Range / Pair ...
    }
    return lhs;
}
```

Read a prefix term; then loop, consuming any infix whose left binding power is at least `minbp`, recursing for the right-hand side with a bound derived from the operator's precedence and associativity. For a left-associative operator the recursive bound is `lbp + 1`, which is what stops it re-associating.

Binding powers are a named enum, higher meaning tighter, and **spaced by ten**:

```
// src/Parser.cpp
enum {
```

```
    BP_OR = 10, BP_AND = 20, BP_ZIP = 25, BP_COMMA = 30, BP_ASSIGN = 40,
    BP_TERNARY = 50, BP_OROR = 60, BP_ANDAND = 70, BP_COMPARE = 80,
    BP_RANGE = 90, BP_CONCAT = 100, BP_REPLICATE = 110, BP_ADD = 120,
    BP_MUL = 130, BP_POW = 140, BP_PREFIX = 150
};
```

The gaps are the whole point, and the next chapter spends them.

Prefix operators (`- ! ~ + ? ++ --`) are handled in `parsePrefix`; postfix ones — `++`, `--`, method calls, subscripts, `.()` — in `parsePostfix`. The two interleave around the primary term, so `-$obj.foo[0]++` composes in the right order without a special case.

4.5 Declarations

The declaration parsers do the bulk of the work, because Raku declarations carry a great deal of information that only matters at run time. None of it is *checked* during parsing; it is recorded as data for the binder and the dispatcher.

`parseSub` reads the name (or an operator declaration, Chapter 5), the signature, a trait loop, an optional return type from `-->` / `of` / `returns`, and the body. It also handles `multi` and `proto`, `method`/`submethod`, alternate signatures `(a) | (b)` sharing one body, and the immediate-call form `sub f($n) {...}(1)`.

The trait loop is worth a note. Built-in traits (`is export`, `is native(...)`, `is rw`, the precedence traits) are consumed into dedicated fields; anything else is captured as a `SubTraitSpec` — a name and an unevaluated argument expression — and dispatched at run time to a user-defined `multi trait_mod:<is>`. That is how a module's `is json-name('x')` reaches its own handler without the parser knowing anything about it.

`parseSignature` builds a `std::vector<Param>`. One `Param` carries a remarkable number of flags, all of them from Chapter 6's struct: positional or named, optional or required, the slurpy kind (`*@` flattening, `**@` non-flattening, `+@` single-arg rule), a default expression, a type constraint, a `:D`/`:U` smiley, a `where` clause, a coercion type `Int(Str)`, `is rw` / `is copy` / `is raw`, and a nested sub-signature for destructuring parameters like `[$a, $b]`.

`parseClass` covers `class`, `role`, `grammar`, `module` and `package` in one function. It parses `is` and `does` parents, then a body of `has` attributes, `method`/`submethod`/`multi`

declarations, and — for grammars — `token`/`rule`/`regex` rules, whose pattern is the opaque `Tok::RegexLit` the lexer captured.

`parseEnum` and **`parseSubset`** leave their interesting parts as expressions: the enum's value list and the subset's `where` clause are evaluated by the interpreter, not the parser.

4.6 String interpolation is parsed, not concatenated

A `"...$x...{ code }..."` literal goes through `parseInterpString(raw)`, which builds an `InterpStr` node whose `parts` alternate literal `StrLit` chunks with parsed sub-expressions. It recognises `$x`, `@a[...]`, `%h{...}`, the capture variables `$/`, `$0`, `$<name>`, backslash escapes including `\x[...]` and `\c[NAME]`, and `{ EXPR }` blocks.

There is a commit rule for method chains that matches Raku's: `"$obj.foo"` interpolates the `.foo` only if a call or a subscript follows it; otherwise the `.foo` stays literal text.

Embedded fragments are parsed by a fresh lexer and parser, which **inherits the program's user-declared operators**:

```
// src/Parser.cpp — parseEmbeddedExpr
sub.userInfix_ = userInfix_;   sub.userInfixRight_ = userInfixRight_;
sub.userPrefix_ = userPrefix_; sub.userPostfix_ = userPostfix_;
sub.userCircumfix_ = userCircumfix_;
sub.userPostcircumfix_ = userPostcircumfix_;
```

so `"{ 5! }"` sees the program's own `postfix:<!>`.

4.7 Compile time: the parser runs nothing

This is a real divergence from Rakudo and it shapes what Raku++ can support.

- **Phasers** become `Block` statements tagged with a phaser name. `BEGIN` is not executed during parsing; the interpreter schedules it after the parse.
- **`constant`** is not folded. It becomes a declaring `VarExpr` that the interpreter binds.
- **`use` and `no`** become `UseStmt` nodes. No module is loaded, no pragma takes effect, and even `no strict` is handled at run time rather than by toggling a parser mode.

- **Named subs are hoisted** — but by the *interpreter*, not the parser. `hoistSubs` pre-registers every named `SubDecl` in a scope before running its statements, which is why subs need not be declared before use.

The one parse-time side effect in the entire front end is registering a user-declared operator, which is lexical bookkeeping rather than execution. `use Foo` participates in exactly that much: `scanModuleOps` finds the module's source and *text-scans* it for operator declarations, so the rest of the importing file parses. It does not lex or parse the module (Chapter 29).

4.8 Errors

```
// src/Parser.h
struct ParseError : std::runtime_error {
    int line;
    std::string exType;    // X::Parameter::Twigil, … ("" = generic)
    std::vector<std::pair<std::string, std::string>> exAttrs;
    bool atEof = false;
};
```

A parse error carries a line, and optionally a *typed* diagnostic — an exception class name plus attributes — so that compile-time failures Roast checks by class can be reported as that class rather than as a generic message. The driver prints `===SORRY!===` and exits 1, matching Rakudo's compile-error shape. `atEof` is the REPL's continuation signal, as in the previous chapter.

`checkRedeclarations` runs after the parse over each scope's statement list and reports duplicate subs and types in the same scope — one of the few checks that happens before anything runs.

4.9 Honest limitations

- **Single pass, declared before use, for operators.** A custom operator used above its declaration will not parse. Subs are hoisted at run time; operator tables are populated textually.
- **A purely symbolic user infix may not be picked up at the use site.** The infix use-site branch keys on a `Tok::Ident`, so word-form infixes (`4 avg 6`) work; prefix, postfix and circumfix handle symbols fine.

- **The operator vocabulary in the lexer is hand-ordered**, so "longest match" is really "first match in a manually ordered table".
- **The block/hash rule is a heuristic.** It implements Raku's documented rules, but pathological cases a backtracking grammar would resolve can still surprise it.
- **No compile-time execution**, which is why a `BEGIN` block cannot influence how later source parses — and, ultimately, why macros and slangs are out of scope.

Chapter 5

User-Defined Operators in a Single Pass

Raku lets a program add operators to the language it is written in:

```
sub postfix:<!>($n) { [*] 1..$n }
say 5!;                                  # 120

sub infix:<avg>($a, $b) is tighter(&infix:<+>) { ($a + $b) / 2 }
say 4 avg 6 + 10;                        # 15 — (4 avg 6) + 10

sub circumfix:<⟦ ⟧>(*@x) { @x.sum }
say ⟦ 1, 2, 3⟧ ;                           # 6
```

That looks like it needs a parser that can rewrite its own grammar. It does not. It needs a parser that keeps a mutable table and reads the file exactly once.

5.1 The tables

```
// src/Parser.h — mutated during the parse, consulted at every operator site
std::map<std::string, int> userInfix_;    // name → left binding power
std::set<std::string> userInfixRight_;    // declared `is assoc<right>`
std::set<std::string> userPrefix_, userPostfix_;
std::map<std::string, std::string> userCircumfix_, userPostcircumfix_;
```

Five containers, all keyed by the operator's spelling. `userInfix_` maps to a binding power so a user infix can sit at any level; the bracket maps go from opening delimiter to closing delimiter.

5.2 Registration happens mid-parse

When `parseSub` reads a sub whose name is one of the operator categories followed by a colon, it pulls the operator's spelling out of the `<...>` and writes it into the tables **on the spot**:

```
// src/Parser.cpp — parseSub, operator declaration
if ((s->name == "infix" || s->name == "prefix" || s->name == "postfix" ||
     s->name == "circumfix" || s->name == "postcircumfix") && isOp(":")) {
    std::string cat = s->name; advance();               // ':'
    std::vector<std::string> w;
    if (isOp("<")) { advance(); w = readAngleWords(">"); }
    std::string opname = w.empty() ? "" : w[0];
    s->name = cat + ":<" + opname + ">";
    if (cat == "infix") { userInfix_[opname] = BP_ADD; declInfix = opname; }
    else if (cat == "prefix")  userPrefix_.insert(opname);
    else if (cat == "postfix") userPostfix_.insert(opname);
}
```

From that token onward, the operator is live. Everything *later* in the file sees it; everything earlier does not. That asymmetry is not a limitation invented here — it is Raku's own rule, and it falls out of the single pass for free.

The declaration is otherwise an ordinary sub. Its name becomes the string `"postfix:<!>"`, and subs live in the environment under a `&`-prefixed key, so at run time it is found exactly like any other routine.

5.3 How `5!` parses

Take `sub postfix:<!>($n) { [*] 1..$n }; say 5!;`.

The declaration runs the branch above and does `userPostfix_.insert("!")`. Later, parsing `5!`: `parseExpr` calls `parsePrefix`, which calls `parsePostfix(parsePrimary())`. `parsePrimary` returns the `5`. `parsePostfix` loops and reaches the user-postfix arm:

```
// src/Parser.cpp — parsePostfix
} else if ((cur().kind == Tok::Op ||
            (cur().kind == Tok::Ident && !cur().spaceBefore)) &&
```

```
            userPostfix_.count(cur().text)) {
    std::string opname = advance().text;
    auto call = std::make_unique<Call>();
    call->name = "postfix:<" + opname + ">";
    call->args.push_back(std::move(base));
    base = std::move(call);
}
```

`5!` becomes a `Call` node named `postfix:<!>` with one argument. The runtime never learns that an operator was involved; it dispatches a named call. Infixes work the same way through a branch at the top of the `parseExpr` loop, building a two-argument `Call` named `infix:<op>` and honouring the recorded binding power and associativity.

That is the whole trick, and it is worth stating plainly: **a user-defined operator is a parse-time spelling for an ordinary sub call.** Every mechanism that works on subs — closures, multi dispatch, `&infix:<avg>` as a value, passing one to `.reduce` — works on operators without further effort.

5.4 Precedence traits and the gaps of ten

A user infix registers at `BP_ADD` by default. A precedence trait resolves the referenced operator's level and slots the new one five away:

```
// src/Parser.cpp — parseSub, `is tighter/looser/equiv(&infix:<+>)`
int refBp = infixBpOf(refOp);
userInfix_[declInfix] = trait == "equiv"   ? refBp
                      : trait == "tighter" ? refBp + 5
                      :                      refBp - 5;
if (kind == "right") userInfixRight_.insert(declInfix);
```

This is what the by-ten spacing in the binding-power enum was reserved for. `is tighter(&infix:<+>)` yields 125: tighter than `+` at 120, looser than `*` at 130. `infixBpOf` resolves the referenced operator whether it is built in or itself user-declared, so traits chain.

The obvious question is what happens when someone declares eight operators each tighter than the last. The answer is that they collide at the same level after the gap is exhausted — the levels are integers with a fixed spacing, not a rational ordering. It has not come up in practice, and the alternative (a real partial order rebuilt on each declaration) has not been worth building.

5.5 Custom brackets

A `circumfix:<  >` or `postcircumfix` declaration registers an open-to-close mapping. `parsePrimary` consults `userCircumfix_` when it meets an unrecognised opening token, and `parsePostfix` consults `userPostcircumfix_` after a term. The token classifiers that decide "does this start a term?" and "does this start a list-operator argument?" consult the tables too:

```
// src/Parser.cpp — startsTermToken / startsListopArg, the Tok::Op case ends:
userPrefix_.count(t.text) || userCircumfix_.count(t.text);
```

so a custom bracket can open an argument to a list operator, not just a standalone term.

One guard is needed: `pcfxClose_` holds the active postcircumfix's closing bracket while its content is parsed, so a bracket pair spelled with the same character on both sides does not re-open itself inside its own contents.

5.6 Operators are lexically scoped, and roll back

A declaration inside a block must not leak out of it. If it did, a `sub postfix:<!!>` in some helper block would eat every later ternary's `!!`.

So every registration is logged, and `parseBlock` rewinds to its entry mark:

```
// src/Parser.h
struct OpUndo {
    char table; std::string name; bool existed; int oldBp; std::string oldClose;
};
std::vector<OpUndo> opUndo_;

void opRollback(size_t mark) {
    while (opUndo_.size() > mark) {
        OpUndo& u = opUndo_.back();
        switch (u.table) {
            case 'i': if (u.existed) userInfix_[u.name] = u.oldBp;
                      else userInfix_.erase(u.name); break;
            case 'p': if (!u.existed) userPrefix_.erase(u.name); break;
            // … 'P' postfix, 'r' right-assoc, 'c'/'C' the bracket maps …
        }
        opUndo_.pop_back();
    }
}
```

The undo record keeps the *previous* value, not just the fact of insertion, so a block that shadows an outer operator at a different precedence restores the outer one on exit rather than deleting it.

This is an undo log rather than a stack of scopes because registration is scattered across `parseSub` and the trait handling, and a single mark-and-rewind call at one place in `parseBlock` is much harder to get wrong than a matched push and pop at each registration site.

5.7 Two escape hatches

The single-pass rule has exactly two ways out, and both are about *another* parse seeing the current one's tables.

`EVAL`. A snippet is parsed by a fresh `Parser`, which would know nothing about the enclosing program's operators. So the evaluator pre-seeds them:

```
// src/Parser.h
void declareUserOp(const std::string& kind, const std::string& name) {
    if (kind == "infix") userInfix_[name] = 120;
    else if (kind == "prefix")  userPrefix_.insert(name);
    else if (kind == "postfix") userPostfix_.insert(name);
}
```

which is why this works:

```
use MONKEY-SEE-NO-EVAL;
sub infix:<z>($a, $b) { $a * 10 + $b }
say EVAL("40 z 2");                    # 402
```

String interpolation, shown at the end of the previous chapter, copies the live tables wholesale into the sub-parser.

5.8 `use Foo` and imported operators

A module that declares an operator changes how the *importing file* parses. Nothing else about a module does. So `use Foo` triggers exactly one compile-time action, and it is not a parse:

```
// src/Parser.h
void scanModuleOps(const std::string& module);
void scanOpsIn(const std::string& src, const std::string& srcPath);
```

`scanModuleOps` resolves the module's *source file* through the same search path the loader uses and **text-scans** it — a substring search over raw bytes, no lexing, no parsing — for `sub`, `multi`, `proto` and `only` declarations of the five operator categories, registering what it finds.

Two consequences worth knowing:

- An operator spelled only in ASCII operator characters is **skipped**. A `multi infix:<*>(Color, Real)` is almost always extending a built-in, and registering it as a user operator would give it the default precedence and silently reshape every expression in the importing file.
- The scan can in principle be fooled by an operator declaration inside a string or a comment. Nothing has tripped on it yet.

Every source the scan read is recorded in `opScanned_` as a (path, content) pair. That is not for the parse; it is for the cache. A file's parse is only valid while the modules it scanned still declare what they declared — an imported module that gains or loses an operator changes how *this* file parses, without this file changing at all. Chapter 28 uses that list as a cache key.

5.9 The line this draws

Everything above is table manipulation: cheap, local, and reversible. That is why all six operator categories work, with precedence and associativity, in a parser that runs no user code.

The features on the other side of the line — `macro`, `quasi`, `RakuAST`, slangs — need the parser to *execute user code mid-parse and then rewrite its own grammar with the result*. None of them are implemented, and the reason is structural rather than a matter of effort.

There is a compensation. Because the language Raku++ accepts stays static enough to know in full at build time, the whole program can be compiled ahead of time — which is what the `--aot` and `--exe` modes exploit. The one remaining genuinely dynamic construct, a runtime-constructed grammar, is exactly what those modes fall back to bundling.

Chapter 6

The AST

The parser's output is a `Program`: a vector of statement pointers. Everything downstream — the interpreter, the two compiling back ends, the linter, the highlighter, the serialiser — consumes that one tree.

```
// src/Ast.h
struct Node { NK kind; int line; virtual ~Node() = default; };
struct Expr : Node { using Node::Node; };
struct Stmt : Node { using Node::Node; std::string label; };
using ExprPtr = std::unique_ptr<Expr>;
using StmtPtr = std::unique_ptr<Stmt>;
struct Program { std::vector<StmtPtr> stmts; };
```

6.1 A class hierarchy here, a fat struct there

The runtime's `Value` is a single struct with a tag and a field for every payload (Chapter 7). The AST is the opposite: a real C++ class hierarchy, one `struct` per node type, each with exactly the fields it needs. Both choices are right, for opposite reasons.

An AST node is **built once, lives for the whole run, and is visited by a `switch` on its tag**. Nodes are few, long-lived, and heterogeneous — some carry two pointers, `ClassDecl` carries seventeen fields including four vectors. A fat struct there would waste far more than it saved, and the virtual destructor costs nothing that matters when construction happens once.

A `Value`, by contrast, is created and destroyed millions of times per second and must be trivially copyable into vectors. Same program, opposite trade.

The `NK` enum tags every node so both the interpreter and the code generator can `switch` rather than dispatch virtually:

```
// src/Ast.h
enum class NK {
    IntLit, NumLit, StrLit, InterpStr, BoolLit, VarExpr, ListExpr,
    Assign, Binary, Unary, Call, MethodCall, Index, Ternary, Range,
    Pair, BlockExpr, ArrayLit, HashLit, NameTerm, RegexLit, SubstLit,
    ChainExpr, SymbolicRef, AllomorphLit, NqpOp,
    ExprStmt, VarDecl, SubDecl, IfStmt, WhileStmt, ForStmt, LoopStmt,
    Block, ReturnStmt, LastStmt, NextStmt, RedoStmt, UseStmt, EmptyStmt,
    GivenStmt, WhenStmt, RepeatStmt, Whatever, ClassDecl, SelfTerm,
    EnumDecl, NamedRegexDecl, SubsetDecl,
};
```

Forty-nine kinds for a language of Raku's size is small, and the reason is that most of Raku's surface syntax lowers into a handful of these. A hyper operator, a metaoperator, a junction constructor and a set operation are all `Binary` nodes with a different `op` string. A `given`/`when` chain is `GivenStmt` plus `WhenStmt`. Anything spelled as an operator, including every user-declared one, is a `Call`.

6.2 The nodes that carry the most

Three node types hold most of the language's declarative weight.

`Param` is not a `Node` at all — it is a plain struct, because signatures are lists rather than trees — and it is the densest thing in the header:

```
// src/Ast.h — Param, abridged
struct Param {
    std::string name;         // includes the sigil; empty for anonymous
    char sigil = '$';
    std::string type;         // constraint name; "" = unconstrained
    ExprPtr whereExpr;        // `where` clause
    ExprPtr litVal;           // literal parameter: MAIN('population')
    ExprPtr defaultVal;
    std::string namedKey;     // external name for :name($var)
    char slurpyKind = 0;      // 'f' = *@, 'n' = **@, '1' = +@
    bool named, slurpy, optional, required, invocant;
    int defConstraint;        // 0 none, 1 = :D, 2 = :U
```

```
    bool coerce;              // Int(Str)
    bool isRw, isCopy, isRaw;
    std::shared_ptr<std::vector<Param>> subSig;   // destructuring
    // …plus decided-once caches, below
};
```

None of that is validated at parse time. It is data for the binder (Chapter 13) and the multi-dispatcher (Chapter 15).

`SubDecl` carries the routine: name, parameters, alternate signatures sharing one body, the body, non-built-in traits kept as unevaluated expressions, the `multi`/`proto`/`method`/`submethod`/private flags, export and `our` scoping, the declared return type, declarator pod, and the four fields that describe an `is native` declaration.

`ClassDecl` covers `class`, `role`, `grammar`, `module` and `package` at once: parents and roles, attributes, methods, grammar rules, the `parameterized` flag for `role R[T]`, the `:ver`/`:auth`/`:api` adverbs both as strings *and* as expressions (because `module Zef:ver($?DISTRIBUTION.meta<version>)` is legal and must be evaluated when the declaration runs), the `is repr(…)` layout for NativeCall, and a statement body for the package forms.

6.3 Fields that are answers about the syntax

Several nodes carry a small mutable field that the parser leaves undecided and the first evaluation fills in:

```
// src/Ast.h — Binary
mutable DecidedOnce<signed char> simpleOp{-1};
mutable DecidedOnce<signed char> fastShape{-1};
mutable DecidedOnce<const void*> litVal{nullptr};
```

```
// src/Ast.h — Index
mutable DecidedOnce<signed char> fastShape{-1};
mutable DecidedOnce<long long> litIdx{0};
```

and similarly:

```
Block::hoistNeed          ForStmt::hasStateCache
StrLit::nfcDone           NumLit::cacheN, cacheD
Param::sigSimple, natSpec, typeKnown
Callable::arityShape, catchScan
```

What they hold is a **fact about the program text**, not a cached result. "Is this binary's left child a plain lexical and its right child a scalar literal?" is true before the program starts and stays true until it exits, because source code does not rewrite itself. Answering it costs several branches; answering it once per node instead of once per evaluation is the entire optimisation, and Chapter 18 measures what that is worth.

Two invariants keep them safe:

- **A value is never cached, only a classification** — except where the value is a literal, which is a constant by definition.
- **The sentinel means undecided**, so a fresh tree and a tree that has been running for an hour behave identically. That is what lets the serialiser skip these fields entirely (Chapter 28).

The `DecidedOnce<T>` wrapper is a relaxed atomic, for the reason given in Chapter 2: under parallel execution several threads may compute the same idempotent answer concurrently, and a plain field would make that a data race.

6.4 Two nodes that only exist sometimes

`NqpOp` implements the `use nqp` compatibility subset. It exists *only* when the parser saw `use nqp` and then met an `nqp::` call:

```
// src/Ast.h
enum class NqpOpc : uint16_t { Stmts, While, Until, IfNull, IseqI, AddI,
                               Substr, Chars, Concat, /* ~50 more */ };
struct NqpOp : Expr {
    NqpOpc op;
    std::vector<ExprPtr> args;
};
```

A program that never says `use nqp` builds no such node, so the interpreter's `NqpOp` arm is never reached and the implementation is dead code for that run. Chapter 30 is about why that zero-cost property is structural rather than an optimisation.

`AllomorphLit` exists because a numeric word inside a `<...>` list is simultaneously a number and its own source spelling: `<42>` is an `IntStr`, `<1/3>` a `RatStr`. The node keeps both the parsed numeric expression and the original text.

6.5 Borrowed pointers, and why the tree is immortal

A `Callable` — the runtime object behind every sub, block and method — does not own its code:

```
// src/Value.h — Callable, excerpt
const std::vector<Param>* params = nullptr;   // borrowed from the AST
const std::vector<StmtPtr>* body = nullptr;   // borrowed from the AST
std::shared_ptr<Env> closure;                 // owned
```

Only the environment is owned. The parameter list and the body are raw pointers into the tree. Calling a sub walks those AST statements directly; there is no copy, no bytecode, and no intermediate representation.

The consequence is that **the AST must outlive every value that refers to it**. For the main program that is automatic. For anything parsed later it is not, so the interpreter keeps them alive explicitly:

```
// src/Interpreter.h
std::vector<std::shared_ptr<Program>> keptPrograms_;   // EVAL'd ASTs
```

and `loadModule` pushes each module's `Program` onto the same vector (Chapter 29). Nothing is ever released from it. That is a deliberate, bounded leak: the number of distinct compilation units a process loads is small, and the alternative — refcounting subtrees — would cost on every call.

6.6 Seeing the tree

```
rakupp --dump-ast prog.raku
RAKUPP_DUMPTOKENS=1 rakupp prog.raku
```

`AstDump.cpp` prints the tree as an indented plain-text outline, one line per node with the fields that matter for that kind. It is the first thing to reach for when a parse produces something unexpected, and it is also how `tools/ast-opportunity.raku` counts syntactic patterns across a corpus — the tool that decided, with numbers, that constant folding was not worth building and node specialisation was (Chapter 18).

6.7 The two emitters that must know every field

Two back ends consume the tree structurally rather than semantically, and both break loudly if a field is added and forgotten.

`AstEmit.cpp` (`--aot`) emits C++ that rebuilds the identical tree. It throws `AstEmitError` on any construct it cannot reconstruct, and the driver answers by bundling the source instead — so a missed field degrades to a slower binary rather than a wrong one.

`AstSerial.cpp` (the precompiled parse) writes and reads a binary encoding. Its defence is stronger, and worth copying: **both directions are driven by one visitor per node type**, so a field cannot be written but not read. That is the failure mode which makes a format like this quietly wrong rather than loudly broken. A version constant in the header is bumped whenever the shape changes, and a mismatched entry is ignored rather than reinterpreted.

Part III

The Value Model

Chapter 7

`Value`: the Fat Tagged Struct

Raku is dynamically typed. A variable holds an `Int` now and a `Hash` later, routines dispatch on the runtime types of their arguments, lists can be infinite, and an object can grow a role while the program runs. Raku++ is written in statically-typed C++17 with no garbage collector. One data structure carries the entire distance between those two facts.

```
// src/Value.h
enum class VT { Nil, Any, Bool, Int, Num, Str, Array, Hash, Code, Range,
                Pair, Type, Whatever, Object, Rat, Regex, Match, Complex };

struct Value {
    VT t = VT::Any;              // the discriminator
    bool b; long long i; double n, im;    // Bool / Int / Num / Complex imag
    CowStr s;                    // Str; also the Type name, the Pair key
    IStr hashKind, enumName, enumType;    // interned secondary tags
    bool isList, itemized, readonly, namedArg, fatRat, objKeyed;
    std::shared_ptr<ValueList> arr;                    // Array / List / Seq
    std::shared_ptr<std::map<std::string, Value>> hash;
    std::shared_ptr<Callable> code;
    std::shared_ptr<Value> pairVal, pairKey;
    std::shared_ptr<ObjectData> obj;
    std::shared_ptr<void> ext;   // opaque: Promise, Channel, lazy-seq state
    std::shared_ptr<BigInt> big, ratN, ratD;
    std::shared_ptr<std::vector<long long>> shape;
    long long rFrom, rTo; bool rExFrom, rExTo, rNum;   // Range
    std::string ofType; int natBits; bool natSigned, natFloat;
};
```

Eighteen tags, eleven `shared_ptrs`, and a pile of flags. It looks indefensible. It is not, and the case for it is the most important argument in this book.

7.1 Why not a union, a variant, or a class hierarchy

The instinct is: a value is one of *N* types, therefore a sum type. All three standard spellings of that instinct are wrong here, each for a different reason.

A `Value` is frequently several fields at once. A union models mutual exclusivity — one active member, selected by the tag — and that is simply not what a Raku value is:

- a **`Match`** sets the subject string `s`, the span `rFrom`/`rTo`, the positional captures in `arr`, *and* the named captures in `hash`, all live together;
- a **`Pair`** is the key in `s` plus `pairVal`;
- an **enum value** is the ordinal in `i` plus `enumName` and `enumType`;
- a **`Rat`** is `ratN` plus `ratD` plus the `fatRat` flag;
- and the cross-cutting flags — `isList`, `itemized`, `readonly`, `namedArg`, `ofType`, `natBits` — are set *regardless of the tag*. A `readonly Int` and an `itemized Array` are both ordinary.

So `VT t` does not mean "which one field is valid". It means "how to read a bag of fields, several of them set". That is a struct.

A raw union of these members is undefined behaviour. They are non-trivial C++ types — a string, eleven `shared_ptrs`. Making it legal means hand-written placement-`new`, a tag-switched copy constructor, move constructor, both assignment operators and destructor. At that point you have rebuilt `std::variant`, which re-imposes the one-active-member model *and* forces `std::get`/`std::visit` on the roughly sixteen thousand lines of interpreter and built-ins that today just read `v.i`, `v.s`, `v.arr`.

The memory a union would save is smaller than it looks. The overlappable members are mostly `shared_ptr`, which is null and therefore allocation-free when unused, and by the first point many of them cannot be overlapped at all.

A class hierarchy would put a virtual call on every field access and a heap allocation on every integer. For a tree-walker whose values are overwhelmingly transient scalars, that is the worst of the three.

7.2 What the fat struct buys

No virtual dispatch and no allocation for scalars. An Int is a Value with t == VT::Int and i set. Building one is a stack operation:

```
// src/Value.h
static Value integer(long long x) { Value v; v.t = VT::Int; v.i = x; return v; }
```

Uniform copy and move. Passing a Value by value, returning it from eval, storing it in a vector — all use the compiler-generated operations. The shared_ptr members make copies of arrays, hashes and objects a refcount bump, and give them **shared identity**, which is exactly what Raku's container semantics need. Mutating an object attribute through a copied self handle works for this reason and no other.

Coercion is a method, not a cast.

```
// src/Value.h
bool truthy() const;      long long toInt() const;   double toNum() const;
std::string toStr() const;  std::string gist() const;
std::string typeName() const;
```

~$x calls toStr, +$x calls toNum, boolean context calls truthy. There is no C++ inheritance making Int "be a" Cool; these six functions encode the numeric and string towers directly, each a switch on t in Value.cpp.

7.3 Clever reuse of fields

Several Raku types have no VT of their own. They are an existing tag plus a tag-like marker, and knowing which is which explains a lot of otherwise puzzling code.

A Junction is an Array tagged by enumName. any(1,2,3) is a VT::Array whose elements are the eigenstates, with enumName set to one of any, all, one, none:

```
// src/Value.cpp — typeName(), the VT::Array case
if (enumName == "any" || enumName == "all" ||
    enumName == "one" || enumName == "none") return "Junction";
```

Set, Bag and Mix are Hashes tagged by hashKind. So are Proxy, Failure, Date, DateTime, Scalar and several more; hashKind is a secondary type tag drawn from a closed vocabulary.

An allomorph is a number that is also its own string. <42> is VT::Int with s holding "42" and hashKind naming IntStr:

```
// src/Value.h
bool isAllomorph() const {
    return (t == VT::Int || t == VT::Rat || t == VT::Num || t == VT::Complex) &&
           (hashKind == "IntStr" || hashKind == "RatStr" ||
            hashKind == "NumStr" || hashKind == "ComplexStr");
}
```

An `Int` grows a bignum only when it must.

```
// src/Value.h
static Value bigint(const BigInt& b) {
    Value v; v.t = VT::Int;
    if (b.fitsLL()) v.i = b.toLL();
    else v.big = std::make_shared<BigInt>(b);
    return v;
}
```

A `FatRat` is a `Rat` with a flag. Same storage, but the flag carries the type identity — contagious through arithmetic — and exempts the value from the denominator spill described in Chapter 10.

A native integer container is a `Value` with a width. `my uint8 $b` sets `natBits = 8`, `natSigned = false`, and every assignment masks to that width.

7.4 `ext`: the opaque slot

One member is deliberately untyped:

```
std::shared_ptr<void> ext;
```

It carries whatever state a value needs that has no business in the struct: a `PromiseState`, a `Channel`, a `TapHandle`, a `LazySeqState`, a `CueState`, or the endpoint objects of a `Range`. Each consumer knows what it parked there and static-casts it back.

That is not as loose as it sounds, because the tag plus `hashKind` always determine which kind of state can be present. But it is the one place in the value model where the type system has been switched off on purpose, and it is worth being aware of when adding a case.

7.5 The identity problem, and how ext solves it

Some Raku types are *reference* types: two of them are the same one only when they are the same object. Buf, Instant and Duration are three. Here they are plain tagged scalars — a Buf is a Str with hashKind = "Buf", an Instant a Num — and with no address to compare, === fell through to comparing the *rendering* and declared two independently built buffers identical.

That is not academic. It made @!outstanding-writes .= grep({ $_ !=== $p }) unemptiable for Promises, and it was worse for Buf, which is mutable: dropping one buffer from a list by !=== $buf threw away every buffer that happened to hold the same bytes.

```
// src/Value.h
inline bool identityScalar(const Value& v) {
    return (v.t == VT::Str && v.hashKind == "Buf") ||
           (v.t == VT::Num && (v.hashKind == "Instant" ||
                               v.hashKind == "Duration"));
}
inline Value& identify(Value& v) { v.ext = std::make_shared<char>(); return v; }
```

Each freshly built one stamps a unique token into ext. A plain Value copy carries the token along — which is precisely what "the same object" means for these types — and === compares it. Blob stays out: it is immutable and compares by value in Rakudo too.

The same trick answers a smaller question. $*INIT-INSTANT is one Instant read many times, and $*INIT-INSTANT === $*INIT-INSTANT must be True, so both reader sites go through one function that hands back the same stored token.

7.6 Ranges remember what they were written with

A Range iterates over integers in rFrom/rTo, or over doubles in n/im when it is fractional. But 1/2 .. 1/3 must keep its Rats and True .. False its Bools for .min, .max, .bounds and rendering.

```
// src/Value.h
struct RangeEnds { Value from, to; };
inline void setRangeEnds(Value& r, const Value& from, const Value& to) {
    auto keep = [](const Value& v) {
        return v.t == VT::Rat || v.t == VT::Num || v.t == VT::Bool ||
               v.t == VT::Nil || v.t == VT::Any || v.t == VT::Type;
```

```
    };
    auto renders = [&](const Value& v) { return keep(v) || v.t == VT::Int; };
    if (!keep(from) && !keep(to)) return;
    if (!renders(from) || !renders(to)) return;
    attachRangeEnds(r, from, to);
}
```

The endpoints are parked in `ext`, and only when they would actually render differently. `..` numifies a `Str` endpoint (`"2"` becomes `2`) and a list endpoint (its element count), so those must *not* be carried; and an `Int` renders identically either way, so carrying it would only cost an allocation on the very hot `1..n` path.

That last clause is the pattern to notice. Almost every field in `Value` is guarded by a test that keeps the common case free.

7.7 The rendering hook

`Value.h` cannot call into `Builtins.cpp`, but a `Range`'s endpoints must render with `.raku`, which lives there. The solution is a function pointer:

```
// src/Value.h
using RakuReprFn = std::string (*)(const Value&);
extern RakuReprFn g_rakuRepr;
```

A raw pointer is zero-initialised before any dynamic initialisation runs, so installing it from another translation unit is order-safe — unlike a `std::function`, which would have its own construction order. There are a handful of these hooks in the runtime (`g_objListItems`, `g_deproxy`), each solving the same layering problem the same way.

7.8 What it costs

	bytes
`long long`	8
`std::string` (libc++)	24
`CowStr`	40
`Value`	**392**, on the build these numbers were taken from

Every `Value` carries every field, live or not. A `ValueList` of integers is about fifty times the size of a `vector<int64_t>`. That is the price of branch-free field access and trivial copyability, and it is a real price: the profile of a method-call-heavy loop puts 31% of the time in heap allocation and 11% in `Value` copy and destruction.

The number moves. It was 392, then 376, then 344 in the course of ordinary optimisation work, and the representation plan intends roughly 204 next. Two of the reductions are already in this chapter: `hashKind`, `enumName` and `enumType` became interned 8-byte handles instead of 24-byte strings (Chapter 9), and the string payload became a copy-on-write type (Chapter 8).

That instability is also the single most important input to the extension ABI in Chapter 32, which is why an extension module never sees this struct at all.

7.9 Honest limitations

- **Memory per value**, as above. It is the accepted cost of the design, not an oversight, but it is the thing to attack first if the design is ever revisited.
- **`ext` is untyped.** Nothing but convention stops two subsystems from parking incompatible state in it for the same tag.
- **Reference semantics are emulated.** The identity-token trick covers the three scalar types that needed it; a fourth would need the same treatment added by hand.
- **The tag set is closed.** Adding a `VT` means auditing every `switch` over it in the interpreter, the code generator, the serialiser and the dumper — which is exactly why new Raku types are added as an existing tag plus a marker instead.

Chapter 8

Strings: `CowStr`

`Value` holds its `Str` payload in a `CowStr`, not a `std::string`. This chapter is about why that class had to be written rather than reached for, and it is the clearest example in the project of a design forced entirely by measurement.

8.1 The problem

A `Value` is copied by value everywhere: every argument pass, every operand evaluation, every list element, every return from `eval`. With a `std::string` member, a long string was `memcpy`'d on every one of those.

That is O(length) **per operation**, which makes any tokenizer written in Raku quadratic. `JSON::Fast` needed 13.9 seconds on a 421 KB document that Rakudo parses in 51 milliseconds, and the profile was copying, not parsing.

Measured, 20,000 operations on a 200 KB string:

	`std::string`	`CowStr`
pass the string to a sub, no work done	142 ms	24 ms
`nqp::ordat` on a global	79 ms	11 ms

8.2 What it is

Two arms, exactly one live:

```
// src/Value.h
class CowStr {
    std::string s_;                          // short: inline
    std::shared_ptr<const StrBody> p_;       // long: shared, immutable
    static constexpr size_t kPromote = 64;

    void take(std::string x) {
        if (x.size() >= kPromote) {
            p_ = std::make_shared<const StrBody>(std::move(x)); s_.clear();
        } else { s_ = std::move(x); p_.reset(); }
    }
public:
    const std::string& str() const { return p_ ? p_->text : s_; }
    std::string& mut() { if (p_) { s_ = p_->text; p_.reset(); } return s_; }
};
```

Short strings stay inline, where `std::string`'s own small-buffer optimisation already makes a copy free and sharing would cost an allocation to save nothing. Long strings are promoted once into a shared immutable body, after which copying is a refcount bump.

Three decisions inside that deserve their own paragraphs.

8.2.1 Promotion is eager

At construction, not lazily on first copy. Raku++ runs work in parallel, and a lazy promotion would have to mutate the source from a `const` copy constructor — two threads copying the same `Value` would race on it.

Eager promotion means **a `const CowStr` is never written to at all**, which is what makes the type safe to share without a lock. That single property is what lets everything else here be lock-free.

8.2.2 The threshold is 64

Above every mainstream small-buffer capacity, measured:

standard library	`sizeof(std::string)`	SSO capacity
libc++ (clang)	24	22
libstdc++ (g++ 14, 16)	32	15

So 64 leaves a band in which a copy is a couple of words and no allocation happens either way, and only promotes when copying is the thing being paid for. It is not a tuned number; it is a deliberately safe one.

8.2.3 The body caches string properties

This is the part no general-purpose string type can do, and the real reason `CowStr` exists rather than a third-party copy-on-write string.

```
// src/Value.h
struct StrBody {
    std::string text;
    mutable std::atomic<signed char> allAscii{-1};
    mutable std::atomic<signed char> crFree{-1};
    mutable std::atomic<long long>   nGraphemes{-1};
};
```

Raku string positions are **grapheme** positions, and Raku++ stores UTF-8 bytes. So every indexing operation must first establish whether a byte index is also a grapheme index. Establishing that by scanning is O(position) per character examined — which is the *other* half of the quadratic, independent of copying.

Because the promoted body is immutable, the answer can be computed once and cached on it. Three flags suffice:

- `allAscii`: every byte below 0x80, so a byte index is a codepoint index;
- `crFree`: no carriage return which, together with `allAscii`, means a byte index is a **grapheme** index — CR LF is the one ASCII sequence that clusters under UAX #29;
- `nGraphemes`: the answer to `.chars`.

They are reached through `cowAllAscii`, `cowByteIsGraphemeIndex` and `cowGraphemeCount` in `BuiltinsShared.h`.

The caches need no lock. Two threads may each compute one, but the text is immutable, so they compute the same answer and the store is idempotent. They live on the body, so they exist only for promoted strings — which is exactly the case that needs them; short strings are cheap to rescan.

8.3 Does C++ have this already? It did, and removed it

libstdc++'s pre-C++11 `basic_string` was refcounted copy-on-write. C++11 made that non-conforming. The string requirements added contiguous storage, O(1) non-`const` `operator[]` and `data()`, and iterator-invalidation rules under which one copy's mutation must not disturb another copy's references. Copy-on-write cannot satisfy all of them at once: non-`const` element access would have to detach, and detaching is O(n). GCC 5's dual ABI (`_GLIBCXX_USE_CXX11_ABI`, `std::__cxx11::basic_string`) exists to carry that break. libc++ was never copy-on-write.

C++11's replacement answer was **move semantics**, which solves "stop copying temporaries" and not this problem. Every copy in the list at the top of this chapter is a genuine copy, not a move: an argument passed to a function that keeps it, an element stored into a list.

None of the reasons copy-on-write was banned apply to `CowStr`, because it is not trying to be a `std::string`. Its `operator[]`, `data()`, `begin()` and `substr` are `const`-only; the single mutation door is `mut()`, which detaches explicitly and hands back a real `std::string&`. What C++11 outlawed was copy-on-write *hiding behind* an interface that promises O(1) non-`const` element access. `CowStr` promises no such thing.

8.3.1 The alternatives, and why each is not it

	why not
`std::string_view`	non-owning; `Value` owns its string
small-buffer optimisation	already the short arm — that is what `s_` relies on
`shared_ptr<const std::string>`	already the long arm; `CowStr` is mostly the *policy* for choosing between the two
`std::atomic<std::shared_ptr<T>>`	only needed if the pointer were rebound concurrently, which eager promotion guarantees it never is

Outside the standard the pattern is thoroughly proven and lives attached to frameworks: `folly::fbstring` is the closest relative — a drop-in `basic_string` with three

tiers, the top one copy-on-write with an atomic refcount; Qt has done "implicit sharing" across its whole container library for about twenty-five years; `absl::Cord` is a tree of refcounted immutable chunks, which wins when *concatenating* large strings rather than copying them; Rust's `Cow<'_, str>` makes the same idea explicit in the type system. LLVM went the other way entirely, with `StringRef` and `Twine` and no copy-on-write anywhere.

Why not adopt one? Folly and Qt are large dependencies for a subset of what is needed, and this project vendors nothing it can write in a page. But the deciding reason is the property cache: `fbstring`, `QString` and `Cord` would all give the copy elision and none of the `allAscii/crFree/nGraphemes` caching, which is the half of the fix that removed the *scanning* quadratic. Bolting that onto a third-party string means a side table keyed on the buffer address, with its own lifetime and thread-safety problem — strictly worse than owning forty lines.

The threshold difference makes the same argument from the other end. `fbstring` promotes at 255 because copy avoidance is all it buys. `CowStr` promotes at 64 because promotion also buys the cache.

8.4 What it costs

	bytes
`std::string` (libc++)	24
`std::shared_ptr`	16
`CowStr`	**40**
`StrBody`	40

Both arms are stored side by side even though only one is ever live, so `CowStr` is the sum rather than the maximum. That is a 16-byte, 4% growth of `Value`, paid on every `Value` everywhere. The performance gate passed and Roast came out marginally up, so it is bought and paid for — but it is a real tax and should be named as one. On libstdc++ the same layout is 48 bytes, because `sizeof(std::string)` is 32 there.

A promoted string also costs **two** allocations: `make_shared` fuses the control block with `StrBody`, but `StrBody::text` heap-allocates its own buffer for anything past the small-buffer capacity.

8.5 Rules for working with it

The type forwards about 1,100 read sites unchanged, which is the point. The places where it does not behave like a `std::string` are worth knowing.

- **`mut()` is the only write door, and it detaches.** After `mut()` the string is inline again and stays inline until it is next *assigned*, which is where promotion happens. A long string mutated in a loop is therefore unpromoted for the whole loop; if that ever shows up in a profile, build into a local `std::string` and assign once at the end.
- **References from `str()` do not survive an assignment.** The same rule as `std::string`, but easier to forget through the implicit conversion.
- **Both converting constructors are `explicit` on purpose.** With an implicit `std::string` to `CowStr` conversion alongside the `CowStr` to `const std::string&` one, every `cond ? value.s : someString` became ambiguous in both directions.
- **Comparisons and `+` are spelled out by hand.** Two real reasons: no conversion is considered for a built-in operator when neither operand is a class type with a candidate, so `CowStr == CowStr` needs its own overload; and `std::string`'s comparisons are templates, and template deduction never considers a user-defined conversion, so `value.s == "Int"` needs a real candidate too. A macro generates the twenty-four resulting overloads.
- **Do not add a non-`const` `operator[]`, `begin()` or `data()`.** That is exactly the interface that made copy-on-write non-conforming for `std::string`. Reach for `mut()`.
- **Do not cache `body()` across an assignment.** It is the raw `StrBody*`, and the `shared_ptr` that owns it can be replaced.

8.6 Deliberately left on the table

Both are real, both unmeasured, and neither should be done on faith — the project's rule is that a performance change is an interleaved A/B against `perf-guard` under the same conditions.

- **A union instead of two side-by-side members.** `fbstring` is a union and stays at `sizeof(std::string)`; the same here returns 16 bytes per `CowStr`. It costs the very readable `p_ ? p_->text : s_` and needs hand-written copy, move and destroy. Worth it only if `Value`'s size shows up in a measurement — and one attempt to blame `Value`'s size has already been measured and rejected.

- **One allocation instead of two** for a promoted string, by storing the refcount and the caches inline ahead of the characters. Only matters if the promotion *rate* is high.

8.7 Checking that it is working

The failure mode is silent: a change that puts a hot string back on the copying path costs time and nothing else. Two cheap checks.

Measure scaling, not speed. Time the operation over inputs of n, $2n$, $4n$, $8n$. Doubling per doubling is fine; quadrupling is the bug this class exists to prevent. That is the test which found `findnotcclass` after four other sites had already been fixed.

Confirm promotion is actually happening. A string that never crosses 64 bytes is never promoted and caches nothing — which is correct, but means a benchmark built from short strings proves nothing about either mechanism.

Chapter 9

Interning and Comparison: `IStr` and `MName`

Two small headers, ninety lines each, both solving the same shape of problem: **a string that is compared far more often than it is created**. They arrived from different directions and ended up with almost the same interface, which is itself the point.

9.1 `MName`: the method name as the dispatcher sees it

Method dispatch on a built-in value is a long ladder of name comparisons in `Built-ins.cpp` and its three continuation files — roughly 1,640 of them. So the comparison itself is hot.

The path to this fix is worth following, because two plausible diagnoses were ruled out by measurement before the real one was found.

A profile of `for ^3000000 { "ab".chars }` put **60% of the time in string comparison**: `_platform_strlen` 19%, an out-of-line `std::operator==(const string&, const char*)` 19%, `DYLD-STUB$$strlen` 14%, `memcmp` 8%.

That looks like the ladder's length. Earlier measurement had seemed to exonerate it: `.chars` sits 177 comparisons into the file and `.uc` 812, yet they cost the same. **Both facts are true.** Position *in the file* is not the number of comparisons *executed*, because the arms are guarded by invocant type — a `Str` invocant only ever runs the `Str` arms, and `.chars`, `.uc` and `.uniname` are all in that same group.

The real problem was that `strlen` was being called **at run time on a string literal**. Clang normally folds that away, and in a small function it does — a 1,700-branch toy reproduction compiles to zero `strlen` calls. But `methodCallInner` was about 8,900 lines with 1,640 comparisons, far past the optimizer's inlining budget: `std::operator==` stayed out of line, and out of line it cannot see the literal, so it measured it with `strlen` on every call.

The fix does not depend on the optimizer's mood. Wrap the name in a type whose `operator==` takes the literal **by reference to array**, so the length is part of the template argument:

```
// src/MethodName.h
struct MName {
    const std::string& s;
    std::size_t n;        // cached length: the first test on every comparison
    std::uint64_t pre;    // first 8 bytes packed, so short names compare as one

    explicit MName(const std::string& v)
        : s(v), n(v.size()), pre(pack(v.data(), v.size())) {}

    template <std::size_t N>
    RAKUPP_ALWAYS_INLINE bool operator==(const char (&lit)[N]) const {
        if (n != N - 1) return false;
        if (N - 1 <= 8) return pre == pack(lit, N - 1);
        return std::memcmp(s.data(), lit, N - 1) == 0;
    }
};
```

One `const MName m{mName};` at the top of the function, and every `m == "chars"` site becomes a size check — which rejects nearly every candidate outright — followed by an inlined comparison. Four compile errors resulted, all of them stream or concatenation overloads, which is why the struct also forwards `+`, `<<` and the handful of `std::string` members the chain uses.

Two details earn their keep.

`always_inline` is not optional. Left to itself the optimizer emits `MName::operator==<N>` out of line in this function too, and an out-of-line call costs more than the comparison it performs.

The first eight bytes are packed into a `uint64_t` at construction, so any name of eight characters or fewer — which is most of them — compares as a single integer against a literal packed at compile time, with no `memcmp` at all.

Measured, one million iterations:

	before	wrapper	+ always_inline & packing
`'ab'.chars`	1.63 s	1.19 s	**1.17 s**
`'ab'.uc`	1.78 s	1.15 s	**0.95 s**
`'ab'.uniname`	1.61 s	1.08 s	**0.71 s**

`.chars` barely moves because it is reached early. `.uniname` is deep in the `Str` arm and gains the most.

9.1.1 Where that leaves dispatch

Name comparison went from 60% of the profile to **8.5%**, which retires it as a target. A perfect-hash dispatch table would only be chasing that 8.5%, and it is not a drop-in: the ladder is not a pure dispatch on the name. Arms are guarded by invocant type and argument shape, and later generic arms deliberately catch what earlier specific ones decline. Turning 1,640 of those into table entries means giving each its own guard *and* preserving the fall-through order between them — a large, risky rewrite for a single-digit percentage.

The 42% now sitting in allocation and `Value` churn is the real remaining target, and it is a different problem (Chapter 15).

9.2 `IStr`: the storable counterpart

`Value` carried four `std::string` members — `hashKind`, `enumName`, `enumType`, `ofType` — that are **empty on almost every value**. An empty `std::string` is not free: it is 24 bytes, and it is constructed, copied and destroyed on every one of the roughly two million `Value` copies a 278 KB JSON parse makes.

Measured on that parse:

- `Value`'s implicit constructor, destructor and copy were **12.8% of self time**, the top line of the profile;
- `std::string == const char*` — which calls `strlen`, then `memcmp` — plus its helpers was about **10% more**, and `hashKind` is compared against a literal on paths that run per parameter per call.

The strings that ever reach these fields are a small closed set in practice: container kinds and type names. So intern them.

```
// src/IStr.h
struct IStr {
    struct Entry { std::string s; std::size_t n; std::uint64_t pre; };
    const Entry* e = nullptr;      // null IS the empty string

    static const Entry* intern(const char* p, std::size_t k) {
        if (!k) return nullptr;
        static std::deque<Entry> storage;   // stable addresses
        static std::unordered_map<std::string, const Entry*> index;
        static std::shared_mutex mu;
        std::string key(p, k);
        { std::shared_lock<std::shared_mutex> rd(mu);
          auto it = index.find(key);
          if (it != index.end()) return it->second; }
        std::unique_lock<std::shared_mutex> wr(mu);
        // … re-check, then append and index …
    }

    // interned, so identity IS equality
    bool operator==(const IStr& o) const { return e == o.e; }
};
```

Eight bytes, trivially copyable, trivially destructible. A default-constructed `IStr` costs a zeroed pointer where a `std::string` cost a constructor call. Comparing two `IStrs` is a pointer comparison, because interning makes identity equality. Comparing against a literal uses the same packed-prefix trick as `MName`.

Interning takes a lock, but only on assignment from text. Copies and comparisons — the operations `Value` does millions of times — never reach it. Readers share the lock; only a miss serialises.

The operator surface is deliberately the same shape as `MName`'s, so the roughly 1,300 existing `.hashKind == "Buf"` and `.ofType.empty()` sites compiled unchanged.

9.2.1 The intern table is never freed

That is deliberate. Entries are type names and kind tags, bounded by the program's own vocabulary, and a stable address is what makes the handle comparable by pointer.

It also imposes a rule, and there is one field that breaks it. `ofType` is **not** interned, and must stay that way until one site moves:

```
// src/Value.h
// `IO::Path`'s `:CWD` rides in `ofType` because a path value has no other
// use for it, so this field can hold a runtime DIRECTORY NAME rather than
// a type name. The intern table is append-only by design, so a program
// walking many directories would add an entry per directory and never
// release it.
std::string ofType;
```

This is the interesting failure mode of interning, and it is worth stating as a general rule: **intern a closed vocabulary, never an open one.** A field that can hold arbitrary runtime data — a path, a user string, a key — turns an append-only table into an unbounded leak. Everything else in `Value` is drawn from the program's own vocabulary of type names and is safe.

9.3 Why they are separate types

`MName` borrows a `const std::string&` and caches its length and prefix. It is a **view**, built at the top of a function and discarded at the bottom; it cannot be stored.

`IStr` **owns** a handle into a global table. It can live in a struct, be copied freely, and outlive the text it was built from.

They could in principle be unified — `IStr` could serve both roles — but a method name is compared once per call and never stored, so paying the intern lock for it would be strictly worse. Two types, two lifetimes, one idea.

Chapter 10

Numbers

Raku's numeric tower is exact where it can be. `1/3` is a rational, not a float. `2**200` is an integer, not an overflow. `0.1 + 0.2 == 0.3` is `True`, because a decimal literal is a `Rat`. Implementing that on a machine with 64-bit registers means three layers, each with a boundary the previous one falls through.

10.1 Layer 0: native integers, with overflow as a signal

Most integers in most programs fit in a `long long`, so that is the representation: `Value` with `t == VT::Int` and the value in `i`. Promotion to bignum happens exactly when an operation overflows, which means overflow must be *detected* rather than avoided.

```
// src/IntOps.h
inline bool add_ovf(long long a, long long b, long long* r) {
#if defined(__GNUC__) || defined(__clang__)
    return __builtin_add_overflow(a, b, r);
#else
    unsigned long long u = (unsigned long long)a + (unsigned long long)b;
    long long s = (long long)u; *r = s;
    return ((a ^ s) & (b ^ s)) < 0;   // both operands' sign differs from result
#endif
}
```

`IntOps.h` exists because MSVC has neither `__builtin_*_overflow` nor `__int128`. It provides `add_ovf`, `sub_ovf`, `mul_ovf`, `ctzll`, `clzll` and a `RAKUPP_HAS_INT128` flag, mapping each onto intrinsics where they exist and a hand-written fallback where

they do not. The multiply fallback for MSVC on ARM64 computes the wrapped product and then verifies it by division, which is slow and correct; the x64 path uses `_mul128` and checks that the high half is the low half's sign extension.

This header is what makes the arithmetic fast paths in Chapters 26 and 27 possible: `rtAdd` can inline the native case precisely because the overflow test is one instruction on the compilers that matter.

10.2 Layer 1: `BigInt`

```
// src/BigInt.h
struct BigInt {
    int sign = 0;                  // -1, 0, +1
    std::vector<uint32_t> mag;     // little-endian limbs, base 1e9
    static const uint32_t BASE = 1000000000u;
};
```

Base 10^9 rather than a power of two. That choice trades some arithmetic density for the thing a language runtime does constantly: **decimal conversion is free**. `toString` walks the limbs and prints nine digits each; `fromString` parses in nine-digit chunks from the right. A binary base would make printing a repeated division by 10, which for a language where every integer is eventually stringified is the wrong trade.

Multiplication is schoolbook, O(n·m). There is no Karatsuba and no FFT: the sizes that appear in practice are small, and the workloads where they are not are dominated by other costs.

Division is base-10^9 long division, and it has one wart worth naming: the per-limb quotient digit is found by **binary search** over `[0, BASE)`, costing about thirty `BigInt` multiplications per limb. That is expensive, which is why the fast path below matters so much.

10.2.1 The 64-bit fast path

Almost every bignum operation in real Raku code is on values that are not, in fact, big. They arrive as `BigInt` because a `Rat` stores its numerator and denominator as `BigInts` unconditionally — and *every* `Rat` construction calls `gcd` and then two `divmods`.

Measured on values that fit in 64 bits, `divmod` cost 2.1 microseconds and `gcd` — which is Euclid over `divmod` — cost 15.8 microseconds. So every decimal literal and every `p/q` in a program cost roughly 10 microseconds to build.

```
// src/BigInt.cpp — divmod, the fast path
// One hardware divide instead of the base-1e9 long division below, whose
// per-limb BINARY SEARCH costs ~30 BigInt multiplications.
if (a.fitsU64() && b.fitsU64()) { /* two 64-bit divides, rebuild */ }
```

```
// src/BigInt.cpp — gcd
// Euclid entirely in registers when both fit — the general loop below
// builds two BigInts per step and calls divmod, and Value::rat() calls
// this on EVERY Rat it constructs.
```

The guard is `fitsU64()`, which is a three-limb comparison against the base-10^9 digits of 2^64 − 1:

```
// src/BigInt.h
bool fitsU64() const {
    if (mag.size() > 3) return false;
    if (mag.size() < 3) return true;
    if (mag[2] != 18u)        return mag[2] < 18u;
    if (mag[1] != 446744073u) return mag[1] < 446744073u;
    return mag[0] <= 709551615u;
}
```

`Rat` construction became about nineteen times faster.

10.2.2 Two conversions to 64 bits, deliberately different

```
// src/BigInt.h
long long toLL() const;                   // SATURATES on overflow
unsigned long long toU64Wrap() const;     // WRAPS, two's complement
```

`toLL` saturates because its callers are indices, codepoints and range bounds, where a silent wrap produces garbage rather than a wrong-but-defined answer. `toU64Wrap` wraps because its caller is a native `uint64/int64` container, which is *defined* to keep the low bits — and saturating there turned every 64-bit digest word into `0x7FFF_FFFF_FFFF_FFFF`.

Two functions with the same signature and opposite policies is a smell in general. Here it is the correct answer, and the comment in the header says so at length precisely because the next person will want to merge them.

10.3 Layer 2: `Rat`

A `Rat` is a normalised pair of `BigInt`s:

```
// src/Value.h
static Value rat(BigInt n, BigInt d) {
    if (d.sign == 0) return ratZ(std::move(n), std::move(d));
    Value v; v.t = VT::Rat;
    if (d.sign < 0) { n = -n; d = -d; }          // sign lives in the numerator
    BigInt g = BigInt::gcd(n, d);
    if (!g.isZero()) { /* divide both by g */ }
    v.ratN = std::make_shared<BigInt>(n);
    v.ratD = std::make_shared<BigInt>(d);
    return v;
}
```

Normalisation is eager: the denominator is positive and the pair is reduced at construction. That is what makes == on `Rat`s a component-wise comparison rather than a cross-multiplication, and it is why `gcd`'s speed matters.

A decimal literal is a `Rat`, built by the parser as a numerator and denominator pair (`NumLit::ratNum`, `ratDen`, with decimal-string fields for the cases that overflow `long long`). The node caches the constructed `BigInt` parts, because a literal in a hot loop would otherwise re-allocate and re-reduce on every evaluation.

10.3.1 The spill, and the type that is exempt

Raku caps a plain `Rat`'s denominator at 64 bits. Arithmetic that would produce a larger one degrades to `Num`:

```
// src/Interpreter.cpp — applyArith, the Rat result
bool fat = (l.t == VT::Rat && l.fatRat) || (r.t == VT::Rat && r.fatRat);
Value v = Value::rat(std::move(n), std::move(d)); v.fatRat = fat;
if (!fat && v.ratD && !v.ratD->fitsU64()) return Value::number(v.toNum());
return v;
```

`FatRat` is the same storage with a flag, and the flag does two things: it carries the type identity, and it exempts the value from the spill so a `FatRat` stays an arbitrary-precision rational forever. The flag is **contagious** — one `FatRat` operand makes the result a `FatRat` — which is the semantics Raku specifies and also the only way a chain of operations can stay exact.

`Rat.new` is subtly different from the `/` operator: a zero denominator must be *preserved* rather than treated as an error, normalised to ±1/0 with 0/0 kept as 0/0. That is `ratZ`, and taking `Str` or `Num` of such a value throws.

10.4 Native integer containers

`my int8 $x` is not a different type at runtime; it is a `Value` carrying a width:

```
// src/Value.h
int natBits = 0;       // 0 = not native
bool natSigned = false;
bool natFloat = false; // num32: truncate to float32 on assignment
```

Every assignment masks to the declared width, so `my uint8 $b = 300` stores 44. The width is derived from the declared type name once and cached, because deriving it involved a `substr` and therefore an allocation per typed parameter per call:

```
// src/Value.h
static int natWidthOfType(const std::string& ofType, bool& sign);
```

```
// src/Ast.h — Param
mutable DecidedOnce<int> natSpec{-1};   // bits<<1 | signed, decided once
```

Typed arrays and hashes use the same table through `ofType`, so `my uint32 @a` stores masked elements.

10.5 Complex

`VT::Complex` reuses `n` for the real part and `im` for the imaginary one — the only tag that uses `im` at all, apart from a fractional `Range`'s upper bound. Coercing a `Complex` back to a real checks the imaginary part against `$*TOLERANCE`, which defaults to 1e-15 and is looked up dynamically first, then lexically.

The language revision matters here: under `use v6.e.PREVIEW`, `sqrt(-1)` is a `Complex` rather than `NaN`. `Interpreter::langRev_` carries the revision and a handful of numeric operations branch on it.

10.6 Where the arithmetic actually happens

One function, `applyArith(op, l, r)`, implements every binary operator for every combination of numeric types, plus string operators, set operators, junction auto-

threading and `Whatever` currying. It is shared by the interpreter and the compiled backend, which is why the two agree byte for byte.

Its structure is a dispatch chain, and the order is performance-critical. Integer-integer and number-number cases are tested first, with a character switch on the operator rather than a chain of string comparisons. String comparisons were originally *late* in that chain, and paid about 110 nanoseconds walking past mixin delegation, negated operators, set operations, junction autothreading, `Whatever` currying and the `Version` branch before reaching the actual work — which is why `~`, `eq` and friends got their own fast path at the top too (Chapter 27).

10.7 A bug worth remembering: numifying by throwing

`Value::toNum()` numified a `Str` with `std::stod`, which **throws `std::invalid_argument`** when the string does not begin with a number. The result was caught and turned into `0.0` — the right answer, arrived at by raising, unwinding and catching a C++ exception.

Speculatively numifying a string that turns out not to be one is completely routine in an interpreter; the invocant of every `"ab".method` call reached that code. So the language's slowest control-transfer mechanism sat on its hottest path, at roughly 7 microseconds per method call.

`std::strtod` reports the same failure by leaving its end pointer at the start of the input, and costs nothing:

```
// src/Value.cpp
static double strToNumOr0(const std::string& s) {
    if (s.empty()) return 0.0;
    const char* p = s.c_str(); char* end = nullptr;
    double d = std::strtod(p, &end);
    return end == p ? 0.0 : d;      // stod would have thrown here
}
```

Measured over 300,000 iterations:

	before	after	Rakudo
`'ab'.chars`	2.23 s	**0.73 s**	0.34 s
`'ab'.uniname`	2.51 s	**0.47 s**	0.36 s

Roast was unchanged across the switch, which is the point: `strtod` and a caught `stod` agree on every input that reaches this path.

The general lesson is not about `stod`. It is that a C++ exception used as a *value-returning* mechanism — rather than for an error that genuinely aborts something — is a performance bug waiting for a profiler. The same realisation drives the cooperative control flow in Chapter 14.

Chapter 11

Containers, Binding, and Copy Semantics

A `Value` copy shares its payload. Raku sometimes wants that and sometimes wants the opposite, and the rules for which are the subtlest part of the value model. This chapter is about where the sharing is broken on purpose, and how the things Raku calls *containers* are modelled without a container object.

11.1 Scopes are hash maps with a parent pointer

```
// src/Interpreter.h
struct Env {
    std::unordered_map<std::string, Value> vars;   // "$x", "@a", "%h", "&sub"
    std::shared_ptr<Env> parent;
    bool routineFrame = false;   // $/ scopes here
    bool loopFrame = false;      // a loop's `state` frame
    std::unique_ptr<EnvExtras> ex;

    Value* find(const std::string& name) {
        auto it = vars.find(name);
        if (it != vars.end()) return &it->second;
        return parent ? parent->find(name) : nullptr;
    }
    Value& define(const std::string& name, Value v) {
        return vars[name] = std::move(v);
```

```
    }
};
```

The **full sigilled name is the key**. Subs live in the same map under a `&`-prefixed key, so `&foo` and `$foo` occupy different namespaces without a second table. Lexical scoping *is* the parent walk in `find`; there is no separate symbol table anywhere in the interpreter.

Reading a variable returns a **copy** of the slot. Writing goes through `lvalue(Expr*)`, which hands back a `Value*` pointing straight into the owning `Env`'s map:

```
// read  $x
if (Value* p = tctx_.cur->find(ve->name)) return *p;      // a COPY

// my $x — make the slot here, hand back its address
tctx_.cur->define(ve->name, std::move(init));
return &tctx_.cur->vars[ve->name];

// $x = 5
*lv = rhs;
```

The three declarators differ only in *which* `Env` holds the slot:

Declarator	Storage
`my`	the current lexical `Env`
`our`	the package env, ultimately the global one; also republished under a qualified name when the package block closes
`state`	a per-`Callable` `stateEnv`, created exactly once and shared across calls

`state` is the only one that needs care under concurrency, and it gets a `std::once_flag` on the `Callable` so the persistent environment is created exactly once even if two threads call the routine simultaneously.

11.1.1 `EnvExtras`: the eight rarely-used maps

An `Env` is constructed for every routine call **and every block**. It also needs somewhere to keep `is rw` write-through links, `temp`/`let` restorations, `is default` values, and the set of names declared `is dynamic` — eight containers which are empty in the overwhelming majority of scopes.

Constructing and destroying eight empty maps per block is pure overhead for the ordinary case, so they live behind one lazily-allocated pointer:

```
// src/Interpreter.h
EnvExtras& x() { if (!ex) ex = std::make_unique<EnvExtras>(); return *ex; }
const EnvExtras& xr() const {
    static const EnvExtras kEmpty;
    return ex ? *ex : kEmpty;
}
```

Writers call `x()` and materialise it; readers call `xr()` and get a shared empty instance, so a lookup never allocates. This is a small pattern but a recurring one: **make the common case cost nothing, and pay only where the feature is actually used.**

11.2 Where the sharing is broken

Copying a `Value` bumps refcounts on the `shared_ptr` members, so a raw copy of an `Array` shares its backing `ValueList`. Raku wants:

```
my @a = 1, 2, 3;
my @b = @a;      # a COPY
@b[0] = 99;
say @a;          # (1 2 3) — unchanged
```

So the interpreter deliberately breaks the sharing at `=` assignment, and **only for the `@` and `%` sigils**:

```
// src/Interpreter.cpp — coerceArray
if (v.t == VT::Array) {
    if (v.itemized) { … }            // an itemized array is ONE element
    if (v.ext) return v;             // a lazy seq stays lazy
    Value r = Value::array(*v.arr);  // *v.arr copies the vector
    r.isList = false; return r;
}
```

`*v.arr` dereferences the pointer and copies the underlying vector, so `@b` gets its own storage. Hashes copy the same way. Scalar assignment is a plain struct overwrite,

so `my $x = @a` stores an `Array` value that *does* still share `@a`'s buffer — because an item container is a reference to one thing.

Two consequences to keep straight:

- **The copy is one level deep**, matching Rakudo. `my @b = @a` copies the top-level buffer, but a nested itemized array inside is copied as a `Value` struct, so its `arr` pointer is still shared.
- **A lazy sequence is not copied.** The `if (v.ext) return v` line is what keeps `my @a = 1, 2, 4 ... *` from trying to drain an infinite list.

The compiled backend has its own mirror of this rule, `rtArrayVal`, with the same fresh-buffer semantics, so interpreted and compiled code agree.

11.3 Binding: := and the Proxy

= copies a value into an existing container. := rebinds the container itself: after `$y := $x`, the two names *are* the same container and a write to either is seen by both.

But `Env` stores `Values` by value in a map, so two map slots cannot literally be the same storage. The alias is faked with a `Proxy`:

```
// src/Interpreter.cpp — $y := $x, scalar case
Value proxy = Value::makeHash(); proxy.hashKind = "Proxy";
(*proxy.hash)["FETCH"] = fetch;    // a builtin Code closing over the owning Env
(*proxy.hash)["STORE"] = store;
*lvalue(target) = proxy;
```

`$y`'s slot holds a `Hash` tagged `Proxy` whose two closures read and write `$x`'s slot in the environment that owns it. Reading a variable notices the tag and calls `FETCH` instead of returning the hash:

```
// src/Interpreter.cpp — reading a variable
Value* p = tctx_.cur->find(ve->name);
if (p) {
    if (p->t == VT::Hash && p->hashKind == "Proxy" && p->hash) {
        auto it = p->hash->find("FETCH");
        if (it != p->hash->end())
            return callCallable(it->second, { *p });
    }
    return *p;
}
```

Binding chains dereference one extra level so `$z := $y := $x` works.

This costs a call on every read of a bound variable, which is why the check is placed inside the *slow* half of the variable-read path and why the fast paths in Chapter 18 decline any value with a non-empty `hashKind`.

Array binding is much cheaper, because sharing a buffer is exactly what a `shared_ptr` copy already does:

```
// src/Interpreter.cpp — @a := @b
if (a->op == ":=" && rhs.t == VT::Array) {
    Value b = rhs; b.isList = false; *lv = b;
}
```

`Value b = rhs` copies the struct and shares `rhs.arr`. This is the deliberate opposite of `@a = @b` above. `constant` reuses the binding path — a constant is `:=` in disguise.

The user-visible `Proxy` type — `Proxy.new(:FETCH{...}, :STORE{...})` — is the same mechanism exposed, so a program can build one explicitly.

11.4 The scalar-container metaphor

Raku's `$` variable is really a *Scalar container* holding a value, with introspectable machinery: `.VAR`, `is default`, type constraints. Raku++ models that without a separate container object, using flags on the `Value` plus per-`Env` side tables.

`.VAR` builds a `Hash` tagged `"Scalar"` reporting the variable's name, value and default.

`is default(v)` and typed defaults live in `EnvExtras::varDefault`, a per-scope map. `my Int $x` stores `(Int)` as both the initial value and the reset default. Assigning `Nil` walks the chain to find it:

```
// src/Interpreter.cpp — $x = Nil restores the container's default
for (Env* en = tctx_.cur.get(); en; en = en->parent.get()) {
    auto di = en->varDefault.find(nm);
    if (di != en->varDefault.end()) { dv = di->second; break; }
    if (en->vars.count(nm)) break;   // owner scope, no declared default
}
*lv = dv;                            // else Any
```

Type constraints on a scalar are enforced at assignment for the core nominal types, throwing `X::TypeCheck::Assignment` on a mismatch.

`readonly` is a flag on the `Value` itself, set when binding a plain `$` parameter. Mutating operations such as `s///` check it and die.

`temp` and `let` push restoration closures onto the scope's extras. `tempRestores` run whenever the scope leaves; `letRestores` run only on an *unsuccessful* exit, which is the distinction the language draws.

11.5 Sigils are mostly a parse-time fact

The runtime dispatches on the `VT` tag, not the sigil. The sigil — the first character of the name — is consulted exactly twice: to pick the empty container shape at declaration (`@` gives an empty `Array`, `%` an empty `Hash`, `$` an `Any`), and to choose the assignment coercion (`coerceArray`, `coerceHash`, or a scalar overwrite).

After that, behaviour follows the tag and two flags:

- **`isList`** marks a `VT::Array` that is a `List` or `Seq` rather than an `Array`. It renders with parentheses instead of brackets and flattens in list context. Same storage, different behaviour.
- **`itemized`**, set by `$(...)` or `$[...]`, marks an array that counts as *one* element in list context rather than flattening.

That pair carries a surprising amount of Raku's list semantics, and most of the one-argument-rule subtleties in the built-ins reduce to testing them.

11.6 Shaped arrays and typed containers

```
// src/Value.h
std::shared_ptr<std::vector<long long>> shape;  // my @a[2;3]
std::string ofType;                              // Array[Int], my Int @a
```

A shaped array is a fixed row-major structure with its dimensions recorded; an unshaped one leaves the pointer null. `makeShapedContainer` builds one, optionally filling it from a flat list.

`ofType` carries the element type for a typed container, comma-joined when there is more than one parameter (`Hash[Int,Str]` becomes `"Int,Str"`). It also drives native-width masking for `my uint8 @a`, through the same `natWidthOfType` table as scalars.

11.7 `is rw`, and writing back to the caller

Because arguments are passed as `Value` copies, a mutated `is rw` parameter has to be written back into the caller's variable. The call site passes the argument *expressions* alongside the values, and the binder records the link:

```
// src/Interpreter.h — EnvExtras
std::map<std::string, std::pair<Expr*, std::shared_ptr<Env>>> rwLinks;
std::map<std::string, Value> rwSynced;
std::map<std::string, Value*> rwDirect;
std::set<std::string> rwDead;
```

There are two mechanisms because there are two situations. `rwLinks` holds the caller's argument expression and environment, so an assignment to the parameter can re-resolve the lvalue and push the value through *immediately* — the caller sees the change mid-call, which is what Raku specifies. `rwSynced` records what was last pushed, so the copy-out backstop at return can skip parameters that are already up to date; without it a late copy-out would re-apply stale values over the callee's own later edits.

`rwDirect` is the simpler case: a hyper-operator element call already has the caller's slot as a pointer, with no expression to re-evaluate. `rwDead` marks a raw or `rw` parameter bound to a literal, so assigning it dies with `X::Assignment::RO` rather than silently writing into a temporary.

This works for **direct** calls, where the caller supplied the argument expressions. Multi-dispatched and indirect calls can lose that fidelity — a known limitation, documented rather than hidden.

11.8 Honest limitations

- **A reference cycle leaks.** Lifetime is refcounting. The interpreter breaks the one cycle it creates systematically — a nested sub whose closure points back at the frame that holds it — and otherwise relies on processes being short-lived.
- **`:=` on a scalar costs a call per read.** The `Proxy` is correct and general, and it is not free.
- **`is rw` write-back is expression-based.** Where the argument expression is not available or not re-resolvable, the write-back does not happen.
- **The one-level copy rule** matches Rakudo, but it surprises people regularly, and no amount of documentation seems to fix that.

Part IV

The Interpreter

Chapter 12

The Tree Walk

There is no bytecode. `Interpreter::eval(Expr*)` returns a `Value`; `Interpreter::exec(Stmt*)` runs a statement and returns its value. Both are recursive, both switch on the node's `NK` tag, and the C++ call stack *is* the Raku call stack.

That decision buys a small implementation and costs throughput, and every optimisation in Chapters 18, 26 and 27 exists because of it.

12.1 The two entry points

```
// src/Interpreter.h
Value eval(Expr* e);
Value exec(Stmt* s, bool sink = false);
Value execBlock(Block* b, std::shared_ptr<Env> scope, bool sink = false);
```

`exec` takes a **sink** flag. When a statement's value is discarded — a loop body, a statement in the middle of a block — the flag says so, and an assignment can skip materialising its (possibly very large) result. It is a small thing that turns the naive `$s ~= ...` in a loop from quadratic into linear, in combination with the in-place append described in Chapter 26.

`execBlock` runs a statement list in a given scope, handling the phaser schedule around it.

12.2 Execution registers

The state that belongs to one thread of Raku execution is gathered into a struct rather than scattered across interpreter members:

```
// src/Interpreter.h — ExecContext, abridged
struct ExecContext {
    std::shared_ptr<Env> cur;           // current lexical scope
    std::vector<Env*> dynStack;         // the dynamic ($*foo) caller chain
    int callDepth = 0;
    Env* curStateEnv = nullptr;
    std::vector<std::shared_ptr<ValueList>> gatherStack;
    std::vector<ValueList*> supplyStack;
    std::vector<Value*> makeTargets;
    std::string pkgPrefix;

    bool returning = false;  Value returnV;      // cooperative return
    uint64_t frameTop = 0, curRoutineFrame = 0;
    int loopCtl = 0;         uint64_t curLoopFrame = 0;
    int givenCtl = 0;        Value givenV;  uint64_t curGivenFrame = 0;

    struct CallSite { int line; const Value* code; };
    std::vector<CallSite> callFrames;           // for callframe(N), backtraces
    int wantLvalue = 0;  Value* lvalueOut = nullptr;
};

static thread_local ExecContext tctx_;
```

It is `static thread_local`, so each real worker thread owns its own set. That is the foundation of the concurrency model in Chapter 33 — with per-thread registers, running Raku on a second thread does not need a register swap at every handover.

Two chains, not one. **`cur`** is the lexical chain: a callee's parent is the scope it was *written* in, reached through its `Callable::closure`. **`dynStack`** is the dynamic chain: the caller's scope, pushed on entry, walked only when resolving a `$*foo`. Keeping them separate is what makes lexical and dynamic scoping both correct and both cheap.

12.3 Evaluating an expression

`eval` is a switch. The interesting arms are the ones that are *not* a straightforward recursion.

VarExpr is the hottest node in most programs. Its fast path is a `find` and a copy:

```
// src/Interpreter.cpp — eval(VarExpr), the plain-lexical fast path
if (Value* p = tctx_.cur->find(ve->name))
    if (!(p->t == VT::Hash && p->hashKind == "Proxy"))
        return *p;
```

with the `Proxy` check placed so that the common case is one comparison. Names with a twigil — `$*dyn`, `$?FILE`, `$^a`, `$/` — never reach this path; each has its own lookup rule, and the fast paths in Chapter 18 exclude them by testing the second character of the name.

Binary dispatches through `applyArith`, but first consults two decided-once fields on the node: `simpleOp`, which records whether this operator needs special handling at all, and `fastShape`, which records the syntactic shape of the operands. Chapter 18 is entirely about those.

Index reads a container element, and has a matching `fastShape`.

Call and **MethodCall** are Chapters 13 and 15.

InterpStr evaluates each part and concatenates. The parts were already parsed into sub-expressions by the front end, so there is no string scanning here at all.

NqpOp exists only under `use nqp` (Chapter 30).

12.4 Evaluating a statement

`exec` is the same shape, with the control-flow statements doing the real work. Two mechanisms recur through all of them.

12.4.1 Hoisting

Before a statement list runs, two things are pre-registered.

hoistSubs walks the list and defines every named `SubDecl` in the scope, so a sub is callable across its whole enclosing scope regardless of textual order. This is why subs need no forward declaration while *operators* do (Chapter 5): sub visibility is a runtime fact, operator visibility a parse-time one.

hoistExprDecls pre-declares `my` variables that are buried inside expressions — in a ternary branch, in an `nqp::` argument — because Raku scopes them to the block, not to the expression.

Both are guarded by a decided-once flag so the walk happens once per block rather than once per entry:

```
// src/Interpreter.h
void hoistExprDecls(const std::vector<StmtPtr>& stmts, Env* env,
                    DecidedOnce<signed char>* cache = nullptr);
```

```
// src/Ast.h — Block
DecidedOnce<signed char> hoistNeed{-1};  // -1 undecided, 0 no, 1 yes
```

For a block with nothing to hoist — the overwhelming majority — the cost after the first entry is reading one byte.

12.4.2 Phasers

`execBlock` runs the phaser schedule around the statement list:

Phaser	When
`ENTER`, `FIRST`	at block entry, in source order
`LEAVE`	at block exit, reverse order, always
`KEEP`	at block exit, only on a successful exit
`UNDO`	at block exit, only on an unsuccessful one
`NEXT`	at the end of each loop iteration
`LAST`	once after a loop's final iteration
`CATCH`, `CONTROL`	as handlers around the block

```
// src/Interpreter.h
void runEnterPhasers(const std::vector<StmtPtr>& stmts);
void runLeavePhasers(const std::vector<StmtPtr>& stmts, bool ok = true);
void runNextPhasers(const std::vector<StmtPtr>& stmts,
                    std::shared_ptr<Env>& scope);
```

The program-level phasers — `BEGIN`, `CHECK`, `INIT`, `END` — are scheduled by `run()` rather than by a block: `BEGIN` in source order after the parse, `CHECK` reversed, `INIT` immediately before the mainline, `END` at process exit. This is where the "the parser runs nothing" decision from Chapter 4 becomes visible: a `BEGIN` block runs *after* the whole file has been parsed, so it cannot influence how later source is read.

12.5 Loop bodies

Every loop form funnels through one function:

```
// src/Interpreter.h
bool runLoopBody(Block* b, std::shared_ptr<Env> scope,
                 const std::string& label = "",
                 bool isFirst = true, bool isLast = true,
                 ValueList* collect = nullptr,
                 const std::function<void()>& rebind = nullptr);
```

It runs one iteration and handles `redo`, `next` and `last` — returning `false` when the loop should stop — plus the `FIRST`/`NEXT`/`LAST` phasers. The `collect` parameter is how a loop used in **value context** works: `my @x = do for @y { ... }` appends each iteration's value rather than discarding it. The `rebind` callback re-binds the loop variable for a `redo`.

Concentrating this in one function is what makes `for`, `while`, `until`, `loop` and `repeat` agree about control flow, which they historically did not.

12.6 Aliasing loops

`for @a { $_ = ... }` must write **into** `@a`, so the topic has to alias the element rather than copy it. That requires knowing what the loop source *expression* is, not just its value, so the interpreter has a small family of functions that peer at the source expression:

```
// src/Interpreter.h
std::shared_ptr<std::map<std::string, Value>> valuesAliasSource(Expr*);
std::shared_ptr<ValueList> derefArrayAlias(Expr*);
bool scalarListAlias(Expr*, std::vector<Value*>& slots);
Value* topicAliasSlot(Expr* topic, bool skip);
Expr* peelGrepFilter(Expr* listExpr, Expr*& pred);
```

The last one is the sort of detail that only shows up against a real test suite: `for %h.values.grep(...) { $_ = ... }` must *still* alias, because Rakudo's `grep` is `is raw`. So the loop peels a trailing `.grep(PRED)` off the source expression, aliases the underlying container, and applies the predicate as a filter during iteration.

12.7 Sink context and `it does not matter what this returns`

Statement values matter in Raku — the last statement of a block is its value — so `exec` returns one. But most statements' values are discarded, and building them can be expensive.

The `sink` flag threads that knowledge down. Its most valuable use is assignment: in sink context, `evalAssign` does not need to produce the assigned value as a result, so `@big = @other` does not build a second copy to throw away. Combined with `rtCatAssign`'s in-place append (Chapter 26), this is what makes string building in a loop linear.

12.8 Recursion, and the stack it needs

Because the C++ stack is the Raku stack, recursion depth is bounded by the thread's stack size. The default for a non-main thread on macOS is 512 KB, which a recursive Raku sub overflows within about a hundred frames.

So every entry point runs the program on a thread with a very large stack:

```
// src/Runtime.h
int rakuppRunBigStack(const std::string& src, std::vector<std::string> args,
                      const std::string& fileName, const std::string& exePath,
                      const std::vector<std::string>& libPaths = {});
int rakuppMainOnBigStack(int (*body)(void*), void* ctx);
```

and worker threads get the same treatment through `BigStackThread`, which reserves 256 MiB of *virtual* address space — committed only as used (Chapter 33). A compiled `--exe` binary calls `rakuppMainOnBigStack` for its own main body, so the recursion budget is the same in every mode.

The interpreter also keeps a `callDepth` register and raises a clean Raku error when it gets too deep, so a runaway recursion produces a diagnostic rather than a segmentation fault.

12.9 What this costs, honestly

Re-dispatching every AST node on every execution is inherently slower than bytecode or a JIT. The profile of a method-heavy loop, after the fixes in Chapters 9 and 10, looks like this:

	share
heap allocate and free	31%
`Value` copy and destroy	11%
method-name comparison	8.5%
the dispatch function's own body	6.1%

Note what is *not* there: no interpreter loop overhead line, no instruction decode. The tree walk itself is not the problem; **allocation and value churn are**, and they would still be there in a bytecode VM built on the same value model.

That is why the optimisation work in this book goes after allocation — the per-call `ValueList`, the per-operation `Value` box, the per-copy string — and not after the dispatch mechanism. The one place the dispatch mechanism itself was worth attacking is Chapter 18, and even there the win came from removing copies rather than from removing branches.

Chapter 13

Calls and Parameter Binding

A call happens in three phases: build the argument list, activate the callee, bind the parameters. Each has a fast path, and each fast path exists because something measurable was in the way.

13.1 Building the argument list

`evalArgs` evaluates the argument expressions into one flat `ValueList`:

```
// src/Interpreter.cpp — evalArgs, abridged
} else if (a->kind == NK::Unary && ((Unary*)a.get())->op == "|") {
    Value v = eval(...);                                  // a Slip: |@a / |%h
    if (v.t == VT::Array || v.t == VT::Range)
        for (auto& x : v.flatten()) args.push_back(x);
    else if (v.t == VT::Hash && v.hash)
        for (auto& kv : *v.hash) {                        // |%h → named args
            Value p = Value::pair(kv.first, kv.second);
            p.namedArg = true; args.push_back(p);
        }
} else {
    Value v = eval(a.get());
    if (v.t == VT::Pair && a->kind == NK::Pair &&
        !((PairExpr*)a.get())->quotedKey && ident)
        v.namedArg = true;
    args.push_back(std::move(v));
}
```

Two things to notice.

An argument list is just a `ValueList`. Named arguments are ordinary `Pair` values carrying a `namedArg` flag; the positional/named split happens later, at bind time. That is why spreading (`|@a`), slurping (`*@rest`) and flattening are all vector operations rather than a separate protocol.

Only a *syntactic* pair is a named argument. `f(k => 1)` and `f(:k(1))` pass a named argument; `f($pair)` and `f(3 => 4)` pass a positional one, even though all four produce a `Pair`. The test is on the *expression* kind, and it also excludes a quoted key — `f('a' => 1)` is positional, per Raku's rule. Getting this wrong makes every module that passes pairs around behave subtly differently, so the check is deliberately narrow.

13.2 Activating the callee

```
// src/Interpreter.h
Value callCallable(const Value& codeVal, ValueList args,
                   const std::vector<ExprPtr>* rwArgs = nullptr,
                   bool ownFrame = false, bool arityCheck = false);
Value callCallableRaw(const Value& codeVal, ValueList args,
                      const std::vector<ExprPtr>* rwArgs,
                      bool ownFrame = false, bool arityCheck = false);
```

`callCallable` is a thin **wrapper layer**: if the routine has been `&r.wrap({...})`'d, it runs the wrapper stack, each level able to `callsame` into the next; otherwise it passes straight through:

```
// src/Interpreter.cpp — callCallable
if (codeVal.code && !codeVal.code->wrappers.empty()) { /* run the stack */ }
return callCallableRaw(codeVal, std::move(args), rwArgs);
```

`callCallableRaw` handles the special callables first — a native FFI sub, a `Format`, junction autothreading, a multi-dispatcher, a builtin — and then activates a user routine:

```
auto env = std::make_shared<Env>();                    // fresh per-call frame
c.stateEnv->parent = c.closure ? c.closure : global_;  // once
env->parent = c.stateEnv;          // frame → stateEnv → closure → … → global
tctx_.dynStack.push_back(caller_scope);                // the OTHER chain
```

Two facts are established here and everything else depends on them.

The callee's parent is its lexical closure, not its caller. Free variables resolve through the scope the routine was *written* in. The caller's scope goes on `dynStack`, used only for `$*foo`.

Each activation bumps `frameTop`, and a routine — as opposed to a bare block — records its own frame number as `curRoutineFrame`. Those two counters drive the cooperative control flow in the next chapter.

13.2.1 The call registers

A handful of one-shot parameters are passed to the next activation without widening every signature:

```
// src/Interpreter.h
static thread_local Value* topicWriteback_;
static thread_local Value* builtinTopicWB_;
static thread_local bool noAutothread_;
static thread_local int loopPhaserCtl_;
static thread_local const std::vector<Value*>* pendingRwSlots_;
```

Each is set immediately before a call and consumed at the top of `callCallableRaw`. `topicWriteback_` is how `@a.grep({ $_++; True })` writes into `@a`'s element: the driver points it at the element, and the mutated implicit `$_` is copied back after the call.

They are `static thread_local` and the comment in the header explains why in one sentence: as plain members they were written by *every* call on *every* thread, and they were ThreadSanitizer's top report — 2,761 lines on a program with no sharing in it at all. The set-then-consume window is contiguous within one thread, so thread-local is not a workaround, it is exactly their semantics.

13.3 Binding parameters

`bindParams` maps the argument list onto the signature. There is a fast path and a general path.

```
// src/Interpreter.cpp — bindParams
if (simple) {                       // all plain positional $ params, no nameds
    for (size_t i = 0; i < params.size(); i++) {
        Value v = i < args.size() ? args[i]
                                  : typedDefault(params[i].type, '$');
        v.readonly = true;
```

```
        env->define(params[i].name, std::move(v));
    }
    return;
}
for (auto& a : args)                // general: split named from positional
    if (isNamedArg(a)) named[a.s] = a.pairVal ? *a.pairVal : Value::any();
    else positional.push_back(a);
```

Whether a signature qualifies for the fast path is a static property of the signature, so it is decided once and stored on the *first* parameter:

```
// src/Ast.h — Param
mutable DecidedOnce<signed char> sigSimple{-1};   // only element [0] is read
```

The general path covers everything else:

- **positional parameters with defaults**, evaluated in the parameter scope so `sub f($g, $a = $g/2)` can see earlier parameters;
- **readonly, `is rw`, `is copy`, `is raw`** — a plain `$` parameter is marked readonly unless it is one of those or the invocant:

  ```
  if (p.sigil == '$' && !p.isRw && !p.isCopy && !p.invocant)
      v.readonly = true;
  ```

- **slurpies**: `*@rest` flattens, `**@rest` does not, `+@rest` applies the single-argument rule, `*%named` collects the rest;
- **named parameters**, including the alias forms `:a(:$b)` and nested `:x(:y(:z($a)))`, where every layer's key answers;
- **sub-signature destructuring**, `sub f([$a, $b])`, recursing through `Param::subSig`;
- the implicit **`$_`** topic, and placeholder parameters `$^a`, `$^b`, bound in sorted order.

13.3.1 What is *not* checked here

Type constraints, `where` clauses and `:D`/`:U` smileys are not enforced for an ordinary, non-multi call. Single dispatch is largely duck-typed at the bind boundary. Those checks live in `scoreCandidate`, which is multi dispatch's business (Chapter 15).

There is one exception, `typeCheckBind`, used when a lone candidate is being bound and a mismatch should raise `X::TypeCheck::Binding`. It caches its verdict on the parameter — but **only once true**:

```
// src/Ast.h — Param
mutable DecidedOnce<signed char> typeKnown{0};
```

The asymmetry is important and the header says why: a type name that is unresolvable now can become resolvable later, when a class further down the file is declared or a module is loaded. Caching the *positive* answer is safe; caching the negative one is a bug that would only show up in a program whose declarations are ordered a particular way.

13.4 The `Callable`, and what it caches

```
// src/Value.h — Callable, abridged
std::string pkg, name;
const std::vector<Param>* params;      // borrowed from the AST
const std::vector<StmtPtr>* body;      // borrowed from the AST
std::shared_ptr<Env> closure;
std::shared_ptr<Env> stateEnv;         // persistent `state` storage
std::once_flag stateInit;
BuiltinFn builtin;                     // set ⇒ this is a builtin
std::vector<Value> candidates;         // multi-dispatch candidates
std::vector<Value> wrappers;           // .wrap stack, outermost last
DecidedOnce<signed char> hoistNeed{-1};
DecidedOnce<signed char> arityShape{-1};
int arityMaxPos = 0, arityReqPos = 0; bool arityUnbounded = false;
DecidedOnce<signed char> catchScan{-1};
Stmt* catchBlkCache = nullptr;
```

The four decided-once fields are all static properties of the routine's AST that `callCallableRaw` used to recompute **on every call**: whether anything needs hoisting, whether an arity pre-check applies and what its bounds are, and whether the body contains an inline `CATCH` block. Each is cheap once and worthless repeated — a parse that calls a routine 73,603 times paid for them 73,603 times.

`stateEnv` is created exactly once, under a `std::once_flag`, and chained between the per-call frame and the closure. That is the whole implementation of `state`: the variable lives in a scope that is created once per routine rather than once per call.

13.5 `is rw` write-back

Because arguments are `Value` copies, a mutated `is rw` parameter must be written back. The call site passes the argument *expressions* alongside the values, and on a normal return each such parameter's final value is written back by re-resolving its expression:

```
// src/Interpreter.cpp — copyOutRw
if ((p.isRw || p.sigil == '\\') && pi < rwArgs->size())
    if (Value* lv = lvalue((*rwArgs)[pi].get()))
        *lv = env->vars[p.name];
```

`setupRwLinks` additionally arranges for an assignment *inside* the callee to push through immediately, so the caller sees the change mid-call. The bookkeeping that keeps those two mechanisms from fighting is described in Chapter 11.

The whole thing is guarded by a sticky flag, `anyRwLinks_`, so a program that never uses `is rw` never runs the per-assignment hook at all.

13.6 Lvalue-mode method calls

`$obj[$i] = $v` on a class whose `AT-POS` is `return-rw @!arr[$i]` must write the *real* element, not a returned copy. That needs a channel from the subscript site into the routine's `return-rw`:

```
// src/Interpreter.h — ExecContext
int wantLvalue = 0;      // 0 off, else the callFrames depth being served
Value* lvalueOut = nullptr;
```

The subscript-lvalue path sets `wantLvalue` to the current frame depth plus one before invoking `AT-POS`; a `return-rw` executing at exactly that depth fills `lvalueOut` with the address of its operand. The pointer survives the frame because its target lives in the object's shared containers.

Matching on depth rather than on a plain flag is what keeps an inner call from accidentally answering an outer call's request.

13.7 What a call costs

Measured against the runtime library, 2 million iterations, minimum of six:

Shape	ns/call
direct C++ call, user-sub shape	46.3
cached `BuiltinFn*` call	47.4
by-name lookup: hash, `unordered_map::find`, `std::function`	55.0

The lookup tax is real but modest at 8 to 9 nanoseconds. The important number is the **floor**: about 46 nanoseconds for a *trivial* call, nearly all of it the `ValueList` — a heap-allocating `std::vector` built per call.

Dispatch was a quarter of the overhead. The argument vector was the rest. That finding is what shaped the optimiser in Chapter 26: its first pass gives fixed-arity subs direct `Value` parameters and removes the vector entirely, which buys more than any lookup cache can.

Chapter 14

Cooperative Control Flow

The natural way to implement `return` in a tree-walker is to throw a C++ exception and catch it at the routine boundary. It is correct, it is short, and it is what Raku++ did first.

It is also slow. On macOS a C++ throw walks the dynamic loader's unwind information under a lock, costing tens of microseconds. In the WebAssembly build, where C++ exceptions run through JavaScript trampolines, it is very much worse. And `return` is not rare — it is on the exit path of a large fraction of all routines.

14.1 The insight

A `return` only needs to unwind C++ frames if there is a **callable boundary** between it and its routine. If the `return` executes directly inside its routine's own statements, or inside a native `for` loop in the interpreter, then no C++ frame between them belongs to another Raku routine — and "unwinding" is just a matter of *stopping the statement loop*.

Distinguishing those two cases needs a way to ask "has a callable been entered since my routine started?", and the answer is two counters.

```
// src/Interpreter.h — ExecContext
bool returning = false;  Value returnV;
uint64_t frameTop = 0;         // incremented per callCallableRaw activation
uint64_t curRoutineFrame = 0;  // frameTop at the enclosing ROUTINE entry
```

Every activation bumps `frameTop`. A routine records the value at its own entry. So the test is an equality:

```
// src/Interpreter.cpp — return
if (tctx_.curRoutineFrame != 0 && tctx_.frameTop == tctx_.curRoutineFrame) {
    tctx_.returning = true; tctx_.returnV = std::move(v);
    return Value::any();                        // set a flag, do not throw
}
throw ReturnEx{v};                              // a boundary was crossed
```

The statement loops check the flag after each statement and bail out, and `callCallableRaw` consumes it at the routine boundary:

```
// src/Interpreter.cpp — after each statement in the activation loop
if (tctx_.returning) {
    if (isRoutine) { tctx_.returning = false; last = std::move(tctx_.returnV); }
    break;                // a bare block just propagates it to its routine
}
```

A bare block does *not* consume the flag — it breaks out and lets it propagate — because a `return` inside `if { … }` returns from the enclosing routine, not from the block.

14.2 The same trick, three more times

`next`, `last` and `redo` use an identical mechanism with their own register and frame counter:

```
// src/Interpreter.h — ExecContext
int loopCtl = 0;              // 0 none, 1 next, 2 last, 3 redo
uint64_t curLoopFrame = 0;    // frameTop of the innermost native loop
```

```
// src/Interpreter.cpp — LastStmt
if (t.empty() && tctx_.curLoopFrame != 0 &&
    tctx_.frameTop == tctx_.curLoopFrame) {
    tctx_.loopCtl = 2; return Value::any();
}
throw LastEx{t};              // labelled, or across a frame: unwind
```

`when`, `default` and `succeed` got the same treatment, and for a specific measured reason. `given $v { when Int {…} when Str {…} … }` executed per row of a data set means a throw per row:

```
// src/Interpreter.h — ExecContext
int givenCtl = 0;            // 1 = a when matched; break the given
Value givenV;
uint64_t curGivenFrame = 0;
```

In each case the rule is the same: **same frame, set a flag; different frame or a label, throw.** Labelled control always throws, because a labelled `last` targets a specific enclosing loop that may be several frames up, and walking to it is exactly what unwinding is for.

14.3 The rest of the control exceptions

Everything that genuinely has to cross frames is still an exception, and they are all tiny structs:

```
// src/Interpreter.h
struct ReturnEx { Value v; };
struct ExitEx { int code = 0; };
struct LastEx { std::string label; };
struct NextEx { std::string label; };
struct RedoEx { std::string label; };
struct DoneEx {};                    // exits a whenever/supply/react body
struct BreakGivenEx { Value v; bool hasVal = false; };
struct LeaveEx { Value v; bool hasVal = false; };
struct ResumeEx {};                  // .resume inside a CATCH
struct StopGatherEx {};              // a lazy gather has enough
struct ProceedEx {};                 // leave a when, keep matching later ones
struct RakuError { Value payload; std::string message; };
struct WorkerAbortEx {};             // unwind a background worker at shutdown
```

`RakuError` is the user-visible one: every Raku exception is a `RakuError` carrying a payload `Value` — normally an exception object — and a message. `WorkerAbortEx` is deliberately *not* a `RakuError`, so a user's `CATCH` cannot accidentally swallow a shutdown (Chapter 33).

14.4 `CATCH`, and where it lives

A `CATCH` block is a `Block` statement with `isCatch` set, sitting inside the block it guards. Finding it means scanning the body — so the answer is cached on the `Callable`:

```
// src/Value.h — Callable
DecidedOnce<signed char> catchScan{-1};   // 1 = the body holds an inline CATCH
Stmt* catchBlkCache = nullptr;            // …and which one
```

When a `RakuError` reaches a block that has one, the handler runs with `$_` and `$!` bound to an exception instance. `exceptionFor` guarantees that instance is always *defined*, whatever was actually thrown, because a `when X::Foo` inside the handler needs something to smartmatch against.

`.resume` throws `ResumeEx`, caught at the throw point. Because a bare `ResumeEx` with nothing to absorb it would reach `std::terminate`, the interpreter tracks how many `CATCH` handlers are live and turns a `.resume` outside any of them into a catchable Raku error:

```
// src/Interpreter.h
int catchDepth_ = 0;
```

Typed exceptions are built rather than hard-coded:

```
// src/Interpreter.h
[[noreturn]] void throwTyped(const std::string& type,
        std::vector<std::pair<std::string, std::string>> attrs,
        const std::string& message);
Value makeTypedEx(const std::string& type,
        std::vector<std::pair<std::string, Value>> attrs,
        const std::string& message);
```

so `X::TypeCheck::Binding` arrives with its `got` and `expected` attributes populated and can be matched on class rather than on message text. That distinction matters: the project's rule is that error *message* wording is only copied from Rakudo where Roast or the documentation actually asserts it — the behaviour is what must match, not the prose.

14.5 Backtraces

```
// src/Interpreter.h — ExecContext
struct CallSite { int line; const Value* code; };
std::vector<CallSite> callFrames;
```

One entry per live routine activation: the line the *call* was written on, and the routine itself. It is pushed next to `dynStack`, which every call already pays for, so it costs a vector append.

That is what `callframe(N)` walks — `Log::Async` stamps every message with `callframe(1)` — and what `Exception.throw` turns into a `BacktraceFrame` list. The routine's declaring file comes from `Callable::declFile`, recorded when the routine was declared, because by the time a backtrace is taken the module that declared it is long gone from the current scope.

14.6 What was gained, and one bug it cost

The cooperative path removes a C++ throw from the exit of most routines and the body of most loops and `given` blocks. On macOS, where a throw is expensive, and in WebAssembly, where it is much worse, that is the difference between a tolerable and an intolerable tree-walker.

The mechanism has one hazard, and it bit: **the counters must be maintained consistently across every activation kind.** Methods go through `invokeMethod` rather than `callCallableRaw`, and for a period that path did not establish the routine frame boundary the same way. The symptom was specific and baffling — a method lost an early `return` when the `return` was inside a loop, but only in some call contexts. The cooperative flag was set, and the frame that should have consumed it did not recognise itself as a routine boundary, so the return value evaporated.

The fix was to establish the boundary in `invokeMethod` exactly as `callCallableRaw` does, and the regression test is a method with a `return` inside a loop, called every way the language allows.

The general lesson is worth stating: **an optimisation that replaces a language-level mechanism with a hand-maintained invariant is only as correct as the least careful place that maintains it.** C++ unwinding could not be got wrong this way, because the compiler maintained it. Frame counters can.

Chapter 15

Dispatch

Raku has more dispatch than most languages: named subs, multiple dispatch on argument types, method dispatch on an invocant, private methods, meta-object calls, and a re-dispatch protocol (`callsame`, `nextsame`, `callwith`, `nextwith`, `samewith`) that lets a routine hand control to the next candidate.

Raku++ implements them as **three distinct mechanisms** that meet at the edges, and knowing which one is in play explains most of the surprises.

15.1 Named calls

`evalCall` resolves every named call the same way:

```
// src/Interpreter.cpp — evalCall, abridged
if (Value* f = tctx_.cur->find("&" + c->name))
    return callCallable(*f, std::move(args), &c->args, false, …);
…
auto it = builtins_.find(c->name);        // built-in fallback
```

The environment first, walking the parent chain; the builtin table second. That order is the whole rule, and it has three consequences worth spelling out.

A lexical `my sub` shadows a builtin. That is Raku's semantics and it comes for free.

A module's exported sub also shadows a builtin, because the loader copies it into the global environment, which is the last link of the chain (Chapter 29).

A compiled program must reproduce this order, which it cannot do by resolving names against the builtin table at compile time. That is a real bug that happened — `--exe` printed the built-in `val`'s answer where the interpreter called a module's exported `val` — and Chapter 27 describes the fix.

The argument *expressions* are passed alongside the values (`&c->args`) so that `is rw` write-back can re-resolve them.

15.2 Multiple dispatch

A `multi sub` or `multi method` is one `Callable` with `isMultiDispatcher` set and a `candidates` vector; each `multi` declaration pushes its `Code` onto it. A `proto` declares the group without being a candidate.

At call time, `scoreCandidate` scores every candidate against the actual arguments, returning `-1` for "does not apply" or a non-negative **specificity**:

```
// src/Interpreter.cpp — scoreCandidate, per positional parameter
if (subsets_.count(p->type)) {
    if (!subsetMatches(p->type, pos[i])) return -1;
    score += 2;
} else if (!typeMatchesArg(pos[i], p->type)) return -1;
if (p->defConstraint == 1 && !isDefined(pos[i])) return -1;   // :D
if (p->defConstraint == 2 &&  isDefined(pos[i])) return -1;   // :U
if (p->defConstraint) score++;
if (!p->type.empty() && p->type != "Any" && p->type != "Mu") {
    score++;                                      // constrained beats not
    if (p->type == pos[i].typeName()) score++;  // exact beats supertype
}
```

Arity gates run first — too few required arguments, or too many for a non-slurpy signature, is an immediate `-1`. Literal parameters (`multi fact(0)`) and sub-signature destructuring are treated as very specific. A required named parameter that was not supplied is `-1`.

The winner is the highest score:

```
const Value* best = nullptr; int bestScore = -1;
for (auto& cand : c.candidates) {
    int s = scoreCandidate(cand, args);
    if (s > bestScore) { bestScore = s; best = &cand; }   // strict >
}
if (!best || bestScore < 0) throw /* X::Multi::NoMatch */;
```

Two honest notes. The comparison is a strict `>`, so on equal scores the *first-declared* candidate wins and there is no `X::Multi::Ambiguous` diagnostic. And a genuine no-match does throw `X::Multi::NoMatch`, with the candidate list in the message.

15.2.1 Subsets

A `subset` is a refinement type that participates in dispatch:

```
// src/Interpreter.h
struct SubsetInfo { std::string base; const Expr* where = nullptr; };
std::unordered_map<std::string, SubsetInfo> subsets_;
bool subsetMatches(const std::string& name, const Value& v, int depth = 0);
```

`subsetMatches` checks the base type and then evaluates the `where` expression with the value as the topic. The `depth` parameter is a recursion guard, because a subset can be defined in terms of another subset and nothing prevents a cycle.

A subset match scores `+2`, above a plain nominal match, which is what makes `multi f(Even $n)` beat `multi f(Int $n)`.

15.3 Method dispatch: two worlds

An `Int`, a `Str` or an `Array` is a native `Value` — it has no `ClassInfo` and no method table. A user class instance is a `VT::Object` with both. So there are two dispatch worlds.

User objects go through `ClassInfo::findMethod`, which *is* the method resolution order:

```
// src/Value.h
Value* findMethod(const std::string& m, ClassInfo** owner) {
    auto it = methods.find(m);
    if (it != methods.end()) { if (owner) *owner = this; return &it->second; }
    if (parent) { if (Value* r = parent->findMethod(m, owner)) return r; }
    for (auto& p : extraParents) if (p)
        { if (Value* r = p->findMethod(m, owner)) return r; }
    if (owner) *owner = nullptr;
    return nullptr;
}
```

Own methods, then the first parent recursively, then each extra parent. It is a depth-first walk, **not C3 linearisation**, and for the diamond hierarchies where the two differ it can pick a different method than Rakudo. The `owner` out-parameter reports

which class the method was found in, so re-dispatch can resume from that class's parent.

Built-in values go through `methodCall`, a very long cascade of branches keyed on the method name and, inside each branch, on the invocant's tag:

```
if (m == "uc")       return Value::str(mapCase(inv.toStr(), true, 0));
if (m == "chars")    return Value::integer(graphemeCount(inv.toStr()));
if (m == "is-prime") { /* Miller–Rabin on inv.toInt() */ }
// … some 1,640 more …
```

The C++ `if`-ladder *is* the built-in method set. That is the pragmatic counterpart to the fat `Value`: since every native value is the same struct, its methods are one dispatch function rather than per-type classes.

The ladder is split across four files as ordered segments (Chapter 2), each returning `std::optional<Value>`. The name is wrapped in an `MName` at the top so the comparisons are integer compares (Chapter 9).

15.3.1 What runs before the ladder

Two things are consulted first, and both are bridges between the two worlds.

Junction autothreading. If the invocant is a junction, a small allow-list of methods (`Bool`, `gist`, `new`, …) act on the whole junction; everything else autothreads, returning a junction of the per-eigenstate results.

`augment` on a built-in type. Methods a program adds to a built-in with `augment class Int { … }` are parked in a map of maps and consulted before the native branches, so they can override built-ins:

```
// src/Builtins.cpp — before the native branches
if (!builtinExt_.empty() && inv.t != VT::Object) {
    std::string tn = inv.t == VT::Type ? inv.s : inv.typeName();
    if (Value* f = lookup(tn))
        return invokeMethod(*f, inv, std::move(args), rwArgs);
    for (const std::string& anc : typeAncestry(tn))
        if (anc != tn) if (Value* f = lookup(anc))
            return invokeMethod(*f, inv, …);
}
```

The ancestry walk is what makes `augment class Cool { … }` reach an `Int` and a `Str`. The guard on `builtinExt_.empty()` is why a program that never augments anything pays one branch.

This is also why two of the inline built-in fast paths in Chapter 27 are `builtinExt_`-guarded: an inlined `abs` must stop inlining the moment a program augments `Int`.

15.4 Re-dispatch

`callsame`, `nextsame`, `callwith`, `nextwith` and `samewith` need to know what the *next* candidate is, and that differs by context: the next-less-specific multi candidate, the same method on the owning class's parent, or a built-in that a user method shadowed.

```
// src/Interpreter.h
struct RedispatchCtx {
    std::function<Value(ValueList)> next;      // callsame / nextsame / …with
    std::function<Value(ValueList)> restart;   // samewith
    ValueList sameArgs;
    bool lastcall = false; bool fromChain = false;
};
static thread_local std::vector<RedispatchCtx> redispatchStack_;
```

Each dispatch site that can be re-entered pushes a context knowing how to invoke the next thing, and the original arguments for the `*same` variants. `invokeMethodChain` is the method version: it invokes a method found from a given class and pushes a context that resumes from that class's parent, recursively.

One field guards a subtle case:

```
// src/Interpreter.h — ExecContext
size_t redispatchFloor = 0;   // frames below this are another routine's
```

Without it, a `callsame` inside a routine could reach a re-dispatch context belonging to a *caller*, and dispatch into something unrelated.

The stack is `static thread_local` for the reason given in Chapter 13: as a plain member it was written by every dispatching call on every thread.

15.5 Private methods, qualified calls, indirect calls

The `MethodCall` node carries the variants as flags:

Written	Flag	Meaning
`$o.m`	—	ordinary dispatch
`$o.?m`	`maybe`	return `Nil` rather than dying when absent
`$o!m`	`bang`	private method, reachable only through `self!m`
`$o.Class::m`	`methodQual`	dispatch to a specific class's method, past overrides
`$o."$name"()`	`methodExpr`	the name is computed at run time
`$o.=m`	`mutate`	call and assign back to the invocant
`$o>>.m`	`hyper`	apply to each element
`$o.^m`	`meta`	a meta-object call

`.^` calls go to the meta-object. Most are answered directly — `.^name`, `.^methods`, `.^attributes`, `.^parents`, `.^roles` — but `.HOW` must return a *persistent* object, because `T.HOW does SomeRole` mixins have to stick. So `ClassInfo` carries its meta-object rather than building one per call:

```
// src/Value.h — ClassInfo
Value howObj;   // the persistent .HOW metaobject
```

15.6 `&builtin` as a value

`&say` must be a `Callable` value, and `&dir.wrap({...})` must mutate *the* `Callable` the call path consults — that is exactly how `File::Find`'s test suite mocks `dir`. So a reference to a builtin is memoised:

```
// src/Interpreter.h
std::map<std::string, Value> builtinRefs_;
```

populated lazily, and empty for programs that never take a builtin reference — which keeps the check on the call path free.

15.7 Where the time goes

After the two fixes in Chapters 9 and 10, a method call on a built-in costs roughly three times what Rakudo charges, down from about twenty. The profile of `for ^6000000 { "ab".uc }`:

	share
heap allocate and free	31%
`Value` copy and destroy	11%
method-name comparison	8.5%
the dispatch function's own body	6.1%

The 42% in allocation and value churn is the remaining target, and it has a known shape: both `methodCall` and `methodCallInner` take their invocant and argument list **by value**, so every call copies a `Value` — with its eleven `shared_ptr` members — and heap-allocates a `ValueList`.

Switching both to `const&` is mechanically small: eighteen compile errors, each either "make a local copy here" or "let this helper take `const&`". It has not been done because it trades away the accidental safety that copying provides against a callee mutating the container its own invocant lives in. That wants an aliasing audit of 178 call sites, not a quick pass.

15.8 Honest limitations

- **The method resolution order is depth-first, not C3.** Diamond hierarchies can resolve differently from Rakudo.
- **No `X::Multi::Ambiguous`.** Equal-specificity candidates resolve silently to the first declared.
- **Role composition is last-writer-wins.** Composing two roles that define the same method copies both into the table with no conflict diagnostic.
- **The ladder's order is load-bearing and undocumented in the code beyond a warning.** An arm moved for readability is a behaviour change.

Chapter 16

The Object System

A user class is a `ClassInfo`. An instance is an `ObjectData` that points at one. There is no per-object method table, no vtable, and no per-class code generation.

```
// src/Value.h — abridged
struct ClassInfo {
    std::string name;
    std::shared_ptr<ClassInfo> parent;
    std::string nativeParent;                          // `is Str`, `is Cool`
    std::vector<std::shared_ptr<ClassInfo>> extraParents;
    std::vector<ClassAttr> attrs;
    std::map<std::string, Value> methods;              // Code values
    std::map<std::string, std::string> rules;          // grammar rules
    std::vector<std::string> ruleOrder;
    bool isGrammar = false, isRole = false;
    std::string repr;                                  // is repr('CStruct')
    std::string ver, auth, api;
    std::set<std::string> requiredMethods, doneRoles;
    Value howObj;                                      // persistent .HOW
    std::shared_ptr<Env> declEnv;                      // for attr defaults
    ClassDecl* decl = nullptr;                         // for role parameters
    std::vector<std::pair<std::string, Value>> roleParamBindings;
};

struct ObjectData {
    std::shared_ptr<ClassInfo> cls;
    std::map<std::string, Value> attrs;
    Value boxed; bool hasBoxed = false;                // for `5 but Role`
};
```

Everything a class *is* lives in that first struct, including the grammar rules if it happens to be a grammar — which is how a grammar is just a class whose methods can be regexes (Chapter 21).

16.1 Registration and lookup

`ClassInfo`s are registered in one flat map at declaration:

```
// src/Interpreter.h
std::unordered_map<std::string, std::shared_ptr<ClassInfo>> classes_;
std::unordered_map<std::string, std::string> classAliases_;
```

Flat, not per-compilation-unit. Two modules declaring the same unqualified class name collide — a real divergence from Rakudo, listed in Chapter 29.

The alias map softens the consequences of flatness. Registering `URI::Path` also aliases its tail, `Path`, unless a real class already claims that name. And a qualified name that is a *partial* path resolves by unique suffix match:

```
// src/Interpreter.h — resolveClassAlias
if (n.find("::") != std::string::npos) {
    const std::string suffix = "::" + n;
    const std::string* hit = nullptr;
    for (auto& kv : classes_) {
        if (...kv.first does not end with suffix...) continue;
        if (hit) return n;          // ambiguous — leave it alone
        hit = &kv.first;
    }
    if (hit) return classAliases_.emplace(n, *hit).first->second;
}
```

so `Globber::Match` inside `unit class IO::Glob` finds `IO::Glob::Globber::Match`. Only qualified names take that road — a bare `Match` must not silently find a nested one — and an ambiguous suffix is left alone rather than guessed at. The successful answer is memoised into the alias map.

16.2 Construction

The default `new` and `bless` share one path. It walks the class chain **parent-first**, gives each attribute its default, folds named arguments into the attribute map, and then runs `BUILD` and `TWEAK`:

```
// src/Builtins.cpp — default construction, abridged
for (auto it = chain.rbegin(); it != chain.rend(); ++it)      // parent-first
    for (auto& at : (*it)->attrs) {
        Value dv = at.hasDefVal ? at.defVal
                 : at.def       ? eval(at.def)
                 : at.sigil == '@' ? Value::array()
                 : at.sigil == '%' ? Value::makeHash() : Value::any();
        od->attrs[at.name] = dv;
    }
for (auto& arg : args)
    if (arg.t == VT::Pair) od->attrs[arg.s] = *arg.pairVal;
if (Value* build = ci->findMethod("BUILD")) invokeMethod(*build, self, args);
if (Value* tweak = ci->findMethod("TWEAK")) invokeMethod(*tweak, self, args);
```

Parent-first matters: a subclass's default for an inherited attribute must win, and it does because it is written last.

Attribute defaults are *expressions*, evaluated at construction in the scope the class was declared in — which is what `ClassInfo::declEnv` is for. The compiled backend precomputes them where it can, into `ClassAttr::defVal`, and falls back to the expression otherwise.

`is required` throws when construction supplies no value, carrying the reason string from `is required("it is a good idea")` into the message. `is built` makes a *private* attribute settable by name at construction anyway.

16.3 Attributes and accessors

`$!x` is a direct read or write of the attribute map. `$.x` is a public accessor: method lookup runs first, and on a miss `findAttr` supplies the attribute. The compiled backend uses the same map through two helpers:

```
// src/Interpreter.cpp
Value rtAttrGet(const Value& self, const std::string& name) {
    if (self.t == VT::Object && self.obj) {
        auto it = self.obj->attrs.find(name);
        if (it != self.obj->attrs.end()) return it->second;
```

```
    }
    return Value::any();                    // absent attribute → (Any)
}
Value& rtAttrRef(Value& self, const std::string& name) {
    return self.obj->attrs[name];           // autovivifies
}
```

`ClassAttr` carries more than storage: the declared type and its `:D`/`:U` smiley, `is rw`, `is required`, `is built`, a container trait (`has %.a is Set`), a `handles` list for delegation, an object-keyed flag, and **user traits**.

That last one is how third-party traits reach their own code. `has $.id is json-name('licenseId')` records `{"json-name", Str}` on the attribute, surfaced on the `Attribute` meta-object, which is exactly what `JSON::Unmarshal`'s role checks and accessors look for. The parser does not know what `json-name` means and does not need to.

Assigning through an accessor is guarded: `$obj.attr = v` on a private or read-only attribute throws, while `$obj!attr` and a plain method stay writable. The declared type is enforced too, which needs the attribute's type at the assignment site — hence:

```
// src/Interpreter.h — ExecContext
std::string lastLvalueAttrType;
```

recorded by the method-call lvalue arm so a role-typed `has C $.x is rw` rejects `42`.

16.4 Roles

A role is a `ClassInfo` with `isRole` set, a set of `requiredMethods`, and a set of `doneRoles`. Composition copies the role's methods and attributes into the class and records membership.

Required methods are checked **at class declaration**, using the same `findMethod` that dispatch uses:

```
for (ClassInfo* role : composed)
    for (const std::string& req : role->requiredMethods)
        if (!ci->findMethod(req))
            throw RakuError{Value::typeObj("X::Role::Unimplemented"), ...};
```

`.does` and `~~` consult `doesRole`, which is true for the role itself, for directly or transitively composed roles, and for roles done by parents:

```
// src/Value.h
bool doesRole(const std::string& rn) const {
    if (isRole && name == rn) return true;
    if (doneRoles.count(rn)) return true;
    if (parent && parent->doesRole(rn)) return true;
    for (auto& p : extraParents) if (p && p->doesRole(rn)) return true;
    return false;
}
```

Attribute composition deduplicates by the address of the *declaring* attribute node, recorded in `ClassAttr::declId`. So a diamond composition — two roles both doing a third — contributes the shared attribute once rather than twice.

Parameterised roles carry their parameters as a `Param` list on the AST declaration, and a composition binds arguments to them:

```
// src/Value.h — ClassInfo
std::vector<std::pair<std::string, Value>> roleParamBindings;
```

The bindings are injected into the scope of the composing class's methods, so the role's body sees `%phase-defaults` in `does Cro::Policy::Timeout[%h]`. `makeRolePun` handles the anonymous case, `R[Int].new`, by building a punned class on the spot.

Role composition is **last-writer-wins**: two roles defining the same method both copy into the table, with no conflict diagnostic. That is a known divergence.

16.5 Mixins: `but` and `does`

`5 but Role` and `%h does R` add a role to a value at run time.

For a value that is already an object, the role is composed into a fresh anonymous subclass. For a **non-object base** there is no `ObjectData` to extend, so the value is *boxed*:

```
// src/Interpreter.cpp — mixinValue
obj = std::make_shared<ObjectData>();
obj->boxed = base;          // the original 5 is kept here
obj->hasBoxed = true;
// … build an anonymous subclass composing the role, wrap in a VT::Object …
```

Dispatch on the result checks the role's methods first; anything not found — and not an identity method — is delegated back to the box:

```
// src/Builtins.cpp — methodCall
if (inv.t == VT::Object && inv.obj && inv.obj->hasBoxed && inv.obj->cls &&
    !inv.obj->cls->findMethod(m) && !inv.obj->cls->findAttr(m)) {
    static const std::set<std::string> keepOnObj =
        {"does","HOW","WHAT","WHICH","defined","DEFINITE"};
    if (!keepOnObj.count(m))
        return methodCall(inv.obj->boxed, m, args, rwArgs);
}
```

So `(5 but Role).succ` runs `Int.succ` on the `5`, while the role's methods and `.does`/`.WHAT` see the mixed object. The `keepOnObj` set is the list of questions that must be answered *about the mixin* rather than about the box.

`$x but Pair` mixes an attribute rather than a role, using the same machinery with a synthesised anonymous role — which is what `anonMixinSeq_` names.

16.6 `augment` and `supersede`

`augment class Foo { … }` on a user class merges methods into the existing `ClassInfo`. On a **built-in type** there is no `ClassInfo`, so the methods go into a side table consulted before the native dispatch ladder:

```
// src/Interpreter.h
std::unordered_map<std::string,
                   std::unordered_map<std::string, Value>> builtinExt_;
```

keyed by type name then method name, and searched along the native ancestry so augmenting `Cool` reaches `Int` and `Str`. Chapter 15 covers the dispatch side; Chapter 27 covers why two compiled fast paths have to check whether this map is empty.

`.^add_method($name, $code)` writes into `ClassInfo::methods` directly, which is the whole of runtime method injection.

16.7 Package stashes and `.WHO`

```
// src/Interpreter.h
std::map<std::string,
         std::shared_ptr<std::map<std::string, Value>>> pkgStashes_;
```

One shared map per package **name**, and the sharing is the point. `.WHO` used to build a fresh empty `Hash` on each call, so `EXPORTHOW.WHO.<grammar> = SomeHOW` wrote into

a temporary and died with “Target is not assignable”. Reading a stash also re-syncs the package’s qualified globals into it, so `our`-scoped symbols appear.

A `module` or `package` has no `ClassInfo` at all, so its `:ver`/`:auth`/`:api` adverbs live in a separate small map, `pkgMeta_`, which is what `.^ver` answers for a package.

16.8 Honest limitations

- **Depth-first method resolution**, as in Chapter 15.
- **Role conflicts are silent**, last writer wins.
- **The class registry is flat**, so unqualified class names from different modules collide. The alias machinery reduces the pain but does not remove the cause.
- **`ClassInfo::decl` is a raw `ClassDecl*`** into the AST, kept alive by the same “the tree is immortal” rule as `Callable::body` (Chapter 6).

Chapter 17

Laziness, Junctions, and the Wilder Values

Three parts of Raku resist a straightforward tree-walker: infinite lists, values that are several values at once, and `gather/take`, which wants a coroutine. None of the three gets a new `VT`. All three are an existing tag plus some state.

17.1 Lazy and infinite sequences

An infinite list like `1, 2, 4 ... *` or `(1..Inf).map(*²)` obviously cannot be a materialised `ValueList`. Raku++ attaches a generator to an otherwise ordinary array value.

```
// src/Interpreter.h
struct LazySeqState {
    std::function<bool(ValueList&)> appendNext;  // one more; false = exhausted
    bool infinite = false;   // truly unbounded: elems/pop/[*-1] must die
};
```

The already-computed **prefix** lives in the `Value`'s `arr`; the generator lives in the opaque `ext` slot — the same slot a `Promise` or a `Channel` would use. `appendNext` appends exactly one element and reports whether more exist.

17.1.1 Building one

`seqOp` implements the `...` operator. It splits the left side into a seed list — a trailing `Code` seed becomes the *generator* — and classifies the right endpoint. A bounded endpoint (`1 ... 10`) is computed eagerly in a loop, capped at a million. An infinite endpoint builds a `LazySeqState`:

```
// src/Interpreter.cpp — seqOp
if (infinite) {
    auto st = std::make_shared<LazySeqState>();
    st->infinite = true;
    st->appendNext = [...](ValueList& cache) -> bool {
        /* feed the last `arity` cached elements to the generator, or step
           the detected progression; push the next value */
    };
    out.ext = st;
}
```

When there is no explicit generator, the operator *detects the progression* from the seed: a constant arithmetic difference, a constant geometric ratio, or — for strings — repeated `succ`/`pred`. That is Raku's rule, and it is why `1, 3, 5 ... 99` and `'a', 'b' ... 'z'` both work with no closure written.

`...` is also **list-associative**: `1 ... 5 ... 1` and `'A'...'Z', 'a'...'z'` are one operator over a list of lists, where each group's first element closes the previous segment. That is `seqOpGroups`, shared with the compiled backend.

A bare `1..Inf` assigned to an array builds a simpler counting generator, in `coerceArray`:

```
auto st = std::make_shared<LazySeqState>(); st->infinite = true;
auto next = std::make_shared<long long>(start);
st->appendNext = [next](ValueList& cache) -> bool {
    cache.push_back(Value::integer((*next)++)); return true;
};
a.ext = st;
```

17.1.2 Forcing elements

Consumers grow the prefix on demand:

```
// src/Interpreter.cpp
void Interpreter::materializeLazy(const Value& v, size_t n) {
    auto st = std::static_pointer_cast<LazySeqState>(v.ext);
```

```
    while (v.arr->size() < n && v.arr->size() < CAP)
        if (!st->appendNext(*v.arr)) break;
}
```

So `@lazy[5]` materialises six elements and then indexes; `.head(3)` materialises three; `.first(&pred)` pulls one at a time until the predicate matches.

Operations that need the *end* of an infinite list cannot complete, and say so:

```
// src/Builtins.cpp
if (m == "elems" || m == "end" || m == "pop" || m == "tail" ||
    m == "reverse" || m == "sort" || m == "sum" || m == "min" ||
    m == "max" || m == "join" || m == "Str" || m == "gist")
    throw RakuError{Value::typeObj("X::Cannot::Lazy"),
                    "Cannot " + m + " a lazy list onto an Array"};
```

A *finite* lazy value — a `gather` that outgrew its probe — instead forces full materialisation, which is the right answer for it.

17.1.3 Lazy `.map` and `.grep`

`.map` over a lazy source builds a **new** `LazySeqState` that pulls from the source on demand, which is what makes `(1..Inf).grep(*.is-prime).head(5)` terminate:

```
// src/Builtins.cpp — .map over a lazy source
st->appendNext = [self, src, fn](ValueList& cache) -> bool {
    size_t si = cache.size();
    self->materializeLazy(src, si + 1);          // pull one more
    if (si >= src.arr->size()) return false;
    cache.push_back(self->callCallable(fn, { (*src.arr)[si] }));
    return true;
};
```

`.grep` loops pulling source elements until its predicate matches; `.skip` builds a shifted view.

17.1.4 The cap

Every materialisation path shares a hard ceiling of **one million elements**. It is a runaway safety net, not a semantic limit, and it means “infinite” is bounded in practice. A consumer that genuinely needs the millionth element of a generated sequence will get it; one that needs the ten-millionth will not.

17.2 `gather` and `take`, without coroutines

`gather { … take … }` should suspend the block at each `take` and resume it when another element is demanded. That is a coroutine, and a tree-walker built on the C++ stack does not have one.

The strategy is **probe and double**. The collector and a per-gather element cap live on the execution context:

```
// src/Interpreter.h — ExecContext
std::vector<std::shared_ptr<ValueList>> gatherStack;
std::vector<size_t> gatherLimits;   // 0 = unlimited
```

The block is first run under a small cap of 64. If it finishes within the cap it was finite, and the result is returned eagerly. If the cap was *hit*, the result becomes a `LazySeqState` that grows by **re-running the block** with a larger cap:

```
// src/Interpreter.cpp
st->appendNext = [this, runGather](ValueList& out) -> bool {
    ValueList grown;
    bool more = runGather(out.size() + std::max<size_t>(64, out.size()), grown);
    for (size_t i = out.size(); i < grown.size(); i++) out.push_back(grown[i]);
    return more;
};
```

Doubling keeps the re-run cost amortised linear. A `take` that pushes past the current cap unwinds the block with an empty marker exception:

```
// src/Builtins.cpp — take
auto& coll = *tctx_.gatherStack.back();
for (auto& x : a) coll.push_back(x);
if (lim && coll.size() >= lim) throw StopGatherEx{};
```

So an infinite `gather` runs its block only far enough to satisfy each demand, stops, and re-enters later for more.

The honest cost of this design: **the block runs more than once, from the start.** A `gather` whose block has side effects — printing, or mutating a counter — will repeat them. That is a genuine divergence from a coroutine implementation, and the reason the initial probe cap is generous enough that most real `gathers` never re-run at all.

17.3 Junctions

A junction has no `VT` of its own. `any(1, 2, 3)` is a `VT::Array` whose elements are the eigenstates, tagged by `enumName`:

```
// src/Value.cpp — typeName(), the VT::Array case
if (enumName == "any" || enumName == "all" ||
    enumName == "one" || enumName == "none") return "Junction";
```

They are built by the `any`/`all`/`one`/`none` routines, the matching methods, and the `|`, `&`, `^` infix operators.

Autothreading — distributing an operation over the eigenstates and recombining — happens at each place a value is consumed, and there are four such places.

Operators. A comparison *collapses* to a single `Bool` according to the junction type; any other operator produces a **new** junction of the per-eigenstate results:

```
// src/Interpreter.cpp — applyArith
Value out = Value::array(); out.enumName = j.enumName;
for (auto& e : *j.arr)
    out.arr->push_back(applyArith(op, jleft ? e : l, jleft ? r : e));
return out;                          // any(1,2) + 10  ==>  any(11, 12)
```

Method calls, with a small allow-list that acts on the whole junction. **Callable invocation**, in `callCallableRaw`. **Smartmatch**, in `~~`.

That is why `if 3 == any(1, 2, 3)` works: the `==` sees a junction on the right, threads the comparison across the eigenstates, and collapses `any` to `True`.

One escape hatch exists, because `Junction.THREAD` must pass each eigenstate — junctions included — through whole:

```
// src/Interpreter.h
static thread_local bool noAutothread_;   // one-shot, consumed by the next call
```

17.4 `Whatever` and `WhateverCode`

`*` is `VT::Whatever`. An expression containing one *curries* into a `WhateverCode` — a `Callable` with a flag and an arity:

```
// src/Value.h — Callable
bool isWhateverCode = false;
long long whateverArity = 0;   // `* + *` consumes 2
```

so `* + 1` becomes a one-argument closure and `* + *` a two-argument one. Composition works because currying an expression that already contains a `WhateverCode` produces another one.

Deciding whether to curry is a syntactic question, answered by walking the expression for a literal `*`:

```
// src/Interpreter.h
static bool exprHasWhateverLit(const Expr* e);
```

In *subscript* position `*` means something else entirely — `@a[*-1]`, `@a[*]` — and is handled by a dedicated path, `idxW`, rather than by currying.

17.5 Hyper operators

»op« and friends apply an operator element-wise, with rules about which side may be extended. All spellings — `>>op<<`, »op«, `>>[&op]<<` — funnel into one core:

```
// src/Interpreter.h
Value hyperCore(Value& l, Value& r, bool strictL, bool strictR,
                const std::function<Value(const Value&, const Value&,
                                          Value*, Value*)>& apply,
                Value* lroot = nullptr, Value* rroot = nullptr,
                bool wantSlots = false);
Value hyperUnary(const std::string& op, Value v);         // -«(...)
Value hyperPostfixApply(const std::string& op, Value v);  // @a»++
```

The `strictL/strictR` flags are the dwimmy-versus-strict distinction Raku draws between `<<` and `>>` on each side; the slot parameters exist because `@a»++` must write *through* to the elements, which is the same write-back problem as an `is rw` parameter and uses the same one-shot register.

17.6 Flip-flops

`ff` and `fff` need per-*site* state — the same operator at two places in a program has two independent latches:

```
// src/Interpreter.h
struct FlipFlop { bool on = false; long long seq = 0; };
std::unordered_map<const void*, FlipFlop> ffState_;
```

keyed on the AST node's address, which is stable for the life of the program. The `seq` counter is there because the result while the latch is on is the *count of elements since it fired*, not a plain `Bool`.

This is the one place in the interpreter where a map keyed on a node pointer is the right answer rather than a hazard. It works here because AST nodes are never freed. Chapter 22 describes a case where the same idea failed for exactly the opposite reason — freed addresses being recycled.

Chapter 18

Node Specialization

This chapter is a single optimisation, described in full, because it is the clearest worked example in the project of the method the whole thing runs on: count the opportunity before building anything, remove work rather than adding cleverness, and keep a control in the measurement.

18.1 What it does

`evalBinary` and `evalIndex` recognise a handful of syntactic **shapes**, record that verdict on the node, and take a short path.

shape	example
`$var OP literal`	`$n < 2`, `$n - 1`
`literal OP $var`	`2 * $n`
`$var OP $var`	`$a + $b`
`@arr[$var]` / `@arr[literal]`	`@a[$i]`, `@a[0]`

Three costs disappear on those paths.

The variable's copy. `eval()` on a `VarExpr` returns the value *by value* — hundreds of bytes, several strings, eleven refcount increments. The fast path takes a `Value*` out of the environment and hands it to `applyArith`, which already took `const&`.

The literal's reconstruction. The `1` in `$n - 1` was rebuilt into a fresh `Value` on every visit. It is now built once and kept on the node.

The probes that cannot apply. Every binary operation ran two string comparisons for `DateTime/Duration` handling, plus the hyper and `Z/X` checks. None of them can match two plain scalars.

For `@arr[$i]` the win is the same in kind: the general path opens by copying the whole container into a local `Value` in order to read one element out of it.

18.2 Why these shapes, and not constant folding

The idea started as "let `-O` fold the tree before walking it". That was measured first, with `tools/ast-opportunity.raku`, which reads `rakupp --dump-ast` and counts the patterns a tree optimizer would rewrite. Across 48 real files — the showcase interpreters, the examples, the tools — 51,353 AST nodes:

pattern	sites	per 1k nodes
constant folding (both operands literal)	37	0.7
binary with ONE literal operand	**1,282**	**25.0**
`$x = $x + ...` self-update	24	0.5
constant condition	0	0.0

Classical constant folding has essentially nothing to fold in real Raku: 0.07% of nodes, and none of it in a loop. Dead branches: none at all. What *is* plentiful is the operand shapes above, and they sit in loop conditions and index arithmetic, where they run millions of times.

That table is the whole reason this is a specialisation pass and not a folding pass. It took two minutes to produce, and it prevented building the wrong thing. **Re-run the tool before assuming any other classical optimisation is worth building**; the static count is an upper bound on the opportunity.

18.3 What "cached" means here

"Cache" is a misleading word for this, so be precise. In most contexts caching means remembering a computed result so you do not recompute it. **Nothing about the result is remembered.** What is remembered is a fact about the source code.

The entire mechanism is two extra fields on each `Binary` node — one byte and one pointer. No table, no map, no key, nothing global:

```
// src/Ast.h — Binary
mutable DecidedOnce<signed char> fastShape{-1};   // which shape this node is
mutable DecidedOnce<const void*> litVal{nullptr}; // the literal, shapes 1 and 2
```

Walk one node, `$n - 1`, through its life.

After parsing, the fields say nothing:

```
op        = "-"
lhs      -> VarExpr("$n")
rhs      -> IntLit(1)
fastShape = -1            <- nobody has looked at this node yet
litVal    = nullptr
```

On the first evaluation the interpreter asks a question *about the syntax*: is the left child a plain lexical, and the right child a scalar literal? Both yes, so it writes the answer down:

```
fastShape = 1             <- "variable, operator, literal"
litVal   -> Value(1)      <- the 1 from the source text
```

On every evaluation after that, including the millionth, the node reads `fastShape == 1` and goes straight to: look up `$n`, check its type, apply the operator. It never asks the question again.

So the two fields hold two different kinds of thing.

- **`fastShape` is a classification of the syntax** — "this expression is written as variable-operator-literal". That is a fact about the *text of the program*. It is true before the program starts and stays true until it exits, because source code does not rewrite itself.
- **`litVal` is the literal's value.** This is cached in the ordinary sense, but a literal is a constant by definition: if the source says `1`, it is 1 forever.

`$n` appears in neither field — not its value, not its address, not the scope it was found in. Every evaluation does a fresh lookup by name through the current environment and a fresh type check on whatever comes back. Which is why all of this behaves normally:

```
my $v = 1;
for ^4 { say $v + 1; $v = $v ~~ Int ?? "10" !! 1 }   # 2, 11, 2, 11
```

One AST node, four evaluations, a variable that changes both value and type underneath it, and the right answer each time. The visits where `$v` holds a `Str` still take the fast path, because `Str` is in the guard; had it become a `DateTime`, that visit alone would have fallen through to the general path.

Why store anything at all, then? Because the classification is not free — it is several branches: check both child node kinds, check the variable name's second character, check the literal is not a bignum or a `Rat`. Asking that once per *node* instead of once per *evaluation* is the whole optimisation. A `fib(29)` run evaluates `$n - 1` about 1.6 million times and answers the question once.

It is less a cache than a **sticky note on the node**, written the first time anyone reads it. The precedent sits directly above it in the same struct: `simpleOp` does exactly the same thing for "is this a plain operator, or one of the special-cased ones?", also decided once, also from the syntax alone.

18.4 The guards

The fast path is declined, and the untouched general path runs, unless:

- **the name is a plain lexical** — second character a letter or _, which excludes every twigilled and special name (`$*dyn`, `$?FILE`, `$^a`, `$/`), each of which has its own lookup rules;
- **the value is `Int`, `Num`, `Str` or `Bool` with an empty `hashKind`**, which excludes `Proxy`, junctions, `DateTime`, `Duration`, `Failure`, allomorphs and undefined values;
- **for a subscript**: no adverb, not multidimensional, not a slice, a plain materialised `Array` (no `ext`, so no lazy sequence), and a **non-negative** in-range integer index.

`Rat` is deliberately left out even though it would probably qualify; so is anything with a shared payload, so that a cached literal can never be mutated through.

Falling through costs nothing incorrect. The only things the fast path evaluates are variable lookups and cached literals, neither of which has side effects, so nothing is ever evaluated twice.

18.5 Why it is not a new node kind

The obvious implementation is a new `NK::` per shape — a superinstruction — rewritten into the tree by a pass. This is deliberately not that.

A new node kind has to be taught to the parser, `AstDump`, `AstEmit` (which serialises the tree for `--aot`), `Codegen`, `Lint`, and every `switch` over `NK`; any of those missing a case is a silent wrong answer. Caching a verdict *inside* the existing node needs none of them, degrades to the old behaviour by construction, and leaves the AST dump and both compiling back ends seeing exactly the tree they saw before.

It also needs no `-O` flag. The transformation cannot change semantics — same `applyArith`, same operands — so it is always on, decided lazily per node.

18.6 Measured

Alternating A/B, medians of seven, both binaries interleaved so that machine drift hits each equally:

kernel	base	specialised	delta
vars — `$a OP $b`	840.8 ms	686.5	**−18.3%**
lits — `$a OP 1`	728.9	600.0	**−17.7%**
fib	478.6	422.4	**−11.7%**
asg	395.1	349.5	**−11.5%**
index — `@a[$i]`	195.3	178.4	**−8.7%**
hash	42.6	40.0	−6.2%
loopsum	208.6	209.6	+0.5%
ctl — method dispatch only	327.9	327.0	**−0.3%**

The control is the point of the table. It is a loop of pure method dispatch, containing nothing the specialisation can touch, and it does not move. Without it the other rows would only be evidence that the machine was quieter the second time.

Roast came out identical: 196,590 assertions, 631 files fully passing, with not one file moving in either direction.

18.7 What went wrong on the way

Three mistakes, all invisible to reasoning and obvious to measurement. They are the reason this chapter exists.

The first version made the control 5.7% *slower*. It bound the right operand through a `std::optional` so the literal could be used by reference — which cost the *general* path its copy elision, since `Value r = eval(...)` constructs in place where `emplace` move-constructs. The lesson is structural: **add an early exit, never restructure the shared code underneath it.** The final version leaves everything after the fast path byte-identical.

The literal cache was a reassignable `shared_ptr`. Under parallel execution a second thread rebuilding it could free the object while this thread held a raw pointer into it. It is now allocated once and never freed: the node outlives every evaluation, so there is nothing to reclaim before exit, and a racing double-build leaks one `Value` instead of dangling.

A negative subscript nearly changed behaviour. The first `evalIndex` version wrapped negative indices from the end, copying logic from a branch further down that plain arrays never reach. `my $i = -1; @a[$i]` must throw `X::OutOfRange`. Rakudo could not have caught this — it rejects the spelling at compile time — so it was found by diffing against the **previous rakupp binary**, which is now the habit: for a change that is supposed to be semantically invisible, the baseline binary is the oracle, not another implementation.

18.8 Codegen already had this

Worth knowing before anyone ports it: `--exe` does not need it. The code generator keeps variables in C++ locals, so there is no lookup and no copy, and it calls `applyArith` directly rather than going through `evalBinary`, so the temporal and hyper probes never existed there. With `-O` it goes one level deeper still, into a raw `int64` lane that never materialises a `Value` for the arithmetic at all (Chapter 26).

What this change really did was close part of the gap between the interpreter and what the code generator had been doing all along — which is why the win was large.

18.9 Extending it

The same treatment fits other shapes, in rough order of expected value:

- `Assign` where the target and the left operand are the same variable (`$x = $x + 1`) — write through the slot instead of building and storing;
- `%h{$k}` and `%h<lit>`, the associative twin of the array path;
- `Unary` on a plain lexical (`-$x`, `!$x`, `++$i`);
- method calls on a plain lexical invocant, which is where the *other* half of the profile lives.

Whatever is added, the two rules that made this one safe still apply: **guard on the runtime value every time, and leave the general path underneath untouched.**

Part V

The Regex and Grammar Engine

Chapter 19

Compiling a Regex

Raku's regex sub-language is not a bolt-on. It has its own quoting rules, its own whitespace conventions, named rules, code assertions, and grammars built out of it. Raku++ implements it as a **second, small recursive-descent compiler** inside the runtime, producing a node tree that a backtracking matcher walks.

The whole thing is one class in one file:

```
// src/Regex.h
class Regex {
public:
    explicit Regex(const std::string& pattern, const std::string& flags = "");
    bool ok() const;
    bool search(const std::string& subject, long startPos, RxMatch& out) const;
    bool matchAt(const std::string& subject, long pos, RxMatch& out,
                 const SubResolver& r,
                 const std::set<std::string>* lexNames = nullptr) const;
};
```

19.1 The pattern text arrives unparsed

The Raku lexer does not tokenize regex syntax (Chapter 3). A `/.../` literal, an `rx//`, an `s///` and a `token NAME { ... }` body all travel through the token stream as raw text. The regex compiler re-lexes that text with its own scanner.

That separation is the reason two grammars can coexist without either contaminating the other, and it is why the constructor takes a `std::string` rather than a token range.

Adverbs arrive as a **flags string** alongside the pattern: `i` for ignorecase, `s` for sigspace, `r` for ratchet, `m` for ignoremark, and so on. A `token` compiles with `r`; a `rule` with `sr`.

19.2 The node tree

```
// src/Regex.h
enum class K {
    Lit, Any, Class, Seq, Alt, Conj, Rep, Group,
    AnchorStart, AnchorEnd, WBLeft, WBRight, Nop,
    Subrule, Look, Code, VarMatch, CapStart, CapEnd
};
```

Nineteen kinds, and the interesting information is in the fields rather than the tags:

```
// src/Regex.h — Node, abridged
struct Node {
    K k;
    std::string lit;                                  // Lit
    std::vector<std::unique_ptr<Node>> kids;
    std::vector<std::pair<unsigned char, unsigned char>> ranges;   // Class
    std::vector<std::pair<uint32_t, uint32_t>> cpRanges;
    std::vector<std::string> clusterMembers;          // multi-codepoint graphemes
    std::string classFlags, negClassFlags, uprop;
    bool negate = false, icase = false, imark = false, multiline = false;
    mutable uint32_t byteset[8];                      // built on first use
    mutable std::atomic<bool> bytesetReady{false};
    long min = 0, max = -1; bool greedy = true, possessive = false;  // Rep
    std::unique_ptr<Node> sep; bool sepTrail = false;    // X+ % Y
    std::string repCode;                              // ** { ... }
    bool runOnly = false, ltmStop = false;            // Code
    bool firstMatch = false, classCombo = false;      // Alt
    int capIndex = -1; std::string capName; bool listCap = false;   // Group
    std::string ruleName, ruleArgs, ruleAlias;        // Subrule
    bool aliasDotted = false, ruleCapture = true;
    mutable const GrammarRuleMeta* metaCache = nullptr;
    bool behind = false;                              // Look
    mutable long lookMin = 0, lookMax = -1;
```

```
    mutable std::unique_ptr<LtmNfa> ltmNfa;         // Alt
};
```

Three fields deserve attention now; the rest appear where they are used.

19.3 The byteset: a per-node character-class cache

A character class has to answer "does this byte match?" for every position it is tested at, taking into account ranges, named classes, negation and case-folding. Computing that from the class's own data each time is a loop.

So each `Class` node caches the answer for all 256 bytes as a bitmap, built on first use:

```
mutable uint32_t byteset[8];                    // 256 bits
mutable std::atomic<bool> bytesetReady{false};
```

The flag is an *acquire/release* atomic rather than a relaxed one, and the comment says why: it gates a 32-byte array, so a relaxed store would not guarantee that a reader which sees the flag also sees the bytes. This is the one place in the codebase where the weaker ordering used everywhere else would be wrong, and it is worth noticing as a general point — `DecidedOnce` is safe because it publishes a *single word*.

Codepoints above 255 do not fit a byteset, so they live in `cpRanges` and are tested separately. Multi-codepoint graphemes — a class member written `\c[A, COMBINING ACUTE ACCENT]` — live in `clusterMembers` and are compared as whole clusters.

19.4 The parser

Five mutually recursive functions, one per precedence level:

```
// src/Regex.h
NodePtr parseAlt();     // a | b   and   a || b
NodePtr parseConj();    // a & b
NodePtr parseSeq();     // concatenation
NodePtr parseQuant();   // * + ? ** N..M, with % separators
NodePtr parseAtom();    // everything else
```

`parseAtom` is the large one. It handles literals and quoted literals, `.`, the escape classes `\d \w \s` and their negations, `<[...]>` and `<-[...]>` character classes with ranges, named classes and Unicode properties `<:Nd>`, groups `[ ]` (non-capturing)

and `( )` (capturing), named captures `$<x>=[...]`, subrule calls `<name>` with their aliasing and capture variants, lookarounds, code assertions `<?{...}>` and `<!{...}>`, side-effect blocks `{...}` and `:my` declarations, `$var` interpolation, the capture-span markers `<(` and `)>`, anchors, and word boundaries.

Two parse-time pieces of state make the adverbs scoped rather than global:

```
// src/Regex.h
bool curIcase_ = false;   // :i / :!i, scoped to the enclosing group
bool curImark_ = false;   // :m / :ignoremark, likewise
int assertDepth_ = 0;     // >0 inside an assertion, so parseSeq stops at `>`
```

so an inline `:i` inside a group applies to that group's nodes and no further — which is why `icase` and `imark` are per-*node* flags rather than per-regex ones.

19.5 Sigspace

Under `:s`, or in a `rule`, whitespace in the pattern is significant and means "match optional whitespace here". That is implemented structurally rather than by a matcher flag:

```
// src/Regex.h
static NodePtr wsWrap(NodePtr inner);   // Seq(inner, <.ws>)
```

Each atom is wrapped in a sequence with a non-capturing `<ws>` subrule after it. The consequence is that sigspace needs no support anywhere in the matcher: the `<ws>` calls are ordinary subrule nodes.

19.6 Ratchet

`token` and `rule` are *ratcheting*: their quantifiers are possessive and their matches commit — no backtracking into a token once it has matched.

```
// src/Regex.h
bool ratchet_ = false;
```

The flag makes every `Rep` node possessive at compile time. It is also what makes the packrat memo in Chapter 21 sound: a rule that does not backtrack has exactly one match at a given position, so caching it is safe.

19.7 Two whole-pattern optimisations

A single-character rule is inlined at its call sites. A grammar full of `token space { <[\ \t]> }` would otherwise pay a full subrule call per character:

```
// src/Regex.h
bool rootIsSingleChar() const;
long trySingleChar(const std::string& s, long pos) const;   // pos+1, or -1
```

The call site tests the property once, caches the answer on the rule's metadata, and thereafter tests the character directly.

Lookbehind width is computed once. A naive lookbehind scans every earlier start position, which is O(pos) per attempt. Instead the inner pattern's possible width is derived and the scan window bounded:

```
// src/Regex.h
std::pair<long, long> nodeWidth(const Node* n, MState& st) const;
```

```
// src/Regex.cpp — the Look case
if (!n->lookWidthReady) {
    auto w = nodeWidth(child, st);
    n->lookMin = w.first; n->lookMax = w.second; n->lookWidthReady = true;
}
long hi = pos - n->lookMin;
long lo = n->lookMax < 0 ? 0 : pos - n->lookMax;
```

O(width) instead of O(position).

19.8 Retired Perl 5 metacharacters

Raku deliberately reuses some Perl 5 regex syntax for other things, and Roast checks that the old spellings produce a specific error rather than silently meaning something else:

```
// src/Regex.h
struct ObsoleteEscape { std::string seq; };
const std::string& obsolete() const;   // non-empty: a retired P5 metachar
```

The constructor records the offending sequence — `\A`, `\z`, `\G`, `\p`, `\Q`, `\1` — so the caller can raise `X::Obsolete` instead of reporting a failed match. Distinguishing "this pattern is wrong" from "this pattern did not match" is worth the extra field.

19.9 Interpolation happens before compilation

`/$var/` and `/@words/` are resolved in the pattern *text*, before the regex is compiled, by three interpreter functions:

```
// src/Interpreter.h
std::string interpRegexPattern(const std::string& in);  // $var atoms
std::string spliceRegexVars(const std::string& pat);    // regex-valued vars
std::string rxInterpArrays(const std::string& pat);     // /@arr/ → alternation
```

An array interpolates as a **longest-first literal alternation**, which is the semantics Raku specifies and which the sorting has to implement explicitly. A regex-valued variable is spliced in as source text at construction, so `rx/ $x /` where `$x` is a `Regex` composes.

A `$var` that must be read *at match time* — because the variable changes during the match — is a different node, `K::VarMatch`, evaluated through the hooks in the next chapter.

19.10 What the compiler produces

The result of construction is a root node plus a little bookkeeping:

```
// src/Regex.h
int ncaps_;                                        // positional capture count
std::set<int> listCaps_;                           // indices under a quantifier
std::shared_ptr<std::set<std::string>> listNames_; // named keys under one
std::shared_ptr<std::set<std::string>> hashNames_; // %<name>= keys
```

The two “list” sets exist because Raku’s capture arity is decided by the *pattern*, not by the match. A capture under a repetition quantifier is an array **even when it matched once or not at all**. That fact is a property of the compiled pattern, so it is computed once by `collectListNames` walking each quantified atom, and then shared — as a `shared_ptr` to a frozen set — into every match result the pattern produces. Sharing rather than copying matters because a grammar parse produces one of these per rule invocation.

Chapter 20

The Backtracking Matcher

The matcher is continuation-passing. One function walks the node tree:

```
// src/Regex.h
bool matchNode(const Node* n, MState& st, long pos, const FnRef& k) const;
```

"Match node `n` at position `pos`; if it succeeds, call `k` with the position after it; return whether the whole thing ultimately succeeded."

That signature is the entire design. Backtracking is the C++ stack unwinding when a continuation returns `false`, and every construct that has alternatives simply tries them in order:

```
// src/Regex.cpp
case K::Seq: {
    // match kids[0], and in its continuation match the rest
}
case K::Alt: {
    // try each branch in ranked order; the first whose continuation
    // succeeds wins
}
case K::Rep: {
    // greedy: try one more repetition first, then k(pos)
    // frugal: try k(pos) first, then one more repetition
}
```

There is no explicit backtracking stack, no thread list, no bytecode. The continuation *is* "the rest of the pattern", and the C++ frame that holds it *is* the choice point.

20.1 FnRef: a continuation that does not allocate

A `std::function` for the continuation would heap-allocate on every node visit, which for a matcher is fatal. Continuations here live only for the duration of the match call chain — they are never stored — so the callable can be borrowed from the caller's stack:

```
// src/Regex.h
struct FnRef {
    void* ctx;
    bool (*fn)(void*, long);
    template <class F, class = std::enable_if_t<
                  !std::is_same_v<std::decay_t<F>, FnRef>>>
    FnRef(F&& f)
      : ctx((void*)&f),
        fn([](void* c, long v) {
              return (*(std::remove_reference_t<F>*)c)(v); }) {}
    bool operator()(long v) const { return fn(ctx, v); }
};
```

Two words, no heap, one indirect call. The safety argument is the lifetime rule stated in the comment: continuations are never stored, so the borrowed lambda always outlives the call. Break that rule — store an `FnRef` anywhere — and the result is a dangling pointer.

20.2 The step budget

Continuation-passing plus backtracking means a pathological pattern can recurse without bound. `/[a*]* b/` against a long non-matching string is the classic.

```
// src/Regex.cpp — the top of matchNode
if (++st.steps > 8000000) throw StepLimitExceeded{};
```

Eight million steps is far beyond any real match and trips in well under a second on the pathological cases. The exception is caught at each match entry point and reported as a failure to match:

```
// src/Regex.h
struct StepLimitExceeded {};
```

The budget is carried *across* the start positions of an unanchored search, so a quadratic scan cannot escape it by restarting.

This is a blunt instrument and named as one: it prevents a hang, it does not prevent a pathological pattern from being slow, and a legitimate match that genuinely needs more than eight million steps would be reported as a non-match. No such match has appeared.

20.3 `MState`: everything one match frame knows

```
// src/Regex.h — MState, abridged
struct MState {
    const std::string& s;
    std::vector<std::pair<long, long>> caps;                 // $0, $1, …
    std::map<std::string, std::pair<long, long>> named;      // $<x>
    std::map<std::string, std::vector<ParseNode>> children;  // subrule trees
    const SubResolver* resolver = nullptr;   // plain-regex subrule path
    class GrammarMatcher* grammar = nullptr; // grammar path
    const std::set<std::string>* lexNames = nullptr;  // my regex NAME shadows
    std::map<int, std::vector<std::pair<long, long>>> capReps;
    long startPos = 0;
    long capFrom = -1, capTo = -1;           // the <( … )> span
    const GrammarHooks* hooks = nullptr;     // interpreter callbacks
    const std::string* curSym = nullptr;     // the proto candidate's :sym<…>
    long firstCode = -1;                     // where the LTM prefix ends
    long litPrefix = -1;                     // leading literal run
    long steps = 0;
};
```

Captures are **byte spans**, not strings. Nothing is copied out of the subject until a `Match` object is built, which keeps backtracking cheap: undoing a capture is restoring a pair of integers.

`capFrom` and `capTo` implement `<( … )>`, which narrows what the *overall match* reports while matching continues outside it.

20.4 Two subrule paths

A `<name>` call inside a pattern resolves one of two ways, and the difference is fundamental.

The grammar path threads the continuation *through* the callee:

```
// src/Regex.cpp — case K::Subrule
if (st.grammar) {
    if (!n->metaCache) n->metaCache = &st.grammar->nameMeta(n->ruleName);
    return st.grammar->matchSubMeta(*n->metaCache, n->ruleName, n->ruleArgs,
                                    ..., st, pos, k, ...);
}
```

Because `k` crosses the call, `<a> <b>` can backtrack **into** `<a>` when `<b>` fails. That is what Raku's grammars require and what most regex engines with "subroutine calls" do not provide.

The plain-regex path uses a `SubResolver`, which is atomic: it matches the named rule at a position and answers with a span. No continuation crosses, so no backtracking into the callee.

Between them sit the built-in rules — `<alpha>`, `<digit>`, `<ident>`, `<ws>` and the rest — resolved directly by `builtinRuleMatch`, with one carve-out:

```
// src/Regex.cpp
if (!(st.lexNames && st.lexNames->count(n->ruleName))) { ... }
```

A lexical `my regex ident { ... }` shadows the built-in of that name. The check is a set lookup on a set that is usually null.

The `metaCache` field is a per-node cache of the name resolution, so the hot path never re-resolves a rule name through a string-keyed map. It is safe because compiled regexes live in the matcher's own cache, so the node and the matcher share a lifetime.

20.5 Lookarounds

A zero-width assertion matches its inner pattern in an **isolated capture state**, so captures inside a lookahead do not leak:

```
// src/Regex.cpp — case K::Look
MState sub{st.s, std::vector<std::pair<long,long>>(ncaps_, {-1,-1}),
           {}, {}, st.resolver, st.grammar};
sub.hooks = st.hooks; sub.startPos = pos;
m = matchNode(child, sub, pos, [](long) { return true; });
...
return (m != n->negate) ? k(pos) : false;
```

The hooks are propagated into the sub-state, so an embedded code block or `$var` inside a lookahead still evaluates against the interpreter's live scope. The final line is the whole of positive-versus-negative: `negate` flips the sense, and either way the position does not advance.

20.6 The hooks: running Raku during a match

A Raku regex can contain executable code — `{ … }` side effects, `<?{ … }>` assertions, `:my` declarations, `$var` atoms, and `** { … }` quantifier bounds. The matcher cannot evaluate Raku, so it calls back:

```
// src/Regex.h — GrammarHooks, abridged
std::function<bool(const std::string&)> hasAction;
std::function<void(const ParseNode&)> onRule;
std::function<bool(const std::string&, long, long,
                   const NamedMap&, const ParamMap&)> assertPass;   // <?{…}>
std::function<void(…)> run;        // :my / {…}
std::function<void(…)> runCaps;    // …with positional captures, so $0 works
std::function<std::string(…)> str;                  // a $var atom
std::function<std::pair<long,long>(…)> range;       // ** { … }
std::function<std::shared_ptr<void>()> saveState;
std::function<void(std::shared_ptr<void>)> restoreState;
std::function<bool(const std::string&, std::string&, std::string&)> namedRule;
```

Every hook is optional. A plain `Regex` with none of them set treats code constructs leniently rather than failing, which is what lets the same engine serve a simple `~~ /…/` and a full grammar parse.

The current named and positional captures are passed *into* the hook, so the block's `$/` can offer `$<x>` and `$0` — the assertion in `/ (\d+) <?{ $0 > 10 }> /` needs them, and they exist only inside the matcher.

`saveState` and `restoreState` exist for one specific purpose: the longest-token ranker in Chapter 22 measures branch lengths by *probing*, and a probe must not leave the interpreter's `:my` variables and deferred `makes` behind. They snapshot and roll back that state around a measurement pass.

20.7 When side effects fire

This is the subtlest semantic decision in the engine, and it differs between the two paths.

On the grammar path, blocks fire eagerly. A completed subrule fires its action method immediately, and a later backtrack does **not** unfire it. That sounds wrong, and it is what Rakudo does: `HTTP::Header` sets header fields from the actions of a parse whose `TOP` ultimately fails on a missing trailing newline, and expects the fields to be there. A later `<?{ ... }>` assertion also needs to see what an earlier `{ ... }` set.

On the plain-regex path, bare blocks are queued. The matcher is continuation-passing and backtracking, so a block sitting on a branch that is later abandoned would otherwise fire anyway — and fire again on every retry. So the plain entry points queue them, unwind the queue when a branch fails, and run them in order only once the overall match is accepted.

The gating flag is why: only the plain-regex entry points turn queuing on and drain it. The grammar path wants eager.

`hasAction` exists so that rules with no action method cost nothing. Assembling a `ParseNode` for a completed rule is not free, so the matcher asks first whether anyone is listening.

20.8 Entry points

```
// src/Regex.h
bool search(const std::string& subject, long startPos, RxMatch& out) const;
bool matchAt(const std::string& subject, long pos, RxMatch& out,
             const SubResolver& r, const std::set<std::string>* lexNames) const;
std::vector<RxMatch> searchExhaustive(const std::string& subject,
             const SubResolver& r, const std::set<std::string>* lexNames) const;
```

`search` is the unanchored scan: try each start position in turn. `matchAt` is anchored, used by grammar subrule calls. `searchExhaustive` implements `:exhaustive` — every match at every start position and every length — which is the only mode that does not stop at the first success.

The result is a `RxMatch`: spans, capture arrays, named spans, the child `ParseNode` trees for subrules, and the shared frozen sets that record which captures are list-valued. The interpreter turns that into a Raku `Match` object, which is a `VT::Match` `Value` carrying the subject in `s`, the span in `rFrom`/`rTo`, positional captures in `arr` and named ones in `hash` — the canonical example of a `Value` with four fields live at once (Chapter 7).

20.9 Substitution

`s///`, `.subst` and `tr///` share one entry point on the interpreter, because the occurrence-selection adverbs are the complicated part:

```
// src/Interpreter.h
std::string substSelect(const std::string& subj, const std::string& pat,
                        Value* replArg, ValueList& args, long& nsub,
                        bool literal = false,
                        const std::string* tmplRepl = nullptr,
                        Value* matchResult = nullptr);
Value substApply(Value* target, const std::string& pattern,
                 const std::string& repl, bool nonMut);
```

`substSelect` handles `:g`, `:x`, `:nth`, `:p`, `:c` and the `same`-family adverbs (`:samecase`, `:samespace`, `:samemark`), which adjust the replacement to match the case, spacing or marks of what it replaced.

`substApply` is the single call the compiled backend emits, so a compiled `s///` and an interpreted one run the same code — including the `readonly` check that makes `s///` on a bound parameter die.

Chapter 21

Grammars

A Raku grammar is a class whose methods can be regexes. That is not a simplification for the book — it is how it is implemented. `grammar G { ... }` produces a `ClassInfo` with `isGrammar` set, its rules in a string map beside its methods:

```
// src/Value.h — ClassInfo, the grammar fields
std::map<std::string, std::string> rules;      // name → pattern text
std::vector<std::string> ruleOrder;            // DECLARATION order
std::map<std::string, std::string> ruleKind;   // token / rule / regex
std::map<std::string, std::vector<std::string>> ruleParams;
bool isGrammar = false;
```

Inheritance works because `findRule` walks the parent chain exactly as `findMethod` does, so `grammar Mine is Base` overrides individual rules.

The pattern text is stored **unparsed**, as the lexer captured it (Chapter 3). Compilation happens lazily, per rule, on first use.

21.1 The grammar matcher

```
// src/Regex.h
class GrammarMatcher {
public:
    struct Rule { std::string pattern, kind; std::vector<std::string> params; };
    std::map<std::string, Rule> rules;
    std::map<std::string, std::vector<std::string>> protos;
    GrammarHooks hooks;
```

```
    bool parse(const std::string& input, const std::string& top, bool subparse,
               ParseNode& out, long& endOut);
    bool matchSub(const std::string& name, const std::string& args,
                  const std::string& capKey, Regex::MState& st, long pos,
                  const FnRef& k);
private:
    std::unordered_map<uint64_t, MemoEntry> memo_;
    std::unordered_map<std::string, NameMeta> nameMeta_;
    std::map<std::string, std::unique_ptr<Regex>> cache_;
    std::vector<std::map<std::string, std::string>> scope_;
};
```

A `GrammarMatcher` is built **per parse**. That single fact removes a whole category of problems: every cache in it — compiled rules, name metadata, the packrat memo, the longest-token automata — dies with the parse, so none of them needs invalidation logic.

`Interpreter::grammarParse` builds one, fills `rules` and `protos` from the `ClassInfo`, wires the hooks to the interpreter, and runs it.

21.2 A subrule call, in full

`<name>` inside a rule reaches `matchSubMeta`, which is where most of the engine's cleverness lives. In order:

1. **Resolve the name** — or rather, read the already-resolved metadata, which the calling node cached (Chapter 20).
2. **Check the memo**, if this rule is ratcheting.
3. **Compile the rule body**, if this is its first use, and cache it.
4. **Bind parameters**, if the call was `<r($x, 'lit')>`.
5. **Match**, threading the caller's continuation through.
6. **Record a `ParseNode`** in the parent's capture frame.
7. **Fire the action method**, if the grammar's action object has one.

21.2.1 The per-name metadata

```
// src/Regex.h
struct GrammarRuleMeta {
    bool ratchet = false; int id = 0;
    Regex* singleChar = nullptr;   // body is one char matcher: inline it
```

```
    const void* rule = nullptr;
    Regex* noArg = nullptr;        // pre-compiled parameterless body
    const std::vector<std::string>* proto = nullptr;
    bool isWs = false;
    bool scoped = false;           // body declares `:my`
    bool dynDep = false;           // ...or reads a dynamic var: do not memoise
    std::string builtinClass;      // unknown name → built-in class flags
    mutable std::shared_ptr<LtmNfa> protoNfa;
    mutable bool protoNfaTried = false;
};
```

Computed once per name and reached through a pointer cached on the calling AST node, so the hot path never touches a string-keyed map.

Two flags on it encode a real correctness rule. `scoped` means the rule's body declares `:my` variables, so the interpreter's scope must be saved and restored around the call. `dynDep` means the body declares `:my` **or reads a dynamic variable** — and therefore its match can depend on caller state that is not part of the memo key, so **it must not be memoised**.

That second one is the kind of condition that is obvious in hindsight and invisible in advance. A memo keyed on (rule, position) is only sound if the rule is a function of (rule, position).

21.2.2 The packrat memo

A ratcheting rule does not backtrack, so its first complete match at a position *is* the match. Caching it collapses the exponential re-descent that recursive longest-token probing would otherwise cause:

```
// src/Regex.h
struct MemoEntry {
    bool matched = false;
    long end = 0;
    long declEnd = 0;      // where the declarative prefix ended
    long litPrefix = 0;    // leading literal run, for the LTM tie-break
    long capFrom = -1, capTo = -1;
    std::vector<std::pair<long, long>> caps;
    std::map<std::string, std::pair<long, long>> named;
    std::shared_ptr<const ChildMap> kids;      // frozen: replays share
    std::shared_ptr<const std::set<std::string>> listNames;
    std::shared_ptr<const std::set<int>> listCaps;
    std::shared_ptr<const std::map<int,
```

```
        std::vector<std::pair<long,long>>>> capReps;
};
std::unordered_map<uint64_t, MemoEntry> memo_;
```

The key is a 64-bit integer built from the rule id, the position and the bound parameters — not a string, because a string key would allocate on every subrule call.

The child map is stored **frozen behind a `shared_ptr`**, which is the detail that makes the memo pay. Replaying a memoised match is then a refcount bump rather than a subtree copy, and a deeply nested grammar replays large subtrees constantly.

21.3 `ParseNode`: the tree the matcher records

```
// src/Regex.h
struct ParseNode {
    std::string name;
    std::string actualRule;    // for a proto: the winning name:sym<...>
    long from = 0, to = 0;
    std::vector<std::pair<long, long>> caps;
    std::map<std::string, std::pair<long, long>> named;
    std::shared_ptr<const ChildMap> kids;     // null = leaf
    std::shared_ptr<const std::set<std::string>> listNames;
    std::shared_ptr<const std::set<int>> listCaps;
    std::shared_ptr<const std::map<int,
        std::vector<std::pair<long,long>>>> capReps;
};
using ChildMap = std::map<std::string, std::vector<ParseNode>>;
```

The child map is a map to a **vector**, so repeated captures of the same name collate into a list — which is Raku's rule for `<pair>*`.

Except that the rule is stronger than "collate when repeated". A capture under a repetition quantifier is list-valued **even when it matched once or not at all**, so the arity has to come from the *pattern* rather than from the match. That is what `listNames`, `listCaps` and `capReps` carry, shared from the compiled `Regex` rather than rebuilt per node.

The interpreter walks the finished tree and builds `Match` values from it, with `$<name>` reading the child map and `$0` reading the positional spans.

21.4 Actions

An action object is an ordinary Raku object whose methods are named after the grammar's rules. When a rule completes, its method runs and can call `make` to attach a value to the match.

The engine fires actions **during** the match, through the hooks:

```
// src/Regex.h — GrammarHooks
std::function<bool(const std::string&)> hasAction;
std::function<void(const ParseNode&)> onRule;
```

`hasAction` gates the node assembly, so a rule with no corresponding method costs nothing. `onRule` fires the method with the completed node.

Three properties of that design, all deliberate:

- **A later backtrack does not unfire an action.** This matches Rakudo; the reasoning is in Chapter 20.
- **A memo replay reuses the first firing** rather than firing again, so packrat caching does not multiply side effects.
- **`make` writes into a target on the execution context** — `ExecContext::makeTargets` — which is how the value attaches to the right match rather than to a global.

21.5 Parameterised rules

`<r($x, 'lit')>` binds arguments to the rule's declared parameter names:

```
// src/Regex.h
std::vector<std::map<std::string, std::string>> scope_;
const std::map<std::string, std::string>& currentParams() const;
Regex* compiled(const std::string& name, const std::string& argstr,
                std::map<std::string, std::string>& boundOut);
std::string interpParams(const std::string& pat,
                         const std::map<std::string, std::string>& sc) const;
```

Bindings are strings, and they are interpolated into the pattern *text* before compilation. So `token line($indent) { $indent \N* }` compiles a different `Regex` for each distinct indent value — and the compiled-rule cache is keyed on name plus argument values for exactly that reason.

That design has a clear cost (a compilation per distinct argument tuple) and a clear benefit: parameterised rules need no support at all in the matcher. It is also what makes an off-side-rule parser — a Python tokenizer written in a Raku grammar — possible, and writing one is how the interpreter's `:my` dynamic-variable restore bug was found.

`currentParams` merges outer dynamic-variable parameters into the current frame's, so a nested rule can see an enclosing rule's parameter.

21.6 Protoregexes

```
proto token statement {*}
token statement:sym<if>    { <sym> \s+ <condition> <block> }
token statement:sym<while> { <sym> \s+ <condition> <block> }
```

A `proto` is a dispatch group. Its candidates are collected by name:

```
// src/Regex.h
std::map<std::string, std::vector<std::string>> protos;
```

and `<sym>` inside a candidate matches that candidate's own `:sym<…>` literal — which is why `MState` carries a `curSym` pointer.

Candidate selection is **longest-token matching**, not declaration order, and that is the next chapter.

`ParseNode::actualRule` records which candidate won, so `$<statement>.made` can tell an `if` from a `while`.

21.7 Entry points and start rules

```
bool parse(const std::string& input, const std::string& top, bool subparse,
           ParseNode& out, long& endOut);
```

`.parse` requires the whole input to be consumed; `.subparse` does not. `:rule<name>` picks a start rule other than `TOP`. Both go through the same function with a flag, so there is one implementation of the anchoring rules.

21.8 Where grammars appear in this book again

Grammars are the one construct the native code generator refuses (Chapter 25), so a program containing one is compiled by bundling rather than transpiling. They are also the heaviest exercise in the test suite: the showcase interpreters for JavaScript, Perl 5, Python 3 and Lisp are grammar-driven, and each of them, when first written, produced a list of genuine bugs in this engine — the `:my` restore, a subrule-alias mis-parse, `|` versus `||` ranking, and an optional-block state corruption among them.

Chapter 22

Longest-Token Matching

Raku's `|` alternation does not try its branches left to right. Each branch is ranked by how far its **declarative prefix** can reach into the input — the leading part of the pattern made of literals, character classes and quantifiers, up to the first procedural construct. The branch with the longest reachable prefix is committed first; ties break by more-literal-first, then by declaration order. `||` is the opt-out: sequential first match, no ranking.

```
say ("abcd" ~~ / [ ab { } cd ] | abc /).Str;   # abc
```

The first branch *could* match four characters, but its declarative prefix ends at the code block, at length 2. `abc`'s prefix is 3, so `abc` wins.

Two properties of that example matter and pull in opposite directions. The ranking is by **prefix** length, not by greedy match length. And ranking must run **no user code** — the `{ }` above must not fire for a branch that loses.

22.1 Two rankers

The probe (the default) runs each branch once for its greedy full-match end, snapshotting and rolling back side effects through the `saveState`/`restoreState` hooks, and commits branches in longest-end-first order.

It is cheap and it is wrong twice: it ranks by the wrong length, and it descends into user-code paths that true longest-token matching never visits.

The NFA (`RAKUPP_LTM=1`) builds a Thompson automaton per alternation, lazily, from the compiled node tree, and answers "how far could each branch's declarative prefix reach?" in one linear scan of the input. No code execution, no backtracking. The commit phase then runs the real engine on the ranked branches, so captures, assertions and side effects behave exactly as before.

An automaton cannot execute code. That is not a limitation here — it is precisely what makes it the structurally correct ranking oracle.

22.2 What ends a declarative prefix

Two very different reasons a prefix stops, and the distinction is the whole correctness argument.

Spec enders — the prefix genuinely ends here, and ranking at this point is *correct*: a bare `{...}` code block, a back-reference (`$0`, `$<name>`), a `||` tail (the first branch's prefix continues, per the specification), runtime quantifier bounds `** {$n}`, and `&` conjunction.

Model gaps — the *builder* could not model the construct, so ranking might unfairly demote this branch: an unexpandable subrule, Unicode-property and grapheme-cluster classes, `:m` on a character *class*, non-ASCII `:i` folding, lookarounds, and a blown build bound (200 depth or 4,000 states).

Zero-width assertions are **transparent**: anchors, word boundaries, `:my` declarations, and the code assertions `<?{...}>` / `<!{...}>` are epsilon transitions for ranking purposes. Roast checks this — `a <?{ 1 }> .+ | aa` against `"aaa"` matches `aa`, so the assertion does not end the prefix.

Over-permissiveness is safe: the commit engine enforces assertions for real, and a branch whose assertion fails simply fails at commit and hands over to the next ranked branch.

22.3 The gap-aware hybrid contract

`RAKUPP_LTM=1` must never be *less* correct than the default. So:

> The NFA decides an alternation only when **every** branch's prefix ended for a spec reason. If any branch hit a model gap, that alternation falls back to the probe.

```
// src/LtmNfa.h
bool anyModelGap() const { return anyGap_; }
```

Each expansion route closed converts more alternations from probe fallback to NFA ranking. A branch whose prefix cannot reach any accept state on the given input is not a candidate at all — the commit never tries it, matching Rakudo's own pruning.

22.4 The automaton

```
// src/LtmNfa.h
struct Pred {
    char kind = 'A';   // 'L' literal cp, 'C' class node, 'A' any,
                       // 'S' whitespace (<ws>), 'F' builtin flag class
    const void* node = nullptr;
    uint32_t lit = 0;
    bool icase = false;
};
struct State {
    std::vector<std::pair<int,int>> eps;
    std::vector<std::pair<int,int>> edges;
    int acceptBranch = -1;
    int litDepth = 0;
};
```

Transitions are codepoint predicates that **reuse the `Class` node's own match data** — byte ranges, codepoint ranges, negation, folding. There is no second Unicode implementation, which is both less code and structurally safer: the ranker cannot disagree with the matcher about what a class contains.

The `Pred` borrows the `Regex`'s `Node`, relying on the regex outliving its cached automaton — the same lifetime the byteset cache already assumes.

Every branch gets its **own** entry epsilon-edge from state 0. Sharing entry states once let a sibling branch's `a*` loop re-anchor other branches, which produced ranks that were not merely wrong but incoherent.

22.5 The cache, and an address-reuse bug

The automaton is cached **on the `Alt` node**:

```
// src/Regex.h — Node
mutable std::unique_ptr<LtmNfa> ltmNfa;
```

The first implementation used a side map keyed on `Node*`, which is the obvious thing and was wrong. A process compiles many regexes, freed node addresses get recycled by the allocator, and the map handed out an automaton built for a completely different pattern. The symptom, found by the phase-1 harness on a Roast alternation file, was empty or nonsense ranks.

That is the exact opposite of the flip-flop state map in Chapter 17, which *is* keyed on a node address and *is* correct — because AST nodes are never freed while regex nodes are. The rule to extract: **a pointer is only a valid key when its target's lifetime is at least the map's.**

For expanded subrules there is a second staleness axis. Named regexes register last-wins, so a re-declared token changes what the same compiled node resolves to. The stored pattern text doubles as a staleness stamp:

```
// src/LtmNfa.h
bool stillValid(const GrammarHooks* hooks) const;
std::map<std::string, std::string> expandStamps_;   // name → pattern text used
```

22.6 Subrule expansion: three routes

A `<name>` inside a branch resolves through an expansion context:

```
// src/LtmNfa.h
struct LtmExpand {
    const GrammarHooks* hooks = nullptr;
    std::function<int(const std::string& name, const void*& regexOut,
                      char& flagOut)> grammar;
};
```

Lexical. Interpreter-registered `my token/rule/regex` bodies, handed over as text plus the match path's exact flag spelling (`rule` becomes `sr`, `token` becomes `r`). The automaton compiles and **owns** the callee, inlining its prefix recursively. Recursion is treated as a spec prefix end.

Grammar. Already-compiled rule bodies from the grammar's own table, so nothing is recompiled and the automaton borrows the `Regex`'s nodes. The resolver answers with a small integer: refuse, here is a compiled body, this is `<ws>`, or this is a single-

character built-in class. Protos, parameterised rules and dynamically-dependent bodies refuse, which becomes a model gap and therefore a probe fallback.

<sym> inlines as the current candidate's :sym<...> literal, for proto dispatch.

<ws> is modelled as \s* for ranking. That is deliberately approximate — the commit engine enforces the real <!ww> word-boundary gate — and it is ranking-grade rather than semantics-grade, which is all the ranker needs.

22.6.1 The composed character class

A class written <[\-+.] +uri_alpha +digit> is parsed into a synthesised first-match Alt. Semantically it is a **one-character union**, so the builder must union its members:

```
// src/Regex.h — Node
bool classCombo = false;   // a SYNTHESIZED Alt for a composed char class
```

Without the tag those nodes were modelled like a user ||, which takes only the first member. That under-matched the prefix and pruned whole branches — the symptom was a URI grammar failing on its scheme rule.

22.7 Proto dispatch

Protoregex candidates are ranked by the same machinery:

```
// src/LtmNfa.h
static std::unique_ptr<LtmNfa> buildForBranches(
    const std::vector<const void*>& regexes,
    const std::vector<std::string>& syms, const LtmExpand* ctx);
```

One union automaton over all candidates' bodies, one branch per candidate, with <sym> inlined per branch. It is cached on the GrammarRuleMeta, and since the matcher is per-parse, that cache dies with the parse and needs no staleness management at all. Any model gap in any candidate discards the automaton and the probe dispatch stands.

22.8 The tie-break must be per-path

Equal prefix length breaks by "more literal wins", and that count must be computed **per path during the scan**, not stored per state.

The builder originally kept a static literal depth on each state, max-merged at join states. In:

```
token TOP { <foo> | <bar> }
token foo { \w\w }
token bar { aa | <foo> }
```

on input `"bb"`, `bar`'s dead `aa` path — two literals — leaked its count onto the join state that `bar`'s live `<foo>` path — zero literals — also reaches. So `bar` stole the tie from the earlier-declared `foo`.

`rank()` now carries, per live state, the length of the leading literal run along the path that reached it, frozen at the first non-literal edge; accepts record the *path's* value, not the state's.

```
// src/LtmNfa.h
struct Ranked { int branch; long prefixEnd; long litPrefix; };
std::vector<Ranked> rank(const std::string& s, long pos) const;
```

sorted by prefix end descending, then literal prefix descending, then declaration order — the specification's tie-break chain.

22.9 Oracle notes

Three behaviours confirmed against Rakudo while building this, each of which would otherwise have been guessed wrong:

- **`multi token` protos do not rank.** Rakudo ranks proto candidates by declarative prefix only when they are declared `token t:sym<x>`; `multi token t:sym<x>` candidates dispatch in declaration order, and so does a direct `.subparse(:rule<proto>)`. Both rankers here rank `multi token` candidates too — a pre-existing divergence, deferred rather than hidden.
- **`||` continues through its first branch.** `a || b`'s declarative prefix is `a`'s prefix: not empty, and not both.
- **Code assertions are transparent** to ranking, as above.

22.10 Debugging and measured results

Variable	Effect
`RAKUPP_LTM=1`	use the NFA ranker where it is gap-free
`RAKUPP_LTM_DEBUG=1`	rank both ways and print disagreements
`RAKUPP_LTM_RANKDUMP=1`	print each NFA-decided alternation's ranking

The debug flag works *without* the feature flag, and it was the phase-1 harness: before anything consulted the automaton's answer, the probe path computed both rankings and printed every disagreement with its pattern context, for classification against Rakudo. It is still the fastest way to classify a suspected ranking bug.

Same binary, same machine, full Roast:

setting	assertions	`longest-alternative.t`	`proto-token-ltm.t`
default (probe)	197,116	45/62	10/10
`RAKUPP_LTM=1`	197,117	47/62	10/10

The flag's failure set on the alternation file is a strict subset of the default's. A grammar benchmark — twenty compiles of a JSON grammar corpus — runs in 28 to 34 milliseconds in both settings, so automaton construction is fully amortised by the node cache. The twenty-eight grammar showcase runs are byte-identical across settings.

The plan keeps the flag available for one release after the default changes, which is the general policy for a switch that changes behaviour rather than speed.

Part VI

Unicode

Chapter 23

Graphemes, Normalization, Collation

Raku specifies strings at the level of **graphemes** — what a reader would call a character — not codepoints and certainly not bytes. `"e\c[COMBINING ACUTE ACCENT]".chars` is 1. Sorting respects the Unicode Collation Algorithm. `"\c[LATIN SMALL LETTER E WITH ACUTE]".uniname` answers a real name from the character database.

None of that is optional, and Raku++ implements all of it from the pinned Unicode 17.0 data with no external library.

23.1 Storage: UTF-8 bytes, grapheme semantics

A `Str` holds UTF-8 bytes in its `CowStr`. Raku's *indices* are grapheme indices. Those two facts have to be reconciled on every string operation, and how that reconciliation is made cheap is the most consequential engineering decision in this part of the system.

Strings are normalised to **NFC** on the way in — Raku's NFG storage model — so a literal is normalised once, in place, on first evaluation:

```
// src/Ast.h — StrLit
bool nfcDone = false;   // NFC-normalized in place on first eval
```

23.2 The fast answer: when a byte index is a grapheme index

Most strings in most programs are ASCII. For those, byte index, codepoint index and grapheme index are all the same number, and every conversion is a no-op — if you can establish it cheaply.

```
// src/BuiltinsShared.h
size_t asciiRun(const std::string& s, size_t limit);
bool allAscii(const std::string& s);
bool byteIsGraphemeIndex(const std::string& s);
bool cowAllAscii(const CowStr& s);
bool cowByteIsGraphemeIndex(const CowStr& s);
long long cowGraphemeCount(const CowStr& s);
```

The `cow*` forms are the memoised ones. They read the flags cached on the promoted string body (Chapter 8), so the answer is computed once per string rather than once per character examined.

The condition is narrow and exact: **all bytes below 0x80, and no carriage return.** CR LF is the one ASCII sequence that forms a single grapheme cluster under UAX #29, so an ASCII string containing a CR can still cluster.

Getting this wrong is not a correctness bug — it is a *complexity* bug. The scanning operations call these once per character examined, so the difference between memoised and not is the difference between a linear tokenizer and a quadratic one.

23.3 The slow answer: `GraphemeMap`

When a string can genuinely cluster, the translation between codepoint indices and grapheme indices is built explicitly:

```
// src/Unicode.h
class GraphemeMap {
public:
    explicit GraphemeMap(const std::vector<uint32_t>& cps);
    bool trivial() const { return starts_.empty(); }
    size_t count() const;                  // graphemes
    size_t cpAt(size_t g) const;           // codepoint index where g starts
    size_t graphemeAt(size_t cp) const;    // grapheme containing codepoint cp
private:
    size_t ncps_;
```

```
    std::vector<size_t> starts_;           // empty when 1 grapheme == 1 cp
};
```

The class exists because `substr`, `index` and `flip` each used to re-derive the translation — or, worse, skip it entirely and index codepoints, which is silently wrong for any text carrying a combining mark.

The common case costs one linear scan and no allocation: a string with no codepoint above U+02FF and no CR cannot cluster, so `starts_` stays empty and `trivial()` is true. Only a string that can actually cluster pays the full UAX #29 walk.

23.4 The subsystems

```
// src/Unicode.h — the interface, abridged
std::vector<uint32_t> uniNormalize(const std::vector<uint32_t>&, int mode);
size_t uniGraphemeCount(const std::vector<uint32_t>& cps);
std::vector<size_t> uniGraphemeStarts(const std::vector<uint32_t>& cps);
size_t uniClusterEndUtf8(const std::string& s, size_t pos, size_t len);
int uniCollate(const std::vector<uint32_t>& a, const std::vector<uint32_t>& b);
int32_t uniCharByName(const std::string& name);
std::string uniNameOf(uint32_t cp);
bool uniNumValue(uint32_t cp, long long& num, long long& den);
std::string uniGeneralCategory(uint32_t cp);
std::string uniScript(uint32_t cp);
bool uniMatchesProp(uint32_t cp, const std::string& prop);
uint32_t uniSimpleUpper(uint32_t cp);
std::vector<uint32_t> uniCaseMap(uint32_t cp, int kind); // 0lo 1up 2ti 3fold
std::string uniBlockOf(uint32_t cp);
int uniBinaryProp(uint32_t cp, const std::string& prop);
```

Normalization implements all four forms — NFD, NFC, NFKD, NFKC — over codepoint sequences: canonical decomposition, canonical ordering by combining class, and canonical composition.

Grapheme segmentation is UAX #29 extended grapheme clusters, which is more than “base plus combining marks”: it covers Hangul syllable composition, regional indicator pairs (flag emoji), emoji modifier sequences and zero-width joiner sequences, and the CR LF rule.

Collation is the Unicode Collation Algorithm against the default table, producing a three-way comparison. That is what `coll` and `.collate` use, and it is why sorting Raku strings is not `strcmp`.

Names go both ways: name to codepoint for `\c[NAME]`, codepoint to name for `.uniname`. This is the single largest file in the tree at 41,174 lines, and it is a binary search plus a hash lookup at run time.

Properties cover general category, script, block, numeric value, bidi class, mirroring, and the binary and enumerated properties. `uniMatchesProp` is what backs `<:Nd>` and `<:L>` inside a regex character class — which is why a regex class with a Unicode property in it is a `Class` node with a `uprop` string rather than a byteset (Chapter 19).

Case mapping has two tiers: simple 1:1 mappings from `UnicodeData`, and full 1:N mappings from `SpecialCasing` and `CaseFolding`. The second is not optional: uppercasing the German sharp s produces two characters, and Turkish dotted and dotless i are language-sensitive.

23.5 The tables are generated, and one generator is written in Raku

```
tools/ucd/                  pinned UCD + UCA 17.0 data files
tools/gen_unicode_*.py      table generators
tools/gen-unicode.raku      ...and the ones written in Raku
src/unicode_*_gen.cpp       the output: 79,400 lines
```

The tables are checked in rather than generated at build time, which keeps the build free of a data dependency and makes the Unicode version a reviewable part of the source tree.

Two of the generators — the names and one of the property tables — are written in **Raku and run by rakupp itself**. That is not a stunt. It is a load-bearing test: generating a 41,000-line table exercises string handling, hashing, sorting and file I/O at a scale no unit test reaches, and a bug in any of them shows up as a table that does not compile or does not match.

It is also the project's standing rule for tooling: build the ecosystem tools in Raku, run them with rakupp, and let the compiler eat its own output.

23.6 Where Unicode shows up elsewhere

In the lexer. Identifiers may contain Unicode letters and combining marks, so `consumeIdentChars` decodes codepoints rather than reading bytes. Superscript digit runs fold into a ** operator.

In the regex engine. A character class carries three parallel representations for three ranges of complexity: a 256-bit byteset for bytes, codepoint ranges for anything above 255, and a list of whole cluster strings for multi-codepoint grapheme members. `:i` folds through `uniCaseMap`; `:m` (ignoremark) compares NFD first-starter base codepoints and consumes the rest of the cluster.

In the longest-token automaton. Its transition predicates reuse the regex `Class` node's data rather than reimplementing any of this — the single most important structural decision in Chapter 22, because two Unicode implementations that disagreed would produce rankings that could not be debugged.

In collation-sensitive built-ins. `cmp`, `leg`, `unicmp`, `coll`, `.sort` and the ordering operators each pick their comparison from this layer.

23.7 Honest limitations

- **Language-sensitive casing is not exposed.** The full case mappings are implemented, but there is no locale parameter, so the Turkish `i` rules are not reachable.
- **Collation is the default table**, with no tailoring and no locale.
- **Normalization is by codepoint sequence**, so a caller that has bytes pays a decode and an encode around it.
- **The Unicode version is pinned in the source tree**, so updating it is a regeneration and a review rather than a dependency bump — deliberate, but it does mean the tables age.

Part VII

The Back Ends

Chapter 24

Four Ways to Run a Program

One front end, four back ends. This chapter takes one small program through all four and shows exactly what each produces.

```
# demo.raku
sub square($n) { $n * $n }
my $total = 0;
for 1..5 { $total += square($_) }
say $total;
```

The CLI in `main.cpp` picks a mode from the flags and calls one of four drivers.

24.1 Mode 1 — interpret

```
demo.raku ─► Lexer ─► Parser ─► Program (AST)
          ─► Interpreter.run(Program)  ─► walk the tree, node by node
```

`main` reads the source and calls `rakuppRunBigStack`, which lexes, parses, and hands the `Program` to `Interpreter::run`. Evaluation is the recursive `eval`/`exec` of Chapter 12: the `for` loop iterates and re-executes its body block, `square($_)` looks the sub up in the environment chain and calls it, `$total += …` re-dispatches through `applyArith`.

Nothing is cached or compiled — the same AST nodes are re-interpreted on every iteration. Startup is about two milliseconds. This is the only mode that runs every construct in the language, so it is the one the language is developed against.

24.2 Mode 2 — `--bundle`

```
demo.raku ──► (embedded verbatim as bytes) ──► generated stub.cpp
          ──► cc stub.cpp librakupp_rt.a ──► ./demo
```

The driver does **not** parse the program at build time. It emits a small stub that embeds the source as a byte array and calls the runtime:

```
static const unsigned char SRC[] = { 115,117,98,32,... };
int main(int argc, char** argv) {
    std::string src(reinterpret_cast<const char*>(SRC), SRC_LEN);
    /* … collect argv … */
    return rakupp::rakuppRunBigStack(src, args, "demo.raku", exe);
}
```

then compiles it and links against the runtime. The result is standalone — no `rakupp` needed on the target machine — but at run time it still lexes, parses and tree-walks. It *is* the interpreter in a box. The win is distribution and a roughly ten-millisecond start; run time equals interpreting.

24.3 Mode 3 — `--aot`

```
demo.raku ──► Lexer ──► Parser ──► Program        ← parsed at BUILD time
          ──► AstEmit.emitAstProgram(Program) ──► C++ that rebuilds the AST
          ──► cc gen.cpp librakupp_rt.a ──► ./demo
```

This is genuine ahead-of-time work: the program is **parsed at build time**, so parse errors are reported then rather than at run time, and `AstEmit` emits one small builder function per AST node:

```
static ExprPtr e0() { auto n = std::make_unique<IntLit>(1LL); return n; }
static StmtPtr s3() { auto n = std::make_unique<ExprStmt>(); n->e = e2();
                      return n; }
// … one builder per node …
int main(int argc, char** argv) {
    Program prog;
    prog.stmts.push_back(s3());  /* … */
    return rakupp::rakuppRunProgramBigStack(prog, args, "demo.raku", exe, "");
}
```

The generated `main` reconstructs the identical `Program` and hands it to `rakuppRun-Program`, which interprets it with **no lexing or parsing at run time**.

Because the interpreter runs the embedded tree, `--aot` handles the **whole language, grammars included**. If `AstEmit` ever meets a node it cannot rebuild, it throws `AstEmitError` and the driver falls back to mode 2.

The trade-off is the generated code size: one builder function per node means a few hundred lines of Raku become tens of thousands of lines of C++, and it builds about ten times slower than bundling. Since both tree-walk at the same speed, `--bundle` is the practical bundler; `--aot`'s only real edge is catching parse errors at build time.

24.4 Mode 4 — `--exe`

```
demo.raku ─► Lexer ─► Parser ─► Program
          ─► Codegen.transpileToCpp(Program) ─► C++ source
          ─► cc gen.cpp librakupp_rt.a ─► ./demo   (no interpreter inside)
```

`Codegen` walks the tree and emits C++ that **implements the program directly** — native control flow, native calls — calling the runtime only for `Value` operations. For `demo.raku`, abridged:

```
#include "Interpreter.h"
using namespace rakupp;
static Interpreter RT;
static Value v_stotal = Value::any();     // top-level `my $total`
static const BuiltinFn* __bfp0 = nullptr; // say — resolved once at startup

static Value u_square(ValueList __a) {
    Value v_sn = rtPos(__a, 0);
    return applyArith("*", v_sn, v_sn);
}
static void __rakupp_register() { __bfp0 = RT.builtinPtr("say"); }

int main(int argc, char** argv) {
    __rakupp_register();
    try {
        v_stotal = Value::integer(0LL);
        {                                          // for 1..5 — a real C++ loop
            long long __lo = Value::integer(1LL).toInt();
            long long __hi = Value::integer(5LL).toInt();
            for (long long __i = __lo; __i <= __hi; __i++) {
                Value v__t0 = Value::integer(__i);          // $_
                try { v_stotal = applyArith("+", v_stotal,
```

```
                                            u_square(ValueList{v__t0})); }
                catch (const NextEx&) { continue; }
                catch (const LastEx&) { break; }
            }
        }
        rtCallB(RT, __bfp0, "say", ValueList{v_stotal});
    } catch (const RakuError& e) { std::cerr << e.message << "\n"; return 1; }
    return 0;
}
```

The loop is a native `for`, the sub call is a direct C++ call, and only value semantics dip into the runtime. This is the only mode whose runtime performance differs, and the only one the optimizer touches.

24.5 Comparing them

	parse when	runs how	whole language	speed
interpret	run time	tree walk	yes	baseline
`--bundle`	run time	tree walk	yes	baseline
`--aot`	build time	tree walk	yes	baseline
`--exe`	build time	native C++	falls back	several times faster on hot loops

The important row is the last column of the third row: **three of the four modes run at the same speed**, because three of them are the same tree walk. Bundling and AOT buy distribution and build-time error reporting, not throughput.

24.6 The fallback that makes --exe total

`Codegen` throws on any construct it cannot transpile:

```
// src/Codegen.h
struct CodegenError { std::string msg; };
```

mainly grammars, and a handful of NativeCall shapes that need copy-back the generated call site cannot express. The driver catches it and **transparently bundles the whole program instead**.

So `--exe` never refuses a program. It native-compiles what it can and bundles the rest, always producing a correct binary. The user is told which happened.

24.7 What every mode shares

One runtime. Everything except the CLI is in `librakupp_rt.a`, so a feature implemented once behaves identically in all four. `Value` is the shared currency, `callBuiltin`/`callCallable` the shared calling convention, `rtIndexRef`/`rtAttrRef`/`rtArrayVal` the shared container operations, and `callNative` the shared FFI.

One big stack. Every entry point runs the program body on a thread with a large reserved stack, including a compiled binary's `main`, so the recursion budget matches across modes.

Modules are always interpreted. `Codegen` emits a call to `Interpreter::rtUse`, a thin mirror of the interpreter's `use` handling, which calls the same `loadModule`. Only the *main program* is compiled; a compiled binary still loads and interprets its modules (Chapter 29) — though it can carry their serialised ASTs inside itself, which is the next section.

24.8 Carrying modules inside a binary

A standalone binary that needs its modules' source files on the target machine is not very standalone. So the compiling modes resolve the `use` graph at build time and embed each module's serialised AST:

```
// src/Interpreter.h
struct BundledModule { std::string name, blob, finish, src; };
std::vector<BundledModule> collectModuleGraph(
    const Program& prog, const std::vector<std::string>& searchPath,
    std::set<std::string>* exportsOut = nullptr);
void rakuppRegisterModule(const std::string& name, const char* blob,
                          size_t blobLen, const std::string& finish);
```

Registration happens once at startup, and `loadModule` consults the embedded table before it looks at the disk.

A module that cannot be found, parsed or serialised is **simply left out** rather than failing the build — the binary falls back to loading it from disk, which is also what has to happen for anything only a running program can name (`require ::($x)`, a computed `use lib`).

`--bundle` needs one extra thing, and it is a subtle one. It parses the main program at *run* time, and that parse has to scan each `used` module for the operators it declares (Chapter 5) — otherwise a program using an imported operator stops parsing once the module tree is gone. So bundled binaries also carry the module **sources**:

```
// src/Parser.h
void rakuppRegisterModuleSource(const std::string& name,
                                const char* src, size_t len);
const std::string* rakuppEmbeddedModuleSource(const std::string& name);
```

`--exe` and `--aot` parse at build time and never need it.

24.9 A fifth target

The same runtime compiled to **WebAssembly** is Raku.js: the interpreter running in a browser tab, with the same `Value` semantics and no server. It is mode 1 with a different host, and the two places it changes the engineering are noted where they arise — C++ exceptions are very expensive there, which is part of why control flow is cooperative (Chapter 14), and there is no shared library to `dlopen`, so NativeCall always takes its fallback path (Chapter 31).

Chapter 25

The Code Generator

--exe transpiles the AST into a self-contained C++ program. The interface is one function:

```
// src/Codegen.h
std::string transpileToCpp(Program& prog, bool optimize = false,
                           const std::string& srcPath = "",
                           const std::set<std::string>& moduleExports = {});
struct CodegenError { std::string msg; };
```

The design principle is stated in one sentence and everything follows from it: **the generator never reimplements the runtime.** It emits C++ that calls the same functions the interpreter calls. That is why the two produce byte-identical output, and why a feature implemented once works in both.

25.1 The mapping

Raku	Generated C++
`$a + $b`, `$a eqv $b`	`applyArith("op", a, b)`, or an inline `rt*` helper
`say`, any named builtin	`rtCallB(RT, __bfpN, "say", ...)`
`.map`, `.sort`, any method	`RT.methodCall(inv, "map", ...)`
a block, `-> $x {...}`, `* + 1`	`Value::closure([=](ValueList& a){ ... })`

Raku	Generated C++
`@a[i]` / `%h{k}` read, write	`rtIndexGet(...)` / `rtIndexRef(...)`
`[+] ...`	`rtReduce("+", ...)`
`class`	register a `ClassInfo` at startup; methods become closures
`$!x`	`rtAttrGet` / `rtAttrRef`
`multi`	one C++ function per candidate, plus a `rtTypeMatch` dispatcher
`enum`	global `Value::enumVal` constants
`gather/take`	push a collector onto the execution context
phasers, `CATCH`	reordered emission, a C++ `try` and a `when` chain
`use Foo`	`RT.rtUse("Foo")` — the module is still interpreted

The `rt*` family in `Interpreter.h` is the generator's vocabulary — about sixty functions, each the runtime's implementation of one construct:

```
// src/Interpreter.h — a sample
Value  rtIndexGet(const Value& base, const Value& key, bool isHash);
Value& rtIndexRef(Value& base, const Value& key, bool isHash);
Value  rtArrayVal(const Value& v);       // list-assignment semantics
Value  rtSlipVal(const Value& v);        // |x as a list element
Value  rtHashLit(const ValueList& items);
Value  rtNamedPair(const std::string& k, Value v);
Value  rtReduce(const std::string& op, const Value& list);
Value  rtTypedDefault(const char* type, char sigil);
ValueList rtMainArgs(const std::vector<std::string>& argv);
```

Several of them exist *only* because the interpreter had the same logic inline and the two would otherwise have drifted. `rtArrayVal` is the compiled twin of `coerceArray`, with the same fresh-buffer rule (Chapter 11), and it was factored out precisely so there is one definition of what `@a = expr` means.

25.2 What is emitted

A generated file has five sections.

A prologue including `Interpreter.h` and defining one static `Interpreter RT` — the runtime object that owns the builtin table, the class registry and the method dispatcher.

Static globals for top-level variables. `my $total` becomes `static Value v_stotal;`, with the sigil encoded into the name (`$` becomes `s`) so that `$x` and `@x` do not collide.

Functions, one per user sub. Their parameters arrive as a `ValueList`, and positionals are pulled out by index:

```
static Value u_square(ValueList __a) {
    Value v_sn = rtPos(__a, 0);
    return applyArith("*", v_sn, v_sn);
}
```

A registration function, `__rakupp_register`, run before `main`'s body. It resolves cached builtin pointers, registers `ClassInfos` for the program's classes, installs named regexes, and registers embedded modules.

`main`, which sets up `@*ARGS`, runs the registration, and executes the mainline inside a `try` with the phaser and exit handling around it.

25.3 Control flow

Raku control flow becomes C++ control flow wherever it can.

A `for` over a literal range becomes a native counting loop with the topic rebuilt per iteration, wrapped in a `try` that catches the control exceptions:

```
for (long long __i = __lo; __i <= __hi; __i++) {
    Value v__t0 = Value::integer(__i);
    try { /* body */ }
    catch (const NextEx&) { continue; }
    catch (const LastEx&) { break; }
}
```

Note that compiled code catches the *exception* forms of `next` and `last` rather than reading the cooperative registers of Chapter 14. It can afford to: the cooperative path exists to avoid a throw in the tree-walker's hot loop, and a compiled loop's body is already native.

`if`, `while`, `until` and the ternary become their C++ equivalents. Conditions that are comparisons use the `bool`-returning helpers (`rtLtB`, `rtEqSB`) rather than building a `Bool Value` and reading it back — a default, not an `-O` feature.

25.4 Closures

A block becomes a C++ lambda wrapped as a `Value`:

```
// src/Value.h
static Value closure(std::function<Value(ValueList&)> fn) {
    Value v; v.t = VT::Code; v.code = std::make_shared<Callable>();
    v.code->builtin = [fn](Interpreter&, ValueList& a) { return fn(a); };
    return v;
}
```

This is where the dual nature of `Callable` pays off. A `Callable` is *either* an AST body plus a closure environment, *or* a C++ function. The interpreter builds the first kind; the code generator builds the second. Everything downstream — `.map`, `.sort`, `callsame`, `&f` as a value, passing a block to a user routine — treats them identically, because it only ever calls through `callCallable`.

The capture is by value (`[=]`), so a compiled closure captures copies of the `Values` it uses. That is the same observable behaviour as the interpreter's environment capture for reads; for a closure that *mutates* a captured variable the code generator has to emit a shared slot instead.

25.5 Classes and multis

A `class` becomes registration code: a `ClassInfo` built at startup, its attributes filled in with precomputed defaults where possible (`ClassAttr::defVal`), and its methods installed as closures. There is no C++ class, no vtable, and no inheritance in the generated code — the runtime's object model does all of it, exactly as it does when interpreting.

A `multi` becomes one C++ function per candidate plus a dispatcher that scores with `rtTypeMatch`. That is a simplification of the interpreter's `scoreCandidate`, which is the honest reason a few multi shapes are on the fallback list.

25.6 Signatures

The generator has to bind arguments without the interpreter's `bindParams`, so there is a small family of binding helpers:

```
// src/Interpreter.h
Value  rtPos(const ValueList& a, size_t idx);
bool   rtHasPos(const ValueList& a, size_t idx);
Value  rtNamed(const ValueList& a, const std::string& key);
bool   rtHasNamed(const ValueList& a, const std::string& key);
Value  rtSlurpyPos(const ValueList& a, size_t from);
Value  rtSlurpyNamed(const ValueList& a);
size_t rtPosCount(const ValueList& a, size_t from = 0);
Value& rtPosRef(ValueList& a, size_t i);   // `is rw`: a ref into the list
```

A signature is compiled into a sequence of these. Defaults become `if (!rtHasPos(...)) ...`; a slurpy becomes one call; `is rw` binds a reference into the caller-visible argument list.

25.7 Sub-language pieces the generator hands back to the runtime

Some constructs are emitted as a single runtime call rather than transpiled, because transpiling them would mean duplicating an engine:

- **regexes and substitutions** — `RT.regexMatch`, `RT.substApply`;
- **`gather`/`take`** — a collector pushed on the execution context;
- **`use`** — `RT.rtUse`;
- **the `...` sequence operator** — `RT.seqOp` and `RT.seqOpGroups`;
- **hyper operators** — `rtHyperMethod` and the hyper core;
- **`nqp::` ops** — `rtNqpOp`, the same function the interpreter calls (Chapter 30).

Each of those is a place where "call the runtime" is not a compromise but the correct answer: there is one implementation, so there is one behaviour.

25.8 What it refuses

`CodegenError` is thrown for anything the generator cannot express, and the driver answers by bundling the whole program. The main cases:

- **grammars**, which need the full engine plus the interpreter hooks;

- a few **NativeCall shapes** — `is native(&lib-sub)`, a library name computed by an expression, `is rw` out-parameters, and buffer or `CArray` parameters that need copy-back;
- anything involving a construct the generator has simply not learned.

The failure is total-per-program rather than per-construct: one refused construct bundles the entire file. That is a deliberate simplification — a hybrid binary that interpreted some routines and compiled others would need the two halves to share an environment, which is a much larger design.

25.9 The correctness constraint on resolving names at compile time

The generator resolves a named call at compile time, deciding whether it is a user sub or a built-in. **A `used` module can take that name away.**

```
# lib/OnlySub.rakumod
unit module OnlySub;
sub val() is export { 'from-module' }
```

The interpreter's `evalCall` looks in the environment *before* the builtin table (Chapter 15), so the exported sub wins. Compiled code never did that lookup and printed the built-in `val`'s answer instead — a silent divergence.

So the generator is given the module graph's exported names, and for those it emits a call that looks in the environment first:

```
// src/Interpreter.h
Value callEnvFirst(const std::string& name, ValueList args);
```

Everything else keeps the cached builtin pointer, so the cost lands only on names a module actually claims. The carve-out is the interpreter's own: a **non**-exported module sub of a built-in's name stays module-private, so the importer still reaches the built-in while the module's own code sees its own.

That fix is a good example of the general hazard in compiling a dynamic language: **any decision made at compile time is a promise about the run-time environment,** and every such promise needs a reason to be true.

25.10 Inspecting the output

```
rakupp --cpp prog.raku          # print the generated C++
rakupp --cpp -O prog.raku       # …with the optimizer on
rakupp --exe -o prog prog.raku  # compile it
```

`--cpp` is the debugging tool for everything in this chapter and the next. When compiled and interpreted output differ, the first step is to read the emitted C++ for the construct that differs; it is ordinary C++, and the divergence is usually visible.

Chapter 26

The –O Optimizer

`--exe` and its inspection twin `--cpp` accept `-O`. Everything under it is **semantics-preserving**: it changes how the compiled program computes, never what it computes. It is opt-in and off by default.

```
rakupp --exe    prog.raku -o prog     # no optimizer
rakupp --exe -O prog.raku -o prog     # optimizer on
rakupp --cpp -O prog.raku             # print the optimized C++
```

26.1 What the generated code looks like without it

By default the transpiler is faithful but generic: every value is a boxed `Value`, operators in value position go through `applyArith`, and every user-sub call packs its arguments into a `ValueList` — a heap allocation per call.

```
// sub fib($n) { $n < 2 ?? $n !! fib($n-1) + fib($n-2) }
static Value u_fib(ValueList __a) {
    Value v_sn = rtPos(__a, 0);
    return rtLtB(v_sn, Value::integer(2LL))       // condition: always inline
         ? v_sn
         : applyArith("+",
             u_fib(ValueList{applyArith("-", v_sn, Value::integer(1LL))}),
             u_fib(ValueList{applyArith("-", v_sn, Value::integer(2LL))}));
}
```

For a hot recursive function that is two costs on every one of about 1.6 million calls: a `ValueList` allocation, and value-position `applyArith` calls that dispatch on the operator *string* before touching the operands.

26.2 Pass 1 — direct-arity calls

A sub whose signature is entirely **plain required positional scalars** — no named, slurpy, optional, defaulted or destructured parameters — gets a direct-`Value` overload, plus a boxed adapter so any unusual call site still resolves. C++ overload resolution picks the right one.

```
static Value u_fib(Value v_sn) { ... }                              // fast
static Value u_fib(ValueList __a) { return u_fib(rtPos(__a, 0)); }  // adapter
```

A call site takes the fast overload when it passes exactly the right number of plain positional arguments — no `:name(...)` pairs, no `|@slurp`. Multi subs, indirect calls through `&fib`, and method calls are untouched.

The same idea covers **37 named builtins**, each of which has a real C++ function (`rtBAbs`, `rtBUc`, `rtBSin`, ...) rather than a `std::function` in a map. Chapter 27 tells that story, because the reason it works is not the one that was first assumed.

26.3 Pass 2 — inline arithmetic

For the common operators the generator emits inline helpers instead of `applyArith`:

ops	helper	fast path
`+ - *`	`rtAdd` / `rtSub` / `rtMul`	native `int64`, overflow to bignum
`**`	`rtPow`	integer power by squaring
`< <= > >= == !=`	`rtLt` ...	`int64` compare
`% %% div`	`rtMod` / `rtDivides` / `rtDiv`	floored `int64`
`~`	`rtConcat`	direct concat when both are `Str`
`eq ne lt gt le ge`	`rtEqS` ...	byte-wise when both are **plain** `Str`

```
// src/Interpreter.h
inline Value rtAdd(const Value& l, const Value& r) {
    long long z;
    if (rtBothInt(l, r) && !rakupp::add_ovf(l.i, r.i, &z))
        return Value::integer(z);
```

```
    return applyArith("+", l, r);     // Rats, bignums, mixed types, coercions
}
```

`rtBothInt` tests that both operands are `VT::Int` and neither has grown a bignum. Overflow promotes to bignum exactly as `applyArith` does.

The string comparisons matter more than the integer ones, because they sat *late* in `applyArith`'s dispatch chain — about 118 nanoseconds against 10 for a direct call. The guard is that both operands are **plain** `Str`: no `Version`/`IO`/`Buf` tag and no enum identity. A tagged value falls back to the full chain, which preserves `Version` part-comparison, enum stringification, junction autothreading and `Whatever` currying.

With both passes plus pass 3's condition lane, `fib` becomes:

```
static Value u_fib(Value v_sn) {
    return ([&]() -> bool {                       // condition lane
        do { if (!(rtIntBox(v_sn))) break;        //   guard the box
             return (v_sn.i < 2LL); } while (0);  //   compare as raw int64
        return rtLtB(v_sn, Value::integer(2LL));  //   guard failed
    }())
         ? v_sn
         : rtAdd(u_fib(rtSub(v_sn, Value::integer(1LL))),
                 u_fib(rtSub(v_sn, Value::integer(2LL))));
}
```

No heap allocation, no string dispatch. Under a real C++ optimizer that collapses to tight native-integer recursion.

26.4 Pass 3 — guarded native-int lanes

Passes 1 and 2 still build a boxed `Value` for every intermediate result: `$sum += $_ * 2 - 1` constructs four of them per evaluation. Pass 3 removes them.

A straight-line integer expression whose leaves are **int literals and plain scalar variables** is computed in raw `int64`. Each leaf is tag-guarded at run time, each operation is overflow-checked, and the result is stored **into the target's existing box** with no `Value` construction at all:

```
// $sum += $_ * 2 - 1     (inside a native range loop)
{ bool __lok = false; do {
    if (!(rtIntBox(v__t0) && rtIntSlot(v_ssum))) break;
    long long __t1; if (rakupp::mul_ovf(v__t0.i, 2LL, &__t1)) break;
    long long __t2; if (rakupp::sub_ovf(__t1, 1LL, &__t2)) break;
```

```
    long long __t3; if (rakupp::add_ovf(v_ssum.i, __t2, &__t3)) break;
    v_ssum.i = __t3; __lok = true;
} while (0);
if (!__lok) { v_ssum = rtAdd(v_ssum,
                  rtSub(rtMul(v__t0, Value::integer(2LL)),
                        Value::integer(1LL))); } }
```

Any guard failure, overflow, or domain failure falls through to the untouched boxed emission. That is sound because **lane leaves are pure** — literals and variable reads — so re-evaluating them on the slow path has no side effects.

The lane applies to statement-position assignment to a plain scalar, statement- position `++`/`--`, and conditions. The store additionally requires `rtIntSlot`: not an enum-typed box, whose stringification is its name rather than its number.

Operations covered: `+ - *` overflow-checked, unary minus, floored `%` mirroring `rtMod` bit for bit, `%%`, and the six comparisons. Everything else — `Num`s, strings, `Rat`s, bignums, array elements, method calls — fails the lane at compile time or its guards at run time.

26.5 Three defaults that are not –O

Three changes look like optimisations and are shipped as defaults in every mode, because each fixes a **correctness wart** rather than adding speed.

In-place `~=`. `$s ~= …` naively rebuilds the whole string each step, so *n* appends do quadratic work. The interpreter and Rakudo both build strings in linear time, so this was a wart:

```
// src/Interpreter.h
inline void rtCatAssign(Value& l, const Value& r) {
    if (l.t == VT::Str && r.t == VT::Str) { l.s += r.s; return; }
    l = applyArith("~", l, r);
}
```

The interpreter pairs it with sink context, so a loop body's assignment does not copy its growing result either.

`.sort($key)` extracts the key once per element. `.sort` takes either a 2-ary comparator or a 1-ary key extractor, and the difference is asymptotic: a comparator is *supposed* to run per comparison, but a key extractor runs **once per element**. Raku++ used to call the 1-ary block inside the comparator:

```
// after — a Schwartzian transform: n calls, then compare the keys
std::vector<Value> keys(items.size());
for (size_t i = 0; i < items.size(); i++)
    keys[i] = callCallable(blk, {items[i]});
std::stable_sort(order.begin(), order.end(), [&](size_t x, size_t y) {
    return valueCmp(keys[x], keys[y]) < 0;
});
```

Sorting codepoints by the length of their Unicode name:

N	before	after	Rakudo
8 K	5.70 s	0.29 s	0.34 s
64 K	18.09 s	0.68 s	0.63 s
128 K	still running at 95 s	1.33 s	1.09 s

The ratio grows with log n, which is the shape the change predicts.

Never numify a string by throwing, which is Chapter 10's `stod` story and was found while profiling the comparator case above.

26.6 Forwarding the C++ optimization level

`--exe` compiles the generated C++ at `-O2` by default; a level on the flag is passed through:

flag	codegen passes	C++ compile
(none)	off	`-O2`
`-O`	on	`-O2`
`-O3` / `-Os` / `-O0`	on	that level

The integer passes only pay off *with* C++ inlining. At `-O0` the `rt*` helpers become real calls with no fast path, so `-O0` is *slower* than the default. It is for inspecting the generated C++, not for speed.

`-O` is a speed switch and not a size lever: the binary is dominated by the statically linked runtime, mostly the Unicode tables, not by the program's own code. `-Os`

shrinks nothing that matters and costs 20 to 50% of the lane speed-up. Use –O for speed and `strip` if size matters.

26.7 Measured

––exe, best of six after a discarded warm-up, startup-inclusive:

Benchmark	––exe	––exe –O	speed-up	what –O reached
fib	166.5 ms	**47.0 ms**	3.5×	direct-arity calls, condition lane
loopsum	27.1 ms	**8.6 ms**	3.2×	the += lane
streq	46.5 ms	**17.5 ms**	2.7×	int lanes atop inline `eq`
arrayops	102.0 ms	**77.8 ms**	1.3×	map/grep body boxing
regex	61.9 ms	60.3 ms	1.0×	the regex engine dominates
hash	15.3 ms	14.9 ms	1.0×	hash slots, not scalars
sortnums	49.9 ms	48.6 ms	1.0×	`.sort` dominates
bigint	29.2 ms	29.2 ms	1.0×	`BigInt` multiply
strcat	3.9 ms	4.8 ms	0.8×	~= already linear by default

The pattern is clear and worth stating: **where the time is inside a runtime method — the regex engine, bignum multiply, `.sort` — `-0` cannot reach it.** What it removes is boxing and allocation in the program's own code.

A dedicated showcase suite in `tools/optbench/`, each program written to lean on one pass, is compiled twice and checked byte-identical against the interpreter and Rakudo before being timed:

benchmark	`--exe`	`--exe -0`	speed-up	rakudo
sieve	1029.3 ms	**25.4 ms**	**40.6**×	994.5 ms
powmod	531.5 ms	**50.6 ms**	10.5×	716.8 ms
intsum	283.1 ms	**35.9 ms**	7.9×	624.0 ms
fibcalls	701.3 ms	**190.9 ms**	3.7×	1353.3 ms
stringbuild	22.3 ms	21.9 ms	1.0×	204.7 ms

`sieve`'s whole inner loop — `while $d * $d <= $n`, `if $n %% $d`, `$d++` — runs as raw `int64` under pass 3, taking it from a tie with Rakudo at plain `--exe` to far ahead. `stringbuild` is flat because in-place `~=` is default in both.

26.8 The box stays; only its per-operation cost goes

A common question: does a native-compiled `my int $x` become a bare C++ `long long`? **No.** The generated code keeps one uniform representation at every optimization level. `my int $s` compiles to `Value v_ss`.

- **Default `--exe`:** `$s = $s + $i` is `rtAdd`, which constructs a fresh `Value` for the result.
- **`-0` pass 3:** the arithmetic runs as raw `int64` in registers and the result is written into the existing box's `.i` slot. No `Value` is constructed per operation — but the *storage* is unchanged, and loops still copy full `Value`s.

Why keep the box? Because the transpile is uniform: any expression must be able to flow anywhere — into a runtime function, an array element, `say()`, an untyped assignment — and Raku's `my int` is readable in `Any` context while a non-native `Int` can hold `Nil` or promote to bignum. Emitting a bare `long long` local is only sound once an analysis proves the variable never escapes into a `Value` context, which is a genuine escape-analysis pass and the natural successor to pass 3.

26.9 Correctness

–O is validated to produce output byte-for-byte identical to the interpreter:

- every benchmark program matches with –O on;
- every deterministic example compiles with ––exe –O and matches its golden output;
- the arithmetic fallbacks are checked directly — int64 overflow to bignum, Int/Num mixes, string coercion, <=> (left on the general path), and += at the int64 boundary;
- the lane fallbacks likewise: +=/++/*= crossing int64, floored % with negative operands, %= 0, and comparisons whose intermediates overflow.

The design leans on the fallback. Anything the fast path does not recognise routes to the same runtime code the non-–O build uses, which is why "identical output" is a checkable claim rather than a hope.

26.10 What is next

- **value-position lanes** — laneable integer expressions inside larger expressions (call arguments, list elements) still box;
- **Num lanes** — the same trick for double, with tag guards and no overflow checks;
- **array-element lanes** — @a[$i] inside the lane, with a bounds and tag guard;
- **native int locals** — the big one, and a real escape-analysis pass: prove a typed local never escapes into a Value context and never leaves Int, then emit a raw long long with no box at all, eliminating both the storage and the per-iteration copies.

Chapter 27

Dispatch Cost in Compiled Code

A compiled Raku++ program reaches code four different ways, and they are not equally fast. This chapter measures each, describes the three cuts made to them, and — because it is the more useful part — records the analysis that turned out to be wrong.

Method: micro-benchmarks against the built runtime, 2 million iterations per shape, seven repetitions, first discarded, minimum reported.

27.1 The four shapes

For `sub square($n) { $n * $n }` plus `say square(5)`, the generated C++ contains all four:

Raku	Generated C++	Dispatch
`square(5)`	`u_square(ValueList{...})`	direct C++ call; inlinable
`say ...`	`rtCallB(RT, __bfp0, "say", ...)`	cached pointer
`$n * $n`	`applyArith("*", ...)`, or `rtMul` under `-O`	string-dispatch chain, or inline
`$x.method`	`RT.methodCall(inv, "m", ...)`	name-keyed `if`-ladder

27.2 What dispatch itself costs

Trivial function body, identical `ValueList` construction in every variant, so the differences are pure dispatch:

Shape	ns/call	
direct C++ call	46.3	the floor
cached `BuiltinFn*` call	47.4	**+1.1 — dispatch eliminated**
by-name: hash, `find`, `std::function`	55.0	+8.7 vs direct
`RT.callBuiltin("chr", ...)` end to end	66.0	the pre-change path

Two readings. The by-name lookup tax is real but modest, and caching the pointer recovers essentially all of it.

The more important reading is that the floor itself is high. About 46 nanoseconds for a *trivial* call, nearly all of it the `ValueList` — a heap-allocating `std::vector` built per call. Dispatch was a quarter of the overhead; the argument vector was the rest.

Operators are a different story, because `applyArith` has integer and number fast paths at the top of its chain:

Op shape	ns/call
`applyArith("+", int, int)` — early fast path	7.8
`rtAdd(int, int)` — the `-O` lane	5.6
`applyArith("~", str, str)` — **late in the chain**	**118.1**
`rtConcat(str, str)` — direct	9.6

The late-chain operators pay for everything they walk past: the mixin-delegation check, negated-operator handling, set operations, junction autothreading, `Whatever` currying, the `Version` branch — about 110 nanoseconds of "is this something special?" before the actual string work.

27.3 Cut 1 — cached builtin pointers

Every `RT.callBuiltin("name", ...)` site now routes through a per-name pointer resolved once at program startup:

```
// src/Interpreter.h
const BuiltinFn* builtinPtr(const std::string& name) const;

inline Value rtCallB(Interpreter& I, const BuiltinFn* f, const char* name,
                     ValueList args) {
    if (f) return (*f)(I, args);
    return I.callBuiltin(name, std::move(args));   // by-name fallback
}
```

The generated call `rtCallB(RT, __bfp0, "say", ValueList{...})` **is** the cached call, not a lookup: `rtCallB` is a three-line `inline` shim whose hot path is one indirect call through the already-resolved pointer, and the `"say"` literal is touched only on the fallback branch — nothing is hashed or searched.

The shim also does a required mechanical job. `BuiltinFn` takes `ValueList&`, which a temporary cannot bind to; `rtCallB`'s by-value parameter materialises it into an lvalue.

This cut is **not –O-gated**: it is plumbing rather than speculation, so plain `--exe` gets it too. The fallback keeps semantics byte-identical for names that are not registered builtins.

One portability landmine, recorded because it cost an afternoon: the cached pointer names were originally `__bfN`, so the seventeenth builtin in a program emitted `__bf16` — which is a **reserved built-in type** (bfloat16) on arm64 clang. Hence the `__bfpN` spelling.

27.4 Cut 2 — inline string comparisons

`eq ne lt gt le ge` get the `rtEqS` family: two *plain* `Str`s compare byte-wise inline, which is exactly what `applyArith`'s tail does for them. Anything tagged falls back to the full chain.

Value-context forms sit in the `-O` table; **bool-context forms join the always-on condition table**, beside the existing integer `rtLtB` family.

End to end, 2 to 3 million iteration loops:

Loop	Before	After	
builtin-heavy (`ord(chr(...))`), `--exe`	407.8 ms	383.5 ms	1.06×
`$c++ if $a eq $b, --exe`	712.0 ms	129.2 ms	**5.5**×
`$c++ if $a eq $b, --exe -O`	621.7 ms	29.0 ms	**21**×

The string-compare cut is the headline. An `eq` in a condition used to compile to a boolified `applyArith` call: build a `Bool Value`, walk the whole dispatch chain, then read the truthiness back out. It now compiles to an inline `l.s == r.s`.

27.5 Cut 3 — true named builtins, and the analysis that was wrong

An earlier revision of this analysis claimed that a *symbol* call like `b_say(...)` had a measured ceiling of about 1 nanosecond per call, and therefore was not worth building.

That figure was correct only for renaming the same call shape — an out-of-line call still taking a `ValueList`. What a real named function unlocks is different in kind:

- **direct `Value` arguments**, so no per-call `ValueList` allocation — which is the actual floor;
- **an inlinable hot path**, because a `std::function` is an opaque wall to the optimizer and an `inline` function is not.

Worse, some builtin lambdas hid extra cost. `abs`'s delegated to `methodCall` — the full method-name ladder on every call, about 370 nanoseconds per iteration in an `abs` loop.

```
// src/Interpreter.h
inline Value rtBAbs(Interpreter& I, const Value& v) {
    if (v.t == VT::Int && !v.big && I.builtinExt_.empty())
        return Value::integer(v.i < 0 ? -v.i : v.i);
    return rtBAbsSlow(I, v);
}
```

`abs`, `chr` and `ord` became real functions. The interpreter's map entries wrap them, **so the interpreter and the generic compiled path get the win too**, and `-O` emits direct calls when a single plain positional argument lines up.

Note the `builtinExt_.empty()` guard: the inline fast path switches itself off the moment a program `augments` a built-in type, so an augmented `.abs` still wins (Chapter 16).

Loop	cached pointer	named function	
`abs`, `--exe -O`	1112.9 ms	198.3 ms	**5.6×**
`abs`, plain `--exe`	1120.5 ms	363.9 ms	3.1×
`chr+ord`, `--exe -O`	393.6 ms	200.8 ms	2.0×

The recipe then swept the rest of the viable set — **36 names**: the numeric family, the string family, the twelve trigonometric and hyperbolic functions, and the I/O quartet.

The I/O quartet is worth its own sentence, because it disproved an assumption. A *buffered* one-argument `say` costs about 125 nanoseconds, so the call plumbing was roughly 45% of it: the `say` loop went from 124.9 to 69.6 milliseconds, with byte-identical output. "I/O dominates, so the call cost does not matter" was simply false at this scale.

Each named function is the old registered lambda's one-argument case **verbatim**. Delegators keep their `methodCall`, so `augment`, user objects and junctions are untouched; only `abs` and `sign` have bypassing fast paths, and both are guarded.

Additional measurements: a `sqrt+sin+floor` loop went 731.5 to 363.6 milliseconds under `-O`; `uc+chars` went 672.5 to 574.3 — a smaller win, because the real work in `mapCase` and `graphemeCount` dominates, as it should.

27.6 The interpreter got the same treatment

A follow-up the same day: `applyArith` gained a character-dispatched `Str/Str` fast path at the top of its chain, beside the existing integer and number ones, so the *interpreter* also skips the late-chain walk for plain-string comparisons and concatenation.

Interpreted `streq` went from 909.7 to 547.9 milliseconds. The remaining gap to Rakudo there is per-node tree-walk cost, which no operator fast path can remove.

27.7 What is deliberately not done

- **The `ValueList` per call** — the dominant cost of the calling convention, about 40 of the 46-nanosecond floor. `-O`'s direct-arity pass removes it for fixed-arity user subs, and cut 3 removes it for the named builtins, but the other ~165 builtins still take `ValueList&` by contract. A wholesale change — small-buffer or span arguments — is a runtime-wide refactor with interpreter implications.
- **`methodCall`'s `if`-ladder.** After the `MName` fix (Chapter 9) name comparison is 8.5% of the profile. A dispatch table would be chasing that 8.5%, and it is not a drop-in: the ladder is not a pure dispatch on the name.
- **The remaining late-chain operators** — `x`, `xx`, `gcd/lcm`, bitwise, `min/max`, `leg/cmp` — still walk the chain. Same recipe as `rtEqS` if any shows up hot; none of the current benchmarks exercise them.

27.8 The general lesson

Three cuts, and the third one contradicted the written analysis of the first two. What made the difference was not better reasoning — it was measuring the *right thing*: the earlier figure benchmarked a rename, not the change that a rename enables.

That is the recurring failure mode in performance work, and the defence is the same one used throughout this book: benchmark the full proposal, not the piece of it that is easy to isolate, and keep a control kernel that the change cannot touch.

Chapter 28

AST Serialization and the Precompiled Parse

Raku++ executes by walking a tree, so **the tree already is the compiled form**; there is nothing further to compile it into. That makes caching straightforward in one sense — store the tree — and delicate in another, because a stored tree must be indistinguishable from a freshly parsed one.

28.1 The format

```
// src/AstSerial.h
inline constexpr uint32_t kAstSerialVersion = 5;
struct AstSerialError { std::string msg; };
std::string serializeAst(const Program& prog);
void deserializeAst(const std::string& blob, Program& out);
```

A self-describing byte string with a header. The version constant is bumped whenever the encoding or the AST changes shape, and an entry carrying a different version is **ignored, never reinterpreted**.

Two properties of the design are load-bearing.

Both directions are driven by one visitor per node type. A field cannot be written but not read, which is the failure mode that makes a format like this quietly wrong rather than loudly broken. Adding a field to `SubDecl` and forgetting the reader is not possible; the single visitor handles both.

The mutable per-node caches are deliberately not stored.

```
Binary::simpleOp, fastShape, litVal
Index::fastShape, litIdx
NumLit::cacheN, cacheD
Block::hoistNeed
StrLit::nfcDone
```

Each has an "undecided" sentinel and is rebuilt lazily on first evaluation (Chapter 18), so a loaded tree behaves identically to a freshly parsed one — it just has not warmed up yet. Storing them would be both larger and more fragile, since `litVal` is a pointer.

Everything else round-trips, **including `line` and `label`**, because diagnostics depend on them.

28.2 Two independent switches

```
rakupp --precomp-modules=on    # cache the parse of `use`d modules
rakupp --precomp-files=on      # cache the main program's own parse
rakupp --precomp-info          # what is on, where it lives, what it holds
rakupp --precomp-clean         # empty it (always safe)
```

Both are **off by default** — nothing is written to disk until asked — and they are separate because they are worth very different amounts.

Modules. A dependency tree is a lot of source and none of it changes between runs:

	no cache	cached
`use XML` (10 files, 1,110 lines)	16.0 ms	**5.7 ms**

Files. A script's own parse is already sub-millisecond, and the few- millisecond floor of a small program is process startup, not parsing:

bare file	no cache	cached	saved
50 lines	2.8 ms	2.4 ms	0.4 ms
1,000 lines	10.0 ms	5.1 ms	4.9 ms

bare file	no cache	cached	saved
20,000 lines	158.8 ms	66.5 ms	**92.3 ms**

Across the twenty-two fastest example programs, turning `files` on made **no measurable difference at all**. There is also a cost on the run that *writes* an entry — about 22 milliseconds at 20,000 lines — so for a script run once, `files` caching is a small net loss.

If you enable one, enable **modules**. That asymmetry is precisely why these are two switches rather than one.

28.3 What invalidates an entry

This is the interesting part, and it is where the naive design would be wrong.

```
// src/Interpreter.h
bool precompLoadProgram(const std::string& srcPath, const std::string& src,
                        const std::vector<std::string>& searchPath,
                        Program& out, std::string& finishOut);
void precompStoreProgram(const std::string& srcPath, const std::string& src,
                         const std::vector<std::string>& searchPath,
                         const Program& prog, const std::string& finish,
                         const std::vector<std::pair<std::string,
                                                     std::string>>& deps);
```

An entry is valid only while **four** things still hold.

The source has not changed. Content-validated, not timestamp-validated.

The rakupp binary is the same one. A tree from another build may not match this build's node shapes.

The search path is the same. This is the one people do not expect, and it is in the signature for a reason: the search path decides which *file* a `use` resolves to for operator scanning, so the same source under a different `-I` can legitimately parse differently.

The operators of everything it `use`s are unchanged. This is the deepest one. A module that gains or loses an operator declaration changes how *this* file parses, without this file changing at all (Chapter 5). So the parser records every source it scanned:

```
// src/Parser.h
std::vector<std::pair<std::string, std::string>> opScanned_;
```

and those pairs go into the entry as dependencies. It is the same list the parser needed anyway; making it a cache key was nearly free.

28.4 Where it lives, and what it reports

Settings persist in a small config file under the user's config directory; `RAKUPP_PRECOMP_MODULES` and `RAKUPP_PRECOMP_FILES` override for one invocation, and `RAKUPP_NO_PRECOMP=1` forces both off.

```
// src/Interpreter.h
struct PrecompEntry {
    std::string source; unsigned long long bytes; bool usable; bool orphan;
};
std::vector<PrecompEntry> precompCacheList();
std::pair<size_t, unsigned long long> precompCacheClear();
```

The `orphan` flag distinguishes the one cache state that is *pure garbage* from the ones that are merely misses waiting to happen: the source file is gone, so the entry can never be reused **or** rewritten. Everything else is derived data, which is why clearing the cache is always safe.

28.5 What is *not* cached

Only the parse. A module's top-level declarations and its `BEGIN` blocks still execute on every run, which is roughly the 30% of module load time that is not parsing.

Rakudo goes further: its `.precomp` serialises the *objects* that compile-time code produced, so `BEGIN` runs once per compile rather than once per run. Matching that would mean serialising live runtime state — class registries, closures, whatever a `BEGIN` block built — which is a different and much larger project.

Compared with the neighbours:

	what it stores	where
Raku++	the AST	a central content-validated directory

	what it stores	where
Rakudo `.precomp`	bytecode plus compile-time objects	beside the source
Python `__pycache__`	bytecode	beside the source

Storing entries centrally rather than beside the source is what makes caching the **main program** defensible at all. Python never caches `__main__`, and the usual objection is that it would litter the source tree; that objection does not apply here.

28.6 The other consumer: embedded modules

The same serialisation carries modules inside a compiled binary (Chapter 24):

```
// src/Interpreter.h
struct BundledModule { std::string name, blob, finish, src; };
void rakuppRegisterModule(const std::string& name, const char* blob,
                          size_t blobLen, const std::string& finish);
```

`collectModuleGraph` resolves the transitive `use` graph at build time and serialises each module, dependencies first. `loadModule` consults the embedded table before the disk.

That the cache format and the embedding format are the *same* format is not a coincidence — it is why a bug in either is found twice as fast.

28.7 The AOT emitter, and where it differs

`--aot` solves a similar problem in a completely different way: instead of a byte encoding, it emits **C++ source that rebuilds the tree**.

```
// src/Ast.h
struct AstEmitError { std::string msg; };
void emitAstProgram(const Program& prog, std::ostream& out,
                    const std::string& fileName, const std::string& finish,
                    const std::vector<BundledModule>& mods);
```

The trade-offs are opposite. The serialiser is compact and fast to read but needs its own reader; the emitter needs no reader at all — the C++ compiler is the reader

— but produces one function per node, so a few hundred lines of Raku become tens of thousands of lines of C++.

Both fail *soft*. An unserialisable module is left out of the binary and loaded from disk; an unemittable node makes `--aot` fall back to bundling. Neither produces a wrong tree.

That shared property is the design rule worth taking away: **a derived-data mechanism should degrade to recomputation, never to a guess.** Every failure mode in this chapter — a version mismatch, a changed dependency, a missing source, an unhandled node — ends in "parse it again", which is slower and correct.

Part VIII

Boundaries

Chapter 29

Modules

A module is **source text that gets lexed, parsed into its own `Program`, and executed exactly once** — at the point its `use` statement runs, inside the same interpreter, on the same object graph as the importing program.

It gets a fresh `Env` chained to the global one for the duration of the load; when the load finishes, that environment's contents are **copied into the global `Env`**. After that the module has no ongoing identity: its subs are ordinary `VT::Code` values in the global scope, its classes ordinary entries in the one flat class registry. Calling into a module is not a different operation from calling a sub in the same file.

That paragraph is also the source of nearly every divergence at the end of this chapter.

29.1 Parse time: only operators are looked at

`use Foo` triggers exactly one compile-time action:

```
// src/Parser.h
void scanModuleOps(const std::string& module);
```

which finds the module's *source file* and **text-scans** it — no lexing, no parsing — for declarations of the five operator categories, registering those names in the parser's tables so the rest of the importing file parses (Chapter 5).

That is the whole compile-time effect, because operators are the only thing about a module that changes how the importing file **parses**:

```
use Vector;          # declares `sub infix:<·>`
say $a · $b;         # a parse error without the scan
```

Two consequences. An operator spelled only in ASCII operator characters is skipped, because `multi infix:<*>(Color, Real)` is almost always extending a built-in and registering it would silently reshape every expression in the importing file. And the scan reads the file a second time — cheap, being a substring search, but it *can* be fooled by an operator name inside a string or a comment.

29.2 Run time: `loadModule`

```
loadModule(name)
 ├─ already loaded?                    → return immediately
 ├─ find the source (see below)
 ├─ Lexer → Parser → Program            (its OWN AST)
 ├─ keptPrograms_.push_back(prog)       (the AST must outlive the load)
 ├─ moduleEnv = new Env, parent = global_
 ├─ bind %?RESOURCES / $?DISTRIBUTION into moduleEnv
 ├─ scan the AST for `is export` names
 ├─ cur = moduleEnv;  curPkgEnv_ = moduleEnv
 ├─ hoistSubs(prog->stmts)
 ├─ exec() every top-level statement
 ├─ publish(): copy moduleEnv->vars into global_
 └─ restore cur / curPkgEnv_ / pkgPrefix / finishData_
```

The module's top-level statements run like any other code, once. `BEGIN` blocks in the module run as part of this — there is no separate compile phase for them to run in.

`loadedModules_` is checked on entry and the name inserted immediately, so recursive and repeated `use`s are no-ops. A cyclic `use` therefore will not hang, but the second module sees a *partially loaded* first one.

29.3 Where the source is found

`libPaths_` starts as `{"lib", ".", "rakulib"}` and is extended:

Source	Position
`RAKULIB`	appended
`ROAST`	appended (adds the test-helper library)
`-I dir`	prepended
`use lib '...'` at run time	prepended

`RAKULIB` accepts **both `,`** and `:` as separators, with one carve-out: a `:` directly after a lone leading letter is a Windows drive letter, not a separator.

That splitting logic is shared rather than duplicated, and the comment in the header says why:

```
// src/Interpreter.h
// Shared so the PARSER's copy of the search path — which is what finds a
// `use`d module's operator declarations while the importing file is still
// being parsed — cannot drift from the interpreter's. It did: the
// interpreter learned ',' and the parser did not, so `RAKULIB=a,b`
// silently stopped importing operators.
std::vector<std::string> splitSearchPath(const std::string& spec);
```

For each base, both `<base>/` and `<base>/lib/` are tried, with the extensions `.rakumod`, `.pm6`, `.raku` and `.pm`.

If nothing matches, the **installed** repositories are searched. Resolution mirrors a real installation repository: SHA-1 the module name, read the short-name index entry, take its content id, and read the source by that id. So Raku++ loads zef-installed modules directly out of Rakudo's own install tree.

```
RAKUPP_TRACE=1 rakupp myprogram.raku
```

```
[LibPaths] [lib] [.] [rakulib]
[Load] JSON::Fast <- source lib/JSON/Fast.rakumod
[Load] Test::Util <- installed /Users/you/.raku dist=E3A0...
```

29.4 Failure is fatal, deliberately

A module that is missing, fails to parse, or throws while loading **aborts the program**.

It did not always. It used to warn and carry on, and that was a divergence with real consequences: a `use` that silently vanished left the program running against a state nobody designed — the module's `BEGIN` blocks never ran, its exports never materialised. It also flattered every measurement. In the module compatibility battery, twenty probes "produced output" purely because the load had been skipped, so the comparison against Rakudo was meaningless for them.

The one exemption is `Slang::*`, which are compile-time grammar mutators Raku++ cannot apply at all; failing on them would reject programs it otherwise runs perfectly.

Pragmas with no file behind them — `strict`, `fatal`, `experimental`, `newline`, `precompilation` — are listed **explicitly** rather than being allowed to fall out of a failed lookup. Ignoring an unimplemented pragma must be a deliberate entry, not an accident.

29.5 Questions people actually ask

29.5.1 Is there an `Env` that holds the module?

During the load, yes; afterwards, no. `moduleEnv` exists only for the duration of `loadModule`. Its real job is scoping the module's *own* code: the module's subs are defined there so they see one another, and each `Callable` captures it as its closure. That closure is what keeps the module's lexical world alive after publication — a module sub calling another module sub still resolves it through `moduleEnv`.

What the importer sees is the *copy* in the global environment. There is no stash object, no per-module symbol table to enumerate, no `Foo::EXPORT::DEFAULT`.

29.5.2 Is there a separate AST?

Yes, and it is never walked as a unit again. Each module gets its own lexer, parser and `Program`. That `Program` is executed once, top to bottom, and then survives only as *storage*: `Callable::params` and `Callable::body` are borrowed raw pointers into it (Chapter 6), which is why it is pushed onto `keptPrograms_` and never released.

Parsing is isolated in **one direction only**. The module is parsed with a fresh `Parser`, so the importing file's operators, lexical regexes and sigilless names do not leak into it. In the other direction the module's operators *do* reach the importer — through the text scan, not the parse.

29.5.3 Is calling a module routine different?

No. `evalCall` resolves every named call the same way (Chapter 15), and `Env::find` walks the parent chain. A local `my sub` is found nearby; a module sub is found in the global environment at the end of the chain. The only difference is a slightly longer walk to *find* it; the call itself is byte-for-byte identical.

The same holds for methods: the class registry is one flat map, so a class from a module and a class from the current file are indistinguishable at dispatch time.

29.5.4 How does `is export` work?

`is export` is scanned for, but it is **not** what makes a symbol visible. Its one real effect today is the built-in collision carve-out in `publish()`:

> A sub that is **not** `is export` and whose name collides with a built-in stays module-private.

That is what lets a module's `our sub run` coexist with the built-in `run`: inside the module, `run` resolves through `moduleEnv` to the module's own; an importer's bare `run(...)` reaches the built-in.

The `sub EXPORT(*@_)` protocol *is* implemented: if the module defines it, it is called with the `use` statement's arguments and the map it returns is defined in the **using** scope. `use Mod <name:alias>` imports a routine under a second name.

29.5.5 `our` and `unit module`?

`tctx_.pkgPrefix` is the mechanism. `unit module Foo;` sets the prefix for the rest of the compilation unit; a braced `module Foo { ... }` sets it for the body and restores it after. Then an `our sub bar` is also published as `&Foo::bar`, an `our` variable as `$Foo::x`, a `my` one not at all, and a class or grammar registers under its qualified name.

That last one was a bug until it was not: compound names skipped the prefix, so `grammar Schema::Core` inside `unit module YAMLish` registered unqualified and was unreachable from any importer.

29.5.6 When do a module's `END` blocks run?

At **process** end, not at load, and *before* the mainline's own `END` blocks, last-in-first-out, so a module loaded later cleans up earlier. They capture the module scope, so they still see its lexicals.

29.5.7 What about `--exe`?

Modules are **not** compiled into the binary. The generator emits a call to `Interpreter::rtUse`, a thin mirror of the interpreter's `use` handling, calling the same `loadModule`. A compiled binary still interprets its modules — though it can carry their serialised ASTs inside itself (Chapter 24), so it needs neither their sources nor a warm cache to start.

29.6 Divergences from Rakudo

These are known and mostly deliberate. They are the reason a module can behave differently under Raku++ than under Rakudo even when nothing is broken.

#	Raku++	Rakudo
1	`publish()` copies the module's **whole** environment to global. A `my sub`, a non-exported `our sub`, and its classes are all visible bare to the importer.	Only `is export` symbols are imported.
2	`need Foo` still makes Foo's symbols visible.	`need` loads without importing.
3	Types resolve **at run time**. `NoSuchType.new` is a run-time "no such method", after output has appeared.	`Undeclared name` at compile time.

#	Raku++	Rakudo
4	The parse is cached; declarations and `BEGIN` still run every time.	`.precomp` caches the parse *and* the objects compile-time code produced.
5	A module's operators reach the importer by **text scan**.	Real lexical export of the operator into the importer's grammar.
6	One flat class registry — no per-compunit type isolation.	Each compunit has its own stash.
7	`use Foo:ver<...>:auth<...>` adverbs are accepted and **discarded**; the first match on the search path wins.	Full version, auth and API resolution.

Divergence 1 is the big one and it cuts both ways. It is why some modules work under Raku++ that would otherwise need more machinery, and it is why a module's internal helper can silently shadow something in the importing program.

Demonstrated with a module declaring `my sub private-sub`, `our sub our-sub`, `sub exported-sub is export` and `class Widget`:

Called from the importer	Raku++	Rakudo
`private-sub()`	`"private"`	not visible
`our-sub()`	`"our"`	not visible
`exported-sub()`	`"exported"`	`"exported"`
`Demo::our-sub()`	`"our"`	`"our"`
bare `Widget.greet`	`"widget"`	not visible

29.7 Debugging a module problem

1. **`RAKUPP_TRACE=1`** — confirm *which* file was loaded. Half of all module bug reports are the wrong copy being picked up, usually because `.` is on the search path or an installed copy shadowed a checkout.
2. **`RAKULIB` takes `,` or `:`.** If the same value also has to drive Rakudo, write `,`.
3. **Instrument the real module, not a reduction.** Copy its `lib/` to a scratch directory, add `note` calls, run it under both engines. Reductions of grammar and module behaviour have repeatedly *passed* while the real module failed — the reduction drops the one piece of context that mattered.
4. **Load failures are loud now**, so a module that appears to do nothing is loading fine and behaving differently, not being silently skipped.

Chapter 30

`use nqp`: a Compatibility Subset

Some ecosystem modules — most importantly `JSON::Fast`, which sits under about 170 other distributions — are written in ordinary Raku but reach for **NQP ops** in their hot paths: `nqp::iseq_i`, `nqp::substr`, `nqp::push_s`.

NQP is Rakudo's bootstrap language, and its `nqp::` operations are the low-level primitives the Raku runtime is built on. Rakudo documents them as **internal and unspecified**; every implementation provides whatever subset its ecosystem happens to need.

Raku++ provides that subset. This chapter is about how, and — more interestingly — about why it costs **nothing** for the programs that do not use it.

30.1 There is no NQP compiler here

The single most important thing to understand: **`nqp::foo(...)` is already valid Raku syntax.** It is a call to a package-qualified routine named `nqp::foo`. The normal parser reads it with no special help.

There is no NQP grammar, no NQP compiler, and no QAST anywhere in Raku++. What is added is a **semantic layer**: when a program opts in with `use nqp`, the parser recognises those already-parsed calls and gives them meaning, compiling them to dedicated AST nodes instead of leaving them as calls to undefined routines.

```
use nqp;                        # (or `use MONKEY-GUTS`)
my int $x = nqp::add_i(2, 3);  # 5
```

Without `use nqp`, `nqp::add_i(...)` is exactly what it looks like — a call to a routine that does not exist — and running it gives the usual undefined-routine error.

30.2 Zero cost when unused

This is a hard design constraint, not an aspiration, and it holds by **construction** rather than by optimisation.

- The parser carries one boolean, `useNqp_`, that starts false and is set only by seeing `use nqp` or `MONKEY-GUTS`. Every recognition site is guarded by it, so in a normal program those checks are a single already-false branch on a token that begins with `nqp::` — which no ordinary identifier does.
- The `NqpOp` node is **only ever constructed** inside that guarded path. A program without `use nqp` builds no such node, so the interpreter's `NqpOp` arm is never reached and the implementation is dead code for that run.
- There are no new fields on any hot struct, no new work in any hot loop, and no change to how normal calls, strings or numbers are handled.

A regression test asserts the invariant directly: that `nqp::add_i` still reports "undefined routine" without `use nqp`.

30.3 How the ops compile

Inside a `use nqp` unit, a recognised `nqp::op(...)` becomes one of three things **at parse time**.

Constants fold to literals. `nqp::const::CCLASS_NUMERIC` is replaced by the integer `8` in the parser; it never exists as a call. All the character-class flags and normalization modes are handled this way.

Control forms become native or lazy nodes, because their arguments must not all evaluate eagerly. `nqp::if` and `nqp::unless` compile to Raku++'s own ternary node, so the untaken branch never runs. `nqp::while`, `nqp::until`, `nqp::stmts` and `nqp::ifnull` become `NqpOp` nodes whose evaluator drives its own argument evaluation.

Leaf ops become eager `NqpOp` nodes — integer maths, string and list primitives — which evaluate their arguments normally and then do the low-level operation over Raku++'s own `Values`.

An `nqp::` op **outside** the implemented subset is left as an ordinary call and fails loudly at run time rather than silently misbehaving. The subset grows by measured demand, never by guessing.

30.4 In code

The parser branch, condensed:

```
// src/Parser.cpp — the term is already lexed as the qualified name `nqp::add_i`
if (useNqp_ && name.compare(0, 5, "nqp::") == 0) {
    if (name.compare(0, 12, "nqp::const::") == 0) {
        long long cv;
        if (nqpConstValue(name.substr(12), cv))
            return std::make_unique<IntLit>(cv);      // folded here
    }
    if (ExprPtr n = makeNqpOp(name.substr(5), callArgs))
        return n;                                     // an NqpOp, or a Ternary
}
```

The node is deliberately tiny:

```
// src/Ast.h
enum class NqpOpc : uint16_t {
    Stmts, While, Until, IseqI, AddI, Substr, /* … */ };
struct NqpOp : Expr {
    NqpOpc op;
    std::vector<ExprPtr> args;
};
```

and the evaluator handles the lazy forms *before* touching their arguments:

```
// src/Builtins.cpp — condensed
Value Interpreter::evalNqpOp(NqpOp* n) {
    auto& a = n->args;
    switch (n->op) {                          // lazy forms drive their own args
        case NqpOpc::While:
            while (boolify(eval(a[0].get())))
                for (size_t i = 1; i < a.size(); i++) eval(a[i].get());
            return Value::nil();
        default: break;
    }
    ValueList v;                              // everything else: eval once
    for (auto& e : a) v.push_back(eval(e.get()));
    auto I = [&](size_t i){ return v[i].toInt(); };
```

```
    switch (n->op) {
        case NqpOpc::AddI:  return Value::integer(I(0) + I(1));   // int64
        case NqpOpc::IseqI: return Value::integer(I(0) == I(1));
        // … about 40 more leaf ops …
    }
}
```

30.5 The argument buffers

An nqp-heavy program is nqp-heavy in the extreme: a tokenizer written in Raku runs about 1.5 million operations on a 278 KB document. Building a fresh `ValueList` per operation is a malloc and a free per op, for a vector of one to four values.

```
// src/Interpreter.h — ExecContext
std::deque<ValueList> nqpArgs;   // one reusable buffer per nesting depth
size_t nqpDepth = 0;
```

The buffers are kept, so their capacity is kept; the contents are still cleared on the way out, so argument lifetimes are exactly what they were.

It is a **deque**, not a vector, and the comment says why: an argument's own evaluation can re-enter `evalNqpOp`, and growing a vector would reallocate under the outer frame's live reference. A deque never moves the elements it already holds.

That is a small illustration of a recurring theme — the reusable-buffer optimisation is easy, and the container choice that makes it *correct* under re-entrancy is the part that takes thought.

30.6 Native compilation

The subset is a first-class part of the transpiler, not just the interpreter. A `use nqp` program native-compiles like any other; it does **not** fall back to bundling.

```
$ rakupp --cpp -e 'use nqp; say nqp::add_i(1, 2)'
…
rtCallB(RT, __bfp0, "say", ValueList{
    ([&]()->Value{ ValueList __na = ValueList{Value::integer(1LL),
                                              Value::integer(2LL)};
                   return rtNqpOp(NqpOpc(10), __na); }())});
```

Two emission strategies, mirroring how the ops evaluate:

- **Eager leaf ops** share their implementation with the interpreter through a free function `rtNqpOp(NqpOpc, ValueList&)` in the runtime. `Interpreter::evalNqpOp` delegates to it and the generator emits a call to the *same* function, so there is exactly one copy of the op logic and the compiled binary cannot drift from the interpreted meaning.
- **Lazy control forms** do not route through a runtime call — they emit native C++ directly, a real `while` and a statement sequence, preserving their non-eager evaluation.

The zero-cost guarantee extends here too: a program without `use nqp` emits zero references to `rtNqpOp` or any nqp machinery.

30.7 What is covered

Around fifty operations, chosen from the actual inventory of the modules in the ecosystem compatibility battery:

Group	Ops
Control (lazy)	`if unless while until stmts ifnull`
Integer (int64, no bignum promotion)	`iseq_i isne_i islt_i isle_i isge_i isgt_i add_i sub_i mul_i bitand_i`
String (codepoint-indexed)	`ordat eqat substr chars concat join index chr strfromcodes strtocodes findnotcclass iscclass`
List	`list list_i list_s elems atpos atpos_i bindpos push push_i push_s pop_s shift_i splice`
Hash	`hash bindkey`
Object / attr	`create istype getattr bindattr p6bindattrinvres null isnanorinf`
Constants	`const::CCLASS_*`, `const::NORMALIZE_*`

Integer ops use plain `int64` semantics, matching NQP's native integers — **no overflow to bignum** — and string ops index by codepoint rather than by grapheme. Both of those are deliberate: an NQP op is supposed to be the low-level primitive, and a module reaching for one is reaching past Raku's semantics on purpose.

30.8 Scope and non-goals

- **Not a general NQP runtime.** This is a compatibility shim sized to the ecosystem. Operations nobody's modules use are not implemented until a module needs them.
- **Semantics match Rakudo's observable behaviour, not its internal representation.** `nqp::create(IterationBuffer)` gives back a plain Raku++ buffer, because what the calling module does with it is all that matters.
- **`use nqp` is an opt-in for module code**, not an invitation for application code. There is no reason to reach for `nqp::` operations in a program you are writing, and no support commitment for operations beyond the measured subset.

30.9 Why this shape is the right one

The temptation with a compatibility layer is to implement the source language. That would have meant an NQP grammar, an NQP compiler, and a second execution model to keep in step with the first — for a feature whose entire purpose is to let a handful of modules load.

Recognising that `nqp::foo(...)` is *already* valid syntax in the host language turned a compiler project into about 250 lines of parser branch and evaluator arm. The general principle generalises: **before implementing a foreign language, check whether its surface syntax is already legal in yours.**

Chapter 31

NativeCall and the libffi Backend

```
use NativeCall;
sub strlen(Str --> size_t) is native {*}
sub hypot(num64, num64 --> num64) is native {*}

say strlen("hello");     # 5
say hypot(3e0, 4e0);     # 5
```

`is native` calls a C function directly. This chapter is about how that works inside, which is a more interesting story than the syntax: Raku++ ships as one portable binary with no third-party dependencies, and it would like to keep that property — so it does not link libffi and does not vendor it.

31.1 libffi is loaded at run time, and its ABI is restated by hand

```
// src/Ffi.h — the header comment, condensed
// Raku++ ships as ONE portable binary with no third-party dependencies, so
// it does not link libffi and does not vendor it. Instead the library is
// loaded at RUNTIME with dlopen ... and the handful of ABI declarations we
// need are restated here.
```

Two things make hand-declaring another project's ABI safe, and they are worth stating because the technique generalises.

We never touch `ffi_cif`'s fields. Its size and layout are target-dependent, so the code hands libffi an over-sized zeroed buffer and only ever passes the pointer back:

```
// src/Ffi.h
struct Cif { alignas(16) unsigned char buf[512] = {}; };
```

512 bytes is far past any target's real size. `ffi_type`'s layout, by contrast, *is* stable public ABI — callers have always had to build struct types by hand — so it is declared normally:

```
// src/Ffi.h
struct Type {
    size_t         size;
    unsigned short alignment;
    unsigned short type;
    Type**         elements;   // null-terminated, for T_STRUCT
};
```

The calling-convention constant is probed, not tabulated. `FFI_DEFAULT_ABI` is an enum whose value differs per target *and* per libffi release. On the machine this was developed on, the arm64 library wanted one value while the x86-64 build of the same program wanted another — neither matching what current libffi headers imply.

So the loader tries candidates until one passes a **self-test of real calls** — `strlen`, `abs`, `ldexp`, `ldexpf` — and disables the backend entirely if none does:

```
// src/Ffi.h
struct Lib {
    bool ok = false;
    std::string path, why;
    int abi = 0;                 // FFI_DEFAULT_ABI for this target, probed
    int (*prep)(void*, int, unsigned, Type*, Type**);
    int (*prep_var)(void*, int, unsigned, unsigned, Type*, Type**);
    void (*call)(void*, void (*)(void), void*, void**);
    void* (*closure_alloc)(size_t, void**);
    int (*prep_closure_loc)(void*, void*,
                            void (*)(void*, void*, void**, void*),
                            void*, void*);
    Type *t_void, *t_uint8, /* … */ *t_pointer;
};
const Lib& lib();          // loads and self-tests on first call; never retries
```

The rule that makes this defensible: **a failed probe disables the backend rather than guessing.** Miscalling through a wrong-shaped interface would produce plausible wrong answers, which is the worst possible failure mode for an FFI.

```
rakupp --ffi-info      # libffi: libffi.dylib (abi 1)
```

31.2 What happens with no libffi

Not everything keeps working, and being clear-eyed about that is part of the design. The fallback is a **narrower** FFI, not a transparent substitute.

Calls go through a fixed prototype that hands eight integer and eight floating-point arguments to the platform ABI and lets it place them. That is correct for a large majority of real C signatures: pointers, `Str`, integers of every width, `num64`, `CArray`, struct and union handles, `is rw` out-parameters, `nativecast`, `cglobal` and synchronous callbacks all behave exactly as they do with libffi.

Four things it cannot do, and each **throws at the point of the call** rather than computing a wrong answer:

a `num32` argument or return	a real C `float` cannot be placed by the fixed prototype
a variadic signature	needs `ffi_prep_cif_var` to say where `...` begins
more than 8 integer or 8 float arguments	the prototype is that wide and no wider
more than 64 distinct callbacks	the trampoline pool holds 64

`RAKUPP_FFI=0` forces this path, which is how the test suite exercises it: the whole suite runs twice, once each way. WebAssembly takes it by construction — there is no shared library to open.

31.3 How a call is made

1. **Resolve.** The library is `dlopen`ed — the name as written, then the platform-decorated forms — and the symbol `dlsym`ed.
2. **Marshal.** Each argument is converted once, at its *declared* width, into a slot libffi reads directly.

3. **Describe.** The signature becomes a call interface, prepared once per sub — with the variadic form when the signature is variadic.
4. **Call**, then box the return: a pointer becomes a live `Pointer` or `CArray`, a `Str` return is read as a C string, a narrow integer is truncated and sign-extended, and `is rw` out-parameters and mutated buffers are copied back.

Steps 1 and 3 are cached **on the `Callable`**, and the reason is measured:

```
// src/Value.h — Callable, the native fields
bool isNative = false;
std::string nativeLib, nativeSym, nativeLibSub;
const Expr* nativeLibExpr = nullptr;
void* nativeSymCache = nullptr;   // dlopen/dlsym once, not per call:
                                  // 5 dlopen candidates cost a flat ~67 µs
void* nativeCifCache = nullptr;   // ffi_prep_cif is ~80 ns — 20% of a whole
                                  // crossing — so it must not run per call
```

The cost of going through libffi rather than calling blind is about **23 nanoseconds per crossing** — 157 milliseconds against 150 for 300,000 calls of `abs`. On a crossing that costs about 490 nanoseconds end to end, that is under 5%, which is why there is one code path rather than a fast one and a general one.

`nativeLibExpr` handles a genuinely awkward case: `is native(EXPR)` where the expression could not be evaluated at declaration time. It is retried once at the first call, and the raw `Expr*` is safe to keep because the AST outlives the interpreter.

31.4 Variadics

Variadic C functions are the easiest thing in an FFI to get quietly wrong.

C's `...` is not a formality. On the SysV AMD64 ABI a variadic argument goes in a register but the callee is told how many; on the Apple ARM64 ABI it goes on the **stack**, while a fixed argument in the same position would go in a register. A caller that does not know which arguments are variadic cannot place them, and the failure is silent — you get a number, it is just the wrong one.

Raku++ spells it with a **slurpy marking the position where C's `...` begins**, which is the same spelling Rakudo uses:

```
use NativeCall;
sub snprintf(Buf, size_t, Str, *@args --> int32) is native {*}

my $buf = buf8.allocate(64);
```

```
my $n = snprintf($buf, 64, "%s scored %d at %.1f%%", "Ada", 97, 99.5e0);
say $buf.decode('latin-1').substr(0, $n);   # Ada scored 97 at 99.5%
```

Everything beyond the declared parameters is variadic and is typed from the value itself, following C's **default argument promotions**:

You pass	C receives
`Int`	a 64-bit integer
`Num` / `Rat`	`double` — never a bare `float`
`Str`	`char*`
`Buf` / `Pointer` / `CArray`	the pointer

That promotion table is why `%.1f` works without saying anything: a float argument to a variadic function *is* a `double` in C.

Until this was fixed, Raku++ placed `...` arguments in registers and answered `"0"` for `snprintf($buf, 16, "%d", 42)`. It is a parity fix, not an extension — and it is what makes `curl_easy_setopt`, the `printf` family, `open(2)` with a mode, and `ioctl` reachable at all.

31.5 The type map

Raku	C	Bytes
`int8` `uint8` `byte`	`int8_t` `uint8_t`	1
`bool`	`bool`	1
`int16` `uint16`	`int16_t` `uint16_t`	2
`int32` `uint32`	`int32_t` `uint32_t`	4
`int` `int64` `long` `size_t`	64-bit integer	8
`num32`	`float`	4
`num` `num64`	`double`	8
`Str`	`char*`	pointer
`Buf` / `Blob`	the bytes, copied back after the call	pointer
`Pointer[T]` `CArray[T]`	the pointer	pointer
a `repr('CStruct')` / `CUnion` class	a pointer to it	pointer

Raku	C	Bytes
`&callback (...)`	a C function pointer	pointer

The table lives in `Interpreter.cpp` and `ffi::scalar(width, sign, isFloat)` maps into it, deliberately keeping the Raku type names in one place.

Structs are laid out with natural alignment; `.new` allocates zeroed native memory, and field reads and writes go to that memory, so a C function can fill one in. `nativesizeof` answers from the same layout code.

```
// src/Interpreter.h — the pointer helpers
Value ncMakePointer(const std::string& type, void* p);
Value ncMakeLiveCArray(const std::string& type, void* p);
static Value ncReadElem(long long addr, const std::string& ofType, long long i);
static void ncWriteElem(long long addr, const std::string& ofType,
                        long long i, const Value& val);
static long long ncFieldOffset(ClassInfo* ci, const std::string& field,
                               std::string& type);
static long long ncStructSize(ClassInfo* ci);
```

31.6 Callbacks

A `Callable` handed to C becomes a real C function pointer built with an `ffi_closure`. Parameters are typed from the block's own signature:

```
// src/Interpreter.h
long runCallback(int slot, long a0, long a1, long a2,
                 long a3, long a4, long a5);
void runFfiClosure(void* closure, void* ret, void** args);
bool onRakuThread() const { return (bool)tctx_.cur; }
```

The last one is the guard that matters. A callback must fire **during** the native call — `qsort`, `bsearch`, an iteration callback. A callback that C stores and fires later from a thread of its own has no Raku scope to run in, so `onRakuThread()` detects it and the callback is ignored with a warning rather than run against a thread the interpreter never entered.

Note what that predicate actually tests: whether this thread has a current lexical scope. A thread the C library made for itself has none.

31.7 Your declaration is a promise nothing can check

A C library exports addresses, not types. Nothing — not Raku++, not Rakudo, not any FFI in any language — can compare a declaration against the real prototype, because that information does not exist at run time. A wrong declaration does not fail; it produces a plausible answer.

```
sub abs(num64 --> num64) is native {*}   # WRONG — C's abs is int abs(int)
say abs(-3.34e0);   # -3.34
say abs(-99.5e0);   # -99.5
```

Every answer is the argument, unchanged, and that is the tell. libffi put the `num64` in the first *floating-point* register, but `int abs(int)` reads the first *integer* register and returns an integer in the integer return register. The floating-point return register is never written, so it still holds what was passed. The call really happened; the wrong question was asked.

Rakudo prints exactly the same three lines. This is not an implementation quirk to work around — it is what an FFI is.

31.8 Tracing

```
RAKUPP_FFI_TRACE=1

[ffi] backend: libffi: libffi.dylib (abi 1)
[ffi] snprintf(0x156f1c9a0, 64, "%s scored %d at %.1f%%", "Ada", 97, 99.5) -
> 22
[ffi] sqlite3:sqlite3_libversion() -> "3.43.2"
```

It shows the resolved C symbol rather than the Raku sub name, and — importantly — **what actually crossed**: the marshalled argument values and the raw return, not the Raku values re-printed. So a marshalling bug shows up in the trace instead of being hidden by it.

An external tracer cannot do this job. `lldb`, `ltrace` and `dtrace` see C symbols, and the interpreter's own C++ calls `strlen` and `malloc` constantly, so a breakpoint on `strlen` cannot distinguish a crossing the program asked for from one the runtime made for itself. Only this layer knows which call came from Raku.

31.9 In compiled code

`--exe` emits the same `callNative`, so `is native` behaves identically interpreted and compiled — and a compiled binary looks for libffi at *its* run time, not at build time.

A few shapes make `--exe` fall back to bundling: `is native(&lib-sub)`, a library name computed by an expression, `is rw` out-parameters, and buffer or `CArray` parameters needing copy-back. The compiler says so and produces a bundled binary instead.

31.10 Not supported

- **C structs passed or returned by value.** Rakudo cannot express this either — its NativeCall maps a struct to one type code with no by-value variant — so adding it here would mean inventing syntax rather than matching an existing spelling. A return of up to 8 bytes can be declared `int64` and unpacked by hand.
- **Callbacks fired from a thread the C library owns**, as above.
- **Embedded structs.** A struct-typed field is a pointer, not an inlined struct.
- `explicitly-manage`.

Chapter 32

The Extension ABI

Raku++ walks an AST. It does not JIT, so a tight loop written in Raku costs it roughly an order of magnitude more than it costs Rakudo — and some jobs, like tokenizing a 300 KB JSON document, are nothing *but* a tight loop.

An extension module is the escape hatch. A distribution ships C source alongside its Raku, the build step compiles it against a published ABI at install time, and the compiled routines become ordinary Raku subs. Perl calls this XS; Python calls it a C extension. The point is the same, and so is the most important property: **the module versions independently of the compiler.** A faster JSON parser should not require a new release of the language implementation.

32.1 Extension or NativeCall?

They solve different problems and the wrong choice is painful.

	NativeCall	extension module
calls	an **existing** C library you did not write	C **you** write for this module
data	C types: ints, nums, pointers, structs	Raku values: Hash, Array, Int, Rat, Str
build	none; the shared object already exists	compiled at install time

	NativeCall	extension module
use when	wrapping libcurl, sqlite3, libgit2	a hot loop in your own module is the bottleneck

If you need to return a *Hash of Rats*, NativeCall cannot express it and an extension can. If you need to call `curl_easy_perform`, use NativeCall.

32.2 The one rule

An extension never sees `Value`. Every Raku value crosses the boundary as an opaque handle.

```
/* src/rakupp_ext.h */
typedef struct RkValueOpaque* RkValue;
typedef struct RkCtxOpaque*   RkCtx;
```

This is not fastidiousness. `sizeof(Value)` moved from 392 to 376 to 344 bytes in a single afternoon of ordinary optimisation work, and the representation plan intends roughly 204 next. An ABI that exposed the struct would have to freeze the interpreter's internals forever, or silently miscompile every extension built against an older header — the failure mode where a module reads a `Str` out of a field that is now an `Int` and nothing crashes until much later.

Handles cost an indirection. They buy the ability to install a module today and keep it working across compiler releases, which is the entire point.

The corollary: **this is plain C.** No C++ types cross the boundary, no exceptions cross it, and nothing received needs freeing. A binding from Rust or Zig would need nothing beyond the header.

32.3 Hello, world

```
#include <rakupp/rakupp_ext.h>

static RkValue answer(RkCtx c) { return rk_int(c, 42); }

static const RkSubDef subs[] = { {"answer", answer}, {0, 0} };
static const RkModule mod = { RAKUPP_EXT_ABI, "Hello::Ext", subs };
```

```
RAKUPP_EXT_EXPORT const RkModule* rakupp_ext_init(unsigned host_abi) {
    return host_abi == RAKUPP_EXT_ABI ? &mod : 0;
}
```

```
use Rakupp::Ext;
rakupp-ext-load('./libhello.dylib');
say answer();      # 42
```

`rakupp-ext-load` installs each of the extension's subs into the **calling scope**, so inside a module they land exactly where an `our sub` would, and that module's `is export` carries them onward.

32.4 The value vocabulary

```
/* constructing */
RkValue rk_any  (RkCtx c);
RkValue rk_bool (RkCtx c, int truthy);
RkValue rk_int  (RkCtx c, long long v);
RkValue rk_int_s(RkCtx c, const char* decimal);       /* arbitrary precision */
RkValue rk_num  (RkCtx c, double v);
RkValue rk_rat_s(RkCtx c, const char* n, const char* d); /* a Rat, normalised */
RkValue rk_str  (RkCtx c, const char* utf8, size_t len); /* copied */
RkValue rk_array(RkCtx c);   void rk_push(RkCtx c, RkValue a, RkValue v);
RkValue rk_hash (RkCtx c);   void rk_set (RkCtx c, RkValue h,
                                         const char* k, size_t kl, RkValue v);
void    rk_list (RkCtx c, RkValue array);   /* mark it a List, not an Array */
void    rk_map  (RkCtx c, RkValue hash);    /* mark it a Map,  not a Hash  */
```

`rk_int_s` and `rk_rat_s` take **decimal strings** rather than integers on purpose: Raku's `Int` has no width, and a document may carry a 40-digit number that no C integer type can hold. `rk_rat_s("1", "7")` gives exactly one seventh, not `0.14285714285714285`.

```
/* inspecting */
RkType      rk_type   (RkCtx c, RkValue v);
long long   rk_int_get(RkCtx c, RkValue v);
const char* rk_str_get(RkCtx c, RkValue v, size_t* len);   /* borrowed */
size_t      rk_elems  (RkCtx c, RkValue v);
RkValue     rk_at_pos (RkCtx c, RkValue array, size_t i);
const char* rk_key_at (RkCtx c, RkValue hash, size_t i, size_t* keylen);
RkValue     rk_val_at (RkCtx c, RkValue hash, size_t i);
```

`RkType` is deliberately coarse:

```
typedef enum {
    RK_ANY = 0, RK_BOOL, RK_INT, RK_NUM, RK_RAT, RK_STR,
    RK_ARRAY, RK_HASH, RK_OTHER
} RkType;
```

It describes **what an extension can do with a value**, not where the value sits in Raku's type hierarchy. Anything the ABI has no vocabulary for arrives as `RK_OTHER`, and the advice is to stringify it. A coarse enum is a deliberate choice: a fine one would have to track the language's type system, which is exactly the coupling this design exists to avoid.

```
/* arguments */
size_t  rk_argc (RkCtx c);                    /* positionals only */
RkValue rk_arg  (RkCtx c, size_t i);          /* NULL if absent */
RkValue rk_named(RkCtx c, const char* name);  /* NULL if not passed */
```

`rk_named` returning `NULL` means *not passed*, which is distinct from being passed an undefined value — so `:immutable` and `:!immutable` remain distinguishable.

32.5 Lifetime: handles belong to the call

Everything an extension creates is allocated in the context's **arena** and released when the sub returns; the single value it returns is copied out first.

So an extension never frees anything, cannot leak, and — the rule that has to be stated because it is invisible until it bites — **a handle stored in a global and used on the next call is dangling.** There is no reference counting to save you. State that must persist across calls belongs in C, not in handles.

Borrowed pointers from `rk_str_get` and `rk_key_at` are valid until the call returns, which is long enough to copy out of them and no longer.

The arena is what makes the boundary safe in both directions: the host cannot be made to leak by a careless extension, and an extension cannot hold a reference into a value model that is free to change underneath it.

32.6 Errors

```
void rk_die(RkCtx c, const char* message);
```

Call it and **return `NULL`**. The host raises a Raku exception at the call site, catchable with `try`/`CATCH` like any other. A second `rk_die` keeps the first message, so an error deep in a recursive parse survives the unwind.

Never throw a C++ exception across the boundary. The host did not compile the extension's code and cannot catch it.

32.7 Versioning

```
#define RAKUPP_EXT_ABI 1u
typedef const RkModule* (*RkInitFn)(unsigned host_abi);
```

One integer, bumped whenever the meaning or order of anything in the header changes. The host looks up one symbol, `rakupp_ext_init`, and passes its own version. **Return `NULL` if you do not recognise it** rather than guessing; the host then reports a clean mismatch instead of calling through a wrong-shaped table.

All three failure modes report themselves and none corrupt:

```
'.../thing.dylib' was built for a different extension ABI (host is 1)
cannot load extension '/nope.dylib'
'/bin/ls' is not a rakupp extension (no rakupp_ext_init)
```

The host side is one function:

```
// src/Interpreter.h
Value extLoadModule(const std::string& path, std::string& errOut,
                    std::vector<std::pair<std::string, Value>>& subsOut);
```

which `dlopen`s the path, checks the ABI, and hands back the subs it declares.

32.8 Writing a module that also runs on Rakudo

A module that *only* works on Raku++ is a poor citizen. The pattern that works everywhere is a compiled fast path with a pure-Raku fallback, chosen at load time — and the load-bearing detail is how the loader is reached:

```
my &ext-load = try &::('rakupp-ext-load');
```

Why not call `rakupp-ext-load(...)` directly? Because Raku resolves sub names at *compile* time. On Rakudo the name does not exist, so the file fails to compile and the fallback never gets the chance to run.

`&::('...')` is a **symbolic lookup**: the name is a string, resolved at run time. The line is valid Raku either way, so the *same source file* compiles on both engines and simply finds nothing on Rakudo.

```
unit module My::Thing;

my &ext-load = try &::('rakupp-ext-load');
my &fast;

sub load-native(--> Bool) {
    return False unless &ext-load;
    for libraries() -> $lib {
        next unless $lib.IO.e;
        next unless try ext-load($lib.Str);
        &fast = try &::('my-thing-native');
        return True if &fast;
    }
    False
}
my Bool $native = load-native();

our sub do-the-thing($x) is export {
    $native ?? fast($x) !! pure-raku-version($x)
}
```

`use Rakupp::Ext` is accepted too and is the discoverable spelling for code that is Raku++-only by design — but it is a `use` statement, so writing it makes the file uncompilable on Rakudo.

Note that `&fast` and `&ext-load` keep their sigil at every mention. Writing `fast.defined` would *call* `fast` and ask the result, producing a baffling `No such method 'CALL-ME'`.

32.9 Packaging

```
My-Thing/
├── META6.json          "resources": [ "libraries/thing" ]
├── Build.rakumod       compiles src/ at install time
├── src/thing.c
├── lib/My/Thing.rakumod
└── resources/libraries/
```

Raku maps the logical resource name to the platform spelling, so **the build must emit `libthing.dylib`, not `thing.dylib`**, or `%?RESOURCES` will not find it.

`Build.rakumod` should **never abort an install**. If the compiler is missing, or the headers are not there, or the engine is Rakudo, warn and return true: the module then runs on its fallback. A build hook that fails turns an optimisation into a hard dependency.

Find the library through `%?RESOURCES`, falling back to a working-directory path. **Do not derive it from `$?FILE`** — under Raku++ a module's `$?FILE` is the *main program's* path, not the module's, so anything computed from it lands somewhere unrelated. That is a recorded bug, not a design.

32.10 The naming convention

A module whose behaviour depends on Raku++ should say so in its name. `Rakupp::JSON` tells a reader at the point of `use` that this is not a portable choice — better than a footnote in a README, and better than discovering it when the deploy target turns out to be Rakudo.

Two rules follow. A `Rakupp::` module is still a normal distribution: installed by zef, carrying a `META6.json`, degrading gracefully elsewhere. And **the compiler must not special-case a module name.** An early draft of `Rakupp::JSON` was built into the interpreter, and `use Rakupp::JSON` intercepted the name before module loading — so the real distribution's tests silently exercised the built-in copy instead of the extension they were written for. If a name can be a module, it must *only* be a module.

32.11 Current limits

- **There is a per-value cost.** Each value crosses as an arena-allocated handle and is then copied into its container — roughly one extra `Value` copy per node. Measured on `Rakupp::JSON`: 5.7 ms against 2.7 ms for the same parser compiled into the interpreter. Still 6.6 times faster than Rakudo on that workload, but it means an extension is worth it for **bulk** work, not for a routine called once with two integers.
- **Hash iteration is index-based**, so `rk_key_at` is O(i) and walking a whole hash is quadratic. A cursor is the obvious version-2 addition, and it is why

`Rakupp::JSON` implements only the parser natively and leaves serialisation in Raku.

- **No `--exe` bundling.** A program using a native extension cannot be compiled into a standalone binary; the shared library is loaded at run time and is not part of the module graph the bundler walks.
- **One context per call, single-threaded within it.** Handles are not safe to pass between threads.

32.12 The other direction

Extensions are **native code called from Raku**. Embedding is **Raku called from native code**. Same boundary, opposite direction — and the observation that shapes the current plan is that the harder half is already built:

> The extension ABI has the value vocabulary, the opaque-handle discipline, version negotiation, an error model, and the rule that makes all of it survivable.

So an embedding API does not get a second value vocabulary. `RkValue` is the value type, `rk_int/rk_str/rk_hash/rk_at_pos` are the accessors, and an earlier proposal for a separate opaque type with its own free function was retracted on exactly that ground.

That is worth stating as a design principle rather than a project note: **when you publish a boundary, publish one.** Two vocabularies for the same values is twice the surface to keep stable, and the whole reason for the handle discipline was to have as little of that surface as possible.

Chapter 33

Concurrency

Raku has a full concurrency surface: `start`, `await`, `Promise`, `Channel`, `Supply`, `react`/`whenever`, `Lock`, atomics, and a scheduler. Raku++ implements it on real OS threads with a **global interpreter lock**, plus an opt-in mode that switches the lock off.

This chapter is the story of getting there in stages, because the stages are still visible in the code and each solved a specific class of bug.

33.1 Stage 1: per-thread execution registers

The state that belongs to one thread of Raku execution — its current lexical scope, the dynamic-variable chain, the recursion depth, the gather and supply collectors — was originally interpreter members. It is now a struct:

```
// src/Interpreter.h
struct ExecContext { /* Chapter 12 */ };
static thread_local ExecContext tctx_;
void saveCtx(ExecContext& c);
void loadCtx(ExecContext& c);
```

Being `static thread_local`, each real worker thread owns its own set, so interpreter execution needs no per-handover register swap. The save and load functions remain because the design allows a parked thread's registers to be stashed and restored; today they merely shuffle within a thread's own copy.

The same treatment was applied, thread by thread, to everything ThreadSanitizer reported:

```
// src/Interpreter.h
static thread_local Value* topicWriteback_;
static thread_local bool noAutothread_;
static thread_local int loopPhaserCtl_;
static thread_local std::vector<RedispatchCtx> redispatchStack_;
static thread_local std::vector<std::shared_ptr<ReactCtx>> reactStack_;
static thread_local bool t_isWorker;
static thread_local Value t_threadSelf;
```

The comment on the call registers records the scale: as plain members they were written by every call on every thread, and were TSan's top report — **2,761 lines on a program with no sharing in it at all**.

One member deliberately stayed shared, and the reasoning is a good example of measuring rather than assuming:

```
// src/Interpreter.h — the current source line, for test diagnostics
// Written on EVERY statement, so a thread_local costs a TLV lookup per
// statement on macOS — measured +6% on loopsum. A relaxed atomic member is
// a plain store, defined under concurrency, and TSan-clean; the value being
// process-wide is the same arbitrariness diagnostics always had.
struct RelaxedLine {
    std::atomic<int> v{0};
    void operator=(int x) { v.store(x, std::memory_order_relaxed); }
    operator int() const { return v.load(std::memory_order_relaxed); }
} curLine_;
```

Thread-local storage is not free on every platform, and a diagnostic field does not need to be exactly right.

33.2 Stage 2: the symbol-table freeze

The shared symbol tables — the class registry, the global environment, the named regex table, the loaded-module set, each class's method map — are mutated freely while the program is single-threaded. Once concurrency engages they must be treated as immutable, so worker threads can read them without a lock.

```
// src/Interpreter.h
std::atomic<bool> symbolsFrozen_{false};
void noteSymbolMutation(const char* what);
```

The flag flips when concurrency engages, and `noteSymbolMutation` is a tripwire wired into every structural writer. Under `RAKUPP_FREEZE_TRACE` it reports any post-freeze mutation and which thread did it.

That is the interesting part: the tripwire is **empirical evidence** for whether lock-free reads are safe, rather than an argument that they are. Behaviour is otherwise unchanged, so the instrumentation costs nothing in a normal build.

33.3 Stage 3: the GIL

```
// src/Interpreter.h
std::mutex gil_;          // held while running Raku
bool gilHeld_ = false;    // engaged once any thread is spawned
void engageGil();         // lazily lock on first async use
```

Only the holder may touch interpreter state. A thread drops it while blocked — in `await`, in a sleep, around a blocking syscall — so another can run.

The lock is **lazy**: a program that never spawns a thread never engages it and pays nothing.

Three operations manage the handover:

```
// src/Interpreter.h
void gilYieldNotify();              // unlock + bump a counter + notify
void yieldToWorker();               // drop the GIL until a worker progresses
bool yieldToWorkerFor(double secs); // …bounded
void sleepYield(double secs);       // sleep with the GIL released
bool gilPark();                     // release around a blocking syscall
void gilUnpark(bool wasParked);
```

`gilPark` has a strict contract, stated in the header: the parked window must touch no interpreter state — only thread-local buffers and syscalls. That is what lets a child-process wait release the lock so sibling workers can spawn their own children concurrently.

33.3.1 Safe points

A background worker doing pure compute has no I/O to yield at, so the interpreter weaves a cheap check into its hot loop:

```
// src/Interpreter.h
inline void safePoint() {
```

```
    if (!t_isWorker) return;                // main-thread loops never park
    if (workerAbort_.load(std::memory_order_relaxed)) throw WorkerAbortEx{};
    if (++t_safePtCtr >= 4096) { t_safePtCtr = 0; workerYield(); }
}
```

Two jobs in four lines. It periodically hands the GIL back, so a compute-bound worker cannot starve the main thread. And it unwinds a worker whose result is no longer wanted, at shutdown, by throwing an exception that is deliberately **not** a `RakuError` — so a user's `CATCH` cannot swallow a shutdown.

The main-thread early return means the check is one predicted branch on a thread-local bool for every loop that is not in a worker.

33.4 Stage 3a: true parallelism, opt in

```
// src/Interpreter.h
bool parallelMode_ = false;   // RAKUPP_PARALLEL
std::mutex sharedMut_;
std::atomic<int> liveWorkers_{0};
```

With `RAKUPP_PARALLEL`, worker threads run interpreter compute concurrently instead of serialising on the GIL — safe now that the registers are thread-local and the symbol tables freeze. The default is off, so **the GIL path is byte-for-byte unchanged**.

The few genuinely shared internals a parallel worker can still touch — the test counters and TAP output, the worker vectors — are guarded by one mutex. User data mutated without a `Lock` is the user's race, as it is in Rakudo.

For user containers there is a striped lock, and its design is the interesting part:

```
// src/Interpreter.h
struct ParStripe {
    std::unique_lock<std::recursive_mutex> l;
    ParStripe(const Interpreter& I, const void* p) {
        if (I.parallelMode_ &&
            I.liveWorkers_.load(std::memory_order_relaxed) > 0)
            l = std::unique_lock<std::recursive_mutex>(atomicStripe(p));
    }
};
```

Two conditions, not one. Parallel mode **and** live workers — because before the first spawn and after the last join, a single thread cannot race itself. A single-threaded program under `RAKUPP_PARALLEL` therefore pays two predicted branches and nothing else.

That second condition was not an optimisation for its own sake: the unconditional stripe tax was pushing compute-heavy Roast files past the timeout, in files with zero threads in them. The rule it encodes — **the machinery must be free when one thread runs** — is the gate every stage of this work had to pass.

33.5 Threads need big stacks

The tree walker recurses deeply, and the default stack for a non-main thread on macOS is 512 KB — a recursive Raku sub inside `start {...}` overflows it within about a hundred frames, producing a bus error and then a process that will not die at exit.

```
// src/Interpreter.h — BigStackThread
const size_t kStack = (size_t)256 << 20;   // 256 MiB, virtual
pthread_attr_setstacksize(&attr, kStack);
if (pthread_create(&h_, &attr, entry, fn) == 0) joinable_ = true;
else { std::unique_ptr<Fn> g(fn); g->f(); }   // creation failed: run inline
```

The reservation is virtual and committed only as used. Windows gets the same through a small shim kept in `Runtime.cpp` so `<windows.h>` stays out of the widely-included header. If thread creation fails, the work runs **inline** rather than being lost.

33.6 Worker bookkeeping, and two real crashes

```
// src/Interpreter.h
struct WorkerSlot {
    BigStackThread th; std::shared_ptr<std::atomic<bool>> done;
};
std::vector<WorkerSlot> workers_;
void addWorker(BigStackThread&& th, std::shared_ptr<std::atomic<bool>> fin);
```

`addWorker` is the **only** safe way to register a worker, and the header explains why in terms of a crash report. In parallel mode spawns happen from worker threads too — a `start` block tapping a supply spawns the interval ticker — and the old

per-site “reap, then push” pattern mutated the vector from several threads at once. `vector::erase` corrupted it and the process segfaulted, with the report naming `erase` inside a supply-interval spawn on thread 5 of 292.

The second crash is inside the fix:

```
// Collect finished slots under the lock, JOIN THEM OUTSIDE IT. Joining
// under sharedMut_ deadlocked: the joinee had set its done flag but was
// still unwinding through code that itself wants sharedMut_ (a worker
// spawning a nested worker) — the reaper held the lock waiting for the
// joinee, the joinee waited for the lock.
```

That is the classic shape — hold a lock across a join — and it is worth remembering that the *first* fix for a concurrency bug introduced it.

There is also backpressure, parallel mode only:

```
void throttleSpawn() {
    if (!parallelMode_) return;
    while (liveWorkers_.load(std::memory_order_relaxed) >= 384)
        { /* reap finished workers, wait for the herd to thin */ }
}
```

A tap-and-close loop over interval supplies — four spawners times a thousand activations, each a real thread with a 256 MiB virtual stack — outran teardown and exhausted the address space. Above the cap a spawner reaps and waits.

Under the GIL this must stay off, and the reason is a nice illustration of how the two modes differ: a spawner spinning while *holding* the lock would keep the very workers it waits for from ever finishing.

33.7 The user-visible layer

Type	Backing
`Promise`	`PromiseState` in `ext`: mutex, condition variable, result or cause, and a list of `.then` continuations fired once on settle
`Channel`	shared state plus a stripe from the atomic pool

Type	Backing
`Supply`	on-demand blocks run through a `SupplyTapCtx`; live suppliers register a tap record
`react/whenever`	a `ReactCtx` with an event queue, a live-source count, and its own mutex and condition variable
`Lock`	a `std::recursive_mutex`, recursive because `protect` re-enters
`Thread`	a `BigStackThread` plus a per-thread identity value
`$*SCHEDULER.cue`	a worker with a `CueState` cancellation flag

Two details in there earn a mention.

A tap's teardown lives on the context that owns it. An earlier design kept a close-callback stack as an interpreter member, which was shared across worker threads — two concurrent `react` blocks corrupted it. Making it thread-local fixed the crash and cost 6% on an unrelated benchmark by reshuffling thread-local storage layout on macOS. The right answer was neither: the context object already travels with the block, is already thread-correct, and is free.

`whenever` activations are deferred. Rakudo runs the `react` body first and only then activates subscriptions, so a `say` after a `whenever` prints before the first emitted value. The synchronous drains queue into `ReactCtx::deferred`, which the `react` implementation runs after the body.

33.8 Signals, and a thing that must be turned off

Signal supplies use the self-pipe trick: a handler writes a byte, a lazily spawned dispatcher thread reads it and runs the `whenever` block under the GIL. When the `react` block ends, externally wired taps must be closed explicitly, or the dispatcher keeps firing the handler after the block is gone — the symptom was a second Ctrl-C re-invoking a stop method on an already-stopped service.

Separately, and simply: **`SIGPIPE` is ignored process-wide.** Without that, a TCP server dies when a client disconnects mid-write. It is set once on the big-stack entry thread.

33.9 Testing it

Concurrency bugs do not appear in unit tests. What found the ones in this chapter:

- **ThreadSanitizer** over the stress suite, which drove the thread-local work;
- **AddressSanitizer** for the lifetime bugs;
- **a stress suite designed to exhaust something** — a thousand tap-and-close activations across four spawners is not a realistic program, and that is the point;
- **crash reports read carefully.** "erase inside a supply-interval spawn on thread 5 of 292" named the bug precisely;
- **`sample` on a hung process.** One apparent hang in an async test turned out to be an unrelated pre-existing bug in a supply transform chain, found by sampling the stuck process rather than by reasoning about the test.

33.10 Honest limitations

- **The GIL is the default.** Parallel mode is opt-in and, while the stress suite passes under it, it is newer.
- **User data races are the user's.** As in Rakudo, mutating shared data without a `Lock` is undefined; the striped locks protect the interpreter's own structural invariants, not the user's semantics.
- **The symbol freeze is enforced by a tripwire, not by the type system.** A new structural writer that forgets to call `noteSymbolMutation` is invisible.
- **Worker stacks are large.** 256 MiB of virtual address space each is cheap but not free, and it is why the spawn throttle exists.

Part IX

Around the Compiler

Chapter 34

Tooling Built on the AST

Once a program is a tree, several useful things become short. This chapter covers the five tools that are built on the front end rather than on the runtime, and one that is built on the call path.

34.1 `--lint`: static analysis that runs nothing

```
// src/Lint.h
struct LintFinding {
    int line = 0;
    char severity = 'W';        // 'W' warning, 'N' note
    std::string rule;           // a stable id, e.g. "unused-variable"
    std::string message;
};
std::vector<LintFinding> lintProgram(Program& prog);
```

It walks an already-parsed tree and reports findings sorted by line and rule. It does not execute the program.

The design constraint is stated in the header and is the whole reason the rule set is small:

> Rules are deliberately conservative — a missed warning is acceptable, a false one is not.

Raku's dynamism is why. Interpolation, dynamic variables, `EVAL`, symbolic references and introspection all mean that a variable which *looks* unused may be reached

by name at run time. An over-eager analyser in this language produces warnings people learn to ignore, at which point the tool is worse than nothing.

Every finding carries a **machine-readable rule id** as well as prose, so a project can suppress or gate on a specific rule without matching message text.

34.2 `--highlight`: colouring with the compiler's own knowledge

```
// src/Highlight.h
std::string highlight(const std::string& source, const std::string& format);
```

Two renderers: `html` produces the same CSS token classes Pygments uses, so existing stylesheets apply unchanged; `ansi` produces terminal escapes.

The class names are borrowed, but the classification is not. A regex-based highlighter has to guess; this one knows. `$obj.role` is coloured as a method call rather than as the keyword `role`, because the lexer and parser already established which it is.

That is the general advantage of building a highlighter inside the compiler, and it is why the output is worth the coupling.

34.3 `--profile`: a routine-level wall-time profiler

```
// src/Profiler.h
namespace prof {
    extern bool on;              // read on the hot path
    void setDest(const std::string& dest);
    void enter(const void* key, const char* name, const char* file);
    void leave();
    void report();
}
```

Two hooks — `callCallableRaw` for routines with a frame boundary, and `invokeMethod` — feed a per-thread shadow stack. Per routine it aggregates call count, inclusive and exclusive wall time.

Three decisions worth naming.

Builtins are attributed to their caller, because the hooks are on *user code* routine entry rather than on the builtin dispatch chain. That is a deliberate simplification: it keeps the hot path to one branch, and a profile that says "this routine spent 40% of its time in built-ins" is usually the answer you wanted anyway.

Recursion is handled the standard way: only the outermost active frame of a routine adds to its inclusive time, so a recursive function's inclusive figure does not count the same seconds many times.

The off-cost is one predicted branch on a plain `bool` per call, measured at zero against the noise band before it was built. A profiler that cannot be compiled in unconditionally is a profiler people forget to use.

The destination follows Rakudo's convention — the extension selects the format: – writes a table to standard error, a path writes the table there, and a path ending in `.json` writes a machine-readable dump.

One portability note is recorded in the header: everything in it is portable C++17, with no `__builtin_expect` and no attributes, because the prototype's GCC-isms broke the MSVC build once already.

34.4 The REPL

```
// src/Repl.h
int rakuppRepl(const std::string& exePath,
               const std::vector<std::string>& libPaths);
bool stdinIsTerminal();
bool replForced();
```

The REPL is entered **only** by a bare `rakupp` attached to a terminal. Anything arriving on a pipe or a redirect is a complete program and keeps running as one; the terminal test in `main` is the whole of that decision.

It lives in the executable rather than the runtime library, so nothing about an interactive session is linked into the binaries the compiling modes produce.

A REPL never calls `run()`. It keeps **one** `Interpreter` alive and feeds it `evalString` per line, so the mainline scope *is* the session. Two functions cover what `run()` would otherwise have done at either end:

```
// src/Interpreter.h
void replStart(std::vector<std::string> args);  // define @*ARGS, arm `state`
```

```
void replFinish();                              // run END phasers, once
std::vector<std::string> replNames() const;     // for tab completion
```

The nicest detail is how an incomplete line is recognised. A parse that dies on end-of-input is a request for more input, not a syntax error:

```
// src/Parser.h — ParseError
bool atEof = false;
```

```
// src/Interpreter.h
Value evalString(const std::string& src, bool mainlinePH = false,
                 bool* incompleteOut = nullptr);
```

The flag is set by the lexer's runaway-construct diagnostics and by the parser when it fails on the end token, so typing `sub f {` at the prompt asks for a continuation line while `sub f )` reports an error. One boolean, threaded from the lexer to the prompt.

34.5 Pod and `--doc`

```
// src/Pod.h
std::vector<Value> parsePod(const std::string& src);
```

Pod blocks are parsed into the `$=pod` DOM — a list of `Pod::Block` values, each a `Hash` tagged `"Pod"` with a class, a name, a level, a configuration and contents. Delimited (`=begin`/`=end`), paragraph (`=for`) and abbreviated (`=head1 …`) forms, nested blocks, and whitespace-collapsed paragraphs.

Note the representation: a Pod block is not a new `VT` or a C++ class. It is a tagged `Hash`, which is the same technique `Set`, `Proxy` and `DateTime` use (Chapter 7). The Raku program manipulating it sees an ordinary object.

Declarator documentation — `#|` above a declaration, `#=` beside or below it — is collected by the lexer keyed by line and attached by the parser, answering `.WHY` on routines, classes and attributes.

`--doc` runs `DOC` phasers and prints the rendered content, and `=finish` data travels from the lexer to `$=finish`.

34.6 --dump-ast, and the tool built on it

```
rakupp --dump-ast prog.raku
RAKUPP_DUMPTOKENS=1 rakupp prog.raku
```

`AstDump.cpp` prints the tree as an indented outline. It is the first thing to reach for when a parse produces something unexpected.

It is also an **interface**, not just a debugging aid. `tools/ast-opportunity.raku` reads its output and counts syntactic patterns across a corpus — which is the tool that measured, in two minutes, that constant folding had almost nothing to fold in real Raku and that operand shapes were 25 per thousand nodes (Chapter 18). A textual tree dump turned out to be a perfectly good static analysis substrate.

34.7 The tools that are not in the binary

A standing rule of the project: **build the ecosystem tooling in Raku, run it with rakupp.** Not because it is elegant, but because it is the largest available body of real-program testing.

Tool	What it does
`tools/run-roast.raku`	the Roast harness
`tools/run-bench.raku`	the benchmark harness, three engines
`tools/perf-guard.raku`	the release performance gate
`tools/run-optbench.raku`	compiles each optimiser showcase twice, checks byte-identical output, then times
`tools/gen-unicode.raku`	generates Unicode tables
`tools/ast-opportunity.raku`	counts AST patterns
`tools/doc-examples-diff.raku`	runs every documented example on both engines
`tools/recheck-divergences.raku`	re-tests the recorded divergence list

Each of these is a substantial Raku program that runs on every release. When one of them breaks, it has usually found a bug in the compiler rather than in itself — which is exactly the point of the rule.

The showcase programs go further. Interpreters for JavaScript, Perl 5, Python 3, Lisp and Forth, each written in Raku, each producing byte-identical output to the real implementation on its examples. Writing one is the single most productive bug-finding activity in the project's history: each of them, on first completion, yielded a list of genuine defects — grammar mis-parses, a lost `return`, dynamic-variable restore, slip flattening, precedence traps.

An interpreter for another language exercises grammars, recursion, closures, string handling and dispatch simultaneously and at a scale no unit test reaches. It is a test suite that writes itself, and it has an oracle: the language it implements already has one.

Chapter 35

Measuring, and Proving

Almost every decision in this book was made against a number. This closing chapter is about how those numbers are produced, why several of them were wrong the first time, and what a release has to pass.

35.1 The benchmark policy

Stated once, applied everywhere:

- **interleave the variants.** Run A, B, A, B — not all of A then all of B — so machine drift and thermal state hit both equally.
- **discard a warm-up**, then report the median or the minimum of several repetitions. Which one depends on the noise shape; the important thing is to say which.
- **keep a control kernel** that the change under test cannot possibly affect.
- **state the conditions**: machine, architecture, compiler, date.

The control is the part people skip and the part that carries the argument. Chapter 18's table has a row for a pure method-dispatch loop, containing nothing node specialisation can touch:

kernel	base	specialised	delta
vars	840.8 ms	686.5	−18.3%
fib	478.6	422.4	−11.7%
ctl — method dispatch only	327.9	327.0	**−0.3%**

Without that last row the others are only evidence that the machine was quieter the second time.

35.2 The performance gate

```
build/rakupp tools/perf-guard.raku --check
```

`perf-guard` compares the current binary against a recorded baseline and **fails** on a regression. The release checklist gates on it.

That it exists at all is a lesson learned twice: a release has shipped with a performance regression because someone eyeballed the numbers and thought they looked fine. Eyeballing does not work — the noise band is a few per cent, the regressions that matter are often a few per cent, and human judgement about which is which is unreliable at exactly that scale.

The rule that follows: **a performance change is an interleaved A/B against `perf-guard`, not an opinion.**

35.3 The correctness gates

Gate	What it is
Roast	the Raku specification suite, run by a Raku harness
the local suite	examples and showcases, byte-compared against golden output
regression tests	one file per fixed bug
stress tests	concurrency and memory, also under TSan and ASan
compiler agreement	every deterministic example must produce identical output interpreted, `--exe`, and `--exe -O`
the second FFI leg	the whole suite run again with `RAKUPP_FFI=0`

Compiler agreement is the one that catches the most. Three execution paths must produce **byte-identical** output for the same program; a divergence is a bug in one of them, and it is found automatically rather than by someone noticing.

Roast is reported two ways, and the difference matters. **Files fully passing** is an all-or-nothing bar; **assertions passing** gives partial credit. Roughly ninety per cent of declared assertions pass. Both numbers are published, because either alone is misleading — a file can fail on one obscure assertion out of two hundred, and a file can "pass" by skipping everything.

35.4 Oracles

A test needs something to be right *against*. Four are used, in decreasing order of authority.

Roast. The specification. If Roast asserts it, it is the answer.

Rakudo. For anything Roast does not cover, the reference implementation is run side by side. The documentation harness does this in bulk: every documented example, on both engines, three-way classified.

The previous rakupp binary. For a change that is supposed to be *semantically invisible* — an optimisation — the baseline binary is the oracle, not another implementation. That habit came directly from a bug: the first `evalIndex` fast path wrapped negative subscripts from the end, and `my $i = -1; @a[$i]` must throw. Rakudo could not have caught it, because Rakudo rejects that spelling at compile time. Diffing against the previous binary did.

The language being implemented. The showcase interpreters compare their output against `node`, `perl`, `python3` and the real Lisp — which is an oracle for thousands of lines of Raku that nobody wrote assertions for.

35.5 Error messages are not behaviour

A rule with a sharp edge: **do not copy Rakudo's message prose unless Roast or the documentation asserts it.**

Behaviour must match. Message wording need not, and chasing it produces churn that no test protects. Where a diagnostic *is* asserted, it is asserted by exception class and attributes rather than by text — which is why typed exceptions are built with attributes rather than formatted strings (Chapter 14).

35.6 What has and has not paid

The pattern across every optimisation in this book is consistent enough to state as a finding.

What paid — all of it removing work or allocation:

Change	Effect
copy-on-write strings	removed a quadratic; 142 ms to 24 ms on a copy benchmark
the property cache on the string body	removed the *other* quadratic
interned tag fields	removed a constructor, destructor and copy from every `Value`
the packed-prefix name comparison	60% of a profile to 8.5%
`strtod` instead of a caught `stod`	removed a C++ throw from the hottest path
node specialisation	removed a `Value` copy and a literal rebuild per evaluation
direct-arity calls	removed the per-call `ValueList`
native integer lanes	removed the per-operation `Value`
the key-once sort	removed an asymptotic factor

What did not pay:

- **constant folding** — measured first, with a corpus of 51,353 nodes; 0.07% of nodes, none of it in a loop. Not built.
- **link-time optimisation and `-mcpu=native`** — measured, no effect.
- **`-Os`** — smaller by nothing that matters, slower by 20 to 50% of the lane speed-up.
- **a dispatch table for the method ladder** — would chase 8.5% of a profile, and is not a drop-in because the ladder is guarded and order-sensitive.

The generalisation: **on this codebase, removing an allocation has always paid and removing a branch almost never has.** That is a property of a tree-walker over a fat value type, and it is worth re-deriving before assuming it holds somewhere else.

35.7 Three ways the measurement itself was wrong

Worth more than the successes.

Benchmarking a rename instead of the change it enables. An early analysis concluded that turning a builtin into a named C++ function was worth about 1 nanosecond per call. That was true for an out-of-line function still taking a `ValueL–ist` — and completely missed what a real named function unlocks: direct `Value` arguments, no allocation, and an inlinable body. The corrected measurement was 5.6 times on an `abs` loop (Chapter 27).

Restructuring the shared path while adding a fast one. The first node specialisation made the *control* 5.7% slower, because binding an operand through an optional cost the general path its copy elision. The rule extracted: add an early exit, never restructure the code underneath it.

Assuming where the cost was. "I/O dominates, so call plumbing does not matter" was false: a buffered one-argument `say` costs about 125 nanoseconds, of which the plumbing was 45%. And the method dispatch ladder was exonerated twice by a correct observation — that a method 177 comparisons in cost the same as one 812 in — before a profiler showed the cost was `strlen` on a literal, not the ladder's length.

35.8 Where the remaining time is

For an interpreted method-heavy loop, after everything above:

	share
heap allocate and free	31%
`Value` copy and destroy	11%
method-name comparison	8.5%
the dispatch function's own body	6.1%

Forty-two per cent in allocation and value churn. The shape of the fix is known — pass the invocant and argument list by reference rather than by value, and shrink `Value` — and both are being approached carefully rather than quickly, because the first trades away an accidental safety property and the second is a representation change that the extension ABI was specifically designed to survive (Chapter 32).

35.9 Honesty as a practice

Every chapter in this book has a "honest limitations" section, and that is a deliberate convention rather than a stylistic tic.

The list for the project as a whole: about ninety per cent of Roast; depth-first method resolution rather than C3; no ambiguity error in multiple dispatch; role composition that is last-writer-wins; modules that publish their whole environment; a flat class registry; no macros, no `RakuAST`, no slangs; laziness capped at a million elements; a `gather` block that can run more than once; a reference cycle that leaks.

Every one of those is a real difference from the reference implementation, and every one is written down where someone hitting it would look. That is the difference between a document and a brochure, and it is the reason the divergence lists in `docs/dev/findings/` are maintained as running logs rather than occasionally cleaned up.

A last observation, which is the closest thing this book has to a thesis. The mechanisms in it are mostly ordinary — a Pratt parser, a tagged struct, a backtracking matcher, a transpiler. What made them work was not cleverness. It was measuring before building, keeping a control, writing down the reason next to the code, and being specific in public about what does not work yet.

Appendix A

The Tag Sets

The three closed enumerations everything else switches on.

A.1 `VT` — runtime value tags

`src/Value.h`. Eighteen tags. Several Raku types are an existing tag plus a marker rather than a tag of their own; those are listed in the second table.

Tag	Live fields	Notes
`Nil`	—	the absent value
`Any`	—	the default undefined value
`Bool`	`b`	
`Int`	`i`, or `big` when it overflows	`enumName`/`enumType` make it an enum value
`Num`	`n`	
`Str`	`s`	
`Array`	`arr`, `isList`, `itemized`, `shape`, `ext`	List, Seq, Junction and lazy sequences all live here
`Hash`	`hash`, `hashKind`, `objKeyed`	Set, Bag, Mix, Proxy, Failure, Date and more

Tag	Live fields	Notes
`Code`	`code`	sub, block, method, builtin, WhateverCode
`Range`	`rFrom`, `rTo`, `rExFrom`, `rExTo`, `rNum`, `ext`	`ext` holds the original endpoint objects
`Pair`	`s` or `pairKey`, `pairVal`, `namedArg`	
`Type`	`s`	a type object; `s` is the type name
`Whatever`	—	*
`Object`	`obj`	a user class instance
`Rat`	`ratN`, `ratD`, `fatRat`	normalised, denominator positive
`Regex`	`s` (pattern), `hashKind` (flags)	
`Match`	`s`, `rFrom`, `rTo`, `arr`, `hash`	five fields live at once
`Complex`	`n` (real), `im` (imaginary)	

A.1.1 Types with no tag of their own

Raku type	Representation
`List`, `Seq`	`Array` with `isList`
an itemized array	`Array` with `itemized`
`Junction`	`Array` with `enumName` in `any/all/one/none`
a lazy or infinite sequence	`Array` with a `LazySeqState` in `ext`
`Set`, `Bag`, `Mix`, and their mutable forms	`Hash` with `hashKind`
`Proxy`	`Hash` with `hashKind == "Proxy"`, holding `FETCH/STORE`
`Failure`	`Hash` with `hashKind == "Failure"`
`Date`, `DateTime`	`Hash` with `hashKind`

Raku type	Representation
Buf	Str with hashKind == "Buf" and an identity token in ext
Blob	Str with hashKind, compared by value
Instant, Duration	Num with hashKind and an identity token
IntStr, RatStr, NumStr, ComplexStr	the numeric tag plus s and hashKind
an enum value	Int with enumName and enumType
FatRat	Rat with fatRat
a native integer container	any numeric tag with natBits and natSigned
Promise, Channel, Supply, Tap, Lock	a tag plus state in ext

A.2 NK — AST node kinds

src/Ast.h. Forty-nine kinds.

Expressions. IntLit, NumLit, StrLit, InterpStr, BoolLit, VarExpr, ListExpr, Assign, Binary, Unary, Call, MethodCall, Index, Ternary, Range, Pair, BlockExpr, ArrayLit, HashLit, NameTerm, RegexLit, SubstLit, ChainExpr, SymbolicRef, AllomorphLit, NqpOp, Whatever, SelfTerm.

Statements. ExprStmt, VarDecl, SubDecl, IfStmt, WhileStmt, ForStmt, LoopStmt, Block, ReturnStmt, LastStmt, NextStmt, RedoStmt, UseStmt, EmptyStmt, GivenStmt, WhenStmt, RepeatStmt, ClassDecl, EnumDecl, NamedRegexDecl, SubsetDecl.

A great deal of Raku's surface syntax lowers into a few of these. Every operator — built-in, meta, hyper, set, and every user-declared one — is a Binary, Unary or Call. class, role, grammar, module and package are all ClassDecl. A phaser is a Block with a name.

A.3 K — regex node kinds

src/Regex.h. Nineteen kinds.

Kind	Construct
`Lit`	a literal string
`Any`	`.`
`Class`	`<[…]>`, `\d`, `<:Nd>`, a named class
`Seq`	concatenation
`Alt`	`|` (ranked) and `||` (first match)
`Conj`	`&`
`Rep`	`*`, `+`, `?`, `** N..M`, with `%` separators
`Group`	`[ ]`, `( )`, `$<x>=[…]`
`AnchorStart`, `AnchorEnd`	`^`, `$`, `^^`, `$$`
`WBLeft`, `WBRight`	`<<`, `>>` word boundaries
`Nop`	an empty branch
`Subrule`	`<name>`, `<.name>`, `<alias=rule>`, `<r($x)>`
`Look`	`<?…>`, `<!…>`, `<?after …>`
`Code`	`{…}`, `<?{…}>`, `<!{…}>`, `:my`
`VarMatch`	a `$var` atom evaluated at match time
`CapStart`, `CapEnd`	`<(` and `)>`

A.4 `NqpOpc` — the `use nqp` subset

`src/Ast.h`. About fifty operation codes in seven groups: lazy control forms, 64-bit integer operations, codepoint-indexed string operations, list and hash primitives, object and attribute helpers, buffer read and write, and folded constants. The full list is in Chapter 30.

A.5 Exception and control structs

`src/Interpreter.h`.

Struct	Meaning
`RakuError`	every user-visible Raku exception: payload plus message
`ReturnEx`	`return` across a callable boundary

Struct	Meaning
`LastEx`, `NextEx`, `RedoEx`	labelled or cross-frame loop control
`BreakGivenEx`	`when` / `succeed` across a boundary
`ProceedEx`	`proceed`
`LeaveEx`	`leave`
`ResumeEx`	`.resume` inside a `CATCH`
`DoneEx`	`done` in a supply, whenever or react body
`StopGatherEx`	a lazy `gather` has produced enough
`ExitEx`	`exit`
`WorkerAbortEx`	unwind a background worker at shutdown
	— deliberately **not** a `RakuError`
`ParseError`	a compile-time diagnostic, optionally typed
`CodegenError`	`--exe` cannot transpile this; bundle instead
`AstEmitError`	`--aot` cannot rebuild this; bundle instead
`AstSerialError`	a cache entry is unreadable; parse instead
`StepLimitExceeded`	a regex exceeded its backtracking budget
`ObsoleteEscape`	a retired Perl 5 metacharacter in a pattern

Appendix B

Flags and Environment Variables

Everything that changes what the compiler does from the outside. This is a reference for the internals; the user-facing command line is documented in `docs/guide/CLI.md`.

B.1 Command-line flags that select a mode

Flag	Effect
(none)	lex, parse, interpret
`-e 'code'`	interpret the argument
`--bundle`	embed the source bytes in a standalone binary
`--aot`	parse at build time, emit C++ that rebuilds the AST
`--exe`	transpile to C++ and compile natively
`--cpp`, `--emit-cpp`	print the generated C++ instead of compiling it
`-O`, `-O2`, `-O3`, `-Os`, `-O0`	run the codegen optimizer; the level is forwarded to the C++ compiler

B.2 Inspection and tooling flags

Flag	Effect
`--ast`, `--dump-ast`	print the parsed tree as an indented outline
`--ast-roundtrip`	serialise and deserialise the tree, then run it — the cache format's own test
`--lint`	static analysis; does not run the program
`--highlight`	syntax-highlight the source
`--html`, `--ansi`, `--terminal`	pick the highlighter's renderer
`--doc`	run `DOC` phasers and print rendered Pod
`--profile[=dest]`	the routine-level wall-time profiler
`--ffi-info`	which FFI backend is live, or why none is
`--precomp-info`	what the parse cache holds
`--precomp-clean`	empty it
`--precomp-modules=on|off`, `--precomp-files=on|off`	the two cache switches
`--version`, `--help`, `--quiet`	as expected

Flags are position-independent, and the perl-compatible one-liner family (`-n`, `-p`, `-a`, `-F`, `-i`, `-0777`, `-M`) is documented in the CLI guide.

B.3 Environment variables

B.3.1 Search and loading

Variable	Effect
`RAKULIB`	extra module search paths; **both `,` and `:`** separate
`ROAST`	adds the specification suite's test-helper library

Variable	Effect
`RAKUPP_HOME`	where the binary considers itself installed
`RAKUPP_CONFIG`	override the settings file location
`XDG_CONFIG_HOME`, `XDG_CACHE_HOME`	the settings and cache directories

B.3.2 The parse cache

Variable	Effect
`RAKUPP_PRECOMP_MODULES`	override the module-cache switch for one run
`RAKUPP_PRECOMP_FILES`	override the file-cache switch
`RAKUPP_NO_PRECOMP=1`	force both off
`RAKUPP_PRECOMP_DIR`	where cache entries live

B.3.3 The foreign-function interface

Variable	Effect
`RAKUPP_FFI=0`	force the no-libffi fallback path — the second CI leg
`RAKUPP_FFI=/path`	use *only* that library; if it fails to load, report and fall back rather than silently substituting the system copy
`RAKUPP_FFI_TRACE=1`	log every crossing, with marshalled arguments and raw returns

B.3.4 Concurrency

Variable	Effect
`RAKUPP_PARALLEL=1`	true parallelism: workers run interpreter compute concurrently instead of serialising on the GIL
`RAKUPP_GIL`	control the lock explicitly
`RAKUPP_FREEZE_TRACE=1`	report any symbol-table mutation after concurrency engaged, and which thread did it

B.3.5 The regex engine

Variable	Effect
`RAKUPP_LTM=1`	use the NFA ranker for longest-token matching where it is gap-free
`RAKUPP_LTM_DEBUG=1`	rank both ways and print every disagreement — works without `RAKUPP_LTM`
`RAKUPP_LTM_RANKDUMP=1`	print each decided alternation's ranking

B.3.6 Diagnostics

Variable	Effect
`RAKUPP_TRACE=1`	the module search path and every resolution
`RAKUPP_DUMPTOKENS=1`	the token stream
`RAKUPP_ACTTRACE=1`	grammar action firing
`RAKUPP_TAP_TRACE=1`	the test harness's own emission
`RAKUPP_KEEPGEN=1`	keep the generated C++ from a compiling mode

Variable	Effect
`RAKUPP_DEBUG_MAKE`, `RAKUPP_DEBUG_REPLAY`	grammar `make` and replay diagnostics

B.3.7 The interactive session

Variable	Effect
`RAKUPP_REPL=1`	force a session even when standard input is not a terminal
`RAKUPP_HISTORY`	the history file

B.3.8 Test-harness variables

`RAKU_TEST_DIE_ON_FAIL` stops a suite after a real failure; `RAKU_EXCEPTIONS_HANDLER=JSON` serialises uncaught exceptions as JSON. Both follow Rakudo's spelling because test harnesses set them.

B.4 Build-time switches

Define	Effect
`RAKUPP_PTR_CENSUS`	count which combinations of `Value`'s eleven pointers are actually live together — the empirical input to shrinking the struct
`_GLIBCXX_USE_CXX11_ABI`	relevant only because it is the ABI break copy-on-write strings caused (Chapter 8)

The pointer census is a good example of the project's habit: before collapsing `Value`'s pointers into tag-dispatched slots, a special build **counts** which sets are live simultaneously, rather than reasoning about what the type tags suggest.

B.5 A note on flags that change behaviour

Two of the variables above change *results* rather than speed: `RAKUPP_LTM` and `RAKUPP_PARALLEL`. Both are off by default, and the policy for both is the same: the old path stays available for at least one release after the default changes, so a regression can be bisected to the switch rather than to the release.

Everything else here either reports something or selects a code path that is required to produce identical output.

Appendix C

Source Map and Glossary

C.1 Where to look for what

If you are changing	Start in	Also read
tokenization, quoting, heredocs	`Lexer.cpp`, `Token.h`	Chapter 3
statement or expression syntax	`Parser.cpp`, `Ast.h`	Chapters 4 and 6
a user-declared operator	`Parser.cpp` `parseSub`, the `userInfix_` family	Chapter 5
what a value *is*	`Value.h`	Chapter 7
string performance	`Value.h` `CowStr`, `BuiltinsShared.h`	Chapters 8 and 23
the number tower	`BigInt.cpp`, `IntOps.h`, `Value::rat`	Chapter 10
scoping, assignment, binding	`Interpreter.cpp` `lvalue`, `evalAssign`	Chapter 11
calls and signatures	`Interpreter.cpp` `callCallableRaw`, `bindParams`	Chapter 13

If you are changing	Start in	Also read
`return`, `next`, `last`, `when`	the cooperative registers in `ExecContext`	Chapter 14
a built-in routine	`Builtins.cpp` `registerBuiltins`	Chapter 15
a built-in **method**	the four `methodCall` segments, **in order**	Chapters 2 and 15
classes, roles, mixins	`Interpreter.cpp` `ClassDecl` handling, `Value.h`	Chapter 16
laziness, `gather`	`LazySeqState`, `seqOp`, the gather stack	Chapter 17
interpreter speed	`evalBinary`, `evalIndex`, the decided-once fields	Chapter 18
regex syntax	`Regex.cpp` `parseAtom`	Chapter 19
regex matching	`Regex.cpp` `matchNode`	Chapter 20
grammars	`GrammarMatcher`, `Interpreter::grammarParse`	Chapter 21
alternation ranking	`LtmNfa.cpp`	Chapter 22
Unicode	`Unicode.cpp`, `tools/ucd/`, the generators	Chapter 23
the CLI and compile drivers	`main.cpp`	Chapter 24
the native compiler	`Codegen.cpp`, the `rt*` helpers in `Interpreter.h`	Chapters 25 to 27
the parse cache	`AstSerial.cpp`	Chapter 28
module loading	`Interpreter.cpp` `loadModule`, `Parser.cpp` `scanModuleOps`	Chapter 29
`nqp::` ops	`Parser::makeNqpOp`, `Interpreter::evalNqpOp`	Chapter 30
NativeCall	`Ffi.cpp`, `Interpreter::callNative`	Chapter 31

If you are changing	Start in	Also read
the extension ABI	`rakupp_ext.h`, `ExtApi.cpp`	Chapter 32
threads, the GIL, supplies	`Interpreter.h`'s concurrency section	Chapter 33
lint, highlight, profile, REPL	`Lint.cpp`, `Highlight.cpp`, `Profiler.cpp`, `Repl.cpp`	Chapter 34

C.2 Rules that are easy to break by accident

Collected here because each has cost real debugging time.

The method-dispatch chain is order-sensitive. The four segments are ordered slices of one function; later arms deliberately catch what earlier ones decline. Moving an arm for readability is a behaviour change.

A decided-once field may hold a fact about the syntax, never a value that can change. The literal cache is the one exception, and a literal is a constant by definition.

Never store an `FnRef`. It borrows the caller's lambda; storing it dangles.

A pointer is only a valid map key when its target's lifetime is at least the map's. AST nodes are never freed, so a flip-flop state map keyed on one is fine. Regex nodes are freed and their addresses recycled, which is why a cache keyed on one produced automata built for other patterns.

Intern a closed vocabulary, never an open one. The intern table is append-only, so a field that can hold arbitrary runtime data would leak an entry per distinct value.

Do not add a non-`const` `operator[]`, `begin()` or `data()` to `CowStr`. That is exactly the interface that made copy-on-write non-conforming for `std::string`.

Add an early exit, never restructure the general path underneath it. The first node specialisation cost the control 5.7% by doing the latter.

A memo is only sound if the thing memoised is a function of the key. A grammar rule that reads a dynamic variable is not a function of (rule, position), which is what the `dynDep` flag records.

`gilPark`'s window must touch no interpreter state. Only thread-local buffers and syscalls.

Never join a thread while holding the lock the joinee might want.

A derived-data mechanism must degrade to recomputation, never to a guess. Every failure mode in the caching and emitting paths ends in "parse it again".

C.3 Glossary

AOT — the `--aot` mode: parse at build time, emit C++ that rebuilds the AST, then interpret it at run time.

Allomorph — a value that is simultaneously a number and its own string, such as `IntStr`. Represented as a numeric tag with the text in `s`.

Autothreading — distributing an operation over a junction's eigenstates and recombining the results.

Bundle — the `--bundle` mode: embed the source bytes in a standalone binary that parses and interprets them at run time.

Byteset — a 256-bit bitmap cached on a regex character-class node, answering "does this byte match?" without re-deriving the class.

Cooperative control flow — implementing `return`, `next`, `last` and `when` with a flag and a frame counter instead of a C++ exception, when no callable boundary was crossed.

Declarative prefix — the leading part of a regex alternative made of literals, character classes and quantifiers, up to the first procedural construct. What longest-token matching ranks by.

Decided-once field — a mutable field on an AST node holding a fact about the syntax, computed on first evaluation and never recomputed.

Eigenstate — one of the values inside a junction.

Fat struct — the `Value` design: one struct with a type tag and a field for every kind of payload, several of which may be live at once.

GIL — the global interpreter lock. Only its holder may touch interpreter state; it is engaged lazily on first concurrent use.

Grapheme — what a reader calls a character. Raku's string indices are grapheme indices; storage is UTF-8 bytes.

Handle — an opaque `RkValue` in the extension ABI. The mechanism by which an extension never sees `Value`.

Model gap — a construct the longest-token automaton builder could not model, as opposed to one that genuinely ends the declarative prefix. A gap forces a fallback to the probe ranker.

NFG — Raku's normalization-form-grapheme storage model. Strings are normalised to NFC on the way in.

Packrat memo — the cache of a ratcheting grammar rule's match at a position, sound because such a rule does not backtrack.

Publish — the step at the end of module loading that copies the module's environment into the global one.

Ratchet — the property of `token` and `rule` that their quantifiers are possessive and their matches commit.

Sigspace — the `:s` adverb and the `rule` declarator, under which whitespace in a pattern means "match optional whitespace here". Implemented by wrapping atoms with a `<ws>` subrule at compile time.

Sink context — a statement whose value is discarded, signalled down so an assignment need not materialise its result.

Slip — `|@a`, a list that splices into the surrounding list or argument list.

Specificity — the score `scoreCandidate` assigns a multi-dispatch candidate.

Superinstruction — a fused node kind representing a common pattern. The approach node specialisation deliberately did *not* take.

Transparent (of a regex construct, in ranking) — treated as an epsilon transition by the longest-token automaton, because the commit engine will enforce it for real.

Twigil — the second sigil character: `*` dynamic, `!/.` attribute, `^` placeholder, `?` compile-time.

Wrapper stack — the `&routine.wrap({...})` layers on a `Callable`, run outermost first, each able to `callsame` into the next.