

The Engineer's Press

Gutenberg's press broke text into pieces of movable type, set them in lines, locked the lines into a page and printed it. A language implementation does much the same to a program. It breaks the source into tokens, arranges them into a tree, checks and transforms the tree, and turns it into something a machine can run. This volume builds every stage of that process in Python you can run: a lexer and parsers, a resolver and type checkers, an interpreter, a bytecode compiler and virtual machine, a garbage collector, an optimiser and a register allocator. Along the way it explains why languages differ in the ways they do.

CHAPTERS

20

CODE EXAMPLES

20

EXERCISES

60

STAGE BUDGET

120 h

Contents

How to use this volume	4
Part I · Reading programs	6
Chapter 1 · How languages are implemented	7
Chapter 2 · Lexing	14
Chapter 3 · Grammars	23
Chapter 4 · Parsing	31
Chapter 5 · Names, scopes and resolution	39
Part II · Types	47
Chapter 6 · Type systems	48
Chapter 7 · Type inference	56
Chapter 8 · Generics and variance	64
Part III · Running programs	71
Chapter 9 · Tree-walking interpreters	72
Chapter 10 · Closures	79
Chapter 11 · Bytecode	86
Chapter 12 · Virtual machines	94
Chapter 13 · Garbage collection	101
Chapter 14 · Tail calls and continuations	109
Part IV · Compiling for speed	117
Chapter 15 · Intermediate representations and SSA	118
Chapter 16 · Optimisation	126
Chapter 17 · Code generation and registers	134
Chapter 18 · JIT compilation	142
Part V · Paradigms and boundaries	150
Chapter 19 · Paradigms and pattern matching	151
Chapter 20 · Foreign function interfaces	159
The deliverable	165

The twenty rules 168

Reading list 170

This is the stage where your tools stop being magic. Every program you write passes through a lexer, a parser, a type checker and either an interpreter or a compiler, and every surprising behaviour of a language is a decision one of them made. Once you've built each of those pieces, a new language becomes a set of familiar choices, and error messages, performance cliffs and odd corner cases start to explain themselves. The table shows where each of the stage's eleven topics is taught here, and what to read or build alongside it.

STAGE 07 TOPIC	CHAPTERS	READ OR BUILD ALONGSIDE
Lexing, parsing, grammars, abstract syntax trees	Ch. 1–4	Crafting Interpreters ch. 4–6
Recursive descent and precedence climbing; parser generators	Ch. 4	Crafting Interpreters ch. 6, 17
Semantic analysis, scope resolution, symbol tables	Ch. 5	Crafting Interpreters ch. 11
Type systems: static and dynamic, inference, generics, variance	Ch. 6–8	Types and Programming Languages
Tree-walking interpreters and bytecode virtual machines	Ch. 9, 11–12	Crafting Interpreters ch. 7–10, 14–24
Intermediate representations, optimisation, code generation, register allocation	Ch. 15–18	Engineering a Compiler
Runtime memory management: reference counting, tracing, generations	Ch. 13	Crafting Interpreters ch. 26
Closures, first-class functions, tail calls, continuations	Ch. 10, 14	Crafting Interpreters ch. 10, 25
Paradigms compared: imperative, object-oriented, functional, logic	Ch. 19	a second language: Haskell, OCaml, Elixir or Clojure
Immutability, algebraic data types, pattern matching	Ch. 19	OCaml or Rust
Foreign function interfaces and language interoperation	Ch. 20	Python's <code>ctypes</code> documentation

Run it, then build it. Each chapter's example builds one stage of a language implementation in Python, small enough to read in a sitting. The early chapters parse a small C-like language. From chapter 9 on, the examples write their programs as S-expressions, which take a dozen lines to read, so each example can spend its lines on its own subject rather than on parsing. Where a chapter compares speeds, it counts the work, in nodes visited, instructions executed or objects copied, so the output shown is exactly what the program prints on any machine. Only chapters 18 and 20 also measure time, because their subject is the cost of running code, and their timings will differ on yours.

The deliverable is a language. The stage asks you to build a complete one, following Robert Nystrom's *Crafting Interpreters*: a scanner, a parser, a resolver and a tree-walking interpreter, then a bytecode virtual machine with its own garbage collector, plus one feature of your own that the book doesn't cover. The appendix maps the book's chapters to this volume's, and lists features worth adding.

THE ONE HABIT

A program is only text until something reads it, and every stage that reads it makes decisions: what's a token, what binds tighter, which variable a name means, what a value's type is, when memory can be reused.

So when a language surprises you, with an error message, a slow loop or a closure that captures the wrong value, **ask which stage of its implementation made that choice, and why.** There is always a stage, and usually a reason.

PART I

Reading programs

Five chapters on the front end of a language implementation, the part that works out what a program means before anything runs it. A lexer turns characters into tokens, a grammar says which arrangements of tokens are programs, a parser builds the tree a grammar describes, and a resolver works out which declaration every name refers to. By the end of this part you can take source text and produce a checked syntax tree, which is where every later stage begins.

How languages are implemented

3 exercises

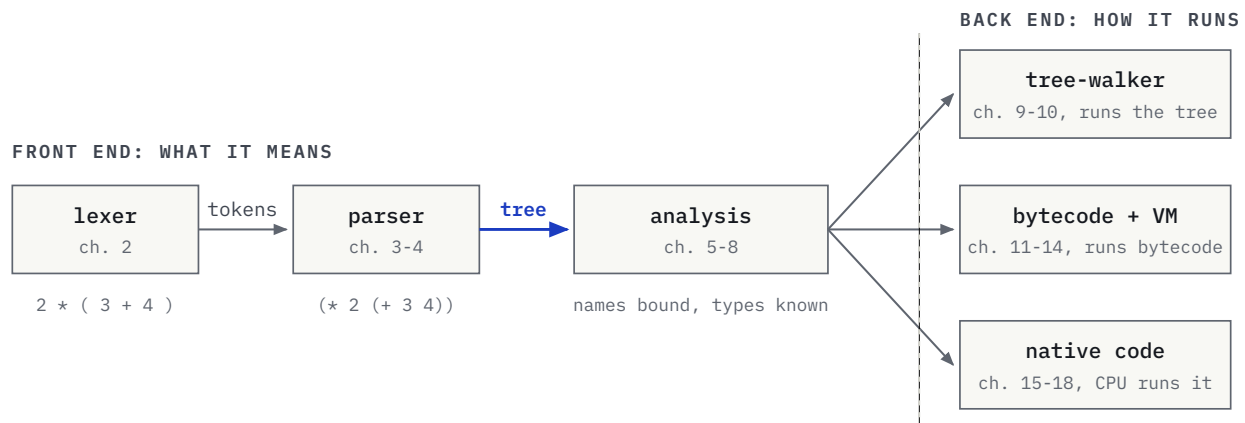
A programming language is a specification: it says what programs look like and what they mean. An implementation is a program that reads programs and makes that meaning happen. Every implementation, from a calculator to an optimising compiler, is built from the same few stages, each turning one representation of the program into the next. This chapter walks one expression through all of them in sixty lines of Python, and sorts real implementations by which stages they have.

A pipeline of representations

Source code arrives as a string of characters. The first stage, the **lexer** or scanner, groups characters into **tokens**: the number `10`, the operator `*`, the name `total`, the keyword `while`. Spaces and comments disappear here. The **parser** then arranges the tokens into a **syntax tree** that records the program's structure: which operator applies to which operands, which statements are inside which loop. The tree for `2 * (3 + 4)` has `*` at its root and the addition below it, and the parentheses are gone, because the shape of the tree now says what they said.

Next comes **semantic analysis**: everything that can be checked about the program's meaning without running it. Which declaration does each name refer to? Is that function being called with the right number of arguments? Does adding these two values make sense? These three stages together are the **front end**. Their output is a tree the rest of the implementation can trust.

The **back end** makes the program run, and here implementations divide. The simplest walks the tree and does what each node says, which is a **tree-walking interpreter**. Others first **compile** the tree into a flat list of instructions. Those instructions may be for a **virtual machine**, a program that imitates a simple processor, or for the real processor, as machine code. Between the tree and machine code there are often several **intermediate representations**, each lower-level than the last, where an optimiser can rewrite the program to do less work.



The stages of a language implementation, and the chapters that build each one. Everything after the parser works on the tree, and the same checked tree can be run in three ways.

The example builds the whole pipeline for arithmetic. A regular expression does the lexing. A parser with one function per grammar rule builds the tree as nested tuples. Then the tree is run twice: once by an interpreter that evaluates it recursively, and once by compiling it to instructions for a stack machine and running those. Both get the answer Python gets.

EXAMPLE

pipeline.py · one expression through a lexer, a parser, a tree-walking interpreter, and a compiler and virtual machine

```
import re

source = "2 * (3 + 4) - 10 / 5"

# 1. Lexing: characters -> tokens
def lex(text):
    tokens = []
    for number, symbol in re.findall(r"\s*(?:(\d+)|(\S))", text):
        tokens.append(("NUM", int(number)) if number else ("OP", symbol))
    return tokens + [("END", None)]

# 2. Parsing: tokens -> a tree. One function per grammar rule (recursive descent):
#   expr = term (("+" | "-") term)* ; term = factor (("*" | "/" ) factor)* ; factor = NUM | "
#   (" expr ")"
def parse(tokens):
    pos = 0
    def peek(): return tokens[pos][1]
    def take():
        nonlocal pos; pos += 1; return tokens[pos - 1]
    def expr():
        node = term()
        while peek() in ("+", "-"): node = (take()[1], node, term())
        return node
    def term():
        node = factor()
        while peek() in ("*", "/"): node = (take()[1], node, factor())
        return node
    def factor():
        kind, value = take()
        if kind == "NUM": return ("num", value)
```

```

    if value == "(":
        node = expr(); take(); return node # skip the ")"
    raise SyntaxError(f"unexpected {value!r}")
    return expr()

def show(node):
    return str(node[1]) if node[0] == "num" else f"({node[0]} {show(node[1])} {show(node[2])})"

OPS = {"+": lambda a, b: a + b, "-": lambda a, b: a - b, "*": lambda a, b: a * b, "/": lambda
a, b: a / b}

# 3a. A tree-walking interpreter: evaluate the tree directly
visits = 0
def evaluate(node):
    global visits; visits += 1
    if node[0] == "num": return node[1]
    return OPS[node[0]](evaluate(node[1]), evaluate(node[2]))

# 3b. Or compile the tree to instructions for a stack machine, then run those
def compile_(node, code):
    if node[0] == "num": code.append(("PUSH", node[1]))
    else:
        compile_(node[1], code); compile_(node[2], code)
        code.append(("{"ADD": "ADD", "-": "SUB", "*": "MUL", "/": "DIV"}[node[0]], None))
    return code
def run(code):
    stack = []
    for op, arg in code:
        if op == "PUSH": stack.append(arg)
        else:
            b, a = stack.pop(), stack.pop()
            stack.append(OPS[{"ADD": "+", "SUB": "-", "MUL": "*", "DIV": "/"}[op]](a, b))
    return stack.pop()

tokens = lex(source)
tree = parse(tokens)
print(f"source: {source}")
print(f"tokens: {' '.join(str(v) for k, v in tokens if k != 'END')} ({len(tokens) - 1}
tokens)")
print(f"tree: {show(tree)}")
print(f"tree-walking interpreter: {evaluate(tree):g}, after visiting {visits} nodes")
code = compile_(tree, [])
print(f"compiled to a stack machine: " + "; ".join(op if arg is None else f"{op} {arg}" for op,
arg in code))
print(f"virtual machine: {run(code):g}, after {len(code)} instructions")
print(f"Python agrees: {eval(source):g}")

```

OUTPUT

```

source: 2 * (3 + 4) - 10 / 5
tokens: 2 * ( 3 + 4 ) - 10 / 5 (11 tokens)
tree: (- (* 2 (+ 3 4)) (/ 10 5))
tree-walking interpreter: 12, after visiting 9 nodes
compiled to a stack machine: PUSH 2; PUSH 3; PUSH 4; ADD; MUL; PUSH 10; PUSH 5; DIV; SUB
virtual machine: 12, after 9 instructions
Python agrees: 12

```

Look at the compiled code. The tree has disappeared into a sequence: push 2, push 3, push 4, add the top two values, multiply the top two values, and so on. The order is the tree's **postorder**, operands before the operator that uses them. No parentheses or precedence

rules are needed to run it, because the parser already settled those questions. The interpreter visited 9 nodes and the virtual machine ran 9 instructions: the same amount of work, organised differently. Chapter 11 shows why the flat organisation is usually faster.

Interpreted or compiled is a property of an implementation

People say "Python is interpreted" and "C is compiled", but a language doesn't decide this. Its implementations do, and most real ones mix both approaches.

- **CPython**, the standard Python, compiles each module to bytecode, caches it in `__pycache__` as `.pyc` files, and interprets the bytecode in a virtual machine. Since 3.11 that interpreter rewrites its own instructions into faster, specialised versions as it learns what types appear (PEP 659), and 3.13 added an experimental JIT compiler. **PyPy** runs the same language with a tracing JIT compiler.
- **Java** is compiled by `javac` to bytecode for the Java Virtual Machine. The HotSpot JVM interprets that bytecode at first, and compiles the methods that turn out to be hot into machine code while the program runs.
- **JavaScript** in V8, Chrome's engine, is parsed to bytecode for an interpreter called Ignition, then compiled by progressively more aggressive compilers as functions get hotter.
- **C, C++, Rust and Go** are usually compiled **ahead of time**, before the program runs, to machine code. Clang and rustc both translate to LLVM's intermediate representation and let LLVM do the optimisation and code generation. But C interpreters exist, and so do Python compilers.
- **TypeScript** is compiled to JavaScript, a different high-level language. A compiler whose target is another source language is often called a **transpiler**.

The useful distinction is about when work happens. Work done before the program runs, by a compiler, is paid once. Work done while it runs, by an interpreter, is paid every time a line executes. An ahead-of-time compiler does as much as it can in advance. An interpreter does almost nothing in advance and starts at once. A JIT compiler waits until it sees which parts of the program matter and how they behave, then compiles those, using what it saw. Chapter 18 builds one.

Why build it in stages

An implementation could go straight from characters to execution, and a very small one sometimes does. Stages win because each one has a single job and a representation suited to it. The lexer can ignore structure, the parser can ignore spelling, the type checker can ignore the order of characters on a line, and the optimiser can ignore source syntax entirely. Each stage reports the errors it is in the best position to see: the lexer reports an unterminated string, the parser a missing parenthesis, the checker an undeclared variable.

Stages also let implementations share work. Without a shared middle, supporting M languages on N processors would need $M \times N$ compilers. With one intermediate representation in the middle, it needs M front ends and N back ends. That's the

arrangement LLVM made standard: Clang, rustc, Swift and Julia all produce LLVM IR, and every processor LLVM supports runs all of them. The JVM and WebAssembly play the same role for virtual machines, which is why Kotlin, Scala and Clojure share one runtime.

Seeing the stages of a real implementation

Python exposes every stage of its own implementation from the standard library, which makes it a good place to look. `python -m tokenize file.py` prints the token stream. `python -m ast file.py` prints the syntax tree, and `ast.parse` returns it as objects you can walk or rewrite. The `symtable` module shows how names were resolved, and `python -m dis file.py` disassembles the bytecode the compiler produced. For C, Clang does the same: `clang -fsyntax-only -Xclang -ast-dump` prints the tree, `clang -S -emit-llvm` writes the LLVM IR, and `clang -S` the assembly. The Compiler Explorer website shows the machine code that many compilers produce for a function, side by side, as you type.

Doing this once for a small function is worth an hour. The tokens look like what you typed. The tree looks like what you meant. The bytecode looks like neither, and the assembly from an optimising compiler sometimes doesn't contain your loop at all, because the compiler worked out its result in advance. The rest of this volume explains each of those transformations.

COMMON TRAPS

Calling a language interpreted or compiled. Ask instead which implementation, and what it does before and during the run.

Starting with the parser generator before writing a parser by hand. Write one by hand first, so you know what the generator is doing for you.

Letting stages leak into each other, such as a parser that evaluates as it goes. It works for a calculator and stops working the day you add functions or loops.

Keeping no source positions in tokens and trees. Every later error message needs a line and column, and adding them afterwards means touching every stage.

Treating the tree as text. Build real data structures, not strings to be parsed again.

EXERCISES

EXERCISE 1.1 Before and during the run

For each of CPython, the HotSpot JVM running a Java program, GCC compiling C, and `tsc` compiling TypeScript, say what is produced before the program runs and what work, if any, is still translation while it runs.

SOLUTION

CPython produces bytecode before running each module, cached in `.pyc` files. While the program runs, the interpreter executes that bytecode instruction by instruction, and since 3.11 also rewrites instructions into specialised forms, which is a small amount of translation at run time. **HotSpot** starts from bytecode that `javac` produced ahead of time. While running, it interprets that bytecode and also compiles hot methods to machine code, sometimes twice, with a quick compiler first and an optimising one later. **GCC** produces machine code before the program runs, and no translation happens during the run. `tsc` produces JavaScript ahead of time, which then goes through a JavaScript engine's own pipeline of parsing, bytecode and JIT compilation when it runs. The types are checked and then erased: none of them exist at run time.

EXERCISE 1.2 Adding an operator

Add the remainder operator `%` to the example, with the same precedence as `*` and `/`. Which stages have to change, and which don't?

SOLUTION

The lexer needs no change, because its pattern already turns any non-space character into an operator token. The parser changes in one place: `term` must accept `%` along with `*` and `/`, which is where its precedence is decided. The interpreter needs an entry `"%": lambda a, b: a % b` in `OPS`. The compiler needs an instruction name, such as `MOD`, in its operator map, and the virtual machine needs the reverse mapping from `MOD` to the operation. So one new operator touches three stages, and each change is small and local. That's the point of the stages: a language feature is a list of small changes, one per stage. Exercise 4.1 adds a postfix operator, which is harder, because it changes the parser's shape.

EXERCISE 1.3 Counting the work

For the source `1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10` , how many nodes does the interpreter visit, and how many instructions does the virtual machine execute? If both do the same amount of work, why might one still be faster?

SOLUTION

The tree has 10 number nodes and 9 addition nodes, so the interpreter visits 19 nodes, and the compiled code has 10 pushes and 9 adds, so the machine executes 19 instructions. For any expression both counts equal the number of nodes. The difference is in the cost of each step. The interpreter makes a recursive function call for every node, with a frame, arguments and a return, and inspects each node to decide what it is. The virtual machine loops over a flat list and keeps its values on one stack. In a Python implementation the gap is modest. In C, where the virtual machine's instructions can be small integers packed into a byte array, it's typically several times. Chapter 11 compares the two on a program with a loop, and explains where the advantage of bytecode comes from.

CHECKLIST

- ☐ Can you name the stages of a language implementation and the representation each produces?
- ☐ Can you say which stages belong to the front end and which to the back end?
- ☐ Can you explain why "interpreted" and "compiled" describe implementations, with a real example of each mix?
- ☐ Can you explain how a shared intermediate representation turns $M \times N$ compilers into $M + N$?
- ☐ Can you print the tokens, tree and bytecode of a Python function on your own machine?

GO DEEPER

BOOK [Crafting Interpreters, chapter 2: A Map of the Territory](#) Robert Nystrom

DOCS [Python Language Services: tokenize, ast, symtable and dis](#) Python documentation

TOOL [Compiler Explorer](#) Matt Godbolt

Lexing

3 exercises

A lexer reads characters and produces tokens: numbers, names, keywords, strings and operators, each with the position where it began. It's the simplest stage of an implementation and the one that touches every character, so it decides questions that look trivial until they aren't: whether `a>=b` contains one operator or two, which escapes a string may hold, and what `1.` means. This chapter builds a hand-written scanner for a small C-like language, shows the rule that settles most ambiguities, and draws the finite automaton that sits underneath every lexer.

Tokens

A **token** is a kind plus, for some kinds, a value. The kind is what the parser cares about:

`NUM`, `IDENT`, `WHILE`, `(`. The value is what later stages need: the number 1071, the name `gcd`, the text of a string. The characters that made up a token are its **lexeme**, and the token records where they were, so that an error in any later stage can point at the source. The lexer also discards what the parser shouldn't see: spaces, newlines and comments. In most languages these only separate tokens. In Python, indentation is significant, and the tokenizer turns changes of indentation into `INDENT` and `DEDENT` tokens, keeping a stack of the indentation levels currently open, so the parser sees blocks as if they were braces.

Most tokens can be recognised by looking at their first character. A digit starts a number. A letter or underscore starts a name, which is then looked up in a table of keywords: `while` becomes the token `WHILE`, and `whilst` stays an identifier. A double quote starts a string. Everything else is an operator or punctuation. A hand-written lexer is one loop with one branch per kind of first character, and that's what the example is.

A scanner by hand

The scanner below reads a small language with functions, loops, strings and comments. It tracks the line and column as it goes, handles numbers with fractions and exponents, translates escapes inside strings, and reports errors with the line printed and a caret under the column where the problem begins. It then lexes three tricky inputs, three broken ones, and checks some numbers against a finite automaton, which the second half of the chapter explains.

EXAMPLE

`lexer.py` · a hand-written scanner with positions, maximal munch, error reports, and the number rule as a DFA

```
KEYWORDS = {"let", "fun", "return", "if", "else", "while", "print", "true", "false", "nil",
```

```

    "and", "or"}
TWO_CHAR = {"==", "!=", "<=", ">="}
ONE_CHAR = set("+-*/%=<>!( ){};,")

class LexError(Exception): pass

def scan(src):
    """A hand-written scanner: one pass over the characters, tracking line and column."""
    tokens, i, line, col = [], 0, 1, 1
    def error(msg, at_col):
        text = src.splitlines()[line - 1]
        raise LexError(f"line {line}, column {at_col}: {msg}\n    {text}\n    {' ' * (at_col - 1)}^")
    while i < len(src):
        c, start, start_col = src[i], i, col
        if c == "\n": i += 1; line += 1; col = 1; continue
        if c in "\t\r": i += 1; col += 1; continue
        if src.startswith("//", i):                                     # a comment runs to the end of
            the line
            while i < len(src) and src[i] != "\n": i += 1
            continue
        if c.isdigit():                                               # 12, 3.5, 1e9, 2.5e-3
            while i < len(src) and src[i].isdigit(): i += 1
            if src[i:i + 1] == "." and src[i + 1:i + 2].isdigit():
                i += 1
                while i < len(src) and src[i].isdigit(): i += 1
            if src[i:i + 1] in ("e", "E"):
                j = i + 1 + (src[i + 1:i + 2] in ("+", "-"))
                if src[j:j + 1].isdigit():
                    i = j
                    while i < len(src) and src[i].isdigit(): i += 1
            tokens.append(("NUM", float(src[start:i]), line))
        elif c.isalpha() or c == "_":                                # identifiers, and keywords
            among them
            while i < len(src) and (src[i].isalnum() or src[i] == "_"): i += 1
            word = src[start:i]
            tokens.append((word.upper(), None, line) if word in KEYWORDS else ("IDENT", word, line))
        elif c == "'":
            i += 1; chars = []
            while i < len(src) and src[i] != "'":
                if src[i] == "\n": error("unterminated string", start_col)
                if src[i] == "\\":
                    i += 1
                    chars.append({"n": "\n", "t": "\t", "'": "'", "\\": "\\"}.get(src[i:i + 1]))
                or
                    error(f"unknown escape \\{src[i:i + 1]}", col + i - start))
                else: chars.append(src[i])
            i += 1
            if i >= len(src): error("unterminated string", start_col)
            i += 1
            tokens.append(("STR", "".join(chars), line))
        elif src[i:i + 2] in TWO_CHAR:                                # maximal munch: the longest
            match wins
            i += 2; tokens.append((src[start:i], None, line))
        elif c in ONE_CHAR:
            i += 1; tokens.append((c, None, line))
        else:
            error(f"unexpected character {c!r}", start_col)
            col += i - start
    return tokens + [("EOF", None, line)]

program = '''// greatest common divisor
fun gcd(a, b) {
    while (b != 0) {
        let t = b; b = a % b; a = t;

```

```

    }
    return a;
}
print gcd(1071, 462) >= 2.1e1; // true
print "done\\n";'''
def show(kind, value):
    if value is None: return kind
    if kind == "NUM": return f"num:{value:g}"
    return f"id:{value}" if kind == "IDENT" else f"str:{value!r}"
toks = scan(program)
print(f"{len(toks)} tokens:")
for n in range(1, program.count("\n") + 2):
    shown = [show(k, v) for k, v, ln in toks if ln == n]
    if shown: print(f"    line {n}: {' '.join(shown)}")

print("\nmaximal munch:")
for text in ("a>=b", "a> =b", "x==1"):
    print(f"    {text!r:9} -> {' '.join(show(k, v) for k, v, _ in scan(text)[:1])}")

print("\nerrors, with positions:")
for bad in ('let s = "never closed;', 'let x = 3 @ 4;', 'print "tab\\q";'):
    try: scan(bad)
    except LexError as e: print("    " + str(e).replace("\n", "\n    "))

# The number rule as a deterministic finite automaton: a table of (state, character class) ->
# state
DFA = {("start", "d"): "int", ("int", "d"): "int", ("int", "."): "dot", ("dot", "d"): "frac",
      ("frac", "d"): "frac", ("int", "e"): "exp", ("frac", "e"): "exp", ("exp", "s"): "sign",
      ("exp", "d"): "power", ("sign", "d"): "power", ("power", "d"): "power"}
ACCEPT = {"int", "frac", "power"}
def run_dfa(text):
    state = "start"
    for ch in text:
        cls = "d" if ch.isdigit() else "e" if ch in "eE" else "s" if ch in "+-" else ch
        state = DFA.get((state, cls), "dead")
    return state
print("\nthe number DFA:")
for text in ("42", "3.14", "1e10", "2.5e-3", "1.", ".5", "1e", "1e+"):
    state = run_dfa(text)
    print(f"    {text:7} ends in state {state:6} -> {'accept' if state in ACCEPT else
          'reject'}")

```

OUTPUT

```

49 tokens:
line 2: FUN id:gcd ( id:a , id:b ) {
line 3: WHILE ( id:b != num:0 ) {
line 4: LET id:t = id:b ; id:b = id:a % id:b ; id:a = id:t ;
line 5: }
line 6: RETURN id:a ;
line 7: }
line 8: PRINT id:gcd ( num:1071 , num:462 ) >= num:21 ;
line 9: PRINT str:'done\n' ; EOF

```

maximal munch:

```

'a>=b'    -> id:a  >=  id:b
'a> =b'   -> id:a  >  =  id:b
'x==1'    -> id:x  ==  -  num:1

```

errors, with positions:

```

line 1, column 9: unterminated string
    let s = "never closed;
        ^
line 1, column 11: unexpected character '@'
    let x = 3 @ 4;
        ^

```

```
line 1, column 12: unknown escape \q
print "tab\q";
      ^
```

the number DFA:

42	ends in state int	-> accept
3.14	ends in state frac	-> accept
1e10	ends in state power	-> accept
2.5e-3	ends in state power	-> accept
1.	ends in state dot	-> reject
.5	ends in state dead	-> reject
1e	ends in state exp	-> reject
1e+	ends in state sign	-> reject

Three details matter in any lexer. First, keywords are found by reading the whole word and then looking it up. A lexer that tried to match `let` as a prefix would split `letter` into a keyword and an identifier. Second, the number `2.1e1` became the value 21: the lexer converts lexemes to values, so later stages never see the spelling. Third, the errors were found and reported by the lexer, with exact positions, before any parsing began. A string that runs to the end of the line can't be fixed by a parser, and it gives a better message if the lexer says so at once.

Maximal munch

The input `a>=b` could be read as `a`, `>`, `=`, `b` or as `a`, `>=`, `b`. Lexers almost always choose the second, by the rule of **maximal munch**: at each position, take the longest token that matches. It's why the example checks the two-character operators before the one-character ones, and why `a> =b`, with a space, gives two separate operators. The rule is simple and usually right, but it occasionally works against the programmer.

- In C++ before 2011, `vector<vector<int>>` was an error, because `>>` was lexed as the shift operator. Programmers wrote `> >` until C++11 changed the rule for that context.
- In C, `x+++y` means `x++ + y`, never `x + ++y`, and `x+++++y` is an error even though a valid reading exists, because maximal munch produces `x ++ ++ + y`.
- `x== -1` in the example lexes as `==`, `-`, `1` because no operator `== -` exists. Early C did have operators spelled `== -` and `== +`, the original forms of `--` and `++`, and exercise 2.1 is about what that did to `x=-1`.

Where the lexer needs help

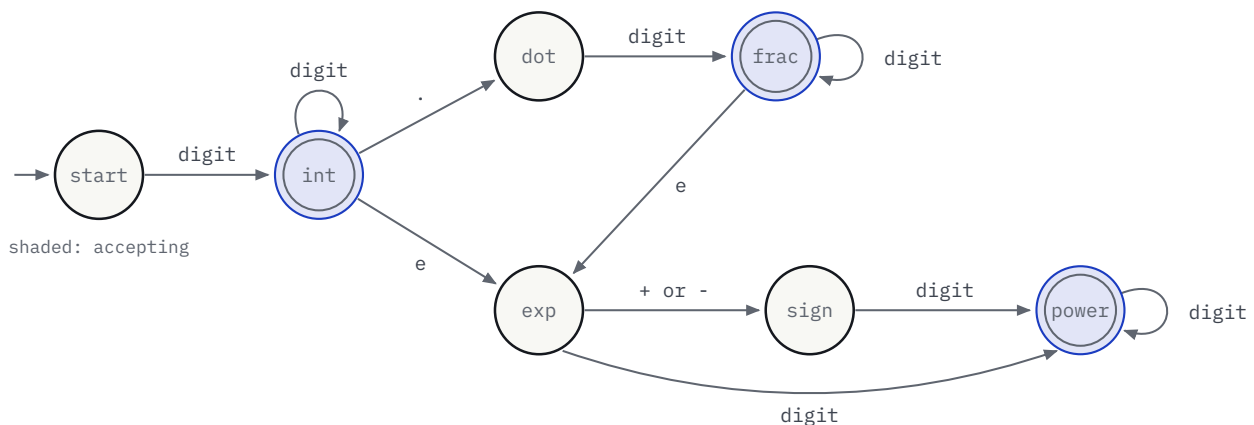
A lexer is meant to work without knowing the program's structure, and in a few famous places it can't. In C, `(a) * b` is a multiplication if `a` is a variable and a cast of `*b` if `a` is a type name declared with `typedef`. Since the parser can't build the right tree without knowing, C compilers let the parser feed type names back to the lexer, which then returns them as a different kind of token. This is known as **the lexer hack**. JavaScript has the opposite problem: a `/` may begin a regular expression literal or be a division, depending

on what came before, so the lexer must know whether the parser expects an operand. And JavaScript's automatic semicolon insertion means a newline sometimes ends a statement: `return` followed by a newline and an object literal returns `undefined`.

Modern languages avoid these problems by design. Go and Rust make type names and expressions distinguishable by their syntax alone. Python 3.10 added `match` as a **soft keyword**, a keyword only where a match statement can begin, so the many existing programs with a variable called `match` kept working. That decision lives in the parser. The lexer still produces an ordinary name.

Regular expressions and finite automata

Every token rule in a typical language can be written as a **regular expression**. A number is `[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?`, an identifier is `[A-Za-z_][A-Za-z0-9_]*`. Every regular expression can be turned into a **deterministic finite automaton**, a DFA: a set of states, a start state, a table saying which state each character leads to, and a set of accepting states. Running a DFA costs one table lookup per character, with no backtracking, whatever the expression was. That's why lexing is fast, and why tools can generate lexers automatically.



The number rule as a deterministic finite automaton. Each character moves the automaton along exactly one arrow. A character with no arrow sends it to a dead state, and the input is a number if it ends in a shaded state. So `1.` ends in `dot`, and is rejected.

The construction goes in two steps. Thompson's construction turns the regular expression into a **nondeterministic** automaton, an NFA, which may have several possible moves for one character; it's a direct translation, a few states per operator in the expression. The subset construction then turns the NFA into a DFA, whose states are sets of NFA states. The DFA can be minimised by merging states that behave identically. A lexer generator such as `lex`, `flex` or `re2c` does this for every token rule at once, with one more rule added: when

several token rules match, the longest match wins, and among equally long matches the rule listed first. That's how `while` becomes a keyword and not an identifier, if the keyword rules are listed first.

The example's DFA rejects `1.`, `.5`, `1e` and `1e+`. C and Python accept the first two as floating-point numbers. Neither choice is wrong. They're decisions about the language, made in the lexer, and they have consequences elsewhere: a language that accepts `1.` must decide what `1..2` or `1.e3` mean, and a language with method calls on numbers must decide what `1.foo` is.

Hand-written or generated

Many production compilers lex by hand: GCC, Clang, Go, Rust and V8 all do. A hand-written lexer is easy to read and debug, handles the awkward cases above with ordinary code, and produces better error messages. Generators remain common where the token rules are many and change often, and re2c-generated lexers are among the fastest. Either way, the lexer usually runs on demand, producing the next token when the parser asks for it rather than lexing the whole file first. That uses little memory, and it's what lets the lexer hack work, because the parser's knowledge is up to date at each request.

Two practical matters affect every lexer. **Unicode:** Python allows identifiers in any script (PEP 3131) and normalises them, so two spellings of the same accented letter name the same variable. And **error recovery:** the example stops at its first error, which is simple but frustrating with a long file. A lexer for an editor or IDE, which sees broken code most of the time, instead emits an error token and carries on.

COMMON TRAPS

Matching keywords as prefixes, which splits `letter` into `let` and `ter`.

Checking one-character operators first, so `>=` is never recognised.

Losing positions, or counting columns in bytes for text that isn't ASCII.

Forgetting the end of the input inside a string or comment, which loops forever or reads past the end.

Using a regular expression for nested comments. Regular languages can't count, as exercise 2.2 shows.

EXERCISES

EXERCISE 2.1 The operator that changed its spelling

In the earliest versions of C, compound assignment was written `==` and `+=`, so `x == 1` meant `x = x - 1`. With maximal munch, what did `x=-1` mean, and why did the language change the spelling to `--`?

SOLUTION

Maximal munch takes the longest token at each position, and `--` was a token, so `x=-1` lexed as `x`, `--`, `1`, which means subtract 1 from `x`. A programmer who wanted to assign minus one had to write a space, `x= -1`, and one who forgot got a program that compiled and did something else. Writing the operator first, `--`, removes the ambiguity: `x-=1` has only one reading, and `x=-1` lexes as `=` followed by `-` because no `--` token exists any more. Kernighan and Ritchie's book mentions the old forms and the reason for the change. The lesson generalises: when designing tokens, check that no new token is a prefix of a common sequence of existing ones.

EXERCISE 2.2 Nested comments

Rust, Swift and Haskell allow block comments to nest, so `/* a /* b */ c */` is one comment. C doesn't. Why can't a regular expression, or any DFA, recognise nested comments? How does a hand-written lexer handle them?

SOLUTION

To know whether a `*/` ends the comment, the lexer must know how many `/*` are still open. That number has no upper bound, but a DFA has a fixed, finite number of states, and so can only distinguish a fixed number of depths. For any DFA with k states, a comment nested $k + 1$ deep makes it confuse two different depths. This is the same argument that shows regular expressions can't match balanced parentheses, and it's why parsers, not lexers, handle nested structure. A hand-written lexer just keeps a counter: add one at each `/*`, subtract one at each `*/`, and stop when it reaches zero. If the input ends first, report an unterminated comment at the position of the first `/*`. C's non-nesting comments are why commenting out a block that already contains a comment breaks, and why C programmers use `#if 0` for the purpose.

EXERCISE 2.3 Indentation as tokens

Describe how to make the scanner produce `INDENT` and `DEDENT` tokens for a language with Python-style blocks. What does it emit for a line indented 8 spaces followed by a line indented 0, and what should happen if the next line is indented 2 spaces where only 0 and 4 are open?

SOLUTION

Keep a stack of open indentation widths, starting with 0. At the start of each line that isn't blank or a comment, measure the leading whitespace. If it's greater than the top of the stack, push it and emit one `INDENT`. If it's equal, emit nothing. If it's less, pop and emit one `DEDENT` for every level greater than the new width. If the width then doesn't equal the new top of the stack, the indentation matches no open block, which is an error: Python reports "unindent does not match any outer indentation level". So with 0, 4 and 8 open, a line at 0 produces two `DEDENT` tokens, and a line at 2 pops 8 and 4, finds 0 on top, and reports the error. Lines inside parentheses or brackets are exempt, since Python allows expressions to continue across lines there, so the lexer also counts open brackets. At the end of the file, emit a `DEDENT` for every level still open.

CHECKLIST

- ☐ Can you write a scanner by hand that records the line and column of every token?
- ☐ Can you state the maximal munch rule and give a case where it surprises programmers?
- ☐ Can you explain how keywords are distinguished from identifiers, including soft keywords?
- ☐ Can you draw the DFA for a token rule and explain why running it is linear in the input?
- ☐ Can you explain why nested comments and balanced brackets need more than a DFA?

GO DEEPER

BOOK [Crafting Interpreters, chapter 4: Scanning](#) Robert Nystrom

ARTICLE [Regular Expression Matching Can Be Simple And Fast](#) Russ Cox

DOCS [The Python Language Reference, chapter 2: Lexical analysis](#) Python documentation

Grammars

3 exercises

A grammar is a finite set of rules that describes an infinite set of programs. It says which sequences of tokens are well formed and, if it's written carefully, what structure each one has: that `1 + 2 * 3` multiplies first, and that an `else` belongs to the nearest `if`. This chapter shows how a grammar can fail to say that, by being ambiguous, how precedence and associativity are built into a grammar's shape, and how to compute the sets that tell a parser which rule to use by looking at a single token.

Rules, symbols and derivations

A context-free grammar has **terminals**, the token kinds from the lexer; **nonterminals**, names for kinds of phrase such as *expression* or *statement*; and **productions**, rules saying what each nonterminal can be made of. The notation is called BNF after John Backus and Peter Naur, who used it to define ALGOL 60:

```
stmt -> "while" "(" expr ")" stmt
      | "{" stmt* "}"
      | expr ";"
expr  -> expr "+" term | term
term  -> term "*" factor | factor
factor -> NUM | IDENT | "(" expr ")"
```

Extended BNF, used here, adds `*` for repetition, `?` for an optional part and `|` for alternatives. A **derivation** starts from the start symbol and repeatedly replaces a nonterminal by one of its productions until only terminals remain. The record of which replacements were made is a **parse tree**, with the start symbol at the root and the tokens at the leaves. The parser's job is to find a derivation for the tokens it's given, or to report that none exists.

Grammars come in a hierarchy, named after Noam Chomsky, by what the rules may say. **Regular** grammars describe what DFAs recognise, which is enough for tokens. **Context-free** grammars allow a nonterminal to contain itself in the middle, which is what nesting needs, and are enough for the syntax of almost every programming language. Rules that depend on context, such as "a variable must be declared before it's used", are beyond context-free grammars. Implementations check those in semantic analysis, the subject of chapter 5, rather than in the grammar.

Ambiguity

A grammar is **ambiguous** if some input has two different parse trees. The obvious grammar for arithmetic, `E -> E + E | E - E | E * E | num`, is badly so: it says which strings are expressions but nothing about their structure. The first part of the example lists every tree

this grammar allows for `1 - 2 - 3` and `1 + 2 * 3`, and evaluates each one. The second counts parses of the classic **dangling else**, and the third builds a grammar a parser can use deterministically.

EXAMPLE

grammars.py · enumerating the trees of an ambiguous grammar, the dangling else, and FIRST and FOLLOW sets for an LL(1) table

```
from functools import lru_cache

# 1. An ambiguous grammar: E -> E + E | E - E | E * E | num. Every way of splitting is a
# parse tree.
def trees(tokens):
    @lru_cache(maxsize=None)
    def span(i, j):
        # all trees for tokens[i..j]
        if i == j: return [tokens[i]]
        out = []
        for k in range(i + 1, j, 2):
            # pick the operator at the root
            for left in span(i, k - 1):
                for right in span(k + 1, j): out.append((tokens[k], left, right))
        return out
    return span(0, len(tokens) - 1)

def value(t):
    if isinstance(t, int): return t
    a, b = value(t[1]), value(t[2])
    return a + b if t[0] == "+" else a - b if t[0] == "-" else a * b

def show(t): return str(t) if isinstance(t, int) else f"({t[0]} {show(t[1])} {show(t[2])})"

print("E -> E + E | E - E | E * E | num is ambiguous:")
for t in trees((1, "-", 2, "-", 3)): print(f"  1 - 2 - 3 as {show(t):19} = {value(t)}")
for t in trees((1, "+", 2, "*", 3)): print(f"  1 + 2 * 3 as {show(t):19} = {value(t)}")
counts = [len(trees(tuple(x for k in range(n) for x in (1, "+"))) + (1,)) for n in range(1, 8)]
print(f"  parse trees for 1 to 7 operators: {counts} (the Catalan numbers)")

# 2. The dangling else: S -> if c then S | if c then S else S | s
def parses(tokens):
    @lru_cache(maxsize=None)
    def S(i, j):
        if j - i == 1 and tokens[i] == "s": return 1
        if j - i < 4 or tokens[i:i + 3] != ("if", "c", "then"): return 0
        n = S(i + 3, j)
        for k in range(i + 4, j - 1):
            if tokens[k] == "else": n += S(i + 3, k) * S(k + 1, j)
        return n
    return S(0, len(tokens))
text = "if c then if c then s else s"
print(f"\n{text!r} has {parSES(tuple(text.split()))} parses: which 'if' owns the 'else'?")

# 3. An LL(1) grammar for expressions, with precedence built in and no left recursion
G = {"E": [["T", "E']], "E'": [["+", "T", "E"], []], "T": [["F", "T']],
      "T'": [["*", "F", "T"], []], "F": [[("(", "E", ")"), ["id"]]}
FIRST = {n: set() for n in G}; FOLLOW = {n: set() for n in G}; FOLLOW["E"].add("$")
def first_of(seq):
    out = set()
    for sym in seq:
        f = FIRST[sym] if sym in G else {sym}
        out |= f - {"ε"}
        if "ε" not in f: return out
    return out | {"ε"}
changed = True
while changed:
    changed = False
    # iterate to a fixed point
```

```

for n, rules in G.items():
    for rule in rules:
        f = first_of(rule)
        if not f <= FIRST[n]: FIRST[n] |= f; changed = True
        for i, sym in enumerate(rule):
            if sym in G:
                rest = first_of(rule[i + 1:])
                new = (rest - {"ε"}) | (FOLLOW[n] if "ε" in rest else set())
                if not new <= FOLLOW[sym]: FOLLOW[sym] |= new; changed = True
print("\nFIRST and FOLLOW sets:")
for n in G: print(f"  {n:3} FIRST {sorted(FIRST[n])!s:20} FOLLOW {sorted(FOLLOW[n])!s:20}")
table = {}
for n, rules in G.items():
    for rule in rules:
        f = first_of(rule)
        for t in (f - {"ε"}) | (FOLLOW[n] if "ε" in f else set()):
            assert (n, t) not in table, "not LL(1)"          # two rules for one cell: a
                                                                conflict
            table[n, t] = rule
def ll1(tokens):
    stack, tokens, steps = ["$"], "E", tokens + ["$"], 0
    while stack:
        top, steps = stack.pop(), steps + 1
        if top == tokens[0]: tokens.pop(0); continue
        if (top, tokens[0]) not in table: return f"syntax error at {tokens[0]!r} after {steps}
        steps"
        stack.extend(reversed(table[top, tokens[0]]))
    return f"accepted in {steps} steps"
print(f"LL(1) table: {len(table)} cells, no conflicts")
for text in ("id + id * id", "( id + id ) * id", "id + * id"):
    print(f"  {text:18} {ll1(text.split())}")

```

OUTPUT

```

E -> E + E | E - E | E * E | num is ambiguous:
  1 - 2 - 3 as (- 1 (- 2 3))      = 2
  1 - 2 - 3 as (- (- 1 2) 3)     = -4
  1 + 2 * 3 as (+ 1 (* 2 3))     = 7
  1 + 2 * 3 as (* (+ 1 2) 3)     = 9
parse trees for 1 to 7 operators: [1, 2, 5, 14, 42, 132, 429] (the Catalan numbers)

'if c then if c then s else s' has 2 parses: which 'if' owns the 'else'?

FIRST and FOLLOW sets:
E  FIRST ['(', 'id']          FOLLOW ['$ ', ')']
E' FIRST ['+', 'ε']           FOLLOW ['$ ', ')']
T  FIRST ['(', 'id']          FOLLOW ['$ ', ')', '+']
T' FIRST ['*', 'ε']           FOLLOW ['$ ', ')', '+']
F  FIRST ['(', 'id']          FOLLOW ['$ ', ')', '*', '+']
LL(1) table: 13 cells, no conflicts
id + id * id      accepted in 17 steps
( id + id ) * id  accepted in 24 steps
id + * id         syntax error at '*' after 8 steps

```

With seven operators the ambiguous grammar allows 429 trees, and the count grows exponentially: it's the sequence of Catalan numbers. Only one of those trees is what the programmer meant. A parser for an ambiguous grammar must either choose, by some rule outside the grammar, or report every tree, which is what some parsers for natural language do. Programming languages remove the ambiguity instead.

Precedence and associativity in the grammar's shape

The grammar at the top of the chapter isn't ambiguous, because it gives each precedence level its own nonterminal. An `expr` is a sum of `term`s and a `term` is a product of `factor`s, so in `1 + 2 * 3` the only way to derive the `*` is inside a `term`, below the `+`. Lower precedence means nearer the root. A language with ten precedence levels has ten such nonterminals, one per level.

Associativity comes from which side the rule recurses on. `expr -> expr "+" term` recurses on the left, so `1 - 2 - 3` can only be derived with the first subtraction deeper in the tree, which is `(1 - 2) - 3`. A rule that recursed on the right, `expr -> term "-" expr`, would force `1 - (2 - 3)`. Both grammars accept exactly the same strings. They differ only in structure, and therefore in meaning.



The same tokens, two grammars, two trees. The side a rule recurses on decides associativity, and so the answer. Subtraction needs the left-recursive rule; exponentiation, which is right-associative, needs the other.

The dangling else

In `if c then if c then s else s`, the `else` could belong to either `if`, and the natural grammar for `if` statements allows both trees. C, Java and JavaScript resolve this with a rule stated outside the grammar: an `else` belongs to the nearest `if` that doesn't yet have one. That's what a hand-written parser does naturally, by taking the `else` as soon as it sees it. The rule can also be encoded in the grammar, with separate nonterminals for statements whose `if`s all have `else`s and those that don't, which is exercise 3.2. Newer languages remove the problem from the syntax: Go, Rust and Swift require braces around every branch, and Python's indentation and `elif` make the owner of every `else` visible.

Grammars a parser can follow

A **top-down** parser, which builds the tree from the root, has to decide which production to use for a nonterminal before it has read the phrase. The most useful case is when one token of lookahead is enough. A grammar is **LL(1)** if, for every nonterminal, the next token always determines which production applies. Two sets make that precise. **FIRST(X)** is the set of tokens that can begin a phrase derived from X, with ϵ included if X can derive nothing

at all. **FOLLOW**(X) is the set of tokens that can come straight after X in some derivation, with **\$** standing for the end of the input. A production applies when the next token is in its **FIRST** set, or, if the production can be empty, in its nonterminal's **FOLLOW** set. If two productions claim the same token, the grammar isn't LL(1).

The grammar at the top of the chapter fails at once: `expr -> expr "+" term` is **left-recursive**, so a top-down parser trying to parse an `expr` would first try to parse an `expr`, forever. The example's grammar is the standard rewriting. `E -> T E'` and `E' -> + T E' | ε` say that an expression is a term followed by any number of `+ term`, which accepts the same strings without left recursion. The program computes **FIRST** and **FOLLOW** by iterating until nothing changes, which is a **fixed point** computation you'll see again in type inference and optimisation, builds the parse table, and finds no cell claimed twice. A parse is then a loop with a stack: pop a symbol, and either match it against the input or replace it by the production the table names.

The rewriting has a cost: the tree for `1 - 2 - 3` that the rewritten grammar produces leans the wrong way, so a parser using it must rebuild left-associative trees as it goes. Hand-written parsers sidestep the issue by writing the repetition as a loop, as chapter 1's parser does with `while peek() in ("+", "-")`. Chapter 4 goes further and handles all precedence levels with one function.

Bottom-up parsing and parser generators

A **bottom-up** parser builds the tree from the leaves. It shifts tokens onto a stack until the top of the stack matches the right side of a production, then reduces them to the nonterminal. LR parsers, introduced by Donald Knuth in 1965, work this way. They handle a larger class of grammars than LL parsers, including left-recursive ones, so grammars can be written more naturally. The tables are too large to build by hand, so they're produced by parser generators such as yacc and its successor Bison, which take the grammar with actions attached to each rule. When the grammar is ambiguous, the generator reports a **shift-reduce** or **reduce-reduce conflict**, and the programmer resolves it by declaring precedences or restructuring the grammar. Learning to read those conflict reports is most of learning Bison.

A third family, **parsing expression grammars** or PEGs, replaces `|` with an ordered choice: try the first alternative, and use the second only if the first fails. A PEG can't be ambiguous, because the order decides, but it can silently prefer the wrong alternative. CPython replaced its LL(1) parser with a PEG parser in Python 3.9 (PEP 617), because the LL(1) restriction had forced awkward rules into the grammar and the parser for years.

COMMON TRAPS

Writing `E -> E + E` and expecting a parser to know about precedence.

Feeding a left-recursive rule to a recursive-descent or LL parser, which recurses forever.

Removing left recursion and forgetting associativity, so subtraction groups the wrong way.

Silencing a parser generator's conflicts without understanding which tree it now builds.

Encoding context-dependent rules in the grammar, such as "declared before use", that belong in semantic analysis.

EXERCISES

EXERCISE 3.1 Adding exponentiation

Extend the grammar `expr -> expr "+" term | term`, `term -> term "*" factor | factor`, `factor -> NUM | "(" expr ")"` with a right-associative `^` that binds more tightly than `*`, so that `2 ^ 3 ^ 2` is 512 and `2 * 3 ^ 2` is 18.

SOLUTION

Add one level between `term` and `factor`, and make it recurse on the right:

```
term -> term "*" power | power
power -> factor "^" power | factor
factor -> NUM | "(" expr ")"
```

Because `power` is below `term`, every `^` is grouped before any `*` around it, so `2 * 3 ^ 2` is `2 * (3 ^ 2) = 18`. Because the recursion is on the right, `2 ^ 3 ^ 2` derives as `2 ^ (3 ^ 2) = 29 = 512`, which is the mathematical convention. A left-recursive rule would give $(2^3)^2 = 64$. Right recursion is also fine for a top-down parser, which only has trouble with left recursion. Where unary minus goes is a separate decision: Python makes `-2 ** 2` equal to `-4`, by placing unary minus above `**` in the grammar, so the exponent binds first.

EXERCISE 3.2 An unambiguous if

Write a grammar for `if c then S`, `if c then S else S` and `s` that allows only the parse in which each `else` belongs to the nearest `if`.

SOLUTION

Split statements into **matched** ones, in which every `if` has an `else`, and **open** ones, which contain at least one `if` without one:

```

S      -> matched | open
matched -> "if" c "then" matched "else" matched | s
open   -> "if" c "then" S
        | "if" c "then" matched "else" open

```

The key restriction is that the statement between `then` and `else` must be `matched`. If it contained an `if` without an `else`, the following `else` would have to belong to that inner `if`, by the nearest-`if` rule. So in `if c then if c then s else s` the only derivation is the outer statement as `open`, with the inner `if c then s else s` as a `matched` statement. Adapting the example's `parses` function to this grammar and counting confirms one parse. The grammar is correct but hard to read and to extend, which is why language references usually state the rule in a sentence.

EXERCISE 3.3 Is it LL(1)?

Is `S -> "a" "b" | "a" "c"` LL(1)? If not, rewrite it so that it is. Then do the same for `stmt -> IDENT "=" expr ";" | IDENT "(" args ")" ";"`, an assignment or a call.

SOLUTION

No: FIRST of both productions is `{ a }`, so on seeing `a` the parser can't choose. The fix is **left factoring**: pull out the common prefix and decide afterwards. `S -> "a" S'` and `S' -> "b" | "c"` accept the same strings, and now the choice is made on the second token, where the alternatives differ. The statement grammar has the same problem, since both start with `IDENT`, and the same fix: `stmt -> IDENT rest`, `rest -> "=" expr ";" | "(" args ")" ";"`. In practice, real assignment targets can be `a.b[i]`, an expression of any length, so no fixed amount of lookahead suffices. Hand-written parsers therefore parse the left side as an ordinary expression, and when they then find `=`, check that the expression is something that can be assigned to. Chapter 4's exercise 4.2 builds exactly that.

CHECKLIST

- ☐ Can you write a grammar in EBNF and draw the parse tree for a sentence it derives?
- ☐ Can you show that a grammar is ambiguous by giving two trees for one input?

- ☐ Can you encode precedence with levels and associativity with the side of recursion?
- ☐ Can you remove left recursion and left-factor a grammar?
- ☐ Can you compute FIRST and FOLLOW sets and say whether a grammar is LL(1)?

GO DEEPER

BOOK Crafting Interpreters, chapter 5: Representing Code Robert Nystrom

PEP PEP 617: New PEG parser for CPython van Rossum, Galindo & Nikolaou

MANUAL Bison manual: Shift/Reduce Conflicts GNU Project

Parsing

3 exercises

Most production compilers parse with hand-written code rather than a generated parser: GCC, Clang, Go, rustc, V8 and the TypeScript compiler all do. They use recursive descent for statements and declarations, and for expressions usually a technique due to Vaughan Pratt that handles every precedence level with one short function. This chapter builds such a parser, explains binding powers, the idea that makes it work, and shows how a parser can report several errors in one pass by recovering after each.

Recursive descent

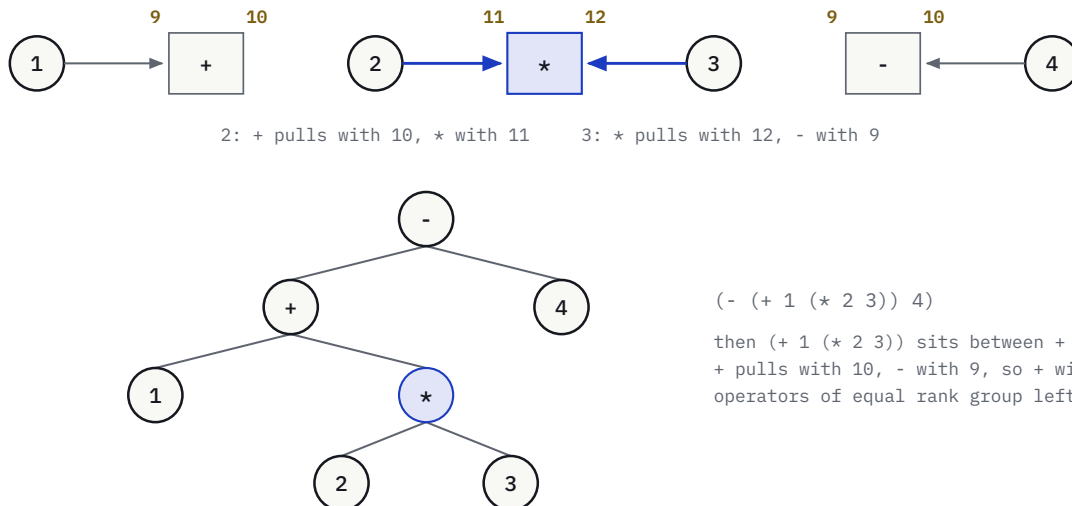
A **recursive-descent** parser has one function per nonterminal, and each function's body follows its production. `while (cond) body` becomes: expect `while`, expect `(`, call `expression()`, expect `)`, call `statement()`. Alternatives become an `if` on the next token, repetition becomes a loop, and nesting is handled by the functions calling each other, with Python's own call stack holding the parser's state. Chapter 1's parser was one, with a function for each of `expr`, `term` and `factor`.

The style is popular because the parser is ordinary code. It can be stepped through in a debugger, given special cases where the grammar is awkward, and made to produce precise error messages, because at every point the code knows what it was trying to parse. Its weakness is expressions. A language with a dozen precedence levels needs a dozen nearly identical functions, and parsing the single token `x` descends through all of them.

Binding power

Pratt's idea, published in 1973, is to give each operator a number, its **binding power**, saying how strongly it holds the operands next to it. In `1 + 2 * 3`, the `2` is between `+` and `*`, and goes to whichever pulls harder. Giving each infix operator a pair of numbers, a left and a right binding power, captures associativity too. For `+` the example uses (9, 10). In `a - b - c`, `b` sits between the first `-`'s right power, 10, and the second's left power, 9, so the first wins, and subtraction is left-associative. For `^` the pair is reversed, (14, 13), so the right-hand operator wins and `2 ^ 3 ^ 2` groups to the right.

EACH OPERAND GOES TO THE OPERATOR THAT BINDS IT MORE TIGHTLY



Binding powers decide the tree. Each operand joins whichever neighbouring operator pulls with the larger number. Once $2 * 3$ has formed, it is an operand too, and the same comparison puts it under the $+$.

The algorithm turns this into a loop. `expr(min_power)` parses one operand, then looks at the next operator. If that operator's left power is less than `min_power`, the operand belongs to some outer call, so it returns. Otherwise it consumes the operator, parses the right-hand side by calling `expr` with the operator's right power as the new minimum, builds the node, and loops. Prefix operators parse their operand with their own power: in the example unary minus has 13, which is tighter than `*` but looser than `^`'s left power of 14. That makes `-2 ^ 2` mean `-(2 ^ 2)`, as in mathematics. Calls fit as a postfix operator: `(` after an expression with power 15 starts an argument list, so `f(x)(3)` is a call of a call.

EXAMPLE

`pratt.py` · a Pratt parser for expressions with prefix operators, calls and right-associative power, and statement-level error recovery

```
import re

def lex(src):
    toks = []
    for line_no, line in enumerate(src.splitlines(), 1):
        for m in re.finditer(r"\s*(\d+|[A-Za-z_]\w*|==|!=|<=|>=|\S)", line):
            toks.append((m.group(1), line_no))
    return toks + [("<eof>", line_no)]

# Binding powers: how tightly each infix operator holds its operands. (left, right) power:
# right > left makes it left-associative; right < left makes it right-associative.
INFIX = {
    "or": (1, 2), "and": (3, 4), "==" : (5, 6), "!=" : (5, 6), "<" : (7, 8), "<=" : (7, 8),
    ">" : (7, 8), ">=" : (7, 8), "+" : (9, 10), "-" : (9, 10), "*" : (11, 12), "/" : (11, 12), "%" : (11, 12),
    "^" : (14, 13), "(" : (15, None)}
PREFIX_POWER = 13
# unary - and ! bind tighter than *
# but not ^

class ParseError(Exception): pass
```

```

class Parser:
    def __init__(self, src): self.toks, self.i, self.errors = lex(src), 0, []
    def peek(self): return self.toks[self.i][0]
    def next(self): self.i += 1; return self.toks[self.i - 1][0]
    def expect(self, t):
        if self.peek() != t: self.fail(f"expected {t!r} but found {self.peek()!r}")
        self.next()
    def fail(self, msg): raise ParseError(f"line {self.toks[self.i][1]}: {msg}")

    def expr(self, min_power=0):
        # Pratt's algorithm
        t = self.next()
        if t.isdigit() or (t[0].isalpha() and t not in INFIX): left = t
        elif t == "(": left = self.expr(); self.expect("(")
        elif t in ("-", "!"): left = f"({t} {self.expr(PREFIX_POWER)})"
        else: self.i -= 1; self.fail(f"expected an expression but found {t!r}")
        while self.peek() in INFIX:
            op = self.peek(); lp, rp = INFIX[op]
            if lp < min_power: break # the operator belongs to an outer call
            self.next()
            if op == "(": # a call: f(a, b)
                args = []
                while self.peek() != ")":
                    args.append(self.expr())
                    if self.peek() != ")": self.expect(",")
                self.next(); left = f"(call {left}{''.join(' ' + a for a in args)})"
            else:
                left = f"({op} {left} {self.expr(rp)})"
        return left

    def statement(self):
        if self.peek() == "let":
            self.next(); name = self.next()
            if not name[0].isalpha(): self.i -= 1; self.fail(f"expected a variable name but found {name!r}")
            self.expect("="); value = self.expr(); self.expect(";")
            return f"(let {name} {value})"
        if self.peek() == "print":
            self.next(); value = self.expr(); self.expect(";"); return f"(print {value})"
        value = self.expr(); self.expect(";"); return value

    def program(self):
        out = []
        while self.peek() != "<eof>":
            try: out.append(self.statement())
            except ParseError as e:
                self.errors.append(str(e)) # record it, then resynchronise:
                while self.peek() not in (";", "<eof>", "let", "print"): self.next()
                if self.peek() == ";": self.next() # skip to the end of the broken statement
        return out

for src in ("1 + 2 * 3", "a - b - c", "2 ^ 3 ^ 2", "-a * b", "-2 ^ 2",
            "!(x == y) or z < 1 and w", "f(x, g(y, 2))(3)"):
    print(f"{src:28} -> {Parser(src).expr()}")

broken = """let a = 1 + ;
print a * (2 + 3;
let = 5;
print a;
let b = a + 1;"""
p = Parser(broken)
good = p.program()
print(f"\na program with three mistakes: {len(p.errors)} errors reported, "
      f"{len(good)} statements still parsed")
for e in p.errors: print(f"    error, {e}")

```

```
for g in good: print(f"    parsed: {g}")
```

OUTPUT

```
1 + 2 * 3          -> (+ 1 (* 2 3))
a - b - c          -> (- (- a b) c)
2 ^ 3 ^ 2          -> (^ 2 (^ 3 2))
-a * b             -> (* (- a) b)
-2 ^ 2             -> (- (^ 2 2))
!(x == y) or z < 1 and w -> (or (! (= x y)) (and (< z 1) w))
f(x, g(y, 2))(3)    -> (call (call f x (call g y 2)) 3)
```

```
a program with three mistakes: 3 errors reported, 2 statements still parsed
error, line 1: expected an expression but found ';'
error, line 2: expected ')' but found ';'
error, line 3: expected a variable name but found '='
parsed: (print a)
parsed: (let b (+ a 1))
```

The whole precedence table is the dictionary `INFIX`, and adding an operator is one line there. That's the practical appeal: precedence lives in data, not in the shape of a dozen functions, and a language can even let programs declare new operators with their own precedence, as Haskell and Swift do, by adding rows to the table. Pratt parsing and the older **precedence climbing** algorithm turn out to be the same algorithm written differently.

Building trees

The example builds its trees as S-expression strings to keep the output short. A real parser builds objects, one class or record per kind of node: `Binary(op, left, right)`, `Call(callee, args)`, `If(cond, then, else_)`. This is the **abstract syntax tree**, the AST. It's abstract because it leaves out what the grammar needed only for parsing: parentheses, semicolons, the chain of single-child nodes that chapter 3's precedence levels would produce for a lone `x`. Each node also records its source position, for error messages in later stages. Python's `ast` module shows a real one: `ast.dump(ast.parse("x + 2 * y"))` prints `BinOp` nodes with `Name` and `Constant` leaves, and every node carries `lineno` and `col_offset`.

Some tools keep everything instead. A **concrete syntax tree** preserves every token, including spaces and comments, so that the source can be reproduced exactly. Formatters and refactoring tools need that, and so do editors: tree-sitter, used by many editors for syntax highlighting, builds concrete trees and updates them incrementally as you type, reusing every subtree that the edit didn't touch.

Errors and recovery

A parser that stops at the first error makes the programmer fix one mistake per compile. Reporting several needs **error recovery**: after an error, get the parser back into a state where it can continue, without reporting a cascade of further errors that are really the same one. The simplest scheme, used by the example, is **panic mode**. When a statement fails to parse, record the error, skip tokens until one that can only appear at a statement

boundary, such as `;` or a keyword that begins a statement, and resume from there. The example's program has three broken statements, and the parser reported all three and still parsed the two good ones.

The quality of messages matters as much as the number. "Expected an expression but found `'`" names what the parser wanted and what it got, and with a source position points at the problem. Better still is a message that guesses the intent: Clang's "expected `'` after expression" with a suggested fix, Rust's suggestions of the likely missing token, and Python 3.10's `"(' was never closed"`, which points at the opening parenthesis rather than the place, possibly many lines later, where the parser noticed. Those messages come from parsers that are written by hand and given special cases for common mistakes, which is one more reason production compilers write their own.

When to use a generator

Parser generators still earn their place. A generator checks the grammar for ambiguity, which a hand-written parser never does, and for a language whose grammar changes often, regenerating is easier than editing code. PostgreSQL's SQL parser is generated by Bison from a grammar of thousands of lines, and ANTLR is widely used for domain-specific languages and data formats. Ruby went the other way: its parser, generated from the famously complex `parse.y`, was replaced as the default in Ruby 3.4 by Prism, a hand-written recursive-descent parser designed to be shared by editors and other Ruby implementations. For learning, write one by hand first, and then a generator's decisions and conflict reports make sense.

COMMON TRAPS

Getting associativity backwards by giving both sides of an operator the same binding power.

Forgetting that unary minus and power interact, so `-2 ^ 2` comes out as 4.

Recovering by skipping one token, which produces a cascade of errors that are all the same mistake.

Throwing away source positions when building tree nodes.

Parsing with unbounded backtracking, which can take exponential time on unlucky input.

EXERCISES

EXERCISE 4.1 A postfix operator

Add postfix `!` for factorial, binding more tightly than every infix operator, so that `2 * 3!` is `(* 2 (! 3))` and `-3!` is `(- (! 3))`. Unary `!` for "not" already exists as a prefix operator. Does that cause a conflict?

SOLUTION

A postfix operator is handled in the loop, like the call operator `(`: after parsing an operand, if the next token is `!` with left power at least `min_power`, consume it and wrap the operand. Give it a left power of 16, above everything else, and no right power, since it takes no right operand:

```
POSTFIX = {"!": 16}
# inside the loop in expr(), before the infix case:
if op in POSTFIX:
    if POSTFIX[op] < min_power: break
    self.next(); left = f"(! {left})"; continue
```

The loop's condition must also accept `!` as an operator, by checking `POSTFIX` as well as `INFIX`. With 16 above unary minus's 13, `-3!` parses the operand of the minus with minimum 13, finds `!` with power 16, and binds it first, giving `(- (! 3))`. There's no conflict with prefix `!`, because the two are never in the same position: a prefix operator appears where an operand is expected, at the start of `expr`, and a postfix one where an operator is expected, in the loop. The parser always knows which it's looking for. A real language would still think twice, because `a ! b` with a missing operator now reads as a factorial followed by a stray name.

EXERCISE 4.2 Assignment as an expression

Add assignment, `a = b = 0`, as a right-associative infix operator with the lowest binding power. How do you reject `1 + a = 3` and `f() = 3`, where the left side can't be assigned to?

SOLUTION

Add `"=": (0.5, 0.4)` to `INFIX`, a lower left power than `or` and a right power below the left, for right associativity, so `a = b = 0` is `(= a (= b 0))`. Then, when building an assignment node, check the left operand: only a variable name, and in a fuller language a field access `a.b` or an index `a[i]`, is an **lvalue**, something with a location to store into. Anything else gets the error "invalid assignment target", reported at the `=`. For `1 + a = 3`, the parser first builds `(+ 1 a)`, because `+` binds more tightly, then finds `=` and rejects `(+ 1 a)` as a target. For `f() = 3`, the target is a call, also rejected. This parse-then-check

approach is what Crafting Interpreters uses, and it solves exercise 3.3's problem: the parser doesn't need to know in advance whether it's reading an assignment, because it parses the left side as an expression and checks it afterwards.

EXERCISE 4.3 Chained comparisons

The example parses `a < b < c` as `(< (< a b) c)`. What does that mean in C, and what does it mean in Python? How would you make the parser produce Python's meaning?

SOLUTION

In C, comparisons are left-associative operators that return 0 or 1, so `a < b < c` compares the result of `a < b`, either 0 or 1, with `c`. With `a = 5, b = 3, c = 2` that's `0 < 2`, true, which is rarely what anyone meant, and compilers warn about it. In Python, `a < b < c` means `a < b` and `b < c`, with `b` evaluated once. To produce that, give comparisons non-associative handling: when the loop has just built a comparison and the next operator is also a comparison, don't nest. Collect the operands and operators into one node, `Compare(a, [<, <], [b, c])`, which is exactly the node Python's `ast` module produces. A third option, taken by Rust, is to make chained comparisons a syntax error, which removes the trap without choosing a meaning.

CHECKLIST

- ☐ Can you write a recursive-descent parser for statements from a grammar?
- ☐ Can you explain binding power and how a pair of powers sets associativity?
- ☐ Can you write a Pratt parser with prefix, infix and postfix operators from memory?
- ☐ Can you explain what an abstract syntax tree leaves out, and when a concrete tree is needed?
- ☐ Can you implement panic-mode error recovery so a parser reports several errors in one run?

GO DEEPER

BOOK Crafting Interpreters, chapter 6: Parsing Expressions Robert Nystrom

ARTICLE Simple but Powerful Pratt Parsing Alex Kladov

Names, scopes and resolution

3 exercises

*The parser knows that `n = n + 1` is an assignment, but not which `n` it assigns: the one declared two lines up, a parameter, a variable in an enclosing function, or a global defined later in the file. Answering that question for every name is **resolution**, and it's the first stage of semantic analysis. A resolver walks the tree once before the program runs. It catches mistakes the grammar can't express, and it turns each variable into a position the running program can reach without searching. This chapter builds one.*

Scope

A declaration's **scope** is the region of the program where its name refers to it. Almost every modern language uses **lexical** scope, also called static scope: the region is determined by the program's text, so you can tell which declaration a name means by reading outward from it, block by block, until you find one. An inner declaration with the same name **shadows** the outer one within its own region. Lexical scope is what makes closures work. A function remembers the variables of the place where it was written, wherever it's later called from, which is the subject of chapter 10.

The alternative is **dynamic** scope, in which a name refers to the most recent binding still active at run time, whichever function made it. Early Lisps used it, Emacs Lisp had nothing else until lexical binding was added in 2012, and a variable declared `local` in a Bash function is dynamically scoped: functions it calls see it. Dynamic scope makes a function's behaviour depend on who called it, which is occasionally useful, as with configuration that should apply to everything a call does, and usually a source of bugs. Exercise 5.2 shows the difference.

Languages also differ in where a scope begins. In C and Java, a variable's scope starts at its declaration. In JavaScript, a `var` is **hoisted** to the top of its function, so it can be read before the line that declares it, and is `undefined` there. The newer `let` is also in scope from the start of its block, but reading it before the declaration throws an error. The region in between is called the **temporal dead zone**. Python decides a variable is local to a function if it's assigned anywhere in the function, which leads to the error in exercise 5.1.

A resolver

A resolver walks the tree keeping a stack of scopes, each a table from name to information about the declaration: its slot number, whether its initialiser has finished, and whether anything has used it. A block or function body pushes a scope and pops it at the end. A declaration adds a name to the innermost scope. A use searches the stack from the innermost scope outward, and the number of scopes it had to go up is its **depth**. A name

found in no local scope is assumed to be global. The example's resolver runs on two programs given as trees: a counter built from a closure, and a function with several mistakes in it.

EXAMPLE

resolver.py · resolving every name to a scope distance and slot, and reporting what the grammar can't catch

```
# A resolver walks the syntax tree once, before the program runs, and works out which
# declaration
# every name refers to. Programs are given as the tree a parser would produce, with line
# numbers.

class Resolver:
    def __init__(self):
        self.scopes, self.in_function, self.notes, self.errors = [], 0, [], []
        self.globals, self.global_uses = {"get"}, [] # get is a built-in function
    def declare(self, name, line):
        if not self.scopes: self.globals.add(name); return # globals are looked up by name at
            run time
        scope = self.scopes[-1]
        if name in scope: self.errors.append(f"line {line}: '{name}' is already declared in
            this scope")
        scope[name] = {"slot": len(scope), "ready": False, "used": False, "line": line}
    def define(self, name):
        if self.scopes: self.scopes[-1][name]["ready"] = True
    def lookup(self, name, line, what="read"):
        for hops, scope in enumerate(reversed(self.scopes)):
            if name in scope:
                v = scope[name]
                if not v["ready"]:
                    self.errors.append(f"line {line}: can't read '{name}' in its own
                        initializer")
                v["used"] = True
                where = f"local, {hops} scope{'s' if hops != 1 else ''} up, slot {v['slot']}"
                self.notes.append(f"line {line}: {what:5} {name:6} -> {where}")
                return
        self.notes.append(f"line {line}: {what:5} {name:6} -> global, found by name at run
            time")
        self.global_uses.append((name, line))
    def begin(self): self.scopes.append({})
    def end(self):
        for name, v in self.scopes.pop().items():
            if not v["used"]: self.errors.append(f"line {v['line']}: warning: '{name}' is never
                used")
    def resolve(self, node):
        kind = node[0]
        if kind == "block":
            self.begin()
            for s in node[1]: self.resolve(s)
            self.end()
        elif kind == "let":
            _, name, init, line = node
            self.declare(name, line); self.resolve(init); self.define(name)
        elif kind == "fun":
            _, name, params, body, line = node
            self.declare(name, line); self.define(name) # defined first, so it can call
                itself
            self.in_function += 1; self.begin()
            for p in params: self.declare(p, line); self.define(p); self.scopes[-1][p]["used"]
                = True
            for s in body: self.resolve(s)
```

```

        self.end(); self.in_function -= 1
    elif kind == "var": self.lookup(node[1], node[2])
    elif kind == "assign": self.resolve(node[2]); self.lookup(node[1], node[3], "write")
    elif kind == "return":
        if not self.in_function: self.errors.append(f"line {node[2]}: 'return' outside a
            function")
        self.resolve(node[1])
    elif kind in ("print", "expr"): self.resolve(node[1])
    elif kind == "while": self.resolve(node[1]); self.resolve(node[2])
    elif kind == "binop": self.resolve(node[2]); self.resolve(node[3])
    elif kind == "call": self.resolve(node[1]); [self.resolve(a) for a in node[2]]
    # "num" needs nothing

def run(title, program):
    r = Resolver()
    for stmt in program: r.resolve(stmt)
    for name, line in r.global_uses:
        # globals may be declared later in
        the file,
        if name not in r.globals:
            # so check them once everything is
            seen
            r.errors.append(f"line {line}: '{name}' is not declared anywhere")
    print(title)
    for n in r.notes: print(f"    {n}")
    for e in r.errors: print(f"    {e}")
    if not r.errors: print("    no errors")

# let count = 0;
# fun make_counter() {
#   let n = 0;
#   fun inc() { n = n + 1; count = count + 1;
#     return n; }
#   return inc;
# }
counter = [
    ("let", "count", ("num", 0), 1),
    ("fun", "make_counter", [], [
        ("let", "n", ("num", 0), 3),
        ("fun", "inc", [], [
            ("assign", "n", ("binop", "+", ("var", "n", 4), ("num", 1)), 4),
            ("assign", "count", ("binop", "+", ("var", "count", 4), ("num", 1)), 4),
            ("return", ("var", "n", 5), 5)], 4),
        ("return", ("var", "inc", 6), 6)], 2)]
run("a counter with a closure:", counter)

# fun average(xs, n) {
#   let total = 0; let unused = 1;
#   let i = 0;
#   while (i < n) { let i = i + 1;
#     total = total + get(xs, i); }
#   let total = total / n;
#   return totl;
# }
# return 0;
mistakes = [
    ("fun", "average", ["xs", "n"], [
        ("let", "total", ("num", 0), 2), ("let", "unused", ("num", 1), 2),
        ("let", "i", ("num", 0), 3),
        ("while", ("binop", "<", ("var", "i", 4), ("var", "n", 4)), ("block", [
            ("let", "i", ("binop", "+", ("var", "i", 4), ("num", 1)), 4),
            ("assign", "total", ("binop", "+", ("var", "total", 5),
                ("call", ("var", "get", 5), [(("var", "xs", 5), ("var", "i",
                    5))], 5))), 5))),
        ("let", "total", ("binop", "/", ("var", "total", 6), ("var", "n", 6)), 6),
        ("return", ("var", "totl", 7), 7)], 1),
    ("return", ("num", 0), 9)]
run("\na function with mistakes:", mistakes)

```

OUTPUT

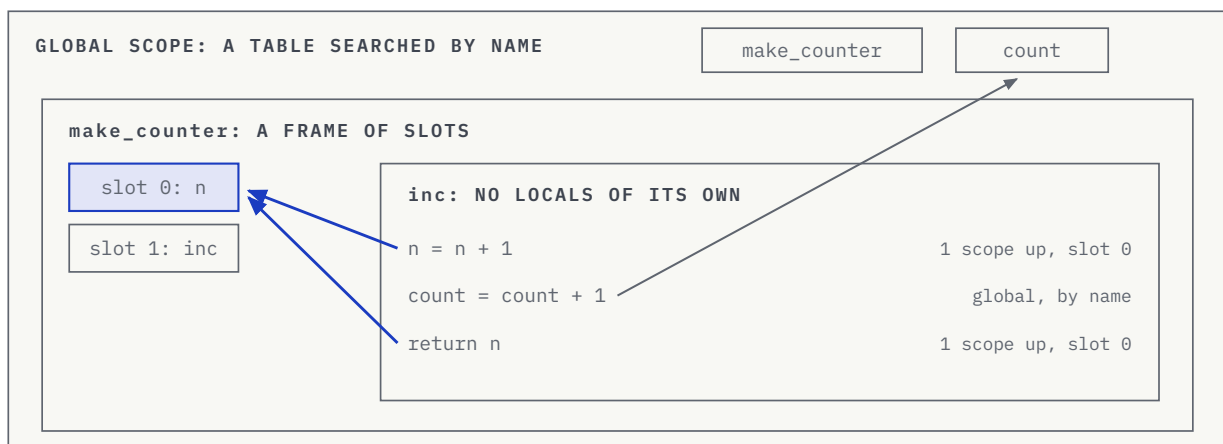
a counter with a closure:

```
line 4: read  n      -> local, 1 scope up, slot 0
line 4: write n      -> local, 1 scope up, slot 0
line 4: read  count  -> global, found by name at run time
line 4: write count  -> global, found by name at run time
line 5: read  n      -> local, 1 scope up, slot 0
line 6: read  inc     -> local, 0 scopes up, slot 1
no errors
```

a function with mistakes:

```
line 4: read  i      -> local, 0 scopes up, slot 4
line 4: read  n      -> local, 0 scopes up, slot 1
line 4: read  i      -> local, 0 scopes up, slot 0
line 5: read  total   -> local, 1 scope up, slot 2
line 5: read  get     -> global, found by name at run time
line 5: read  xs      -> local, 1 scope up, slot 0
line 5: read  i      -> local, 0 scopes up, slot 0
line 5: write total   -> local, 1 scope up, slot 2
line 6: read  total   -> local, 0 scopes up, slot 5
line 6: read  n      -> local, 0 scopes up, slot 1
line 7: read  totl    -> global, found by name at run time
line 4: can't read 'i' in its own initializer
line 6: 'total' is already declared in this scope
line 6: can't read 'total' in its own initializer
line 2: warning: 'unused' is never used
line 9: 'return' outside a function
line 7: 'totl' is not declared anywhere
```

For the counter, each use of `n` inside `inc` resolves to "1 scope up, slot 0": the first variable of the function one level out. That pair of numbers is all the running program needs. An interpreter can follow one link outward and index slot 0, instead of looking up the string "n" in a dictionary at every step. Chapter 9 counts what that saves. `count` is global, and globals are handled differently: they're looked up by name when the program runs, for a reason exercise 5.3 explores.



What the resolver works out for the counter. Local variables become a scope distance and a slot number, so the running program reaches them directly. Globals stay names, looked up when the line runs.

The second program shows the errors a resolver can report. None of them is a syntax error, since each statement is well formed, and none needs types.

- **Reading a variable in its own initialiser.** In `let i = i + 1` inside the loop, the new `i` is declared but not yet ready, so the `i` on the right refers to it, not to the outer `i`. Most languages forbid this, because the value would be undefined. The resolver marks a variable "declared" when its name is added and "defined" only after its initialiser is resolved.
- **Declaring a name twice in one scope**, as with the second `let total`. Shadowing an outer variable is allowed. Redeclaring in the same scope is almost always a mistake.
- **A variable that's never used.** This one is a warning, not an error: `unused` was declared and never read.
- **return outside a function.** The resolver knows whether it's inside a function body, so it can reject this.
- **A name declared nowhere**, such as the misspelt `totl`. The check is made only after the whole file has been resolved, because a global may be declared later in the file than the function that uses it.

Symbol tables

A compiler keeps the results of resolution in a **symbol table**: for every declaration, its name, kind, scope, type once known, and storage location. Every later stage consults it. The type checker records types in it, the code generator looks up where each variable lives, and the debugger uses a copy of it, saved in the executable as debugging information, to show variables by name. Python's `symtable` module shows CPython's. For each function it lists which names are local, global, free (used here but belonging to an enclosing function) and cell (belonging here but captured by an inner function). Those last two categories are how CPython decides which variables a closure must keep alive, the subject of chapter 10.

Resolution also has to handle names that aren't variables. Fields are looked up in the type of the object, which the resolver may not know until type checking. Methods may be found through a chain of classes. Modules add another level: `import` creates a name whose members belong to another file. Overloaded functions, as in C++ and Java, share one name, and choosing between them needs the argument types, so resolution and type checking are interleaved. The more a language puts in its names, the later resolution has to happen.

COMMON TRAPS

Resolving names while executing instead of in a separate pass, so every loop iteration repeats the search.

Adding a name to the scope before resolving its initialiser, without tracking that it isn't ready yet.

Reporting undeclared globals too early, before the rest of the file has declared them.

Confusing shadowing with redeclaration, and rejecting the first or allowing the second.

Forgetting that loops, conditionals and functions each open their own scope, or don't, depending on the language.

EXERCISES

EXERCISE 5.1 UnboundLocalError

In Python, this raises `UnboundLocalError` on the `print` line, although `x` is a global with the value 1 when it runs. Explain, in terms of resolution, why. What are the two ways to fix it?

```
x = 1
def f():
    print(x)
    x = 2
f()
```

SOLUTION

CPython resolves names when it compiles the function, not when each line runs, and its rule is that a name assigned anywhere in a function is local to the whole function. Because of `x = 2`, every `x` in `f` refers to a local slot, including the one in `print(x)` on the line before. When that line runs, the slot hasn't been assigned yet, and Python reports it as an unbound local. It doesn't fall back to the global, because the decision was made at compile time and the bytecode reads a local slot: `dis` shows `LOAD_FAST_CHECK`, a local read that checks the slot has been assigned. One fix is to declare `global x` in `f`, if the function is meant to change the global. The other is to use a different name for the local. The same rule produces the error inside nested functions, where `nonlocal x` is the declaration that says "this `x` belongs to the enclosing function".

EXERCISE 5.2 Lexical or dynamic

What does this program print under lexical scope, and what under dynamic scope?

```
let x = "global";
fun show() { print x; }
fun test() { let x = "local"; show(); }
test();
```

SOLUTION

Under **lexical** scope it prints `global`. The `x` in `show` is resolved by reading outward from where `show` is written, and the only enclosing declaration is the global one. `test`'s local `x` is invisible to `show`, because `show` isn't written inside `test`. Under **dynamic** scope it prints `local`. The `x` in `show` means the most recent active binding at the time of the call, and when `show` runs, `test`'s binding is active. A resolver can only do its work in advance under lexical scope, since under dynamic scope the answer depends on the run-time call chain. Bash behaves the second way for variables declared `local`, which is worth knowing when a shell function seems to read a variable it never declared.

EXERCISE 5.3 Why globals are late

The example's resolver gives local variables slots, but leaves globals to be looked up by name at run time, and only reports an undeclared global after the whole file is resolved. Give two programs that would break if globals were resolved like locals, at the point of use.

SOLUTION

First, **mutual recursion** between top-level functions: `fun is_even(n)` calls `is_odd`, and `fun is_odd(n)`, declared after it, calls `is_even`. Whichever is resolved first refers to a function that isn't declared yet. Resolving globals by name, at run time or after the whole file is seen, handles any order. Second, the **REPL**, where a programmer defines a function that uses a helper and defines the helper on the next line: the file is never complete, so each global must be found when it's used. Some languages make the other choice for safety. C requires a declaration before use, which is why C has prototypes and header files: `int is_odd(int);` at the top lets `is_even` call it before its definition. Crafting Interpreters' Lox takes the dynamic approach for globals and the static one for locals, as the example does.

CHECKLIST

- ☐ Can you explain lexical and dynamic scope, and shadowing, with an example of each?
- ☐ Can you write a resolver that turns local variables into a scope depth and slot?
- ☐ Can you list errors a resolver catches that a parser can't?

- ☐ Can you explain hoisting, the temporal dead zone and Python's local-if-assigned rule?
- ☐ Can you read the output of Python's `symtable` for a function with a closure?

GO DEEPER

BOOK Crafting Interpreters, chapter 11: Resolving and Binding Robert Nystrom

DOCS The Python Language Reference: Execution model, naming and binding
Python documentation

DOCS MDN: let, and the temporal dead zone Mozilla

PART II

Types

Three chapters on what a type checker knows and how it knows it. A type system classifies every expression by the kind of value it produces, and rejects programs that would combine values in ways that make no sense. Chapter 6 builds a checker and compares checking before a program runs with checking while it runs. Chapter 7 shows how a checker can work out the types itself, with no annotations at all. Chapter 8 adds generics and subtyping, and the rules, called variance, that say when a list of cats is a list of animals.

3 exercises

A type is a set of values together with the operations that make sense on them. A type error is an operation applied to a value outside its set: adding a string to a number, calling an integer. Every language detects type errors somewhere. A dynamically typed language detects them when the operation is about to run; a statically typed one proves, before the program runs, that they can't happen. This chapter builds a static type checker as a short recursive function, runs the same programs without it, and sorts out the vocabulary that people use loosely: strong and weak, sound and unsound, nominal and structural.

Checking as a walk over the tree

A type checker is another pass over the syntax tree, like the resolver of chapter 5, except that it computes a type for every expression instead of a location for every name. The rules are local: the type of a node follows from the types of its children. A number literal has type `int`. A variable has whatever type the environment, the checker's symbol table, records for it. `a + b` has type `int` if both `a` and `b` do, and is an error otherwise. Type theorists write rules like that last one as an inference rule, with the premises above the line and the conclusion below. Γ is the environment, and $\Gamma \vdash e : T$ reads "in environment Γ , expression e has type T ":

$$\frac{\Gamma \vdash a:\text{int} \quad \Gamma \vdash b:\text{int}}{\Gamma \vdash a + b:\text{int}}$$

A checker is those rules written as code, one case per kind of node, which is exactly what the example's `check` function is. A function literal declares its parameter types and result type; the checker extends the environment with the parameters, checks the body, and compares the body's type with the declared result. A call checks that the callee is a function, that the number of arguments is right, and that each argument has the declared parameter type. An `if` requires a `bool` condition and branches of the same type, because the result of the `if` is one or the other.

EXAMPLE

```
typecheck.py · a static type checker for a small language, and the same programs run
without it
```

```
import re

def read(src):
    """Read an S-expression: (f a (g b)) becomes ['f', 'a', ['g', 'b']]."""
    tokens = re.findall(r'\(|\)|"([^\"]*)"|^\s\(|'+',', src)
```

```

def form():
    t = tokens.pop(0)
    if t != "(": return int(t) if t.lstrip("-").isdigit() else t
    out = []
    while tokens[0] != ")": out.append(form())
    tokens.pop(0); return out
return form()

class TypeError_(Exception): pass

def show(t): return t if isinstance(t, str) else f"fn({'', '.join(map(show, t[0]))}) -> {show(t[1])}"

def check(e, env):
    """Return the type of expression e, or raise. Types: 'int', 'bool', 'str', or (params, result)."""
    if isinstance(e, int): return "int"
    if isinstance(e, str):
        if e.startswith("'"): return "str"
        if e in ("true", "false"): return "bool"
        if e not in env: raise TypeError_(f"unknown variable '{e}'")
        return env[e]
    op, *args = e
    if op in ("+", "-", "*", "<"):
        for a in args:
            if (t := check(a, env)) != "int": raise TypeError_(f"{'op'} needs int operands, got {show(t)}")
        return "bool" if op == "<" else "int"
    if op == "++":
        for a in args:
            if (t := check(a, env)) != "str": raise TypeError_(f"{'++'} joins strings, got {show(t)}")
        return "str"
    if op == "if":
        if (c := check(args[0], env)) != "bool":
            raise TypeError_(f"{'if'} needs a bool condition, got {show(c)}")
        a, b = check(args[1], env), check(args[2], env)
        if a != b: raise TypeError_(f"the branches of 'if' have different types: {show(a)} and {show(b)}")
        return a
    if op == "let":
        # (let (name value) body)
        (name, value), body = args
        return check(body, {**env, name: check(value, env)})
    if op == "fn":
        # (fn ((x int) ...) result body)
        params, result, body = args
        got = check(body, {**env, **{p: t for p, t in params}})
        if got != result: raise TypeError_(f"function declared to return {result} returns {show(got)}")
        return ([t for _, t in params], result)
    f = check(op, env)
    # a call
    if isinstance(f, str): raise TypeError_(f"{show(f)} is not a function")
    if len(args) != len(f[0]): raise TypeError_(f"expected {len(f[0])} arguments, got {len(args)}")
    for i, (a, want) in enumerate(zip(args, f[0]), 1):
        if (got := check(a, env)) != want:
            raise TypeError_(f"argument {i} should be {want}, got {show(got)}")
    return f[1]

def evaluate(e, env):
    # the same language, run without checking
    if isinstance(e, int): return e
    if isinstance(e, str):
        if e.startswith("'"): return e.strip("'")
        return {"true": True, "false": False}.get(e, env.get(e))
    op, *args = e
    if op == "if": return evaluate(args[1] if evaluate(args[0], env) else args[2], env)

```

```

if op == "let": (name, value), body = args; return evaluate(body, {**env, name:
    evaluate(value, env)})
if op == "fn":
    params, _, body = args
    names = [p for p, _ in params]
    return lambda *vs: evaluate(body, {**env, **dict(zip(names, vs))})
vals = [evaluate(a, env) for a in args]
if op == "+": return vals[0] + vals[1]
if op == "-": return vals[0] - vals[1]
if op == "*": return vals[0] * vals[1]
if op == "<": return vals[0] < vals[1]
if op == "++": return "".join(vals) # fails if a value isn't a string
return evaluate(op, env)(*vals)

programs = {
    "fine": '(let (sq (fn ((x int)) int (* x x))) (+ (sq 3) 1))',
    "a function value": '(fn ((n int) (s str)) str (++ s s))',
    "wrong argument": '(let (sq (fn ((x int)) int (* x x))) (sq "3"))',
    "too many arguments": '(let (sq (fn ((x int)) int (* x x))) (sq 3 4))',
    "unknown name": '(+ 1 cuont)',
    "branches disagree": '(let (label (fn ((n int)) str (if (< n 0) "negative" n))) (++ (label
        5) "!"))',
}
for name, src in programs.items():
    try: print(f"{name:20} type {show(check(read(src), {}))}")
    except TypeError_ as e: print(f"{name:20} rejected before running: {e}")

src = programs["branches disagree"]
print("\nthe last program, run without the checker:")
label = evaluate(read(src)[1][1], {})
print(f"    label(-1) = {label(-1)!r}: fine, the bug isn't on this path")
try: print(evaluate(read(src), {}))
except TypeError as e: print(f"    label(5) ++ '!' fails only when it runs: TypeError: {e}")

```

OUTPUT

```

fine                type int
a function value     type fn(int, str) -> str
wrong argument       rejected before running: argument 1 should be int, got str
too many arguments   rejected before running: expected 1 arguments, got 2
unknown name         rejected before running: unknown variable 'cuont'
branches disagree    rejected before running: the branches of 'if' have different types: str
                    and int

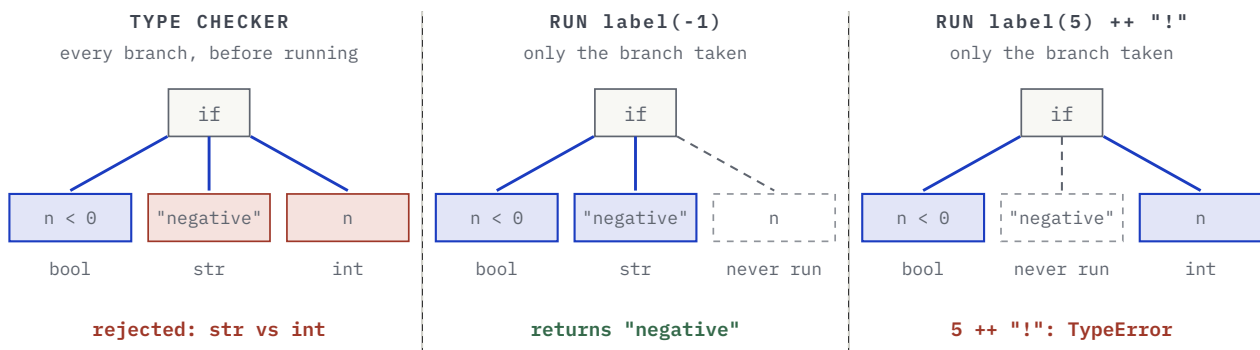
the last program, run without the checker:
label(-1) = 'negative': fine, the bug isn't on this path
label(5) ++ '!' fails only when it runs: TypeError: sequence item 0: expected str instance,
int found

```

Every one of the four bad programs was rejected before running, with a message naming the problem. The last of them is the interesting one. Its function `label` returns a string for negative numbers and the number itself otherwise, so its result type depends on its argument, and the checker rejects it. Run without the checker, the same function works for `-1`. The bug appears only when some later call passes a positive number and the caller tries to join the result to a string. The program fails far from the mistake, in a line that's correct by itself.

Static and dynamic checking

That example is the whole trade-off in miniature. A static checker examines every path through the program once, before it runs, so it finds the error in the branch that the tests never took. A dynamic check examines only the path actually taken, each time it's taken, and so finds only errors that happen. Python, JavaScript, Ruby and Lisp check dynamically: every value carries a tag saying its type, and every operation checks the tags of its operands. Java, C#, Go, Rust, Haskell and TypeScript check statically, and most of them also keep some dynamic checks for what the static system can't express, such as array bounds and casts.



The same function, checked and run. A static checker examines every branch before the program runs. At run time only the branch taken is examined, so the bug surfaces only for arguments that reach it, and only where the result is used.

Static checking has a price, which follows from a basic result in computability. Whether a program will ever perform a bad operation is, in general, undecidable: Rice's theorem says no algorithm can answer such questions exactly for every program. So any static checker must be **conservative**. It rejects some programs that would never actually go wrong. The example's checker rejects `(if true 1 "one")`, although the string branch can never run. The art of type system design is to make the rejected-but-correct programs rare and unnatural, while keeping the rules simple enough for programmers to predict.

In return, the static checker gives more than error detection. Types are documentation that can't go out of date. They let an editor complete method names and a refactoring tool rename safely. And they let a compiler generate better code: if `a` and `b` are known to be machine integers, `a + b` is one instruction, while a dynamic language must check both tags and choose an operation at run time. Chapter 18 shows how JIT compilers recover much of that speed by guessing types and checking the guess.

Sound and unsound

A type system is **sound** if a program it accepts can't produce a type error at run time. Robin Milner put it as "well-typed programs cannot go wrong", in his 1978 paper on type inference for ML. Soundness is a theorem about the language, usually proved by showing that each step of evaluation preserves types and that a well-typed program is never stuck. Standard ML and Haskell, without their explicit escape hatches, are sound in this sense.

Most popular languages aren't, by design or by accident. C's casts let any pointer be treated as any other, and that's the source of a large class of security bugs. Java's arrays are covariant, which the type checker can't make safe, so the JVM checks every store into an array at run time; chapter 8 shows why. TypeScript is deliberately unsound in several places, such as treating method parameters covariantly and trusting `as` casts, because its designers judged that strictness there would reject too much ordinary JavaScript. Rust's `unsafe` blocks mark the places where the programmer, not the checker, is responsible for safety. Knowing where a language's system is unsound tells you where its type errors can still come from.

Strong, weak, nominal, structural

"Strongly typed" and "weakly typed" have no agreed meanings, and it's clearer to ask two specific questions instead. First, does the language convert values implicitly? JavaScript evaluates `"1" + 1` to `"11"` and `"3" * "4"` to 12, converting as needed. Python raises an error for both. Second, can a program reinterpret memory, reading the bytes of a float as an integer or writing past the end of an array? C allows both, and memory-safe languages, Python and Java and safe Rust among them, allow neither. A language can be statically typed and memory-unsafe, like C, or dynamically typed and memory-safe, like Python.

The other distinction is how a language decides two types are compatible. In a **nominal** system, the one Java and C# use for classes, a type is compatible with another only if it's declared to be, by name: a class must say `implements Comparable`. In a **structural** system, a type is compatible with any other that has the right shape. Go's interfaces are satisfied by any type with the required methods, with no declaration, and TypeScript compares object types by their fields. Python's type hints support both: classes are nominal, and `typing.Protocol` (PEP 544) adds structural types, which is how static checking describes duck typing.

Gradual typing

Python and JavaScript programs now often carry type annotations that the language itself ignores at run time. Tools such as mypy and Pyright check Python's annotations, introduced by PEP 484, and TypeScript is JavaScript with a static checker attached whose types are erased in the output. This is **gradual typing**: annotated and unannotated code mix freely, and a special type, `Any` in Python and `any` in TypeScript, is compatible in both directions with everything, so the checker stays silent at the boundary. The benefit is that a large dynamic codebase can be typed one module at a time. The cost is that a wrong annotation, or an `Any` that leaks, gives no protection, because nothing checks at run time that values match their annotations unless a library such as Pydantic is used to do it explicitly.

COMMON TRAPS

Believing tests make types unnecessary, when tests check the paths they take and a type checker checks every path

.....
CHECKER CHECKS EVERY PATH.....

Believing types make tests unnecessary, when most bugs are wrong values of the right type.

Treating annotations in Python or TypeScript as run-time checks. They're checked only by tools, and erased or ignored when the program runs.

Letting `Any` spread until the checker silently checks nothing.

Arguing about "strong" typing instead of asking about implicit conversions and memory safety.

EXERCISES

EXERCISE 6.1 Rejected but correct

Give two programs that the example's checker rejects although they would run without a type error. For each, say what a more powerful type system would need to accept it.

SOLUTION

First, `(if true 1 "one")`. The string branch can never run, but the checker types both branches without evaluating the condition, and they disagree. Accepting it would need the checker to evaluate constant conditions, which some do for dead-code warnings, or a type system that tracks which branch is possible from the type of the condition, as dependent types can. Second, a function used at two types: `(let (id (fn ((x int)) int x)) ...)` can't also be applied to a string, because its parameter type is fixed at `int`. A second copy with a `str` parameter would work, and so would a type system with type parameters, `id : forall a. a -> a`, which chapter 7 infers automatically and chapter 8 calls generics. In general, each feature that makes a type system accept more correct programs makes it more complex to implement and to understand.

EXERCISE 6.2 Checking a let with a declared type

The example's `let` takes its variable's type from the value. Add an optional declared type, `(let (x int (+ 1 2)) body)`, that must match the value's type. What new error does this catch, and where should the message point?

SOLUTION

In the `let` case, accept a three-element binding `(name declared value)`. Check the value, compare its type with the declared one, and raise "`x is declared int but its value is str`" if they differ. Then check the body with `x` bound to the **declared** type, not the value's type: the declaration is a promise about how the variable will be used, and other code depends on it. The new error catches a value that doesn't match the programmer's intent, at the declaration, instead of wherever the variable is first misused, which may be far away. The message should point at the value expression, since that's what disagrees, and mention the declaration. The same reasoning is why many teams require annotated function signatures even in languages with inference: the annotation moves type errors to the boundary where a mistake is easiest to understand.

EXERCISE 6.3 Tags at run time

A dynamically typed language must check the types of `a` and `b` before doing `a + b`. Where does the information it checks come from, what does it cost, and why can a statically typed language like C omit it?

SOLUTION

Each value carries a tag recording its type. In CPython every object starts with a header containing a pointer to its type object, so `a + b` reads both type pointers and looks up the addition for that pair of types. Other implementations pack small tags into the value itself, for instance by using spare bits of a pointer or the unused bit patterns of a NaN float, so that small integers need no separate object. The cost is memory for the tags, a check and a branch for every operation, and the loss of optimisations that need to know types in advance. C omits all of this because its checker proved the types before the program ran. The compiled code for `a + b`, with both `int`, is a single add instruction with no tags anywhere. The price in C is that when the proof is broken by a cast or a memory error, nothing at run time notices: the bytes are simply used as whatever type the code assumes.

CHECKLIST

- ☐ Can you write a type checker for expressions, functions and calls as one recursive function?
- ☐ Can you read and write a typing rule with premises and a conclusion?

- ☐ Can you explain why every static checker must reject some correct programs?
- ☐ Can you say what soundness means and name an unsound feature in a language you use?
- ☐ Can you distinguish nominal from structural typing, and implicit conversion from memory safety?

GO DEEPER

BOOK **Types and Programming Languages, chapters 1, 8 and 9** Benjamin C. Pierce

ESSAY **Type Systems** Luca Cardelli

PEP **PEP 483: The Theory of Type Hints** van Rossum & Levkivskyi

Type inference

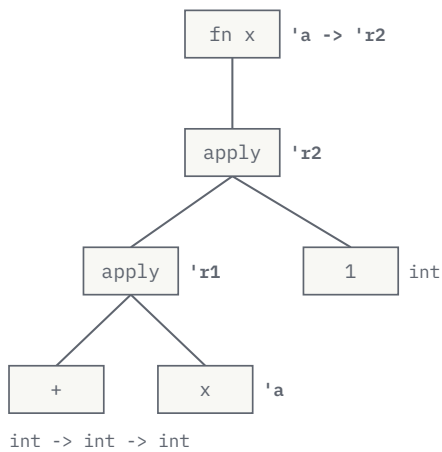
3 exercises

*Chapter 6's checker needed every parameter's type written down. ML, Haskell and OCaml need none: given `fn f (fn g (fn x (f (g x))))`, they work out that its type is `('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`, the most general type it can have. The algorithm, due to Hindley and Milner, treats each unknown type as a variable, collects equations between types from the shape of the program, and solves them by **unification**. This chapter builds it in about a hundred lines, and explains where its limits are and why most mainstream languages infer types only locally.*

Unknowns and equations

Inference begins by giving every type it doesn't know a fresh **type variable**: `'a`, `'b` and so on. Each node of the tree then imposes an equation. If `x` has type `'a` and the program applies `+`, whose type is `int -> int -> int`, to `x`, then `'a` must equal `int`. If `f` is applied to an argument of type `'b` and the result has type `'c`, then `f`'s type must equal `'b -> 'c`. An `if` says the condition's type equals `bool` and the two branches' types equal each other. A program is well typed exactly when all its equations can be satisfied at once.

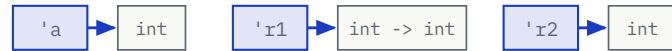
Unification solves an equation between two types by making them identical. If one side is an unbound variable, bind it to the other side. If both sides are constructors, such as two function types, they must be the same constructor, and their parts are unified pairwise: `'a -> 'r = int -> bool` means `'a = int` and `'r = bool`. Anything else, such as `int = bool`, is a type error. Algorithm W, as Milner called it, walks the tree once, unifying as it goes, so the solution builds up incrementally.



CONSTRAINTS, SOLVED IN ORDER

- 1 x gets a fresh variable 'a
- 2 (+ x): int -> (int -> int) = 'a -> 'r1
so 'a := int and 'r1 := int -> int
- 3 ((+ x) 1): 'r1 = int -> 'r2
so 'r2 := int
- 4 the whole function: 'a -> 'r2 = int -> int

THE SOLUTION IS POINTERS



unify sets a variable's ref; prune follows the chain

Inferring the type of `(fn x (+ x 1))`. Every unknown gets a variable, every application gives an equation, and unification solves them by pointing each variable at what it must be.

The implementation represents a binding as a pointer. Each type variable has a `ref`, initially empty. Binding a variable sets its `ref`, and `prune` follows a chain of bound variables to the type at the end. Nothing is ever substituted through the whole program. This is the same idea as the union-find structure for sets, and it makes unification nearly linear in practice. One check is essential: before binding `'a` to a type, the **occurs check** makes sure `'a` doesn't appear inside that type. Binding `'a` to `'a -> 'b` would describe an infinite type, a function taking itself as argument forever.

EXAMPLE

inference.py · Hindley-Milner inference with unification, the occurs check and let-polymorphism, on eight programs

```
import re

def read(src):
    tokens = re.findall(r"\\(|\\)|\\[\\^\\s()]+", src)
    def form():
        t = tokens.pop(0)
        if t != "(": return int(t) if t.isdigit() else t
        out = []
        while tokens[0] != ")": out.append(form())
        tokens.pop(0); return out
    return form()

# Types are type variables, which unification may later bind, or constructors such as int, ->,
# list.
class TVar:
    count = 0
    def __init__(self): self.ref = None; TVar.count += 1; self.id = TVar.count
class TCon:
    def __init__(self, name, *args): self.name, self.args = name, list(args)
def fn(a, b): return TCon("->", a, b)
INT, BOOL = TCon("int"), TCon("bool")

def prune(t):
    # follow a chain of bound variables to
    # its end
```

```

while isinstance(t, TVar) and t.ref is not None: t = t.ref
return t
def occurs(v, t):
    t = prune(t)
    return t is v or (isinstance(t, TCon) and any(occurs(v, a) for a in t.args))
def unify(a, b):
    a, b = prune(a), prune(b)
    if isinstance(a, TVar):
        if a is b: return
        if occurs(a, b): raise TypeError(f"infinite type: {show(a)} = {show(b)}")
        a.ref = b
    elif isinstance(b, TVar): unify(b, a)
    elif a.name != b.name or len(a.args) != len(b.args):
        raise TypeError(f"can't match {show(a)} with {show(b)}")
    else:
        for x, y in zip(a.args, b.args): unify(x, y)

def free(t):
    t = prune(t)
    return {t} if isinstance(t, TVar) else set().union(*(free(a) for a in t.args) if t.args
        else set())
def generalize(t, env):
    # quantify the variables the environment
    # doesn't pin
    pinned = set().union(*(free(ty) - set(vs) for vs, ty in env.values())) if env else set()
    return (list(free(t) - pinned), t)
def instantiate(scheme):
    vs, t = scheme
    fresh = {v: TVar() for v in vs}
    def copy(t):
        t = prune(t)
        if isinstance(t, TVar): return fresh.get(t, t)
        return TCon(t.name, *[copy(a) for a in t.args])
    return copy(t)

def infer(e, env):
    if isinstance(e, int): return INT
    if e in ("true", "false"): return BOOL
    if isinstance(e, str):
        if e not in env: raise TypeError(f"unknown variable {e}")
        return instantiate(env[e])
    head = e[0]
    if head == "fn":
        # (fn x body)
        v = TVar()
        return fn(v, infer(e[2], {**env, e[1]: ([], v)}))
    if head == "let":
        # (let (x value) body): x is generalised
        (x, value), body = e[1], e[2]
        return infer(body, {**env, x: generalize(infer(value, env), env)})
    if head == "letrec":
        # like let, but the value may refer to x
        (x, value), body = e[1], e[2]
        v = TVar(); unify(v, infer(value, {**env, x: ([], v)}))
        return infer(body, {**env, x: generalize(v, env)})
    if head == "if":
        unify(infer(e[1], env), BOOL)
        then = infer(e[2], env); unify(then, infer(e[3], env)); return then
    f = infer(head, env)
    # application, one argument at a time
    for arg in e[1:]:
        result = TVar(); unify(f, fn(infer(arg, env), result)); f = result
    return f

def show(t, names=None):
    names = {} if names is None else names
    t = prune(t)
    if isinstance(t, TVar): return names.setdefault(t, "" + "abcdefgh"[len(names)])
    if t.name == "->":
        # arrows group to the right: a -> b -> c
        arg, a = prune(t.args[0]), show(t.args[0], names)
        if isinstance(arg, TCon) and arg.name in ("->", "*"): a = f"({a})"

```

```

    return f"{a} -> {show(t.args[1], names)}"
    if t.name == "*": return f"{show(t.args[0], names)} * {show(t.args[1], names)}"
    return t.name if not t.args else f"{t.name} {show(t.args[0], names)}"

def scheme(build):
    # a built-in with fresh, fully general variables
    a, b = TVar(), TVar()
    t = build(a, b)
    return (list(free(t)), t)
BUILTINS = {
    "+": scheme(lambda a, b: fn(INT, fn(INT, INT))), "<": scheme(lambda a, b: fn(INT, fn(INT,
        BOOL))),
    "pair": scheme(lambda a, b: fn(a, fn(b, TCon("*", a, b)))), "nil": scheme(lambda a, b:
        TCon("list", a)),
    "cons": scheme(lambda a, b: fn(a, fn(TCon("list", a), TCon("list", a)))),
    "head": scheme(lambda a, b: fn(TCon("list", a), a)),
    "tail": scheme(lambda a, b: fn(TCon("list", a), TCon("list", a))),
    "null": scheme(lambda a, b: fn(TCon("list", a), BOOL)),
}
programs = [
    ("identity", "(fn x x)"),
    ("add one", "(fn x (+ x 1))"),
    ("compose", "(fn f (fn g (fn x (f (g x))))"),
    ("let-polymorphism", "(let (id (fn x x)) (pair (id 1) (id true)))"),
    ("lambda-bound id", "(fn id (pair (id 1) (id true)))"),
    ("map", "(letrec (map (fn f (fn xs (if (null xs) nil (cons (f (head xs)) (map f (tail
        xs)))))) map)"),
    ("self-application", "(fn x (x x))"),
    ("mismatched if", "(fn n (if (< n 0) 1 false))"),
]
for name, src in programs:
    try: print(f"{name:17} {show(infer(read(src), BUILTINS))}")
    except TypeError as e: print(f"{name:17} error: {e}")

```

OUTPUT

identity	'a -> 'a
add one	int -> int
compose	('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
let-polymorphism	int * bool
lambda-bound id	error: can't match int with bool
map	('a -> 'b) -> list 'a -> list 'b
self-application	error: infinite type: 'a = 'a -> 'b
mismatched if	error: can't match int with bool

No program has a single annotation. The inferred types are the most general possible: `identity` works for any type, and `compose`'s type says exactly how its three arguments must fit together and nothing more. That property, called **principal types**, is the remarkable thing about Hindley–Milner: every well-typed program has one most general type, of which all its other valid types are instances, and the algorithm always finds it. `map` was inferred from its recursive definition alone, including that it takes a function and a list and returns a list of the function's results.

Let-polymorphism

Two of the programs look nearly identical. In `(let (id (fn x x)) (pair (id 1) (id true)))` the identity function is used on an integer and a boolean, and the program is accepted, with type `int * bool`. In `(fn id (pair (id 1) (id true)))` the same uses are

rejected. The difference is **generalisation**. When a `let` binds a value, its inferred type's free variables, those not pinned by the surrounding environment, are quantified: `id` gets the **type scheme** "for all `'a` , `'a -> 'a` ", and each use **instantiates** the scheme with fresh variables, so the two uses don't constrain each other. A function's parameter can't be generalised, because its type is chosen by whoever calls the function. Inside the body, `id` has one unknown type, and using it on both `1` and `true` requires that type to be `int -> ...` and `bool -> ...` at once.

Accepting the second program would need a parameter whose type is itself polymorphic, a **rank-2** type. Type systems with such types exist, notably System F, but Wells proved in 1994 that inference for System F is undecidable. Hindley–Milner sits at a sweet spot: polymorphism only at `let` , in exchange for complete inference. Haskell and OCaml allow higher-rank types when the programmer annotates them, which is the common way to go beyond the sweet spot: infer what can be inferred, and ask for annotations at the places where inference would be undecidable.

Generalisation has one more subtlety when a language has mutation. If `let r = ref []` gave `r` the scheme "for all `'a` , a reference to a list of `'a` ", the program could store a list of integers through one use and read a list of strings through another. ML's fix is the **value restriction**: only generalise a `let` whose right side is syntactically a value, such as a function literal, and not one that performs a computation such as allocating a reference.

Inference in mainstream languages

Most languages that call themselves typed with inference do much less than this. `auto` in C++, `var` in Java and C#, `:=` in Go and `let` in Swift and Kotlin infer a variable's type from its initialiser, which is checking in one direction rather than solving equations. TypeScript and Scala use **bidirectional** type checking, which pushes expected types down into expressions and synthesises types up from them, and handles subtyping and overloading, which Hindley–Milner can't. Rust runs Hindley–Milner-style unification within each function body, so `let mut v = Vec::new(); v.push(1)` infers `Vec<i32>` from the later use, but requires every function signature to be written out.

Requiring signatures is a deliberate choice, and the reason is error messages. When unification fails, it reports the place where two types finally clashed, which can be far from the mistake that caused the clash. In a large program with no annotations, a wrong argument in one function can produce an error in another. Signatures at function boundaries stop errors spreading across them, document the interface, and keep inference fast. Haskell programmers, who don't have to write top-level signatures, conventionally write them anyway.

One theoretical caveat: Hindley–Milner inference can take exponential time, because nested `let` s can produce types whose size doubles at each level. Mairson showed in 1990 that the problem is complete for deterministic exponential time. Real programs never do this, and in practice the algorithm runs in close to linear time.

COMMON TRAPS

Omitting the occurs check, so a self-application loops forever or is wrongly accepted.

Generalising variables still pinned by the environment, which lets a type change between uses.

Substituting through the whole program at every binding, instead of binding variables by reference.

Trusting the location of an inference error. Look for the earlier expression that fixed the type the wrong way.

Generalising mutable references, which needs the value restriction or something like it.

EXERCISES

EXERCISE 7.1 Inference by hand

Infer the type of `(fn f (fn x (f (f x))))`, applying a function twice, by giving variables to `f` and `x` and solving the equations. Then check your answer by adding it to the example's programs.

SOLUTION

Give `f` the variable `'f` and `x` the variable `'x`. The inner application `(f x)` needs `'f = 'x -> 'r1`. The outer application `(f (f x))` applies `f` to something of type `'r1`, so `'f = 'r1 -> 'r2`. Unifying the two equations for `'f` gives `'x = 'r1` and `'r1 = 'r2`, so all three are one variable, say `'a`, and `'f = 'a -> 'a`. The whole expression is `'f -> 'x -> 'r2`, which is `('a -> 'a) -> 'a -> 'a`. That makes sense: a function can only be applied to its own result if its result type equals its argument type. Adding `("twice", "(fn f (fn x (f (f x))))")` to the list prints exactly that type.

EXERCISE 7.2 The occurs check

`(fn x (x x))` was rejected with "infinite type". Trace the unification that fails. What would happen if the occurs check were removed from `unify`?

SOLUTION

`x` gets `'a`. Applying `x` to `x` requires the function's type to equal argument-to-result: `'a = 'a -> 'b`. Unify sees an unbound variable on the left and is about to bind `'a` to `'a -> 'b`, but `'a` occurs in that type, so it reports the infinite type instead. Without the check, `'a`'s `ref` would point at a function type containing `'a` itself, a cycle. Any later operation that walks the type, such as `show`, `free` or another `unify`, would follow the cycle forever, or until Python's recursion limit stopped it. Some Prolog systems omit the occurs check for speed, and the logic program in chapter 19 has the same issue: unification without it can produce cyclic terms and wrong answers. A few type systems do allow such recursive types on purpose, but they then need special algorithms for comparing them.

EXERCISE 7.3 Which variables to generalise

In `(fn y (let (f (fn x y)) (pair (f 1) (f true))))`, the inner function `f` ignores its argument and returns `y`. What is the type of the whole expression, and why must `generalize` leave `y`'s variable unquantified when it generalises `f`?

SOLUTION

`y` gets `'y` and `x` gets `'x`, so `f`'s type is `'x -> 'y`. When the `let` generalises it, `'x` is free only in `f`'s type, so it's quantified. `'y` also appears in the environment, as the type of the parameter `y`, so it must not be. Each use of `f` then gets a fresh copy of `'x` but shares `'y`: `(f 1)` and `(f true)` both type-check, and both return `'y`. The whole expression has type `'a -> 'a * 'a`. If `'y` were wrongly quantified, each use of `f` would get its own fresh result type, unconnected to `y`, and the checker would accept, for example, `(+ (f 1) 1)` while `y` is later passed a boolean, which is exactly the unsoundness that the "pinned" set in `generalize` prevents.

CHECKLIST

- ☐ Can you infer the type of a small function by hand, writing down its equations?
- ☐ Can you implement unification with an occurs check, binding variables by reference?
- ☐ Can you explain let-polymorphism, generalisation and instantiation?
- ☐ Can you explain why a lambda-bound variable can't be polymorphic under Hindley–Milner?

- ☐ Can you say how inference in C++, Java, Rust and TypeScript differs from Hindley–Milner?

GO DEEPER

BOOK Types and Programming Languages, chapter 22: Type Reconstruction
Benjamin C. Pierce

PAPER A Theory of Type Polymorphism in Programming Robin Milner, 1978

BOOK OCaml Programming: Type Inference Cornell CS 3110

Generics and variance

3 exercises

*A generic function or type is written once with a type parameter and used at many types: one `List<T>` for lists of strings, of numbers and of lists. Languages implement generics in strikingly different ways, and the choice shows up in compile times, binary sizes and what a program can ask about its types at run time. Generics also raise a question that subtyping alone doesn't: if a `Cat` is an `Animal`, is a list of cats a list of animals? The answer depends on whether the list can be written to, and this chapter works out the rules, called **variance**, and shows what happens in a language that gets them wrong.*

Parametric polymorphism

Chapter 7's `map` had the type `('a -> 'b) -> list 'a -> list 'b`. Its type variables are **parameters**: the function behaves the same whatever types are substituted for them, because it never looks inside the values. That's **parametric polymorphism**, and generics in Java, C#, Rust, Swift and, since 1.18, Go provide it with explicit type parameters: `<T> List<T> reverse(List<T> xs)`. Parametricity has a striking consequence, which Wadler called "theorems for free": a function of type `'a -> 'a` that never fails or loops can only be the identity, because it has no way to make a value of an unknown type other than returning the one it was given.

Most generic code needs a little more than "any type". Sorting needs elements that can be compared, a hash map needs keys that can be hashed. **Bounded** type parameters say so: `<T extends Comparable<T>>` in Java, `T: Ord` in Rust, a constraint interface in Go, a type class such as `Ord a` in Haskell. The bound tells the checker which operations are allowed on `T` inside the function, and tells callers which types they may supply.

Three ways to implement generics

The type checker treats generics the same way everywhere. The back end does not, and there are three main strategies.

- **Erasure**, used by Java. The compiler checks the types and then removes them: `List<String>` and `List<Integer>` both compile to one class, `List`, holding `Object` references, with casts inserted where values come out. There is one copy of the code, and old code that predates generics still works. The cost is that the types are gone at run time: `new T[10]` and `x instanceof List<String>` are compile-time errors, and a list of integers must hold boxed `Integer` objects, not raw `int`s.

- **Monomorphisation**, used by C++ templates and by Rust. The compiler generates a separate copy of the generic code for every set of type arguments actually used, so `Vec<u8>` and `Vec<String>` become two unrelated types with code specialised for each. That's as fast as hand-written code, since each copy can be optimised for its types, but it costs compile time and binary size. A generic function used with ten types is compiled ten times.
- **Dictionary passing**, used by Haskell for type classes and, in a mixed form, by Go. One copy of the code is compiled, and it receives a hidden extra argument, a table of the operations the bound promises, such as the comparison function for `T`. Go's implementation generates one copy per "shape" of type, grouping all pointer types together, and passes a dictionary for the rest.

C# and the .NET runtime combine strategies: generic types are **reified**, keeping their type arguments at run time, so `typeof(T)` works. The runtime generates specialised code for each value type, such as `List<int>` without boxing, and shares one copy among all reference types. Rust offers the dictionary approach as an option too: a `&dyn Trait` is a pointer paired with a table of methods, one copy of the code, chosen when binary size matters more than speed.

Subtyping

A type `S` is a **subtype** of `T`, written `S <: T`, if a value of type `S` can be used anywhere a `T` is expected. For classes in nominal languages the relation is declared: `class Cat extends Animal`. The interesting cases are types built from other types. When is a list of `S` a subtype of a list of `T`, or a function of one type a subtype of a function of another? The example implements the rules for three type constructors, read-only lists, mutable arrays and functions, and then shows what Java's choice for arrays costs.

EXAMPLE

variance.py · subtyping rules for lists, arrays and functions, and Java's covariant arrays with and without the store check

```
# Subtyping: S <: T means a value of type S can be used wherever a T is expected.
PARENT = {"Cat": "Animal", "Dog": "Animal", "Animal": "Object"} # nominal: declared by the
                        programmer

def subtype(s, t):
    """Types: a class name, ('list', T) read-only, ('array', T) mutable, ('fn', (params...),
        result)."""
    if s == t: return True, "the same type"
    if isinstance(s, str) and isinstance(t, str):
        while s in PARENT:
            s = PARENT[s]
            if s == t: return True, "declared subclass"
        return False, "not a subclass"
    if isinstance(s, str) or isinstance(t, str) or s[0] != t[0]:
        return False, "different kinds of type"
    if s[0] == "list": # covariant: lists are
        only read
        ok, _ = subtype(s[1], t[1]); return ok, "lists are covariant"
    if s[0] == "array": # invariant: read and
        written
```

```

        return subtype(s[1], t[1])[0] and subtype(t[1], s[1])[0], "arrays are invariant"
    if s[0] == "fn":
        parameters,
        params_ok = len(s[1]) == len(t[1]) and all(subtype(tp, sp)[0] for sp, tp in zip(s[1],
        t[1]))
        return params_ok and subtype(s[2], t[2])[0], "parameters contravariant, result
        covariant"

def show(t):
    if isinstance(t, str): return t
    if t[0] == "fn": return f"({', '.join(map(show, t[1]))}) -> {show(t[2])}"
    return f"{t[0]}[{show(t[1])}]"

queries = [("Cat", "Animal"), ("Animal", "Cat"), ("Dog", "Object"),
            (("list", "Cat"), ("list", "Animal")), (("array", "Cat"), ("array", "Animal")),
            (("fn", ("Animal",), "Cat"), ("fn", ("Cat",), "Animal")),
            (("fn", ("Cat",), "Cat"), ("fn", ("Animal",), "Cat")),
            (("list", ("fn", ("Animal",), "Cat")), ("list", ("fn", ("Cat",), "Animal")))]
for s, t in queries:
    ok, why = subtype(s, t)
    print(f"{show(s):>21} <: {show(t):<21} {'yes' if ok else 'no ':3} {why}")

# Why mutable arrays can't be covariant: Java allows it, and pays with a check on every store
class Cat:
    def meow(self): return "meow"
class Dog:
    def bark(self): return "woof"
class JavaArray:
    def __init__(self, element_type, items, check=True):
        self.element_type, self.items, self.check = element_type, list(items), check
    def store(self, i, value):
        if self.check and not isinstance(value, self.element_type):
            name = self.element_type.__name__
            raise TypeError(f"ArrayStoreException: {type(value).__name__} into {name}[]")
        self.items[i] = value

print("\nJava lets a Cat[] be used as an Animal[], so this compiles:")
for check in (True, False):
    cats = JavaArray(Cat, [Cat(), Cat()], check=check)
    animals = cats # Animal[] animals = cats; (covariance)
    try:
        animals.store(0, Dog()) # animals[0] = new Dog(); fine for an
        Animal[]
        print(f"    without the run-time check: the store succeeds, then cats[0].meow() fails:
        ", end="")
        cats.items[0].meow()
    except (TypeError, AttributeError) as e:
        print(f"    with the run-time check: {e} " if check else f"{type(e).__name__}")

```

OUTPUT

```

        Cat <: Animal          yes  declared subclass
        Animal <: Cat          no   not a subclass
        Dog <: Object          yes  declared subclass
        list[Cat] <: list[Animal] yes  lists are covariant
        array[Cat] <: array[Animal] no   arrays are invariant
        (Animal) -> Cat <: (Cat) -> Animal yes  parameters contravariant, result covariant
        (Cat) -> Cat <: (Animal) -> Cat    no   parameters contravariant, result covariant
        list[(Animal) -> Cat] <: list[(Cat) -> Animal] yes  lists are covariant

```

Java lets a Cat[] be used as an Animal[], so this compiles:

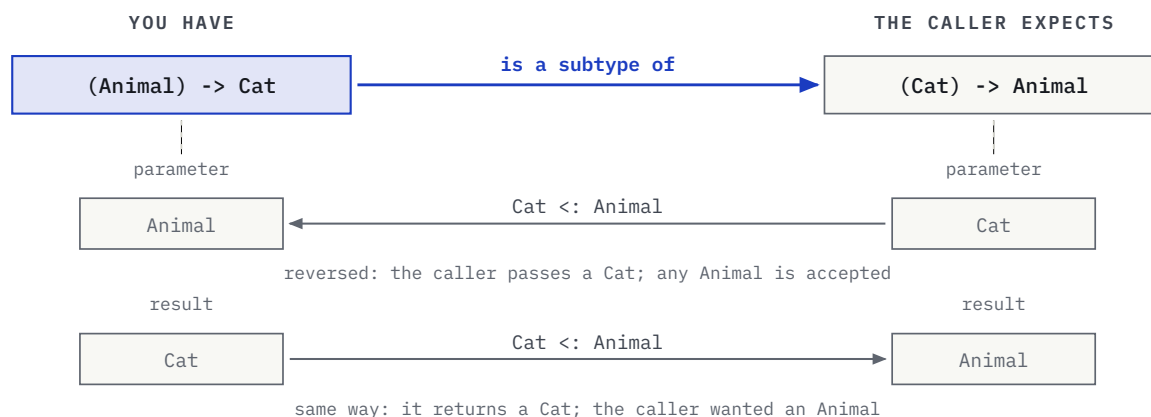
with the run-time check: ArrayStoreException: Dog into Cat[]

without the run-time check: the store succeeds, then cats[0].meow() fails: AttributeError

The variance rules

A type constructor is **covariant** in a parameter if subtyping goes the same way: `Cat <: Animal` implies `list[Cat] <: list[Animal]`. That's safe for a list that is only read. Anyone reading an animal from it gets a cat, which is an animal. It's **contravariant** if subtyping goes the opposite way, which is the rule for function parameters. And it's **invariant** if neither direction is allowed, which is the rule for anything that is both read and written.

Functions show why the reversal happens. Code expecting a function of type `(Cat) -> Animal` will call it with cats and use its results as animals. A function of type `(Animal) -> Cat` is fine there: it accepts any animal, so certainly the cats it's given, and it returns cats, which are animals. The function with the more general parameter and the more specific result is the subtype. The opposite pair, `(Cat) -> Cat` where `(Animal) -> Cat` is expected, fails, because a caller may pass a dog.



Why function parameters are contravariant. For one function type to stand in for another, its result must be a subtype, the same direction as the whole, and its parameter a supertype, the opposite direction.

The general rule: a type parameter that appears only in outputs can be covariant, one that appears only in inputs can be contravariant, and one that appears in both must be invariant. A mutable array has both: reading an element is an output and storing one is an input. So `array[Cat]` can't safely be an `array[Animal]`. If it were, code holding it as an array of animals could store a dog in it, and code holding it as an array of cats would later read that dog and call `meow`.

Java made its arrays covariant anyway, in its first version, years before it had generics, so that one sorting method could accept any array of objects. To keep the type system from being broken, every store into an array checks at run time that the value fits the array's real element type, and throws `ArrayStoreException` if not. The example shows both outcomes: with the check, the bad store fails at once; without it, the store succeeds and the program fails later, at the `meow`, far from the mistake. That check runs on every array store in every Java program. C# copied the design, and TypeScript has covariant arrays with no check at all, one of its deliberate unsound spots.

Variance in real languages

Languages let programmers choose variance in one of two places. With **declaration-site variance**, the author of a generic type marks each parameter. C# writes `IEnumerable<out T>` for a covariant parameter and `Action<in T>` for a contravariant one, Kotlin uses the same keywords, and Scala writes `+T` and `-T`. The checker then verifies that an `out` parameter really only appears in outputs. With **use-site variance**, the user of a type says what they need. Java's generics are invariant, and wildcards relax them where they're used: `List<? extends Animal>` accepts a list of cats but allows only reading, and `List<? super Cat>` accepts a list of animals but allows only adding cats. The Java mnemonic is "producer extends, consumer super".

Python's type checkers follow the same rules. `list` is invariant, since lists are mutable, so mypy rejects passing a `list[Cat]` where a `list[Animal]` is expected. `Sequence`, which has no methods for changing it, is covariant, and so are tuples. `Callable` is contravariant in its parameter types. Since Python 3.12, a class declared with the new type parameter syntax (PEP 695) has the variance of each parameter inferred from how the class uses it.

COMMON TRAPS

Assuming a list of a subtype is a list of the supertype. It's true only if the list can't be written to.

Getting function parameters the wrong way round, and accepting a function that handles fewer inputs than callers will pass.

Expecting generic type information at run time in a language that erases it.

Monomorphising without limit, so compile times and binaries grow with every new type argument.

Annotating a Python parameter as `list` when the function only reads it, which rejects callers needlessly. Use `Sequence`.

EXERCISES

EXERCISE 8.1 Read-only lists in Python

A function `def total_weight(animals: list[Animal]) -> float` only reads its argument. Calling it with a `list[Cat]` is rejected by mypy. Why is mypy right to reject it as written, and what one change to the signature fixes it?

SOLUTION

mypy checks the signature, not the body. A parameter of type `list[Animal]` promises the function may do anything a list allows, including `animals.append(Dog())`. If a `list[Cat]` were accepted, that `append` would put a dog into the caller's list of cats, exactly the array problem from the example. Because `list` is mutable, it's invariant, and the rejection is correct. The fix is to ask for what the function needs: `animals: Sequence[Animal]`, from `collections.abc` or `typing`. `Sequence` has no mutating methods, so it's covariant, and a `list[Cat]` is a `Sequence[Animal]`. The same reasoning says to accept `Iterable` if the function only loops over its argument, and `Mapping` rather than `dict` for a dictionary it only reads. The narrowest interface in parameters makes a function accept the most callers.

EXERCISE 8.2 The signature of sort

Java declares `static <T> void sort(List<T> list, Comparator<? super T> c)`. Why `? super T` and not plain `Comparator<T>`? Give a call that the wildcard allows.

SOLUTION

A comparator only consumes values of its type parameter: it takes two and returns an `int`. So it's naturally contravariant. A comparator that can compare any two animals, say by weight, can certainly compare two cats. With plain `Comparator<T>`, which Java treats as invariant, sorting a `List<Cat>` would demand exactly a `Comparator<Cat>`, and the existing `Comparator<Animal> byWeight` would be rejected. With `? super T`, `sort(cats, byWeight)` compiles, because `Animal` is a supertype of `Cat`. That's the "consumer super" half of the mnemonic. The list itself is `List<T>` without a wildcard because `sort` both reads and writes it, which requires invariance. In C# the same thing is declared once on the interface, `IComparer<in T>`, and every use gets it automatically.

EXERCISE 8.3 Erased or copied

A generic function `largest<T: Ord>(xs: &[T]) -> &T` is called with 12 different element types in a Rust program. How many copies of its machine code are there? How many would the equivalent Java method have, and what can each do that the other can't?

SOLUTION

Rust monomorphises, so there are 12 copies, one per element type, each with the comparison inlined and specialised: comparing `u8 s` becomes a single byte comparison in a tight loop. Identical copies may be merged by the linker, but in general the code grows with the number of types used, and so does compile time. Java erases, so there's one copy that works on `Comparable` references and calls `compareTo` through a virtual call, with primitives boxed. The Rust version is faster and works on unboxed values. The Java version is smaller, compiles once, and can be called from code compiled later with types it has never seen, such as classes loaded at run time. Rust can get Java's trade-off where a trait allows it, by taking trait objects such as `&dyn Display`: one copy of the code, with calls through a table. C++ programmers make the same choice between templates and virtual functions.

CHECKLIST

- ☐ Can you explain parametric polymorphism and what a type bound adds?
- ☐ Can you compare erasure, monomorphisation and dictionary passing, with a language for each?
- ☐ Can you state when a type constructor can be covariant, contravariant or invariant?
- ☐ Can you explain why function parameters are contravariant, with an example?
- ☐ Can you explain why Java's covariant arrays need a run-time check, and how Java's generics avoid it?

GO DEEPER

[DOCS](#) Python typing: Generics, including variance [Python typing documentation](#)

[DOCS](#) Covariance and contravariance in generics [Microsoft .NET documentation](#)

[PAPER](#) Theorems for free! Philip Wadler, 1989

PART III

Running programs

Six chapters on the back end that most dynamic languages use: an interpreter. It starts with the simplest kind, which walks the syntax tree, and adds closures, the feature that forces an interpreter to think about where variables live. Then it compiles the tree to bytecode and builds a virtual machine to run it, with call frames on a stack of its own. Two chapters cover what the machine needs beyond that: a garbage collector to reclaim memory, and proper handling of calls in tail position, with continuations, the idea underneath generators and async functions. From here on, the example programs are written as S-expressions, so the examples can spend their lines on running code rather than parsing it.

Tree-walking interpreters

3 exercises

The quickest way to run a program is to walk its syntax tree and do what each node says: evaluate the children of an addition and add them, evaluate an `if`'s condition and then one branch. An interpreter built that way is one recursive function, easy to write, easy to extend and easy to debug, and it's how most languages start. This chapter builds one with functions and recursion, counts exactly what it does while computing `(fib 20)`, and finds the two costs that later chapters remove: repeated work on the tree, and recursion that borrows the host language's stack.

S-expressions

The examples from here on write programs in the notation of Lisp: `(+ 1 2)` instead of `1 + 2`, `(if c a b)` instead of `if (c) a else b`. Every compound expression is a list whose first element says what it is. The syntax is the tree, written out with parentheses, so a reader for it takes a dozen lines, the same function at the top of every example, and each node's kind is its first element. Nothing about interpreting depends on this choice. A parser from Part I would produce the same trees from conventional syntax.

Evaluating a tree

An interpreter's core is a function `evaluate(node, env)` that returns the node's value in an **environment**, the set of variables visible at that point. Numbers evaluate to themselves. A name is looked up in the environment. An `if` evaluates its condition and then exactly one branch, which is why `if` can't be an ordinary function: a function would evaluate both. A `lambda` creates a **closure**, the function's parameters and body together with the environment where the `lambda` was evaluated. Calling a closure evaluates the arguments, creates a new environment that binds the parameters to them, whose parent is the closure's environment, and evaluates the body in it.

That last rule is lexical scope, implemented. The new environment's parent is the environment the function was **defined** in, not the one it was **called** from, so a function sees the variables around its definition, as chapter 5 described. Each environment here is a dictionary with a link to its parent, and looking up a name searches the chain from the innermost scope outward. The example counts each step: every node evaluated, every call, every lookup, and every dictionary probed during the searches.

EXAMPLE

`treewalk.py` · a tree-walking interpreter with closures and lexical scope, instrumented while it computes `fib 20`

```

import re, sys

def read_all(src):
    """Read S-expressions: (f a (g b)) becomes ['f', 'a', ['g', 'b']]; numbers become ints."""
    tokens = re.findall(r"\(|\)|[^\s()]+", src)
    def form():
        t = tokens.pop(0)
        if t != "(": return int(t) if t.lstrip("-").isdigit() else t
        out = []
        while tokens[0] != ")": out.append(form())
        tokens.pop(0); return out
    forms = []
    while tokens: forms.append(form())
    return forms

class Env:
    """One scope: a dict of variables and a link to the enclosing scope."""
    def __init__(self, names=(), values=(), parent=None):
        self.vars, self.parent = dict(zip(names, values)), parent
    def find(self, name):
        env, hops = self, 0
        while name not in env.vars:
            env, hops = env.parent, hops + 1
            if env is None: raise NameError(f"undefined variable '{name}'")
        stats["lookups"] += 1; stats["probes"] += hops + 1
        return env

class Closure:
    def __init__(self, params, body, env): self.params, self.body, self.env = params, body, env

stats = {"nodes": 0, "lookups": 0, "probes": 0, "calls": 0, "depth": 0, "max_depth": 0}
PRIMS = {"+": lambda a, b: a + b, "-": lambda a, b: a - b, "*": lambda a, b: a * b,
        "<": lambda a, b: a < b, "=": lambda a, b: a == b}

def evaluate(x, env):
    stats["nodes"] += 1 # every node of the tree passes through here
    if isinstance(x, int): return x
    if isinstance(x, str): return env.find(x).vars[x]
    op = x[0]
    if op == "define": env.vars[x[1]] = evaluate(x[2], env); return None
    if op == "lambda": return Closure(x[1], x[2], env) # capture the scope it was created in
    if op == "if": return evaluate(x[2] if evaluate(x[1], env) else x[3], env)
    f = evaluate(op, env)
    args = [evaluate(a, env) for a in x[1:]]
    if callable(f): return f(*args)
    stats["calls"] += 1; stats["depth"] += 1
    stats["max_depth"] = max(stats["max_depth"], stats["depth"])
    result = evaluate(f.body, Env(f.params, args, f.env)) # a new scope for each call
    stats["depth"] -= 1
    return result

program = read_all("""
(define fib (lambda (n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
(fib 20)
""")
glob = Env(PRIMS.keys(), PRIMs.values())
for form in program: result = evaluate(form, glob)
print(f"(fib 20) = {result}")
print(f"tree nodes evaluated: {stats['nodes']:,}")
print(f"function calls: {stats['calls']:,}, nested at most {stats['max_depth']} deep")
print(f"variable lookups: {stats['lookups']:,}")
print(f"dictionaries probed: {stats['probes']:,} searching scope by scope")
print(f"with a resolver: {stats['lookups']:,}, one per lookup")
per_call = stats["nodes"] / stats["calls"]
stats["nodes"] = stats["nodes"] - stats["calls"] * per_call

```

```

per_call = stats[ nodes ] / stats[ calls ], stats[ probes ] / stats[ calls ]

print(f"    per call: {per_call[0]:.1f} nodes and {per_call[1]:.1f} probes")

# Deep recursion runs out of the host language's stack: the interpreter's recursion is Python's
# recursion
sys.setrecursionlimit(1000)
count = read_all("(define down (lambda (n) (if (= n 0) 0 (down (- n 1)))))")[0]
evaluate(count, glob)
for n in (100, 1000):
    try: evaluate(["down", n], glob); print(f"(down {n}) finished")
    except RecursionError: print(f"(down {n}) overflowed: each call it makes costs several
        Python frames")

```

OUTPUT

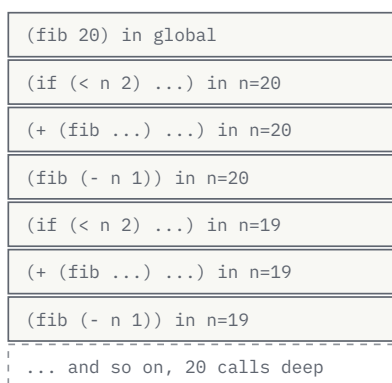
```

(fib 20) = 6765
tree nodes evaluated: 273,636
function calls:      21,891, nested at most 20 deep
variable lookups:    131,344
dictionaries probed: 207,960 searching scope by scope
with a resolver:     131,344, one per lookup
per call: 12.5 nodes and 9.5 probes
(down 100) finished
(down 1000) overflowed: each call it makes costs several Python frames

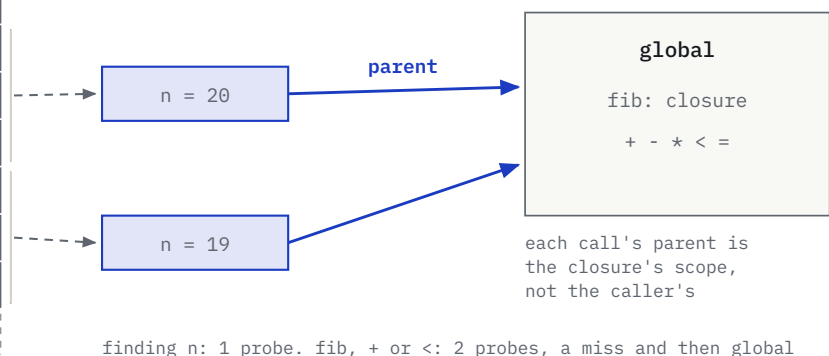
```

Computing the 20th Fibonacci number took 21,891 calls, as it must with this doubly recursive definition, and evaluated 273,636 tree nodes, 12.5 per call. That's the first cost of walking a tree: the interpreter rediscovers the same structure every time. Each of the 21,891 calls asks of the body "is this an `if`?" and of its children "is this a name, a number or a call?", although the answers never change. A compiler asks once.

PYTHON STACK



SCOPES ON THE HEAP



What the interpreter's state looks like during `(fib 20)`. Python's own stack holds the chain of `evaluate` calls. Each call of `fib` gets a scope on the heap whose parent is where `fib` was defined, so every name except `n` costs a failed probe before it's found.

The second cost is in the lookups. There were 131,344 of them, but 207,960 dictionary probes, because every name other than `n` is first looked for in the call's own scope, missed, and then found in the global one. A resolver, chapter 5, would have worked out in advance

that `n` is in the innermost scope and `fib`, `+` and `-` are global, so each lookup could go straight to the right place: 131,344 probes, one per lookup. Real interpreters go further and replace each local variable's name with a slot number in an array, so a lookup is an index operation with no hashing at all.

The host's stack is the interpreter's stack

The last lines of the example show the third cost. `(down 1000)` is a function that calls itself a thousand times, which any language should handle. It overflowed. The interpreter implements a call in the interpreted language with calls in Python: `evaluate` of the call node calls `evaluate` of the body, which calls `evaluate` of the `if`'s branch, which is the next call. So each level of the program's recursion costs two Python frames, and Python's default limit of 1,000 frames is reached at about 500 levels. The interpreted language inherits not just the host's stack, but its limits and its cost per frame.

There are three ways out, and later chapters take each of them. Chapter 12's virtual machine keeps the program's calls on a stack it manages itself, so the depth of the program's recursion and the depth of the interpreter's recursion become unrelated. Chapter 14 shows that a call in tail position, like the one in `down`, needs no new frame at all. And an interpreter can be written so that it never recurses, keeping an explicit stack of what remains to be done; that's the CEK machine of programming language theory, and it turns out to be the same thing as the continuations of chapter 14.

Tree-walkers in practice

For many languages, a tree-walker is all they ever need. Bash interprets its syntax tree, as do many configuration and template languages, and Ruby's original interpreter walked the tree until version 1.9 introduced the YARV virtual machine in 2007. Crafting Interpreters builds its first interpreter, `jlox`, this way in Java, with one `visit` method per node class. When a tree-walker is too slow, the usual next step is the bytecode of chapter 11.

There is a remarkable exception. The Truffle framework, part of GraalVM, makes tree-walking interpreters fast by letting nodes **rewrite themselves** as the program runs. An addition node that has only ever seen integers replaces itself with an integer-only addition node; a variable lookup replaces itself with a direct slot access. A JIT compiler then compiles the specialised tree into machine code. Implementations of JavaScript, Python, Ruby and R built on Truffle reach speeds comparable with hand-built virtual machines, from interpreters whose structure is as simple as this chapter's.

COMMON TRAPS

Giving a call's new scope the caller's environment as parent, which is dynamic scope, not lexical.

Implementing `if`, `and` or `or` as ordinary functions, so both branches or both operands are always evaluated

Looking names up by string at every step, when a resolver could have computed where each one lives.

Forgetting that the host's recursion limit is now the language's, and testing only shallow programs.

Mixing evaluation with printing or parsing, which makes the interpreter impossible to reuse in a REPL or test.

EXERCISES

EXERCISE 9.1 Where the probes come from

Account for the 131,344 lookups and 207,960 probes exactly. Count the lookups and probes in one call of `fib` with `n ≥ 2` and in one with `n < 2`, given that `(fib 20)` makes 10,945 calls of the first kind and 10,946 of the second.

SOLUTION

A call with `n < 2` evaluates `(< n 2)` and then `n`: three lookups, `<`, `n` and `n`. `n` is found in the call's own scope at the first probe, and `<` needs two, a miss and then the global scope, so that's 4 probes. A call with `n ≥ 2` evaluates the condition and then `(+ (fib (- n 1)) (fib (- n 2)))`: lookups of `<`, `n`, `+`, `fib`, `-`, `n`, `fib`, `-`, `n`, which is 9, costing 2 probes for each of the six globals and 1 for each of the three `n`s, 15 in all. So lookups are $10,945 \times 9 + 10,946 \times 3 = 131,343$, and probes are $10,945 \times 15 + 10,946 \times 4 = 207,959$. The last lookup and probe is the top-level `(fib 20)` itself, which finds `fib` in the global scope at once. Counting like this, rather than timing, tells you which change will help: here, resolving the six global names saves 76,616 probes.

EXERCISE 9.2 `set!` and `define`

Add `(set! name value)`, which changes an existing variable, alongside `define`, which creates one. Why must `set!` search the environment chain while `define` doesn't, and what should `set!` do with a name that isn't defined anywhere?

SOLUTION

`define` always creates a binding in the current scope, `env.vars[name] = value`, so it never searches. `set!` must change the binding the name already refers to, which may be in an enclosing scope, so it finds that scope with `env.find(name)` and assigns there:

```
env.find(x[1]).vars[x[1]] = evaluate(x[2], env)
```

If it assigned in the current scope instead, a function that updates a counter defined outside it would create a new local each call and the counter would never change. That's the bug chapter 10's counters depend on avoiding. A name defined nowhere should be an error, "undefined variable", rather than silently creating a global. Python makes the same distinction differently: plain assignment always means the current scope, and `nonlocal` or `global` are needed to assign further out, which is what exercise 5.1 was about.

EXERCISE 9.3 How deep can it go?

With Python's recursion limit at 1,000, about how deep can `down` recurse before the interpreter overflows? Why exactly that, and how would adding a `begin` form around the body change it?

SOLUTION

Each level of `down` costs two nested `evaluate` frames: one for the call node, which evaluates the body, and one for the `if`, which evaluates its chosen branch, the next call. The arguments and the condition are evaluated and return before the next level begins, so they don't add to the depth that persists. With about a thousand frames available, a few already in use by the program that started the interpreter, the limit is near 500 levels, which is why `(down 100)` finished and `(down 1000)` didn't. Wrapping the body in `(begin ...)` adds a third frame per level, for the `begin` node, and the limit drops to about 330. Every layer of syntax between a function's entry and its recursive call costs a frame of the host. This is a reason interpreters written in C crash with a segmentation fault on deep recursion, when the C stack runs out, unless they check the depth themselves, as CPython does.

CHECKLIST

- ☐ Can you write `evaluate` for numbers, names, `if`, `lambda` and calls?
- ☐ Can you explain why a call's new environment has the closure's environment as parent?
- ☐ Can you explain why `if` must be a special form and not a function?

- ☐ Can you count the work a tree-walker does and say which costs a resolver or compiler removes?
- ☐ Can you explain why a tree-walker's recursion depth is limited by its host language's stack?

GO DEEPER

BOOK Crafting Interpreters, chapter 7: Evaluating Expressions Robert Nystrom

BOOK Structure and Interpretation of Computer Programs, section 4.1: The Metacircular Evaluator Abelson & Sussman

ARTICLE (How to Write a (Lisp) Interpreter (in Python)) Peter Norvig

Closures

3 exercises

A closure is a function together with the variables it captured from where it was written. Closures are what make functions first-class values: they can be returned, stored and called later, long after the function that created them has returned, and still see the variables they were born with. That last part is the hard one for an implementation, because it means some local variables can't live on the stack. This chapter shows closures capturing state, the classic bug where they capture a loop variable, and how compilers decide which variables must move to the heap.

Functions that remember

Chapter 9's interpreter already had closures: a `lambda` evaluated to the pair of its code and the environment it was evaluated in, and calling it made a new scope whose parent was that environment. A function created inside another function therefore keeps a reference to the outer function's scope. When the outer function returns, its scope isn't discarded, because the closure still points at it. The scope lives on as long as the closure does, and the garbage collector reclaims both together.

That's enough to build objects with private state, as the first part of the example does.

`make_counter` creates a variable `n` and returns a function that increments and returns it. Each call of `make_counter` creates a new scope with its own `n`, so two counters count independently, and nothing outside can reach either `n` except through its counter. The observation that a closure with mutable captured variables is an object with private fields and one method, and that an object is a bundle of closures over shared state, is old enough to be a programming koan: closures are a poor man's objects, and objects are a poor man's closures.

EXAMPLE

closures.py · counters with private state, the loop-variable trap in Python and in our interpreter, and which variables a closure captures

```
import re

def read(src):
    tokens = re.findall(r"(\(|\)|[^\s()]+", src)
    def form():
        t = tokens.pop(0)
        if t != "(": return int(t) if t.lstrip("-").isdigit() else t
        out = []
        while tokens[0] != ")": out.append(form())
        tokens.pop(0); return out
    return form()

class Env:
```

```

def __init__(self, names, values, parent): self.vars, self.parent = dict(zip(names,
values)), parent
def find(self, name):
    env = self
    while name not in env.vars: env = env.parent
    return env

def evaluate(x, env):
    if isinstance(x, (int, tuple)) or callable(x): return x      # a number, or a value already
    if isinstance(x, str): return env.find(x).vars[x]
    op = x[0]
    if op == "lambda": return ("closure", x[1], x[2:], env)      # code plus the scope it was
        born in
    if op == "set!": env.find(x[1]).vars[x[1]] = evaluate(x[2], env); return None
    if op == "if": return evaluate(x[2] if evaluate(x[1], env) else x[3], env)
    f, args = evaluate(op, env), [evaluate(a, env) for a in x[1:]]
    if callable(f): return f(*args)
    _, params, body, defined_in = f
    scope = Env(params, args, defined_in)                          # parent: where it was
        defined,
    for form in body: result = evaluate(form, scope)                # not where it's called
    return result

top = Env(["+", "*"], [lambda a, b: a + b, lambda a, b: a * b], None)
make_counter = evaluate(read("(lambda () ((lambda (n) (lambda () (set! n (+ n 1)) n)) 0))"),
    top)
c1, c2 = evaluate([make_counter], top), evaluate([make_counter], top)
calls = [evaluate([c], top) for c in (c1, c1, c1, c2)]
print(f"two counters, each with its own captured n: c1 c1 c1 c2 -> {calls}")

# The loop-variable trap: closures capture variables, not values
fs = [lambda: i for i in range(3)]
print(f"\nPython, [lambda: i for i in range(3)], called:    {[f() for f in fs]} (one shared
    i)")
fs = [lambda i=i: i for i in range(3)]
print(f"with i=i, a parameter evaluated at creation time: {[f() for f in fs]}")
shared = Env(["i"], [0], top)
made_shared, made_fresh = [], []
for i in range(3):
    shared.vars["i"] = i                                          # like JavaScript's var: one
        binding
    made_shared.append(evaluate(read("(lambda () i)"), shared))
    fresh = Env(["i"], [i], top)                                  # like let: a new binding
        each time
    made_fresh.append(evaluate(read("(lambda () i)"), fresh))
print(f"our interpreter, one binding for the loop:    {[evaluate([f], top) for f in
    made_shared]}")
print(f"our interpreter, a fresh binding per iteration: {[evaluate([f], top) for f in
    made_fresh]}")

# Which variables must outlive their stack frame? The ones some inner function captures.
def free_vars(x, bound):
    if isinstance(x, int): return set()
    if isinstance(x, str): return {x} - bound
    if x[0] == "lambda":
        inner = bound | set(x[1])
        return set().union(*(free_vars(b, inner) for b in x[2:]))
    if x[0] == "set!": return ({x[1]} - bound) | free_vars(x[2], bound)
    return set().union(*(free_vars(e, bound) for e in x))
def captured(fn_src):
    fn = read(fn_src)
    locals_ = set(fn[1])
    inner = [b for b in fn[2:] if isinstance(b, list) and b[0] == "lambda"]
    caught = set().union(*(free_vars(i, set()) for i in inner)) & locals_
    return sorted(caught), sorted(locals_ - caught)
src = "(lambda (start step scale) (* scale 2) (lambda (n) (set! start (+ start (* n step))))"

```

```

    start))"
caught, stay = captured(src)
print(f"\nin (lambda (start step scale) ...), the inner function captures {caught}: they move
    to the heap")
print(f"{stay} is never captured, so it can stay in the stack frame and vanish when the call
    returns")

# Python does the same: captured variables live in "cells" the closure carries with it
def make_account(balance):
    def deposit(amount):
        nonlocal balance
        balance += amount; return balance
    return deposit
d = make_account(100); d(50)
print(f"\nPython: deposit.__code__.co_freevars = {d.__code__.co_freevars}, "
    f"cell holds {d.__closure__[0].cell_contents}")

```

OUTPUT

two counters, each with its own captured n: c1 c1 c1 c2 -> [1, 2, 3, 1]

Python, [lambda: i for i in range(3)], called: [2, 2, 2] (one shared i)
 with i=i, a parameter evaluated at creation time: [0, 1, 2]
 our interpreter, one binding for the loop: [2, 2, 2]
 our interpreter, a fresh binding per iteration: [0, 1, 2]

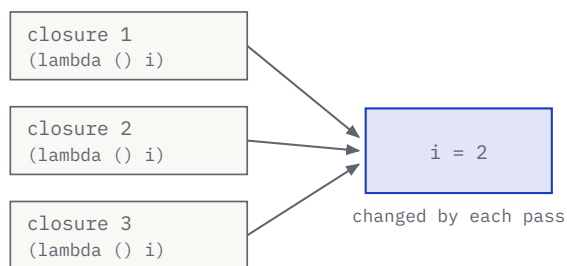
in (lambda (start step scale) ...), the inner function captures ['start', 'step']: they move to
 the heap
 ['scale'] is never captured, so it can stay in the stack frame and vanish when the call returns

Python: deposit.__code__.co_freevars = ('balance',), cell holds 150

Closures capture variables, not values

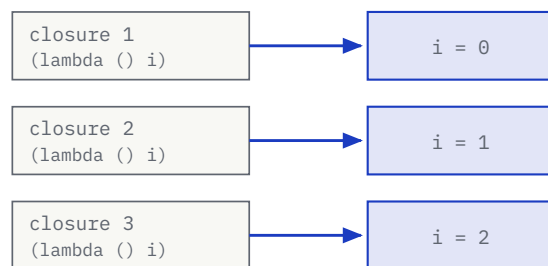
The second part of the example is one of the most common bugs in languages with closures. `[lambda: i for i in range(3)]` builds three functions, and all three return 2. Each `lambda` captured the **variable** `i`, not its value at the moment the `lambda` was created, and there is only one `i`, which the loop updated to 2 before any of the functions was called. The interpreter reproduces the bug exactly when the loop uses one binding for all iterations, and fixes it when each iteration gets a fresh binding.

ONE BINDING FOR THE LOOP



calls give [2, 2, 2]

A FRESH BINDING EACH TIME



calls give [0, 1, 2]

The loop-variable trap. Each closure keeps a pointer to a scope, not a copy of a value. If the loop reuses one scope, every closure sees its final value.

Languages have repeatedly changed their rules to remove this trap. In JavaScript, `var` declares one variable for the whole function, so closures made in a loop share it, while `let` in a `for` loop creates a fresh binding for each iteration. C# 5 changed `foreach` to use a fresh variable per iteration in 2012, and Go 1.22 made the same change for `for` loops in 2024, after years of the bug in production code. Python kept the shared variable. Its idiom for capturing the current value is a default argument, `lambda i=i: i`, because default values are evaluated when the function is created.

Other languages make the capture explicit. Java's lambdas may only capture local variables that are never reassigned, "effectively final", so the question of value or variable never arises. C++ lambdas list their captures, by value with `[=]` or by reference with `[&]`, and a lambda that captures a local by reference and outlives the function holds a dangling reference, which is undefined behaviour. Rust's borrow checker turns that same mistake into a compile error, and the `move` keyword makes a closure take ownership of what it captures.

Where captured variables live

An interpreter whose every scope is a heap-allocated dictionary pays for closures everywhere, even in functions that create none. A compiler wants local variables in a stack frame, or better in registers, which disappear when the function returns. It can do that safely for every variable that no inner function captures. So the compiler finds, for each function, the variables that some nested function refers to without declaring them. These are the inner function's **free variables**. Only those must outlive the frame. The third part of the example computes this for a function with three parameters: two are captured by the inner function and must move to the heap, and the third, `scale`, can stay on the stack.

Implementations differ in how they move them. CPython puts each captured variable in a **cell**, a small heap object holding one value. The outer function's frame and the closure both point to the cell, so assignments through either are seen by both. The code object records the decision, in `co_cellvars` for the function that owns the variable and `co_freevars` for the closure that uses it, as the last lines of the example show. Lua and the clox interpreter of *Crafting Interpreters* use **upvalues**: a captured variable stays in the stack frame while the outer function runs, with the closure pointing into the stack, and is copied to the heap, "closed", only when the frame is about to be popped. That keeps the common case, closures that don't outlive their creator, as fast as plain locals.

The general technique behind these decisions is **escape analysis**: working out whether a value can be reached after the function that created it returns. Go's compiler uses it for every allocation. A `&x` that never leaves the function lets `x` stay on the stack, while one that is returned or stored in a global moves `x` to the heap, and `go build -gcflags=-m` prints each decision. The JVM's JIT compilers use the same analysis to remove allocations of objects that never escape.

Creating closures in a loop in a language where the loop variable is shared, and getting its last value everywhere.

Expecting a closure to copy the values it uses. It keeps the variables, and sees later changes.

Capturing by reference in C++ and returning the lambda, leaving it pointing at a dead stack frame.

Keeping large objects alive by accident, because a long-lived closure captured the scope that refers to them.

Assigning to a captured variable in Python without `nonlocal`, which creates a new local instead.

EXERCISES

EXERCISE 10.1 Adders

What does `[f(10) for f in [lambda x: x + i for i in range(3)]]` evaluate to in Python, and why? Give two ways to make it produce `[10, 11, 12]`.

SOLUTION

It gives `[12, 12, 12]`. The list comprehension has one variable `i`, in the comprehension's own scope. All three lambdas capture it, and by the time any of them is called the comprehension has finished, leaving `i` at 2, so each returns `10 + 2`. The first fix captures the value at creation time with a default argument: `lambda x, i=i: x + i`. Its drawback is that callers can pass `i` by accident. The second creates a new scope per value with a factory function, `def adder(i): return lambda x: x + i`, then `[adder(i) for i in range(3)]`: each call of `adder` has its own `i`, which is the "fresh binding per iteration" of the example. `functools.partial(operator.add, i)` does the same without a named helper.

EXERCISE 10.2 Two closures, one variable

Write a function in the example's language that returns two closures over the same variable, one that increments it and one that reads it, and show that the reader sees the incrementer's changes. What, in the environment structure, makes this work?

SOLUTION

Make the outer function return a pair of lambdas created in the same scope:

```
(lambda ()
  ((lambda (n)
    (pair (lambda () (set! n (+ n 1)))
          (lambda () n)))
   0))
```

with `pair` added as a built-in that returns a Python tuple. Calling the increment closure twice and then the reader returns 2. Both closures were created while evaluating the inner `lambda (n)`'s body, so both hold a pointer to the same scope, the one where `n` was bound to 0. `set!` finds `n` in that scope and changes it there, and the reader looks `n` up in the same scope. Nothing is copied, so there is only ever one `n`. CPython does the same with one cell referenced from both closures' `__closure__` tuples, which you can check with `is`.

EXERCISE 10.3 Returning a local

In C, a function that returns a pointer to one of its local variables has a serious bug. A JavaScript function that returns a closure reading one of its local variables doesn't. What does the JavaScript implementation do that C doesn't, and what does it cost?

SOLUTION

In C, every local lives in the function's stack frame, which is reused by the next call as soon as the function returns. The returned pointer then points into memory that something else will overwrite, and reading through it is undefined behaviour. The compiler doesn't move the variable, because C's semantics say locals end with their function. The JavaScript engine sees, while compiling the function, that an inner function captures the variable, so it allocates that variable in a heap object, V8 calls it a context, instead of in the stack frame. The closure holds a reference to the context, and the garbage collector keeps it alive for as long as the closure is reachable. The cost is a heap allocation when the function is called, an extra indirection on each access to the captured variable, and more work for the collector. Variables that aren't captured stay in registers or on the stack, so functions without closures don't pay it.

- ☐ Can you explain what a closure contains and why its environment outlives the call that created it?
- ☐ Can you build an object with private state from closures alone?
- ☐ Can you explain the loop-variable trap and how JavaScript's `let`, C# and Go fixed it?
- ☐ Can you work out which of a function's variables are captured, and so must leave the stack?
- ☐ Can you describe cells and upvalues, and what escape analysis decides?

GO DEEPER

BOOK [Crafting Interpreters, chapter 25: Closures](#) Robert Nystrom

ARTICLE [Fixing For Loops in Go 1.22](#) David Chase & Russ Cox

DOCS [MDN: Closures](#) Mozilla

Bytecode

3 exercises

A tree-walker rediscovers the program's structure every time it runs a line. A bytecode compiler discovers it once, and writes the result down as a flat sequence of simple instructions for an imaginary machine. Expressions become pushes and arithmetic on a stack, variables become numbered slots, and loops and conditionals become jumps. CPython, the JVM, Lua, Ruby, .NET and WebAssembly all run bytecode of some kind. This chapter compiles a small language with loops to stack bytecode, explains how forward jumps are filled in after the fact, and is honest about where the speed comes from.

A stack machine

The simplest instruction set for compiled expressions works on a **stack** of values. `CONST 3` pushes a constant. `LOAD 1` pushes the value of variable slot 1. `ADD` pops two values and pushes their sum, and `STORE 1` pops a value into slot 1. Compiling an expression is the postorder walk of chapter 1: compile the operands, which leave their values on the stack, then emit the operator, which consumes them. No instruction needs to name where its operands are, because they are always at the top of the stack. That makes instructions short, usually one byte for the operation and sometimes one or two more for an operand, and it makes the compiler simple.

Two tables accompany the code. The **constant pool** holds the literal values the program uses, so `CONST 2` means "the constant at index 2", not the number 2, and large or non-numeric constants such as strings take no space in the instruction stream. The **slot table** maps each variable name to a number. Both are the resolver's work from chapter 5, done during compilation, and after it no name is ever looked up as a string at run time.

Control flow becomes jumps

A tree represents an `if` by nesting its branches inside it. Bytecode has no nesting, so structure becomes **jumps**. `JUMP n` continues at instruction *n*, and `JUMP_IF_FALSE n` pops a value and jumps only if it's false. An `if` compiles to its condition, a conditional jump over the then-branch, the then-branch, an unconditional jump over the else-branch, and the else-branch. A `while` loop compiles to its condition, a conditional jump out, the body, and a jump back to the condition.

The compiler meets a problem with forward jumps: when it emits the jump over the then-branch, it doesn't yet know where the else-branch will start, because it hasn't compiled the then-branch. The standard solution is **backpatching**. Emit the jump with an empty target, remember its position, compile the code it jumps over, and then go back and fill in the

target, which is now the current position. Backward jumps, such as the jump to the top of a loop, need no patching, because their targets are already known. The example's `patch` method is the whole technique.

EXAMPLE

bytecode.py · compiling a loop with an if-else to stack bytecode with backpatched jumps, and running it beside a tree-walker

```
import re

def read(src):
    tokens = re.findall(r"\(|\)|\[^\s()]+\|", src)
    def form():
        t = tokens.pop(0)
        if t != "(": return int(t) if t.lstrip("-").isdigit() else t
        out = []
        while tokens[0] != ")": out.append(form())
        tokens.pop(0); return out
    return form()

program = read("""
(begin
  (set total 0)
  (set i 1)
  (while (< i 11)
    (begin
      (if (< (* i i) 50) (set total (+ total i)) (set total (- total 1)))
      (set i (+ i 1))))
  (print total))
""")

class Compiler:
    """Turns the tree into a flat list of instructions for a stack machine."""
    def __init__(self): self.code, self.consts, self.slots = [], [], {}
    def emit(self, op, arg=None): self.code.append([op, arg]); return len(self.code) - 1
    def patch(self, at): self.code[at][1] = len(self.code) # fill in a forward jump's
        target
    def expr(self, x):
        if isinstance(x, int):
            if x not in self.consts: self.consts.append(x)
            self.emit("CONST", self.consts.index(x))
        elif isinstance(x, str): self.emit("LOAD", self.slots[x])
        else:
            self.expr(x[1]); self.expr(x[2])
            self.emit({"+": "ADD", "-": "SUB", "*": "MUL", "<": "LESS"}[x[0]])
    def stmt(self, x):
        op = x[0]
        if op == "begin":
            for s in x[1:]: self.stmt(s)
        elif op == "set":
            self.expr(x[2]); self.emit("STORE", self.slots.setdefault(x[1], len(self.slots)))
        elif op == "print": self.expr(x[1]); self.emit("PRINT")
        elif op == "if":
            self.expr(x[1])
            to_else = self.emit("JUMP_IF_FALSE") # target not known yet
            self.stmt(x[2])
            to_end = self.emit("JUMP")
            self.patch(to_else) # backpatch: else starts
                here
            self.stmt(x[3])
            self.patch(to_end)
        elif op == "while":
```

```

        top = len(self.code)
        self.expr(x[1])
        to_exit = self.emit("JUMP_IF_FALSE")
        self.stmt(x[2])
        self.emit("JUMP", top)                                # backwards: target
        already known
        self.patch(to_exit)

c = Compiler(); c.stmt(program); c.emit("HALT")
names = {v: k for k, v in c.slots.items()}
print(f"constants {c.consts}, variables {c.slots}")
for i, (op, arg) in enumerate(c.code):
    note = (f" ({c.consts[arg]})" if op == "CONST" else f" ({names[arg]})" if op in ("LOAD",
        "STORE")
        else "")
    print(f" {i:2} {op:14}{' ' if arg is None else arg}{note}".rstrip())

def run(code, consts):
    stack, slots, pc, executed, out = [], [0] * len(c.slots), 0, 0, []
    while True:
        op, arg = code[pc]; pc += 1; executed += 1
        if op == "CONST": stack.append(consts[arg])
        elif op == "LOAD": stack.append(slots[arg])
        elif op == "STORE": slots[arg] = stack.pop()
        elif op in ("ADD", "SUB", "MUL", "LESS"):
            b, a = stack.pop(), stack.pop()
            stack.append({"ADD": a + b, "SUB": a - b, "MUL": a * b, "LESS": a < b}[op])
        elif op == "JUMP": pc = arg
        elif op == "JUMP_IF_FALSE":
            if not stack.pop(): pc = arg
        elif op == "PRINT": out.append(stack.pop())
        elif op == "HALT": return out, executed

out, executed = run(c.code, c.consts)
size = sum(1 if arg is None else 2 for _, arg in c.code)        # 1 byte of opcode, 1 of
operand
def count_nodes(x): return 1 + sum(count_nodes(y) for y in x[1:]) if isinstance(x, list) else 1
print(f"\nprinted {out}; {len(c.code)} instructions, {size} bytes of bytecode; "
    f"{executed} instructions executed")

steps = 0
def walk(x, env):                                              # the same program, tree-
    walked
    global steps; steps += 1
    if isinstance(x, int): return x
    if isinstance(x, str): return env[x]
    op = x[0]
    if op == "begin":
        for s in x[1:]: walk(s, env)
    elif op == "set": env[x[1]] = walk(x[2], env)
    elif op == "print": print(f"tree-walker printed {walk(x[1], env)} after", end=" ")
    elif op == "if": walk(x[2] if walk(x[1], env) else x[3], env)
    elif op == "while":
        while walk(x[1], env): walk(x[2], env)
    else:
        a, b = walk(x[1], env), walk(x[2], env)
        return a + b if op == "+" else a - b if op == "-" else a * b if op == "*" else a < b
walk(program, {})
print(f"visiting {steps} tree nodes; the tree itself has {count_nodes(program)} nodes")

```

OUTPUT

```

constants [0, 1, 11, 50], variables {'total': 0, 'i': 1}
0  CONST      0  (0)
1  STORE      0  (total)
2  CONST      1  (1)

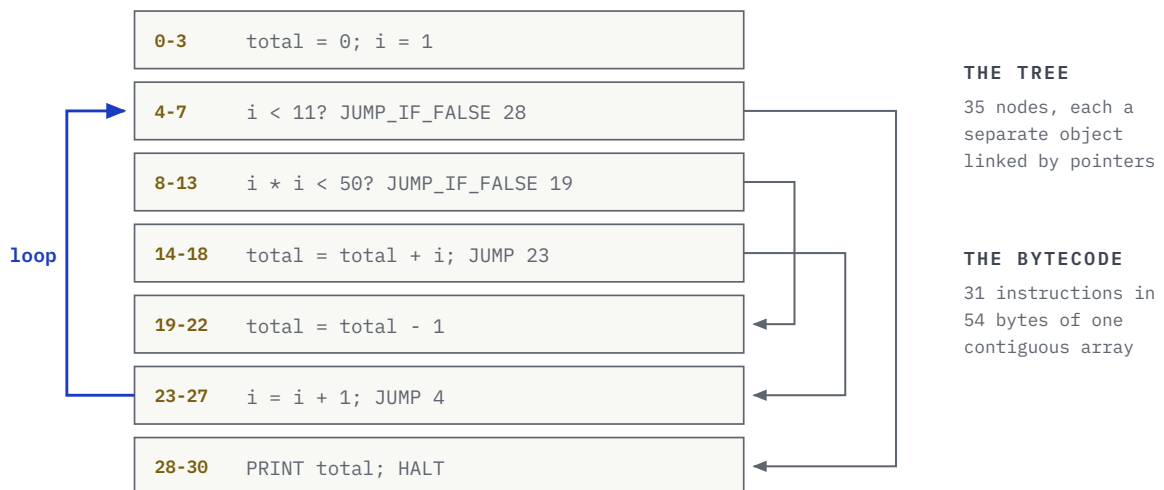
```

```

3  STORE      1  (i)
4  LOAD       1  (i)
5  CONST      2  (11)
6  LESS
7  JUMP_IF_FALSE 28
8  LOAD       1  (i)
9  LOAD       1  (i)
10 MUL
11 CONST      3  (50)
12 LESS
13 JUMP_IF_FALSE 19
14 LOAD       0  (total)
15 LOAD       1  (i)
16 ADD
17 STORE      0  (total)
18 JUMP       23
19 LOAD       0  (total)
20 CONST      1  (1)
21 SUB
22 STORE      0  (total)
23 LOAD       1  (i)
24 CONST      1  (1)
25 ADD
26 STORE      1  (i)
27 JUMP       4
28 LOAD       0  (total)
29 PRINT
30 HALT

```

printed [25]; 31 instructions, 54 bytes of bytecode; 208 instructions executed
tree-walker printed 25 after visiting 191 tree nodes; the tree itself has 35 nodes



The compiled program as runs of straight-line code joined by jumps. The forward jumps, on the right, were emitted before their targets existed and patched afterwards; the loop's backward jump, on the left, was not.

Where the speed comes from

The numbers need reading with care. The virtual machine executed 208 instructions, and the tree-walker visited 191 nodes. Bytecode didn't do fewer steps; it did slightly more, because operations like `JUMP` exist only to replace structure. The gain is in what each step

costs, and in what it doesn't do.

- **No repeated decoding.** The tree-walker inspects each node's type and shape on every visit. The virtual machine reads an opcode and an operand and acts. The question "what kind of node is this?" was answered once, at compile time.
- **Compactness.** The whole program is 54 bytes in one contiguous array, while the tree is 35 separately allocated objects connected by pointers. Walking an array uses the processor's caches and prefetcher well. Chasing pointers to objects scattered in memory doesn't.
- **No host recursion.** The virtual machine is a loop. Evaluating a deeply nested expression or a long chain of calls doesn't recurse in the host language, which also removes the stack-depth problem of chapter 9. Chapter 12 shows how calls work in that loop.
- **Names are gone.** Variables are slot numbers, so access is an array index.

In Python, which is itself interpreted, these advantages mostly cancel out, because every instruction of our virtual machine costs dozens of CPython instructions either way. In C, where the dispatch loop compiles to a few machine instructions per bytecode, a bytecode interpreter is typically several times faster than a tree-walker.

The remaining overhead is the **dispatch** itself: fetching the next opcode and branching to its handler, a branch the processor finds hard to predict because the same indirect jump goes to every handler. Production interpreters attack it in several ways. CPython uses **computed gotos** where the C compiler supports them, so each handler jumps directly to the next one, giving the branch predictor one prediction site per handler. **Superinstructions** fuse common sequences into one opcode, which chapter 12's example does for an operator with a constant operand. And since 3.11, CPython **specialises** instructions as it runs, replacing a generic `BINARY_OP` with an integer-only version after seeing that its operands are integers.

Bytecode as a file format

Bytecode is also a convenient thing to store and ship. CPython caches it in `.pyc` files to skip recompiling unchanged modules. Java distributes programs as `.class` files of JVM bytecode, which any JVM on any platform can run, and `javap -c` disassembles them. WebAssembly is a bytecode designed for the web: a stack machine, like this chapter's, with a compact binary encoding that browsers compile to machine code. Formats that are loaded from untrusted sources need **verification**: before running a class file, the JVM checks that every instruction's operands have the right types and that the stack can't underflow on any path, so that bytecode crafted by hand can't break the virtual machine's safety. CPython does no such check, and it's possible to crash it with handcrafted bytecode, which is one reason `.pyc` files should never be accepted from untrusted sources.

Not every bytecode is a stack machine. Lua switched to a **register machine** in version 5.0, in 2003, whose instructions name their operands, such as `ADD R1 R2 R3`, and the Dalvik virtual machine used for Android apps was register-based too. Chapter 12 compares the two designs.

COMMON TRAPS

Forgetting to patch a forward jump, so it jumps to instruction 0 or to whatever placeholder was left.

Leaving values on the stack after an expression statement, so the stack grows on every loop iteration.

Assuming bytecode executes fewer steps than a tree-walker. It usually executes slightly more, each much cheaper.

Encoding jump targets in one byte, which works until the first function longer than 255 instructions.

Loading bytecode from untrusted sources into a virtual machine that doesn't verify it.

EXERCISES

EXERCISE 11.1 Compile by hand

Compile `(if (< a b) (set m b) (set m a))`, with `a`, `b` and `m` in slots 0, 1 and 2, starting at instruction 10. Write each instruction with its index, and say which targets had to be backpatched.

SOLUTION

```
10  LOAD 0          a
11  LOAD 1          b
12  LESS
13  JUMP_IF_FALSE 17  patched after the then-branch
14  LOAD 1
15  STORE 2         m = b
16  JUMP 19         patched after the else-branch
17  LOAD 0
18  STORE 2         m = a
19  ...
```

Both jumps are forward and were emitted before their targets existed, so both were backpatched: the conditional jump at 13 once the then-branch ended and the else-branch was about to start, at 17, and the unconditional jump at 16 once the else-branch ended, at 19. A good compiler would notice that both branches store into the same slot and compile `(set m (if ...))` instead, with one `STORE` after the branches join. That's the kind of rewriting chapter 16 does on a lower-level representation.

EXERCISE 11.2 Short-circuit and

Add `(and a b)`, which evaluates `b` only if `a` is true, and whose value is `a` if it's false and `b` otherwise. Which instructions does it compile to? Why can't it be compiled like `ADD`?

SOLUTION

Compiled like `ADD`, both operands would always be evaluated, breaking idioms like `(and (not-null p) (field p))`, where the second operand is only safe when the first is true. It needs a jump. One encoding uses an instruction that peeks instead of popping: compile `a`, emit `JUMP_IF_FALSE_KEEP end`, which jumps without popping if the top is false, then `POP`, compile `b`, and patch `end` to here. If `a` is false, the jump leaves `a` on the stack as the result. Otherwise the `POP` discards it and `b`'s value becomes the result. CPython's compiler does essentially this, using `COPY` to duplicate the value and a conditional jump that pops. `or` is the mirror image, jumping if the value is true.

EXERCISE 11.3 Reading real bytecode

Disassemble `def f(n): return sum(i * i for i in range(n) if i % 3)` with `dis.dis(f)` in your Python, and find: where the constants are loaded, where a variable is read by slot, and where a jump implements the `if`. What surprises you?

SOLUTION

The details depend on the Python version, but the structure is always visible. `LOAD_CONST` or `LOAD_SMALL_INT` loads 3 from the constant pool or directly from the operand. `LOAD_FAST`, or a variant such as `LOAD_FAST_BORROW` in recent versions, reads `i` or `n` by slot number, with the name shown in parentheses only as a courtesy of the disassembler. The `if i % 3` becomes a `BINARY_OP` with the remainder operator, a `TO_BOOL`, and a conditional jump, `POP_JUMP_IF_TRUE` or `POP_JUMP_IF_FALSE` depending on the version, which is the equivalent of the example's `JUMP_IF_FALSE`. The usual surprises: the generator expression is a separate code object, compiled as its own function and disassembled after `f`; the loop is driven by `FOR_ITER`, which calls the iterator and jumps out when it's exhausted; superinstructions such as `STORE_FAST_LOAD_FAST` do two things at once; and `sum` is loaded with `LOAD_GLOBAL`, a name lookup at run time, because globals and built-ins are resolved late, as chapter 5 explained. Running `dis.dis(f, adaptive=True)` after calling `f` a few times shows the specialised instructions that have replaced the generic ones.

CHECKLIST

- ☐ Can you compile expressions, assignments, `if` and `while` to stack bytecode?
- ☐ Can you explain backpatching and why only forward jumps need it?
- ☐ Can you explain where a bytecode interpreter's speed comes from, given that it executes more steps?
- ☐ Can you describe dispatch overhead and two techniques that reduce it?
- ☐ Can you disassemble a Python function and map its instructions back to the source?

GO DEEPER

BOOK [Crafting Interpreters, chapter 14: Chunks of Bytecode](#) Robert Nystrom

DOCS [dis: Disassembler for Python bytecode](#) Python documentation

SPEC [WebAssembly Core Specification](#) W3C WebAssembly Working Group

Virtual machines

3 exercises

A virtual machine is a loop that fetches an instruction, decodes it and executes it, plus the state the instructions work on: a stack of values, a stack of call frames, and an instruction pointer. Chapter 11's machine had no functions. Adding calls means deciding where arguments go, how a function finds its locals and how a return finds its way back, and those decisions shape everything from performance to how exceptions work. This chapter adds calls to the machine, measures the effect of fusing common instruction pairs, and compares stack machines with register machines.

Frames on the value stack

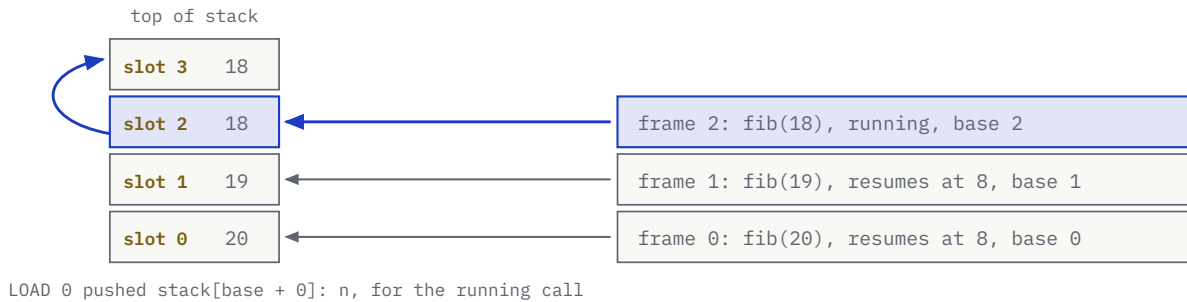
A function call needs somewhere to keep its arguments and locals, a record of where to resume the caller, and a way to deliver the result. The arrangement used by clox, Lua and the example avoids copying. The caller pushes the arguments onto the value stack, where they already are after evaluating them, and executes `CALL`. The machine saves the caller's position in its frame record, and starts a new frame whose **base** is the stack index of the first argument. Inside the function, `LOAD 0` means "the value at `base + 0`", the first parameter, so the arguments become the callee's locals in place. `RET` pops the result, discards everything from `base` upward, pops the frame, and pushes the result for the caller.

A frame record holds three things: the function's code, the instruction pointer to resume at, and the base. The instruction pointer is the **return address**, exactly what a processor's `call` instruction pushes onto the hardware stack, and the base plays the role of the frame pointer register. The virtual machine is a model of the real one in chapter 17, with the parts that a processor provides in hardware written as ordinary data structures.

ONE VALUE STACK

CALL FRAMES: CODE, RETURN IP, BASE

CALL pushes a frame whose base is the first argument's slot; RET pops it, discards the slots above base and pushes the result



The machine three calls into `(fib 20)`, just after `LOAD 0`. Every call's argument is already on the shared stack when the call begins, so a frame is just a record of where its slots start and where to go back to.

Because the frames are the machine's own data, the depth of recursion in the program is independent of recursion in the machine. The example's `run` is one loop that never calls itself, and the only limit on the program's call depth is the memory for the two stacks. A production machine sets a maximum anyway, so that runaway recursion produces a clean "stack overflow" error instead of exhausting memory.

EXAMPLE

vm.py · a stack VM with call frames running fib 20, with and without fused operator-and-constant instructions

```
import re, textwrap

def read(src):
    tokens = re.findall(r"\(|\)|[^\s()]+", src)
    def form():
        t = tokens.pop(0)
        if t != "(": return int(t) if t.lstrip("-").isdigit() else t
        out = []
        while tokens[0] != ")": out.append(form())
        tokens.pop(0); return out
    return form()

ARITH = {"+": "ADD", "-": "SUB", "<": "LESS"}
def compile_fn(params, body, fused):
    """Compile one function. Parameters live in slots counted from the frame's base."""
    code = []
    def expr(x):
        if isinstance(x, int): code.append(("CONST", x))
        elif isinstance(x, str): code.append(("LOAD", params.index(x)))
        elif x[0] == "if":
            expr(x[1]); jf = len(code); code.append(None)
            expr(x[2]); j = len(code); code.append(None)
            code[jf] = ("JUMP_IF_FALSE", len(code)); expr(x[3]); code[j] = ("JUMP", len(code))
        elif x[0] in ARITH:
            expr(x[1])
            if fused and isinstance(x[2], int):
                # one instruction: operator and constant
```

```

        code.append((ARITH[x[0]] + "_K", x[2]))
    else: expr(x[2]); code.append((ARITH[x[0]], None))
else:
    # a call: push the arguments,
    then CALL
    for a in x[1:]: expr(a)
    code.append(("CALL", (x[0], len(x) - 1)))
expr(body); code.append(("RET", None))
return code

def run(functions, entry, args):
    stack = list(args)
    frames = [(functions[entry], 0, 0)]
    # (code, instruction pointer,
    # base of locals)
    executed, deepest = 0, 1
    code, ip, base = frames[-1]
    while True:
        op, arg = code[ip]; ip += 1; executed += 1
        if op == "CONST": stack.append(arg)
        elif op == "LOAD": stack.append(stack[base + arg])
        elif op in ("ADD", "SUB", "LESS"):
            b = stack.pop(); a = stack.pop()
            stack.append(a + b if op == "ADD" else a - b if op == "SUB" else a < b)
        elif op in ("ADD_K", "SUB_K", "LESS_K"):
            a = stack.pop()
            stack.append(a + arg if op == "ADD_K" else a - arg if op == "SUB_K" else a < arg)
        elif op == "JUMP": ip = arg
        elif op == "JUMP_IF_FALSE":
            if not stack.pop(): ip = arg
        elif op == "CALL":
            name, nargs = arg
            frames[-1] = (code, ip, base)
            # save where to come back to
            code, ip, base = functions[name], 0, len(stack) - nargs
            frames.append((code, ip, base)); deepest = max(deepest, len(frames))
        elif op == "RET":
            result = stack.pop(); del stack[base:]
            frames.pop()
            if not frames: return result, executed, deepest
            stack.append(result)
            code, ip, base = frames[-1]

src = "(define (fib n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))"
_, (name, *params), body = read(src)
for fused in (False, True):
    fns = {name: compile_fn(params, body, fused)}
    result, executed, deepest = run(fns, "fib", [20])
    label = "with operator-plus-constant instructions" if fused else "plain stack code"
    print(f"{label}: {len(fns['fib'])} instructions")
    def text(op, a): return op if a is None else f"{op} {a[0] if isinstance(a, tuple) else a}"
    listing = "; ".join(text(op, a) for op, a in fns["fib"])
    print(textwrap.indent(textwrap.fill(listing, 90), " "))
    print(f"    fib(20) = {result}: {executed:,} instructions executed, {deepest} frames deep at most")

# Stack machine or register machine? The same statement, a = b + c * d, both ways
stack_code = ["LOAD b", "LOAD c", "LOAD d", "MUL", "ADD", "STORE a"]
register_code = ["MUL t1, c, d", "ADD a, b, t1"]
print(f"\na = b + c * d on a stack VM: {len(stack_code)} instructions: {'; '.join(stack_code)}")
print(f"a = b + c * d on a register VM: {len(register_code)} instructions: {'; '.join(register_code)}")
print("    (each register instruction is wider: it names its operands instead of using the stack)")

```

OUTPUT

plain stack code: 16 instructions

```

LOAD 0; CONST 2; LESS; JUMP_IF_FALSE 6; LOAD 0; JUMP 15; LOAD 0; CONST 1; SUB; CALL fib;
LOAD 0; CONST 2; SUB; CALL fib; ADD; RET
fib(20) = 6765: 229,852 instructions executed, 20 frames deep at most
with operator-plus-constant instructions: 13 instructions
LOAD 0; LESS_K 2; JUMP_IF_FALSE 5; LOAD 0; JUMP 12; LOAD 0; SUB_K 1; CALL fib; LOAD 0;
SUB_K 2; CALL fib; ADD; RET
fib(20) = 6765: 186,071 instructions executed, 20 frames deep at most

a = b + c * d on a stack VM:    6 instructions: LOAD b; LOAD c; LOAD d; MUL; ADD; STORE a
a = b + c * d on a register VM: 2 instructions: MUL t1, c, d; ADD a, b, t1
(each register instruction is wider: it names its operands instead of using the stack)

```

Instruction design

The example compiles `fib` twice. The second time, an arithmetic operator whose right operand is a constant becomes a single instruction carrying the constant, so `CONST 1; SUB` becomes `SUB_K 1`. That shrank the function from 16 instructions to 13 and the run from 229,852 executed instructions to 186,071, 19% fewer, without changing what the program computes. Each removed instruction is one fewer trip round the dispatch loop: one fewer fetch, one fewer unpredictable branch, one fewer push and pop.

This is how real instruction sets are tuned: compile representative programs, count which instruction sequences are most frequent, and give the common ones their own opcodes. The JVM has `iconst_0` to `iconst_5` and `iload_0` to `iload_3` because small constants and the first few locals are overwhelmingly common. CPython has `LOAD_FAST_LOAD_FAST` for loading two locals at once. The cost of every new opcode is a larger, more complicated interpreter, and eventually the dispatch code itself stops fitting comfortably in the processor's instruction cache.

Stack machines and register machines

A **stack machine's** instructions take their operands from the top of the stack. A **register machine's** instructions name their operands, which are slots in the current frame called virtual registers. The last lines of the example show the difference for `a = b + c * d`: six stack instructions, or two register instructions. Register code executes fewer instructions, so it pays the dispatch cost fewer times, and it avoids the pushes and pops that only move values to where the next instruction expects them. Each instruction is wider, though, because it carries two or three operand fields, and the compiler must allocate registers, a simple version of chapter 17's problem.

Measurements bear this out. A 2008 study by Shi, Casey, Ertl and Gregg, translating JVM stack code to a register form, found the register version executed about 47% fewer instructions with bytecode about 25% larger, and ran faster overall. Lua moved from a stack machine to a register machine in version 5.0 for these reasons, and Dalvik, Android's original virtual machine, was register-based. Stack machines remain popular for formats that must be small and simple to verify: the JVM, .NET and WebAssembly are all stack machines, and the engines that run them compile the stack code to registers anyway, as machine code.

What else the machine needs

Value representation. Every slot on the stack must hold any value the language allows, with its type, as chapter 6 discussed. clox uses a tagged union: a type tag and a field large enough for a number or a pointer. Faster machines pack the tag into the value.

JavaScriptCore and LuaJIT use **NaN-boxing**, storing pointers and small integers inside the unused bit patterns of IEEE floating-point NaNs, so every value fits in 64 bits and floating-point numbers need no conversion. V8 stores small integers directly in tagged pointer-sized words, telling them apart from pointers by their lowest bit.

Exceptions. A `throw` must find the nearest handler, which may be several frames up, pop those frames, restore the value stack to the height it had when the handler's `try` began, and jump to the handler. Older designs pushed a handler record at every `try` and popped it at the end, which costs time even when nothing is thrown. The JVM and, since 3.11, CPython use **exception tables** instead: a per-function table mapping ranges of instruction addresses to handlers, consulted only when an exception is actually raised, so entering a `try` block costs nothing.

Calls to the host. Built-in functions written in the host language, like Python's `len` or the example's arithmetic, need a calling convention of their own: the machine passes them a slice of the stack and pushes what they return. That boundary between the virtual machine and native code is the subject of chapter 20.

COMMON TRAPS

Copying arguments into a new frame instead of using them in place on the stack.

Forgetting to discard a frame's slots on return, so the stack grows with every call.

Adding opcodes without measuring, until the interpreter is large and no faster.

Recurring in the host on every call, which brings back the stack limit the virtual machine was meant to remove.

Restoring the instruction pointer on a `throw` but not the stack height, leaving the handler with stray values.

EXERCISES

EXERCISE 12.1 Accounting for the saving

The fused instructions saved exactly 43,781 executed instructions.

Explain that number, given that `(fib 20)` makes 10,945 calls with `n ≥ 2` and 10,946 with `n < 2`.

SOLUTION

Fusing removes one executed instruction each time a `CONST k` followed by an operator runs. The body has three such pairs: `(< n 2)`, `(- n 1)` and `(- n 2)`. A call with `n < 2` runs only the comparison and returns `n`, so it executes one pair and saves 1 instruction. A call with `n ≥ 2` runs the comparison and both subtractions, saving 3. The total is $10,945 \times 3 + 10,946 \times 1 = 32,835 + 10,946 = 43,781$. That's exactly $229,852 - 186,071$. Exact agreement is the point of counting instead of timing: it confirms that the change did what was intended and nothing else. The same accounting shows the limit of this optimisation. Even with every constant fused, the machine still executes the calls, loads and jumps, which are most of the remaining 186,071.

EXERCISE 12.2 Locals that aren't parameters

The example's functions have only parameters. Add `(let (x value) body)` inside functions, with `x` stored in the frame. Where does `x`'s slot go, and how does the compiler keep track of slot numbers when `let` s nest?

SOLUTION

Give the compiler a list of names in scope, starting with the parameters, so that a name's slot is its index in the list. To compile `(let (x value) body)`, compile `value`, which leaves its result on the stack, at slot `base + len(names)` because that's the next free position. Then append `x` to the list, compile the body, which refers to `x` as `LOAD len(names) - 1`, and remove `x` from the list afterwards. At run time the value simply stays where it was computed, so the binding costs no instruction at all. At the end of the `let`, the body's result is on top and `x`'s value is below it, so emit an instruction that removes the slot under the top, or arrange for `RET` to discard it with the rest of the frame if the `let` is the whole body. Nested `let` s just keep appending, so their slots are allocated like a stack, which matches how their lifetimes nest. This is how clox handles local variables.

EXERCISE 12.3 Throw and catch

Design `(try body handler)` and `(throw value)` for the example's machine, using an exception table rather than instructions that push handlers. What does each table entry contain, and what does the machine do when `THROW` executes?

SOLUTION

When compiling `try`, record the instruction range of the body, `start` to `end`, the address where the handler's code begins, and the stack height relative to the frame's base at the start of the body, and add the entry to the function's exception table. The body compiles normally, followed by a jump over the handler. `THROW` pops the thrown value and searches: in the current frame's table, find an entry whose range contains the current instruction pointer. If there is one, truncate the stack to `base` plus the recorded height, push the thrown value for the handler, and jump to the handler address. If there isn't, pop the frame, set the instruction pointer to the caller's saved position, and search again in the caller's table. If no frame has a handler, report an uncaught exception with the chain of frames as the traceback, which is exactly the information the frame records already hold. Entering a `try` costs nothing at run time, which is why CPython 3.11 called its version "zero-cost" exceptions.

CHECKLIST

- ☐ Can you implement `CALL` and `RET` with frames that use arguments in place on the stack?
- ☐ Can you explain what a frame records and how it corresponds to a processor's return address and frame pointer?
- ☐ Can you explain how fused instructions save dispatch, and account for a saving exactly?
- ☐ Can you compare stack and register machines, with a real virtual machine of each kind?
- ☐ Can you describe tagged values and exception tables?

GO DEEPER

BOOK Crafting Interpreters, chapter 24: Calls and Functions Robert Nystrom

PAPER The Implementation of Lua 5.0 Ierusalimschy, de Figueiredo & Celes

DOCS CPython internals: The bytecode interpreter CPython developers

Garbage collection

3 exercises

Closures, lists and objects live on the heap, and a program in a managed language never frees them. The runtime must work out which objects the program can still reach and reuse the memory of the rest. There are two families of answers: count the references to each object, or periodically trace everything reachable from the program's roots. This chapter builds two tracing collectors over a simulated heap, a copying collector and a generational one, measures how much work each does, and shows the one-line omission that makes a generational collector free an object that's still in use.

Reachability

A garbage collector can't know which objects the program will use again, so it uses a safe approximation: an object that can't be reached by following pointers from the **roots** can never be used again. The roots are the places the program can read directly: global variables, the slots of every active stack frame, and the machine's registers. Everything reachable from them, directly or through other objects, is **live**. Everything else is garbage, even if it still points to other garbage. The example's heap makes the roots explicit: the program reaches objects only through a list called `roots`, as compiled code reaches them only through registers and stack slots, so the collector can find, follow and update every reference.

Reference counting

The first family keeps a count in every object of the references pointing to it, adds one when a reference is copied, subtracts one when a reference is overwritten or goes out of scope, and frees the object when its count reaches zero. Memory is reclaimed immediately and incrementally, with no pauses, which makes behaviour predictable: a file object closes the moment its last reference disappears. CPython uses reference counting as its primary mechanism, Swift's ARC inserts the count updates at compile time, and Rust offers it explicitly with `Rc` and `Arc`.

Reference counting has two weaknesses. Every pointer copy writes to memory, including in code that only reads data, and with several threads those writes must be atomic, which is expensive. CPython's free-threaded build, introduced by PEP 703, uses **biased** reference counting, cheap non-atomic counts for the thread that owns an object and atomic ones for others, for this reason. And reference counting can't reclaim **cycles**: two objects that point to each other keep each other's counts above zero forever after the program drops them. CPython therefore also runs a separate tracing collector, the `gc` module, whose only job is to find unreachable cycles.

Tracing: mark-sweep and copying

The second family, invented by John McCarthy for Lisp in 1960, ignores individual pointer updates and periodically traces the live set. **Mark-sweep** marks every object reachable from the roots, then sweeps the whole heap, freeing each unmarked object. It leaves the free memory fragmented, scattered between surviving objects, so allocation must search for a hole big enough. **Mark-compact** slides the survivors together afterwards, at the cost of another pass.

A **copying** collector does both jobs at once. The heap is split into two halves, and the program allocates in one of them by bumping a pointer, the fastest allocation there is. When that half fills, the collector copies every live object into the empty half, packed together, and the old half, with all its garbage, is abandoned whole. Cheney's 1970 algorithm needs no recursion and no extra memory for the traversal: the to-space itself is the queue, with one pointer for the next object to scan and another for the next free slot. When an object is copied, its old copy is overwritten with a **forwarding pointer**, so that other references to it are redirected to the same new copy rather than copying it twice. The example's `Copying` class is Cheney's algorithm.

The crucial property of a copying collector is that its work is proportional to the **live** data, not to the garbage: dead objects are never touched. That's why it suits programs that allocate many short-lived objects. It wastes half the memory, though, and long-lived objects are copied again at every collection, which is what the next design fixes.

Generations

Measurements of real programs consistently show that **most objects die young**: temporaries, intermediate results, iterator objects. The objects that survive a while tend to survive a long time. A **generational** collector, introduced by David Ungar in 1984, exploits this. New objects are allocated in a small **nursery**, which is collected often, in a **minor** collection, by copying its few survivors into an **old** space, which is collected rarely. A minor collection touches only the live young objects, and the long-lived ones are no longer copied again and again.

EXAMPLE

gc.py · Cheney's copying collector against a generational collector on the same workload, and the bug a missing write barrier causes

```
# A simulated heap. The program reaches objects only through 'roots', as compiled code reaches
# them
# through registers and stack slots, so the collector can move objects and fix the roots.
class Obj:
    def __init__(self, fields, young=True):
        self.fields, self.forward, self.young, self.dead = fields, None, young, False

    def forwarded(fields):
        # after a collection, follow
        forwarding pointers
        return [f.forward if isinstance(f, Obj) and f.forward else f for f in fields]

class Copying:
```

```

"""Cheney's semispace collector: copy everything reachable into an empty space, abandon the
rest."""
def __init__(self, capacity): self.cap, self.space, self.copied, self.gcs = capacity, [],
0, 0
def alloc(self, fields, roots):
    if len(self.space) >= self.cap: self.collect(roots); fields = forwarded(fields)
    obj = Obj(fields); self.space.append(obj); return obj
def write(self, obj, i, value): obj.fields[i] = value
def collect(self, roots):
    self.gcs += 1; to = []
    def copy(o):
        if not isinstance(o, Obj): return o
        if o.forward is None:
            o.forward = Obj(list(o.fields)); to.append(o.forward); self.copied += 1
        return o.forward
    roots[:] = [copy(r) for r in roots]
    scan = 0
    while scan < len(to):
        queue
        to[scan].fields = [copy(f) for f in to[scan].fields]; scan += 1
    for o in self.space:
        if o.forward is None: o.dead = True
    self.space = to

class Generational:
    """A nursery for new objects; survivors are promoted to an old space that's rarely
    collected."""
    def __init__(self, nursery, barrier=True):
        self.cap, self.barrier, self.nursery, self.old = nursery, barrier, [], []
        self.remembered, self.copied, self.gcs = set(), 0, 0
    def alloc(self, fields, roots):
        if len(self.nursery) >= self.cap: self.minor(roots); fields = forwarded(fields)
        obj = Obj(fields); self.nursery.append(obj); return obj
    def write(self, obj, i, value):
        obj.fields[i] = value
        if self.barrier and not obj.young and isinstance(value, Obj) and value.young:
            self.remembered.add(obj) # the write barrier: an old -> young
        pointer
    def minor(self, roots):
        self.gcs += 1; queue = []
        def promote(o):
            if not (isinstance(o, Obj) and o.young): return o
            if o.forward is None:
                o.forward = Obj(list(o.fields), young=False); self.old.append(o.forward)
                queue.append(o.forward); self.copied += 1
            return o.forward
        roots[:] = [promote(r) for r in roots]
        for o in self.remembered: o.fields = [promote(f) for f in o.fields] # extra roots
        while queue:
            o = queue.pop(); o.fields = [promote(f) for f in o.fields]
        for o in self.nursery:
            if o.forward is None: o.dead = True
        self.nursery, self.remembered = [], set()
    collect = minor

def workload(heap):
    roots = [None] * 11 # [0]: a long-lived list; [1:]: recent
    temporaries
    for k in range(2000):
        roots[0] = heap.alloc([k, roots[0], None], roots)
    for i in range(30000):
        t = heap.alloc([i, None], roots)
        roots[1 + i % 10] = t
        if i % 1000 == 0: heap.write(roots[0], 2, t) # the old list's head points at a new
        object
    heap.collect(roots) # one last collection, then check the

```

```

    heap
    dangling, seen, todo = 0, set(), [r for r in roots if r]
    while todo:
        o = todo.pop()
        if id(o) in seen: continue
        seen.add(id(o))
        for f in o.fields:
            if isinstance(f, Obj):
                if f.dead: dangling += 1
                else: todo.append(f)
    return len(seen), dangling

allocations = 32000
for name, heap in (("semispace copying, 6,000-object spaces", Copying(6000)),
                   ("generational, 1,000-object nursery", Generational(1000)),
                   ("generational, without the write barrier", Generational(1000,
                                   barrier=False))):
    live, dangling = workload(heap)
    print(f"{name}:")
    print(f"    {heap.gcs} collections, {heap.copied:,} objects copied, "
          f"{heap.copied / allocations:.2f} per allocation; {live:,} objects reachable at the"
          f"end")
    if dangling:
        print(f"    BROKEN: a live object points at {dangling} freed object{'s' if dangling > 1}"
              f"else '{}'")

```

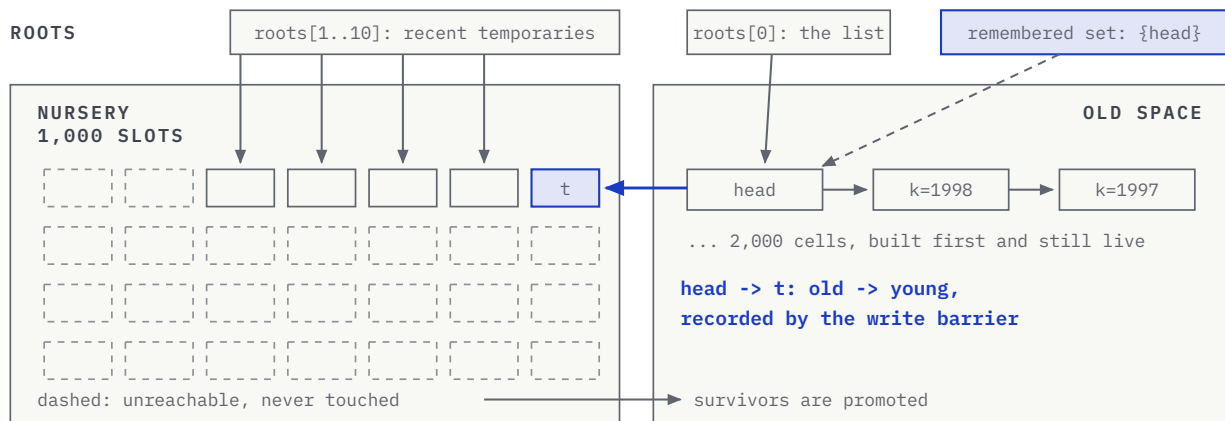
OUTPUT

```

semispace copying, 6,000-object spaces:
    8 collections, 16,088 objects copied, 0.50 per allocation; 2,011 objects reachable at the
    end
generational, 1,000-object nursery:
    32 collections, 2,330 objects copied, 0.07 per allocation; 2,011 objects reachable at the
    end
generational, without the write barrier:
    32 collections, 2,300 objects copied, 0.07 per allocation; 2,010 objects reachable at the
    end
BROKEN: a live object points at 1 freed object

```

The workload builds a 2,000-element list that lives to the end, then allocates 30,000 temporaries of which only the last ten are kept. The copying collector ran 8 times and copied the long list at every one: 16,088 copies, half an object copied for every object allocated. The generational collector ran 32 smaller collections and copied 2,330 objects in all, a seventh as many, because the list was promoted once and never copied again.



A minor collection traces from the roots and from the remembered set, never from the rest of old space. The write barrier is what puts the list head in the remembered set when the program stores a pointer to a young object in it.

Minor collections have a problem. To find the live young objects, they trace from the roots, but some young objects are reachable only through an old object, as when the program stores a new object into a field of the long list's head. Tracing all of old space to find such pointers would defeat the purpose. So the program must tell the collector whenever it creates an old-to-young pointer. Every store of a pointer into an object goes through a **write barrier**, a few instructions that check whether an old object now points to a young one and, if so, add it to a **remembered set**. A minor collection treats the remembered set as extra roots. Production collectors usually track this at the granularity of memory regions, **card marking**, rather than objects, to keep the barrier to a couple of instructions.

The third run of the example omits the barrier. Everything looks the same, the same number of collections and almost the same copying, but the final check finds a live object pointing at a freed one. A minor collection didn't know the list head referred to a young object, so it treated the object as garbage, and the list head now holds a **dangling pointer**. In a real runtime, that memory would be reused for another object, and the program would eventually read the wrong data or crash, far from the cause. Barrier bugs are among the hardest bugs in runtime systems, which is one reason write barriers are emitted by the compiler, never written by hand.

Pauses

A collector that stops the program while it works causes **pauses**, which matter for interactive programs and servers with latency targets. Two techniques reduce them. An **incremental** collector does its work in small slices interleaved with the program, and a **concurrent** one works on another thread while the program runs. Both must cope with the program changing pointers in the middle of a trace, and the standard tool is **tri-colour marking**, from Dijkstra and colleagues in 1978: objects are white (not yet seen), grey (seen, children not yet scanned) or black (done), and a write barrier ensures the program never makes a black object point to a white one unnoticed. Go's collector is concurrent mark-

sweep with pauses usually well under a millisecond. Java's ZGC and Shenandoah move objects concurrently too. CPython 3.14 made its cycle collector incremental, cutting its worst pauses on large heaps.

None of this is free: barriers slow every pointer store, concurrent collectors need spare processor time, and every collector trades throughput, pause time and memory overhead against one another. Choosing and tuning a collector, which the JVM exposes with flags like `-XX:+UseZGC`, is choosing a point on that triangle.

COMMON TRAPS

Thinking garbage collection prevents memory leaks. An object reachable from a forgotten cache or listener list is live, and will never be collected.

Relying on finalisers to release files or locks, when they run late or never. Use `with` or `try/finally`.

Creating reference cycles in a reference-counted system without weak references or a cycle collector.

Storing a pointer into an old object without a write barrier in a runtime you're writing.

Tuning a collector without measuring pauses and throughput, the two things it trades against each other.

EXERCISES

EXERCISE 13.1 Why 16,088

Explain the copying collector's 8 collections and 16,088 copies. What would doubling the semispace to 12,000 objects do to both numbers, and what does that say about the trade-off between memory and collection work?

SOLUTION

Once the 2,000-element list exists, about 2,011 objects are live at any moment: the list, ten temporaries and the one temporary stored into the list head. Each collection copies exactly the live set, and 16,088 is $8 \times 2,011$. A collection happens when the space fills. The first comes after 6,000 allocations; after it, 2,011 slots are occupied, so each later one comes about 4,000 allocations after the last. That gives seven during the 32,000 allocations, and the workload asks for one more at the end, making 8. With 12,000-object semispaces, about 10,000 allocations fit

between collections, so there are three automatic collections and the final one: 4 in all, copying 8,044 objects, half as many. Collection work is roughly $live\ data \times allocations / (space - live\ data)$: more memory means less collection, without limit. That's the fundamental memory-time trade-off of garbage collection. Given enough spare memory a copying collector is very cheap, and given too little it thrashes, collecting constantly while freeing almost nothing.

EXERCISE 13.2 Promoted garbage

The generational collector copied 2,330 objects. Roughly where do they come from? Some of those promoted objects die soon after promotion. What happens to them, and what is the cost?

SOLUTION

About 2,000 are the list cells, each promoted once when a minor collection found it live. Each of the 32 minor collections also promotes whatever is live in the nursery at that moment, essentially the ten temporaries in `roots`, which accounts for about 300 more, and a few dozen are the objects referenced through the remembered set. But the temporaries are only live because they were caught at the moment of collection: each one is replaced moments later and dies in old space. That's **premature promotion**. The dead objects sit in old space until a major collection, which the example never runs, so in a long-running program they accumulate until old space fills and a full, expensive collection is needed. Real collectors reduce it by requiring an object to survive several minor collections, aging it in survivor spaces before promotion, as the JVM does, so that only objects that are genuinely long-lived reach old space.

EXERCISE 13.3 A cycle in CPython

In CPython, two objects `a` and `b` refer to each other, and the program then deletes its last references to both. When is their memory reclaimed, and how would you verify it? How could the program avoid creating the cycle at all?

SOLUTION

Not when the references are deleted: each object still holds a reference to the other, so both counts stay at 1 and reference counting never frees them. They're reclaimed the next time the cycle collector examines the generation containing them, which happens automatically after enough allocations, or immediately on `gc.collect()`. To verify, keep a

`weakref.ref` to one of them: it returns the object while it lives and `None` once it's freed, so checking it before and after `gc.collect()` shows the cycle being collected. `gc.collect()` also returns the number of unreachable objects it found. To avoid the cycle, make one direction of the link weak: a child that refers back to its parent through `weakref.ref(parent)` doesn't keep the parent alive, so dropping the parent frees both by reference counting alone. That's the standard design for trees with parent pointers, observers and caches.

CHECKLIST

- ☐ Can you define the roots and the live set, and explain why reachability is a safe approximation?
- ☐ Can you compare reference counting and tracing, including what each can't handle?
- ☐ Can you implement Cheney's copying collector, with forwarding pointers?
- ☐ Can you explain the generational hypothesis, minor collections and promotion?
- ☐ Can you explain what a write barrier records and what goes wrong without one?

GO DEEPER

BOOK [Crafting Interpreters, chapter 26: Garbage Collection](#) Robert Nystrom

ARTICLE [A Guide to the Go Garbage Collector](#) The Go team

BOOK [The Garbage Collection Handbook](#) Jones, Hosking & Moss

Tail calls and continuations

3 exercises

*When a function's last act is to call another function, its own frame has nothing left to do, and an implementation can reuse it for the callee. With that one change, recursion that walks a list or runs a state machine uses constant stack space, and a loop can be written as a function calling itself. Behind the change is a broader idea: a **continuation**, "the rest of the computation", which every call implicitly carries and which generators, async functions and exceptions all manipulate. This chapter eliminates tail calls in an interpreter, builds a trampoline for a language that lacks them, and shows a generator as the state machine a compiler makes of it.*

Tail position

A call is in **tail position** if its result becomes the caller's result with nothing else done to it. In `return total(n - 1, acc + n)` the recursive call is a tail call. In `return n + total(n - 1)` it isn't, because the caller still has to add `n` when the call returns, so the caller's frame, which holds `n`, must survive. Tail position is a syntactic property that a compiler can determine: the body of a function is in tail position, both branches of an `if` in tail position are, the last expression of a `begin` in tail position is, and the operands of an operator never are.

A frame that makes a tail call is dead the moment the call begins. **Tail-call elimination** discards it before the call instead of after, so the callee's frame takes its place. A chain of tail calls, however long, then uses one frame. Scheme's standard requires this, under the name proper tail recursion, so in Scheme a loop is ordinarily written as a tail-recursive function. Most functional languages guarantee it too. C compilers often do it as an optimisation without guaranteeing it, and Clang's `__musttail` attribute makes it a requirement for one call, which interpreters written in C use to build fast dispatch loops.

EXAMPLE

tailcalls.py · a tail call in Python, a trampoline, tail-call elimination in an interpreter, CPS, and a generator as a state machine

```
import sys
sys.setrecursionlimit(1000)

# 1. A tail call is a call that's the last thing a function does. Python still pushes a frame
# for it.
def total(n, acc=0):
    return acc if n == 0 else total(n - 1, acc + n)      # a tail call
for n in (500, 5000):
    try: print(f"total({n}) = {total(n)}")
    except RecursionError: print(f"total({n}): RecursionError, {n} frames for a loop in
                               disguise")
```

```

# 2. A trampoline: return "what to call next" instead of calling it, and let a loop make the
calls
def total_t(n, acc=0):
    return acc if n == 0 else (lambda: total_t(n - 1, acc + n))
def trampoline(result):
    bounces = 0
    while callable(result): result = result(); bounces += 1
    return result, bounces
value, bounces = trampoline(total_t(100_000))
print(f"trampoline total(100000) = {value:}, {bounces:} bounces, with a stack one frame
deep")

# 3. Tail-call elimination in an interpreter: evaluate a call in tail position by looping, not
recursing
depth = max_depth = 0
def evaluate(x, env, tco):
    global depth, max_depth
    depth += 1; max_depth = max(max_depth, depth)
    try:
        while True:
            if isinstance(x, int): return x
            if isinstance(x, str): return env[x]
            if x[0] == "if":
                x = x[2] if evaluate(x[1], env, tco) else x[3]      # the chosen branch is in
                           tail position
                continue
            if x[0] in ("+", "-", "="):
                a, b = evaluate(x[1], env, tco), evaluate(x[2], env, tco)
                return a + b if x[0] == "+" else a - b if x[0] == "-" else a == b
            params, body = env[x[0]]                                # a call to a user function
            args = [evaluate(a, env, tco) for a in x[1:]]
            new_env = {**env, **dict(zip(params, args))}
            if tco: x, env = body, new_env; continue                # reuse this Python frame
            return evaluate(body, new_env, tco)
        finally: depth -= 1
    body = ["if", ["=", "n", 0], "acc", ["loop", ["-", "n", 1], ["+", "acc", "n"]]]      # (loop n
acc)
    loop = {"loop": (["n", "acc"], body)}
    for tco in (False, True):
        max_depth = 0
        try:
            r = evaluate(["loop", 5000, 0], dict(loop), tco)
            print(f"interpreter with tail calls: (loop 5000) = {r:}, {max_depth} nested evaluate()
calls")
        except RecursionError:
            print("interpreter without tail calls: (loop 5000) overflowed Python's stack")

# 4. Continuation-passing style: every call gets a function, k, that says what to do with its
result
def fact_cps(n, k):
    return k(1) if n == 0 else fact_cps(n - 1, lambda v: k(n * v))
print(f"\nfactorial(10) in CPS = {fact_cps(10, lambda v: v)}")
def to_cps(x, k):
    """Rewrite (f (g a) (h b)) so every intermediate result is named and passed on
explicitly."""
    counter = [0]
    def walk(x, k):
        if not isinstance(x, list): return k(x)
        def with_args(args, done):
            if not args: return done([])
            return walk(args[0], lambda a: with_args(args[1:], lambda rest: done([a] + rest)))
        def call(parts):
            counter[0] += 1; v = f"v{counter[0]}"
            return f"({' '.join(parts)} (lambda ({v}) {k(v)}))"
        return with_args(x, call)

```

```

    return walk(x, k)
print("(+ (f x) (g y)) in CPS: " + to_cps(["+", ["f", "x"], ["g", "y"]], lambda v: f"(k {v})"))

# 5. A generator is a function whose continuation is saved at each yield. Written out by hand:
def countdown(n):
    while n > 0:
        yield n
        n -= 1
    yield "liftoff"
class Countdown:                                # the state machine a compiler makes
    of it
    def __init__(self, n): self.n, self.state = n, 0
    def __next__(self):
        if self.state == 0:
            if self.n > 0: self.state = 1; return self.n
            self.state = 2; return "liftoff"
        if self.state == 1:
            self.n -= 1; self.state = 0; return self.__next__()
        raise StopIteration
    def __iter__(self): return self
print(f"\ngenerator:      {list(countdown(3))}")
print(f"state machine: {list(Countdown(3))}")

```

OUTPUT

```

total(500) = 125250
total(5000): RecursionError, 5000 frames for a loop in disguise
trampoline total(100000) = 5,000,050,000 in 100,000 bounces, with a stack one frame deep
interpreter without tail calls: (loop 5000) overflowed Python's stack
interpreter with tail calls: (loop 5000) = 12,502,500, 3 nested evaluate() calls

factorial(10) in CPS = 3628800
(+ (f x) (g y)) in CPS: (f x (lambda (v1) (g y (lambda (v2) (+ v1 v2 (lambda (v3) (k v3)))))))

generator:      [3, 2, 1, 'liftoff']
state machine: [3, 2, 1, 'liftoff']

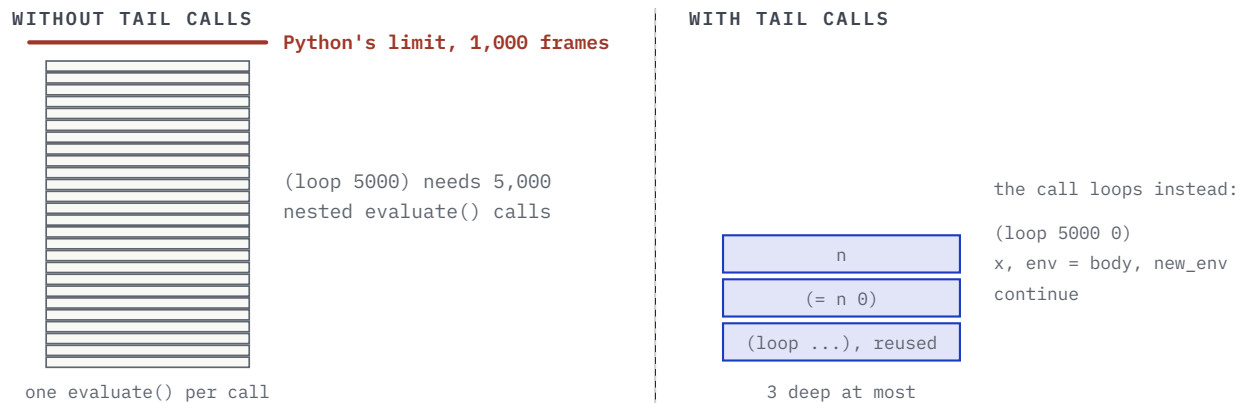
```

Python doesn't eliminate tail calls, so `total(5000)`, a loop in disguise, exhausted the stack. Guido van Rossum explained the decision in 2009: eliminated frames vanish from tracebacks, which makes debugging harder, and he considered recursion a poor substitute for loops in Python anyway. The example's **trampoline** shows how to get the effect without the language's help. Instead of making the tail call, the function returns a small function that would make it, and a loop outside calls whatever it's given until it gets a value that isn't callable. Each "call" returns to the loop first, so the stack stays one frame deep. Clojure's **trampoline** function uses the technique for mutual recursion, and so do many compilers that target JavaScript, which doesn't reliably eliminate tail calls either.

Eliminating tail calls in an interpreter

In a tree-walker, eliminating tail calls is a small change to `evaluate`. Wrap the body in a loop. When evaluating an `if`, instead of recursively evaluating the chosen branch and returning its value, replace the current expression with that branch and go round the loop again. When evaluating a call to a user function, after evaluating the arguments, replace the expression with the function's body and the environment with the new scope, and loop. The interpreter's Python frame is reused for the callee, because the caller had nothing left

to do with it. In the example, the version without this overflowed Python's stack on 5,000 iterations, and the version with it needed at most 3 nested `evaluate` calls, for evaluating arguments and conditions, which aren't in tail position.



The same program, two interpreters. Without tail calls every iteration adds Python frames until the host's limit is reached. With them, the frame for the call is reused and the stack never grows beyond what evaluating one iteration needs.

A bytecode machine does the same with a `TAIL_CALL` instruction that the compiler emits for calls in tail position. It moves the new arguments down over the current frame's slots and jumps to the callee's code without pushing a new frame. Lua does exactly this, which is why a Lua function can tail-call itself forever.

Continuations

At every point in a program's execution there is a **continuation**: what will be done with the value currently being computed. While evaluating `(g y)` in `(+ (f x) (g y))`, the continuation is "add this to the result of `(f x)`, then do whatever the caller does with the sum". An ordinary implementation represents continuations implicitly, as the stack of frames waiting for return values. **Continuation-passing style**, CPS, makes them explicit: every function takes an extra argument `k`, a function to call with its result, and never returns normally. The example's `fact_cps` computes factorial this way, and its `to_cps` rewrites an expression so that every intermediate result is named and passed to the next step.

CPS looks perverse, but it's a useful compiler representation. Every call in a CPS program is a tail call, every intermediate value has a name, and the order of evaluation is fixed by the program text, which simplifies optimisation, and some compilers for Scheme and ML have used CPS as their intermediate representation. And with continuations as values, control flow becomes programmable. Scheme's `call/cc` hands the current continuation to a function, which can call it later to return to that point, once or many times. Exceptions, early exits, backtracking search and coroutines can all be built from it.

Generators and async functions

Mainstream languages expose restricted continuations, and the two most common are generators and async functions. A generator's `yield` suspends the function and saves its continuation, the instruction pointer and local variables, so that the next call to `next` resumes from the same point. CPython does this by keeping the generator's frame on the heap instead of the stack. Languages without heap frames compile the function into a **state machine** instead, an object whose fields hold the locals and the current state, and a `next` method that switches on the state. The example's `Countdown` class is that translation of the generator above it, written by hand, and produces the same values.

Async functions are the same transformation applied to `await`. C# compiles each `async` method into a state machine class, and Rust compiles each `async fn` into an anonymous type implementing `Future`, whose `poll` method advances the machine to its next `await`. In both, the locals that are alive across an `await` become fields of the object, which is the closure-conversion question of chapter 10 again. JavaScript engines and Python implement `await` by suspending a heap-allocated frame, as with generators. Either way, an `await` is a point where the function's continuation is packaged up and handed to an event loop, to be resumed when the awaited operation completes.

The most general modern form is **effect handlers**, added to OCaml in version 5: a function performs an effect, and the nearest handler receives the effect together with the continuation, which it can resume, store or discard. Generators, async code, exceptions and backtracking all become library code instead of separate language features.

COMMON TRAPS

Believing a call is a tail call when something remains after it, such as an addition, a `try` block or a destructor.

Writing deep recursion in Python, Java or JavaScript, none of which reliably eliminate tail calls.

Losing debugging information in an implementation that eliminates tail calls, without any way to see the frames that were reused.

Holding large objects in locals across a `yield` or `await`, which keeps them alive as long as the suspended function.

Calling a continuation in CPS code and then doing more work, which breaks the rule that every call is a tail call.

EXERCISES

EXERCISE 14.1 Which calls are tail calls?

In `(define (f n) (if (= n 0) (g n) (if (h n) (f (- n 1)) (+ 1 (f (- n 2))))))`, mark each call as a tail call or not: `(= n 0)`, `(g n)`, `(h n)`, `(f (- n 1))`, `(- n 1)`, `(f (- n 2))`.

SOLUTION

The body is an `if` in tail position, so its two branches are in tail position and its condition isn't. `(= n 0)` is a condition: not a tail call. `(g n)` is the first branch: a tail call. The second branch is another `if`, also in tail position, so `(h n)`, its condition, is not a tail call, and `(f (- n 1))`, its first branch, is. `(- n 1)` is an argument, evaluated before the call, so it isn't. `(f (- n 2))` is an operand of `+`, which still has work to do with the result, so it isn't a tail call, and this branch will grow the stack. The rule to remember is that only the branches of a tail-position conditional, and the final expression of a tail-position sequence, inherit tail position.

EXERCISE 14.2 Making recursion tail-recursive

Rewrite `def length(xs): return 0 if not xs else 1 + length(xs[1:])` so that the recursive call is in tail position, then make it run in constant stack space in Python with the example's trampoline.

SOLUTION

The pending work is the `1 +`. Move it into an extra parameter, an **accumulator**, so that it's done before the call instead of after:

```
def length(xs, acc=0):
    return acc if not xs else length(xs[1:], acc + 1)

def length_t(xs, acc=0):
    return acc if not xs else (lambda: length_t(xs[1:], acc + 1))
value, bounces = trampoline(length_t(list(range(100_000))))
```

The first version is tail-recursive but still overflows in Python, because Python doesn't reuse the frame. The trampolined version returns a thunk for the next step instead of calling it, and the trampoline's loop makes the calls one at a time. Accumulator-passing works whenever the pending operation is associative, like addition. For a tree recursion like `fib`, two pending calls, the general method is CPS, which turns the pending work into a continuation function passed along instead of an accumulator. (The slicing `xs[1:]` also copies the list each time, so this `length` takes quadratic time; an index parameter would avoid that.)

EXERCISE 14.3 Two yields in a loop

Write the state machine for this generator, in the style of the example's `Countdown`. Which locals become fields, and how many states are there?

```
def pairs(n):
    for i in range(n):
        yield ("start", i)
        yield ("end", i)
```

SOLUTION

`n` and `i` are alive across the yields, so both become fields. Three states suffice: 0, about to test the loop and yield the start; 1, after the first yield, about to yield the end; and 2, finished.

```
class Pairs:
    def __init__(self, n): self.n, self.i, self.state = n, 0, 0
    def __iter__(self): return self
    def __next__(self):
        if self.state == 0 and self.i < self.n:
            self.state = 1; return ("start", self.i)
        if self.state == 1:
            self.state = 0; self.i += 1; return ("end", self.i - 1)
        self.state = 2; raise StopIteration
```

Each `yield` becomes a `return` that first records which state to resume in, and the code between two yields becomes the branch for that state. The loop has disappeared into the transitions: state 1 increments `i` and goes back to state 0, which tests the loop condition. C#'s compiler and Rust's `async` lowering produce code of exactly this shape, with one state per suspension point and one field per variable that's alive across one.

CHECKLIST

- ☐ Can you identify tail positions in a function and explain why a tail call needs no new frame?
- ☐ Can you add tail-call elimination to a tree-walking interpreter?
- ☐ Can you write a trampoline and convert a recursive function to use it?
- ☐ Can you explain continuations and continuation-passing style, with an example?

- ☐ Can you translate a generator into a state machine, and explain how `async` functions are compiled?

GO DEEPER

ARTICLE Tail Recursion Elimination Guido van Rossum

ARTICLE By example: Continuation-passing style in JavaScript Matt Might

BOOK Asynchronous Programming in Rust Rust async working group

PART IV

Compiling for speed

Four chapters on the parts of a compiler that make programs fast. A syntax tree is the wrong shape for optimisation, so compilers lower it into an intermediate representation of simple instructions in basic blocks, usually in static single assignment form, where every value has exactly one definition. On that representation, passes such as constant folding, common subexpression elimination and dead-code elimination rewrite the program to do less work. Then the code generator maps values onto the processor's few registers. The last chapter does all of this while the program runs, as a JIT compiler.

Intermediate representations and SSA

3 exercises

*An optimiser needs to ask questions like "where was this value computed?" and "is this computation used anywhere?", and a syntax tree makes them hard to answer. So compilers lower the tree into an **intermediate representation**: instructions that each do one operation on named values, grouped into basic blocks connected by a control-flow graph. The form most compilers then use is **static single assignment**, in which every name is assigned exactly once. This chapter lowers a loop to three-address code, builds its control-flow graph, computes dominators, and converts it to SSA with phi-functions placed exactly where they're needed.*

Three-address code

A syntax tree nests: `s + i * 2` is an addition whose right child is a multiplication. **Three-address code** flattens it into instructions with at most one operator, each writing a named result: `t1 = i * 2`, `t2 = s + t1`. Every intermediate value now has a name, so a pass can refer to it, record facts about it and replace it. Control flow becomes labels and jumps, as in chapter 11's bytecode, but now the representation is designed to be analysed rather than executed. LLVM IR, GCC's GIMPLE and the Java HotSpot and V8 compilers' internal graphs are all variations on this idea.

The instructions are then grouped into **basic blocks**: maximal runs of straight-line code that are entered only at the top and left only at the bottom. Inside a block, if the first instruction runs, they all run, in order. The blocks are the nodes of the **control-flow graph**, the CFG, whose edges show which block can follow which. Most analyses work block by block, propagating facts along the edges, and a loop shows up in the CFG as an edge back to an earlier block.

EXAMPLE

ssa.py · lowering a loop to three-address code, cutting it into basic blocks, computing dominators, and converting to SSA

```
import re

def read(src):
    tokens = re.findall(r"\(|\)|\s|"+, src)
    def form():
        t = tokens.pop(0)
        if t != "(": return int(t) if t.lstrip("-").isdigit() else t
        out = []
        while tokens[0] != ")": out.append(form())
        tokens.pop(0); return out
    return form()

# fn(n): s = 0; i = 0; while i < n: (if i % 2 == 0: s = s + i else: s = s - 1); i = i + 1;
```

```

    return s
body = read("""(begin (set s 0) (set i 0)
  (while (< i n) (begin (if (= (% i 2) 0) (set s (+ s i)) (set s (- s 1))) (set i (+ i 1))))
  (return s))""")

# 1. Lower the tree to three-address code: each instruction does one operation on named values
code, temps, labels = [], [0], [0]
def temp(): temps[0] += 1; return f"t{temps[0]}"
def label(): labels[0] += 1; return f"L{labels[0]}"
def expr(x):
    if not isinstance(x, list): return x
    a, b, t = expr(x[1]), expr(x[2]), temp()
    code.append(("op", t, x[0], a, b)); return t
def stmt(x):
    if x[0] == "begin": [stmt(s) for s in x[1:]]
    elif x[0] == "set": code.append(("copy", x[1], expr(x[2])))
    elif x[0] == "return": code.append(("ret", expr(x[1])))
    elif x[0] == "if":
        then, other, end = label(), label(), label()
        code.append(("br", expr(x[1]), then, other))
        code.append(("label", then)); stmt(x[2]); code.append(("jump", end))
        code.append(("label", other)); stmt(x[3]); code.append(("jump", end));
        code.append(("label", end))
    elif x[0] == "while":
        top, inside, out = label(), label(), label()
        code.append(("jump", top)); code.append(("label", top))
        code.append(("br", expr(x[1]), inside, out))
        code.append(("label", inside)); stmt(x[2]); code.append(("jump", top));
        code.append(("label", out))
stmt(body)

def show(ins):
    k = ins[0]
    if k == "op": return f"{ins[1]} = {ins[3]} {'==' if ins[2] == '=' else ins[2]} {ins[4]}"
    if k == "copy": return f"{ins[1]} = {ins[2]}"
    if k == "phi": return f"{ins[1]} = phi({'', '.join(f'{v} from {b}' for b, v in ins[2])}"
    if k == "br": return f"if {ins[1]} goto {ins[2]} else {ins[3]}"
    return {"jump": f"goto {ins[1]} if len(ins) > 1 else ''", "ret": f"return {ins[1]}"}[k]

# 2. Cut it into basic blocks: straight-line runs entered only at the top and left only at the bottom
blocks, order, current = {}, [], "entry"
blocks[current] = []; order.append(current)
for ins in code:
    if ins[0] == "label":
        current = ins[1]; blocks[current] = []; order.append(current); continue
    blocks[current].append(ins)
succ = {b: [] for b in order}
for i, b in enumerate(order):
    last = blocks[b][-1] if blocks[b] else None
    if last and last[0] == "br": succ[b] = [last[2], last[3]]
    elif last and last[0] == "jump": succ[b] = [last[1]]
    elif last and last[0] == "ret": succ[b] = []
    else: succ[b] = [order[i + 1]]
pred = {b: [p for p in order if b in succ[p]] for b in order}
print(f"{len(code)} three-address instructions in {len(order)} basic blocks:")
for b in order:
    print(f"    {b} (then {'', '.join(succ[b]) or 'exit'})")
    for i in blocks[b]: print(f"        {show(i)}")

# 3. Dominators (every path to B passes through D), then dominance frontiers
dom = {b: set(order) for b in order}; dom["entry"] = {"entry"}
changed = True
while changed:
    changed = False
    for b in order[1:]:

```

```

        new = {b} | set.intersection(*(dom[p] for p in pred[b])) if pred[b] else {b}
        if new != dom[b]: dom[b], changed = new, True
    idom = {b: max(dom[b] - {b}, key=lambda d: len(dom[d])) for b in order if b != "entry"}
    frontier = {b: set() for b in order}
    for b in order:
        if len(pred[b]) >= 2:
            for p in pred[b]:
                runner = p
                while runner != idom[b]: frontier[runner].add(b); runner = idom[runner]

# 4. Put phi-functions where a variable's definitions meet, then rename every definition
uniquely
variables = {i[1] for b in order for i in blocks[b] if i[0] == "copy"}
for v in sorted(variables):
    sites = {b for b in order for i in blocks[b] if i[0] == "copy" and i[1] == v}
    work, placed = list(sites), set()
    while work:
        for f in frontier[work.pop()]:
            if f not in placed:
                placed.add(f); blocks[f].insert(0, ["phi", v, []]); work.append(f)
count = {v: 0 for v in variables | {"n"}}; stacks = {v: [v] for v in variables | {"n"}}
def rename(b):
    pushed = []
    def use(v): return stacks[v][-1] if v in stacks else v
    for ins in blocks[b]:
        if ins[0] == "op": ins[3:] = [use(ins[3]), use(ins[4])]
        elif ins[0] in ("copy",): ins[2] = use(ins[2])
        elif ins[0] in ("br", "ret"): ins[1] = use(ins[1])
        if ins[0] in ("copy", "phi") and ins[1] in variables or (ins[0] == "phi"):
            base = ins[1].rstrip("0123456789") if ins[0] == "phi" else ins[1]
            count[base] += 1; new = f"{base}{count[base]}"; stacks[base].append(new);
            pushed.append(base)
        ins[1] = new
    for s in succ[b]:
        for ins in blocks[s]:
            if ins[0] == "phi": ins[2].append((b, stacks[ins[1].rstrip("0123456789")][-1]))
    for child in [c for c in order if idom.get(c) == b]: rename(child)
    for base in pushed: stacks[base].pop()
blocks = {b: [list(i) for i in blocks[b]] for b in order}
rename("entry")
print("\nin SSA form, every name is assigned exactly once:")
for b in order:
    print(f"    {b}")
    for i in blocks[b]: print(f"        {show(i)}")

```

OUTPUT

```

24 three-address instructions in 7 basic blocks:
entry (then L1)
    s = 0
    i = 0
    goto L1
L1 (then L2, L3)
    t1 = i < n
    if t1 goto L2 else L3
L2 (then L4, L5)
    t2 = i % 2
    t3 = t2 == 0
    if t3 goto L4 else L5
L4 (then L6)
    t4 = s + i
    s = t4
    goto L6
L5 (then L6)
    t5 = s - 1
    s = t5

```

```

    goto L6
L6 (then L1)
    t6 = i + 1
    i = t6
    goto L1
L3 (then exit)
    return s

```

in SSA form, every name is assigned exactly once:

```

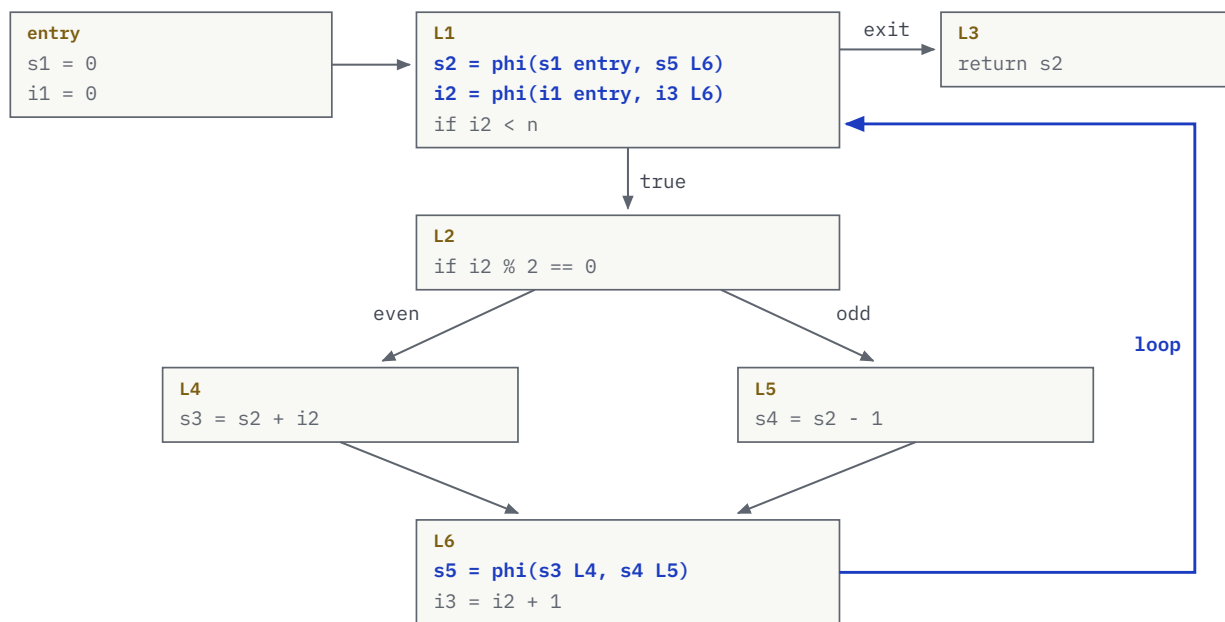
entry
    s1 = 0
    i1 = 0
    goto L1
L1
    s2 = phi(s1 from entry, s5 from L6)
    i2 = phi(i1 from entry, i3 from L6)
    t1 = i2 < n
    if t1 goto L2 else L3
L2
    t2 = i2 % 2
    t3 = t2 == 0
    if t3 goto L4 else L5
L4
    t4 = s2 + i2
    s3 = t4
    goto L6
L5
    t5 = s2 - 1
    s4 = t5
    goto L6
L6
    s5 = phi(s3 from L4, s4 from L5)
    t6 = i2 + 1
    i3 = t6
    goto L1
L3
    return s2

```

Static single assignment

In the first listing, `s` is assigned in three places: `entry`, `L4` and `L5`. To know which value `s + i` in `L4` reads, a pass must work out which assignments can reach that point, which depends on the paths through the loop. **SSA form** removes the question. Every assignment gets a new name, `s1`, `s3`, `s4`, so every use refers to exactly one definition, and the answer to "where was this computed?" is written in the name.

Where control flow merges, a variable may arrive with different definitions on different edges. At the top of the loop, `s` is `s1` coming from `entry` and `s5` coming back from `L6`. SSA resolves this with a **phi-function**, `s2 = phi(s1 from entry, s5 from L6)`, which means "the value from whichever predecessor we arrived from". A phi isn't an instruction a processor executes. It's a statement of how values flow, and it's removed again before code generation. The second listing is the program in SSA form: eight names for what were two variables, three phi-functions, and every name assigned once.



The loop in SSA form. Phi-functions, highlighted, sit at the two places where different definitions of a variable meet: the loop header, where the initial value meets the value from the previous iteration, and the join after the `if`.

Where phi-functions go

Putting a phi for every variable at every join would be correct and wasteful. Cytron, Ferrante, Rosen, Wegman and Zadeck showed in 1991 how to place exactly the necessary ones, using **dominance**. Block *D* dominates block *B* if every path from the entry to *B* passes through *D*. A definition in *D* is then certainly the only one of its variable that reaches *B*, unless another definition intervenes. The **dominance frontier** of *D* is the set of blocks where *D*'s dominance ends: blocks that *D* doesn't strictly dominate but that have a predecessor *D* does dominate. Those are exactly the places where a definition in *D* meets definitions from paths that avoid *D*, so a phi is needed in each block of the frontier of every block that assigns the variable. A phi is itself a new definition, so the process repeats until no more are added.

The example computes dominators by iterating to a fixed point, intersecting the dominator sets of each block's predecessors, then finds frontiers by walking up from each predecessor of a join to its immediate dominator. `L4` and `L5` both assign `s`, and both have `L6` in their frontier, so `L6` gets a phi for `s`. That phi is a definition in `L6`, whose frontier contains `L1`, so `L1` gets one too. `i` is assigned only in `entry` and `L6`, so its only phi is in `L1`: no phi for `i` is needed at `L6`, because only one definition of `i` reaches it. The renaming step then walks the **dominator tree**, keeping a stack of current names for each variable, so that each use gets the name of the nearest dominating definition.

Why SSA helps

SSA makes many analyses almost trivial. Each name has one definition, so the chains linking definitions to uses are just the uses of the name. A value is a constant if its single definition is a constant. An instruction is dead if its name has no uses. Two instructions compute the same value if they apply the same operator to the same names, because names never change meaning. Chapter 16's passes all rely on this. Optimisations that, on ordinary variables, need data-flow analyses over the whole CFG become a single pass over the instructions.

Nearly every modern optimising compiler uses SSA. LLVM IR is in SSA form from the start, with one trick: Clang first emits every local variable as a stack slot with loads and stores, and LLVM's `mem2reg` pass promotes those slots to SSA values with phi-functions, using the algorithm above, so the front end never has to place phis itself. GCC converts GIMPLE to SSA. HotSpot's C2 compiler uses a graph form called the **sea of nodes**, in which instructions float free of blocks, ordered only by their data and control dependences. V8 used the same design for years, and in 2025 published its reasons for moving back to a conventional CFG: the free-floating graph was hard to debug and made some optimisations slower to compile, which is a reminder that IR design is an engineering trade-off.

Before code generation the phis must go. Each phi becomes copies on its incoming edges: `s2 = phi(s1, s5)` becomes `s2 = s1` at the end of `entry` and `s2 = s5` at the end of `L6`. Doing this correctly has two well-known pitfalls, both covered in exercise 15.2, and the register allocator of chapter 17 then tries to give the copied names the same register, so that most copies disappear.

COMMON TRAPS

Optimising the syntax tree instead of a lower representation, and finding every pass needs its own tree walk and special cases.

Placing phi-functions at every join, which is correct but floods the program with dead phis.

Treating phis as sequential instructions. All the phis at the top of a block take effect at once, on entry.

Removing phis with copies that overwrite a value still needed, the "swap" problem.

Forgetting that SSA names are values, not variables: `s2` and `s5` may need the same register later, or different ones.

EXERCISES

EXERCISE 15.1 Dominators by hand

List the dominators of each block in the example's CFG, and the immediate dominator of each. Which block dominates L6, and why doesn't L4?

SOLUTION

entry dominates every block, since every path starts there. L1, the loop header, is reached from entry and from L6, and every path to any later block passes through it, so it dominates L2, L3, L4, L5 and L6. L2 dominates L4, L5 and L6. L4 and L5 dominate only themselves. So the immediate dominators are: L1 for L2 and L3, entry for L1, and L2 for L4, L5 and L6. L4 doesn't dominate L6 because there's a path to L6 through L5 that avoids it; that path is exactly why L6 is in L4's dominance frontier and needs a phi for s. The dominator tree is therefore entry, then L1, with children L2 and L3, and L2's children L4, L5 and L6, which is the order the renaming visits them.

EXERCISE 15.2 Leaving SSA

A loop header has $a2 = \text{phi}(a1, b2)$ and $b2 = \text{phi}(b1, a2)$: each iteration swaps a and b. Replace the phis with copies at the end of the back-edge predecessor, in the obvious order, and show what goes wrong. How do compilers avoid it?

SOLUTION

The obvious translation puts $a2 = b2$ and then $b2 = a2$ at the end of the loop body. The second copy reads a2 after the first has already overwritten it, so both end up holding the old b2, and the swap is lost. That's the **swap problem**, and it happens because the phis at the top of a block take their values simultaneously, while copies run one at a time. Compilers treat the copies for one edge as a **parallel copy** and sequentialise it carefully: copy any value that's about to be overwritten into a temporary first, so here $t = a2$; $a2 = b2$; $b2 = t$. The other pitfall is the **lost-copy problem**: on a **critical edge**, one from a block with several successors to a block with several predecessors, there's no place to put the copies without affecting other paths. The fix is to split such edges by inserting an empty block on them before leaving SSA.

EXERCISE 15.3 Dead code in SSA

Suppose the loop body also computed `t7 = s2 * 3` and never used `t7`. Describe a dead-code elimination pass for SSA that removes it in one sweep, and explain why the same pass on the original, non-SSA code would need more care.

SOLUTION

In SSA, count the uses of every name. Put every instruction whose result has zero uses and no side effects on a worklist. Remove each one, and decrement the use counts of the names it read; any that fall to zero join the worklist. When the worklist is empty, everything left is used, directly or indirectly, by something with an effect. `t7` has no uses, so it's removed at once, and if it had been the only use of some other temporary, that would go next. On the original code, a variable's name can be reused: `s = s + i` both reads and writes `s`, so "no use of the name after this point" depends on the path, and a pass must compute **liveness**, which names may be read before being overwritten, by data-flow analysis over the whole CFG before it can decide whether an assignment is dead. SSA turned that analysis into counting.

CHECKLIST

- ☐ Can you lower expressions and control flow to three-address code and cut it into basic blocks?
- ☐ Can you draw a control-flow graph and compute its dominators and dominance frontiers?
- ☐ Can you explain what a phi-function means and where it must be placed?
- ☐ Can you convert a small function to SSA by hand, and back out again?
- ☐ Can you explain why SSA makes constant propagation and dead-code elimination simple?

GO DEEPER

BOOK Engineering a Compiler, chapters 5, 8 and 9 Keith Cooper & Linda Torczon

BOOK SSA-based Compiler Design Rastello & Bouchez Tichadou, editors

DOCS LLVM Language Reference Manual LLVM Project

Optimisation

3 exercises

An optimiser is a sequence of small passes, each of which recognises one pattern of wasted work and removes it without changing what the program does. No single pass achieves much, but each one exposes opportunities for the others, so compilers run them repeatedly. This chapter writes six classic passes over three-address code, runs them until nothing changes, and watches ten instructions collapse to two. It then turns to loops, where nearly all of a program's time goes, and to the rule that every optimisation must obey.

The rule

An optimisation may change anything about a program except its **observable behaviour**: its output, its effects on files and the network, the values it returns, whether it terminates. It may delete computations whose results are never observed, reorder independent ones, and replace one computation with a cheaper one that gives the same result. Everything in this chapter is a consequence of that rule, and most optimiser bugs are a case where a pass believed two computations were equivalent and they weren't, usually because of floating point, overflow, aliasing or a side effect the pass didn't see.

Six local passes

The example's block computes a result through ten instructions, with plenty of redundancy built in. Six passes attack it.

- **Constant propagation and folding.** If `a = 4`, uses of `a` can use 4, and an operation whose operands are all constants can be computed now: `b = a * 2` becomes `b = 8`.
- **Algebraic simplification.** Identities remove operations: `x * 1` is `x`, `x + 0` is `x`, `x * 0` and `x - x` are 0, for integers.
- **Strength reduction.** A cheap operation replaces an expensive one with the same result: `h * 2` becomes `h + h`, and on real processors a multiplication by a power of two becomes a shift.
- **Copy propagation.** After `d = c`, uses of `d` can use `c` directly, which makes `d`'s definition dead.
- **Common subexpression elimination.** `b + x` is computed twice with nothing in between changing `b` or `x`, so the second becomes a copy of the first. This version treats `+` and `*` as commutative, so `x + b` would match too. The general technique is **value numbering**: give each distinct computation a number and reuse it.

- **Dead-code elimination.** Working backwards from the result, keep only instructions whose values are needed. `unused = x * x` goes, along with everything the other passes left unused.

EXAMPLE

optimize.py · six optimisation passes run to a fixed point on a block, then loop-invariant code motion and strength reduction counted

```
# Optimisation passes over three-address code: (destination, operator, operand, operand).
block = [("a", "const", 4, None), ("b", "*", "a", 2), ("c", "+", "b", "x"), ("d", "+", "b",
    "x"),
    ("e", "*", "d", 1), ("f", "-", "e", "c"), ("g", "*", "f", "y"), ("h", "+", "c", "g"),
    ("unused", "*", "x", "x"), ("r", "*", "h", 2)]
RESULT = "r" # the block returns r

def run(code, x, y):
    env = {"x": x, "y": y}
    val = lambda v: env[v] if isinstance(v, str) else v
    for dst, op, a, b in code:
        env[dst] = (val(a) if op in ("const", "copy") else val(a) + val(b) if op == "+" else
            val(a) - val(b) if op == "-" else val(a) * val(b))
    return env[RESULT]

def fold(code): # constant propagation and folding
    known, out = {}, []
    for dst, op, a, b in code:
        a, b = known.get(a, a), known.get(b, b)
        if op == "const" or (op != "copy" and isinstance(a, int) and isinstance(b, int)):
            v = a if op == "const" else a + b if op == "+" else a - b if op == "-" else a * b
            known[dst] = v; out.append((dst, "const", v, None))
        else: out.append((dst, op, a, b))
    return out

def simplify(code): # algebra and strength reduction
    out = []
    for dst, op, a, b in code:
        if op == "*" and b == 1: out.append((dst, "copy", a, None))
        elif op == "*" and 0 in (a, b): out.append((dst, "const", 0, None))
        elif op == "+" and b == 0: out.append((dst, "copy", a, None))
        elif op == "+" and a == 0: out.append((dst, "copy", b, None))
        elif op == "-" and a == b: out.append((dst, "const", 0, None))
        elif op == "*" and b == 2: out.append((dst, "+", a, a)) # a multiply becomes
            an add
        else: out.append((dst, op, a, b))
    return out

def copy_prop(code): # after d = c, use c wherever d
    appears
    alias, out = {}, []
    for dst, op, a, b in code:
        a, b = alias.get(a, a), alias.get(b, b)
        if op == "copy": alias[dst] = a
        out.append((dst, op, a, b))
    return out

def cse(code): # common subexpressions: compute b + x
    once
    seen, out = {}, []
    for dst, op, a, b in code:
        key = (op, *sorted(map(str, (a, b)))) if op in ("+", "*") else (op, a, b)
        if op not in ("const", "copy") and key in seen: out.append((dst, "copy", seen[key],
            None))
        else: seen.setdefault(key, dst); out.append((dst, op, a, b))
    return out

def dce(code): # dead code: keep only what the result
```

```

needs
live, out = {RESULT}, []
for dst, op, a, b in reversed(code):
    if dst in live:
        live.discard(dst); live |= {v for v in (a, b) if isinstance(v, str)}
        out.append((dst, op, a, b))
return out[::-1]

def show(code):
    def one(d, op, a, b):
        return f"{d} = {a}" if op in ("const", "copy") else f"{d} = {a} {op} {b}"
    return "; ".join(one(*i) for i in code)

def ops(code): return sum(op not in ("const", "copy") for _, op, _, _ in code)
print(f"original: {len(block)} instructions, {ops(block)} arithmetic operations, "
      f"result for x=5, y=7: {run(block, 5, 7)}")
code, round_ = block, 0
while True:
    anything
    round_ += 1; before = code
    for name, p in (("constant folding", fold), ("algebra, strength reduction", simplify),
                  ("copy propagation", copy_prop), ("common subexpressions", cse)):
        new = p(code)
        if new != code:
            print(f"    round {round_}, {name:28} operations {ops(code)} -> {ops(new)}, "
                  f"    result {run(new, 5, 7)}")
            code = new
    if code == before: break
code = dce(code)
print(f"    then dead-code elimination: {len(code)} instructions left, result {run(code, 5, 7)}")
print(f"    {show(code)}")

# Loop-invariant code motion and strength reduction, counted by running the loop
def run_loop(pre, body, n, a=6, b=7):
    env, muls = {"a": a, "b": b, "s": 0, "i": 0}, 0
    def step(ins):
        nonlocal muls
        dst, op, x, y = ins
        vx, vy = (env[v] if isinstance(v, str) else v for v in (x, y))
        muls += op == "*"
        env[dst] = vx + vy if op == "+" else vx * vy
    for ins in pre: step(ins)
    while env["i"] < n:
        for ins in body: step(ins)
    return env["s"], muls
naive = ([], [(("t", "*", "a", "b"), ("u", "*", "t", "i"), ("s", "+", "s", "u"), ("i", "+", "i", 1))])
hoisted = ([("t", "*", "a", "b")], [(("u", "*", "t", "i"), ("s", "+", "s", "u"), ("i", "+", "i", 1))])
reduced = ([("t", "*", "a", "b"), ("u", "+", 0, 0)],
            by adding t
            [(("s", "+", "s", "u"), ("u", "+", "u", "t"), ("i", "+", "i", 1))])
print("\na loop summing a * b * i for i below 1,000:")
for name, (pre, body) in (("as written", naive), ("a * b hoisted out of the loop", hoisted),
                          ("t * i replaced by a running sum", reduced)):
    s, muls = run_loop(pre, body, 1000)
    print(f"    {name:32} result {s:,}    {muls:,} multiplication{'s' if muls != 1 else ''}")

```

OUTPUT

```

original: 10 instructions, 9 arithmetic operations, result for x=5, y=7: 26
round 1, constant folding          operations 9 -> 8, result 26
round 1, algebra, strength reduction operations 8 -> 7, result 26
round 1, copy propagation         operations 7 -> 7, result 26
round 1, common subexpressions    operations 7 -> 6, result 26

```

```

round 2, copy propagation      operations 6 -> 6, result 26
round 3, algebra, strength reduction operations 6 -> 5, result 26
round 4, constant folding      operations 5 -> 5, result 26
round 4, algebra, strength reduction operations 5 -> 4, result 26
round 5, constant folding      operations 4 -> 4, result 26
round 5, algebra, strength reduction operations 4 -> 3, result 26
round 5, copy propagation      operations 3 -> 3, result 26
then dead-code elimination: 2 instructions left, result 26
c = 8 + x; r = c + c

```

```

a loop summing a * b * i for i below 1,000:
as written                      result 20,979,000    2,000 multiplications
a * b hoisted out of the loop    result 20,979,000    1,001 multiplications
t * i replaced by a running sum  result 20,979,000    1 multiplication

```

ORIGINAL	WHAT HAPPENED TO IT	RESULT
a = 4	propagated into b, then dead	
b = a * 2	folded to the constant 8, then dead	
c = b + x	b replaced by 8	c = 8 + x
d = b + x	same as c: common subexpression	
e = d * 1	times 1 removed, then copy of c	
f = e - c	became c - c, which is 0	
g = f * y	0 times anything is 0	
h = c + g	c + 0 is c	
unused = x * x	never used: dead code	
r = h * 2	h is c; times 2 became an add	r = c + c

10 instructions, 9 operations same result for every x, y 2 instructions, 2 operations

Where each of the ten instructions went. Only two survive, and each of the others was removed by a chain of passes, not by one: **f** needed CSE, then copy propagation, then algebra, before it became a constant 0.

Phase ordering

The trace shows why the passes run in rounds. Folding turns **b** into 8. CSE finds that **c** and **d** both compute **8 + x** and turns **d** into a copy, which copy propagation pushes into **e**, which makes **f = c - c**, which algebra turns into 0, which folding pushes into **g**, and so on. Each pass's output is the next pass's opportunity, and it took five rounds before a full round changed nothing. The example simply repeats the whole sequence until it reaches a **fixed point**. That terminates because no pass ever increases the number of operations, and a round that changes nothing changes nothing forever after.

The order still matters, because a different order can reach a different fixed point, or reach it more slowly. This is the **phase-ordering problem**, and it has no general solution. Production compilers settle on a fixed pipeline tuned by measurement: LLVM's **-O2** runs a sequence of dozens of passes with some, such as instruction combining and CSE, appearing several times. Running **opt -passes='default<O2>' -print-after-all** on a small file shows what each pass did, in the style of the example's trace.

Loops

A statement inside a loop that runs a million times matters a million times more than one outside, so the most valuable optimisations are about loops. The second half of the example counts multiplications in a loop that sums `a * b * i`.

- **Loop-invariant code motion** notices that `a * b` has the same value on every iteration and moves it before the loop. Multiplications fell from 2,000 to 1,001.
- **Strength reduction of induction variables** notices that `t * i`, where `i` goes up by one each iteration, goes up by `t` each iteration. It replaces the multiplication with a running sum that adds `t`. Multiplications fell to 1, the one computing `t`.

Other loop transformations work at a larger scale. **Unrolling** repeats the body several times per iteration to cut the overhead of the loop test and jump. **Vectorisation** turns a loop over arrays into SIMD instructions that process four, eight or sixteen elements at once. And the most important optimisation of all is not about loops but enables most of the others: **inlining**, which replaces a call with a copy of the called function's body. A small function inlined into a loop can then be folded, hoisted and vectorised together with the code around it, which a call would have hidden.

When equivalences aren't

The example works on integers, where its identities are true. Many fail for floating point. `x + 0.0` isn't always `x`, because `-0.0 + 0.0` is `+0.0`, and `x * 0.0` isn't always 0, because it's NaN for infinities and NaNs. Floating-point addition isn't associative, so `(a + b) + c` can't be reordered to `a + (b + c)` without changing results. That's why vectorising a floating-point sum needs a flag such as `-ffast-math`, which tells the compiler you accept the difference.

C and C++ give compilers the opposite kind of freedom. Signed integer overflow is undefined behaviour, so a compiler may assume it never happens: `x + 1 > x` can be folded to true, and a loop counter can be widened to 64 bits without checking for wraparound. Those assumptions make real loops faster, and they're also why a C program with an overflow bug can behave in ways that seem impossible: the optimiser reasoned correctly from a premise that the program violated.

COMMON TRAPS

Applying integer identities to floating point, where `x + 0`, `x * 0` and reassociation all change results.

Treating a call as pure and eliminating or moving it, when it has side effects.

Hoisting an operation that can fault, such as a division, out of a loop that might run zero times.

Running each pass once, missing the opportunities each pass creates for the others.

Benchmarking without the optimiser on, or with it on and a result it can prove unused, so it deletes the work being measured.

EXERCISES

EXERCISE 16.1 Value numbering

Apply local value numbering by hand to `a = x + y; b = y + x; c = a * 2; d = b * 2; e = c - d`. Which instructions become copies, and what does the block reduce to after copy propagation, algebra and dead-code elimination, if only `e` is needed?

SOLUTION

Give `x` and `y` value numbers 1 and 2. `a = x + y` computes `+(1, 2)`, a new value, 3. `b = y + x` is `+(2, 1)`, which, because addition commutes, is normalised to `+(1, 2)`, value 3 again, so `b` becomes a copy of `a`. `c = a * 2` is `*(3, 2)`, value 4. `d = b * 2` is also `*(3, 2)`, value 4, so `d` becomes a copy of `c`. `e = c - d` is then `c - c`, which algebra turns into the constant 0. With only `e` needed, dead-code elimination removes everything else, leaving `e = 0`: five instructions with five operations became one with none. The example's `cse` pass does the same with its sorted key for commutative operators.

EXERCISE 16.2 A pass that's wrong for floats

Someone extends the example to floating-point values unchanged. Which of its rewrites become incorrect, and for which inputs? Which remain correct?

SOLUTION

`x * 0 → 0` is wrong when `x` is infinite or NaN, since the product is NaN, and gives the wrong sign when `x` is negative, since the product is `-0.0`. `x - x → 0` is wrong for infinities and NaN, which give NaN. `x + 0 → x` is wrong when `x` is `-0.0`: the sum is `+0.0`, which is observable through division or `math.copysign`. `0 + x → x` fails the same way. Correct ones remain: `x * 1 → x` is exact for every float including NaN, `x * 2 → x + x` is exact because both round the same exact result, and constant folding is correct as long as the compiler rounds exactly as the target machine would. CSE and copy propagation

are also correct, since they don't change any arithmetic. In general an optimiser needs one set of rules for each type, and IEEE floating point admits very few identities.

EXERCISE 16.3 Hoisting safely

In `for i in range(n): total += x / y`, the division is loop-invariant. Is it always safe to move it before the loop? What does a compiler do when it isn't sure?

SOLUTION

Not always. If `n` is 0 the loop body never runs, so the original program never divides, and if `y` is also 0 the hoisted division raises an error, or on a processor traps, that the original program never did. That changes observable behaviour, so it's forbidden. Hoisting is safe for operations that can't fail or have effects, like `a * b` on integers, or when the compiler can prove the loop runs at least once. Otherwise compilers either leave the operation in the loop, or transform the loop into `if n > 0:` followed by the hoisted computation and a loop that runs at least once, a transformation called **loop rotation** or loop inversion, which LLVM performs routinely precisely to make hoisting safe. JIT compilers have a third option: hoist speculatively, guarded by a check, and fall back to unoptimised code if the check fails, as chapter 18 shows.

CHECKLIST

- ☐ Can you state the rule every optimisation must obey, and give an example that breaks it?
- ☐ Can you implement constant folding, copy propagation, CSE and dead-code elimination over three-address code?
- ☐ Can you explain why passes are iterated, and what the phase-ordering problem is?
- ☐ Can you apply loop-invariant code motion and induction-variable strength reduction by hand?
- ☐ Can you explain why many integer identities are invalid for floating point?

GO DEEPER

BOOK Engineering a Compiler, chapters 8 and 10: introduction to and scalar optimisations Keith Cooper & Linda Torczon

DOCS

LLVM's Analysis and Transform Passes LLVM Project

ARTICLE

What Every C Programmer Should Know About Undefined Behavior Chris
Lattner

Code generation and registers

3 exercises

*The intermediate representation has as many values as the program needs; the processor has sixteen or thirty-two general-purpose registers. The back end must choose machine instructions for each operation, decide which value lives in which register at each moment, and, when there aren't enough registers, move some values to memory and back. That last problem, **register allocation**, is where most of a back end's effort goes, because a value in a register costs nothing to read and a value in memory costs a load. This chapter allocates registers for a block by graph colouring and by linear scan, with spilling, and runs the result on a simulated machine to check it.*

Instruction selection

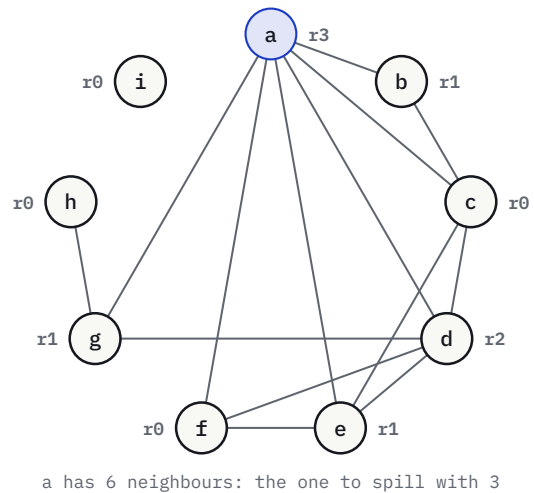
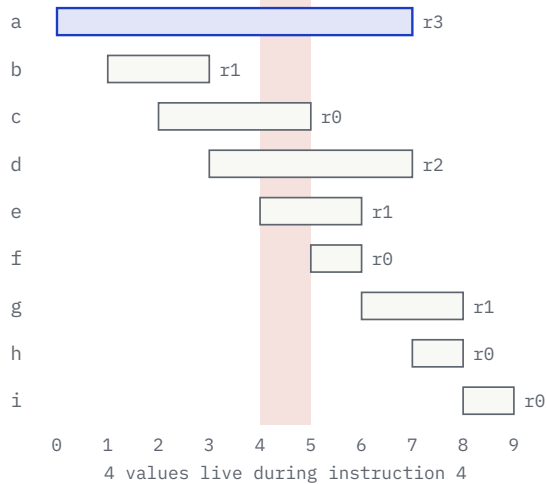
The first step maps IR operations onto the target's instructions. On a simple load-store machine like RISC-V it's nearly one to one: an IR addition becomes `add`, a load becomes `lw` or `ld`. Richer instruction sets offer instructions that do several IR operations at once, and choosing well matters. On x86-64, `leaq rax, [rdi + rsi*4 + 8]` computes a shift, two additions and writes the result in one instruction, and memory operands let `add rax, [rbx]` load and add together. Instruction selection is usually framed as **tiling**: cover the IR's expression trees or graphs with patterns, one per machine instruction, at the lowest total cost. LLVM's SelectionDAG and GlobalISel frameworks do this from pattern descriptions of each target.

The output still uses unlimited **virtual registers**, one per IR value. Mapping them onto the real ones is the next step, and the one this chapter is about.

Liveness and interference

A value is **live** from the instruction that defines it to the last instruction that uses it. Two values that are live at the same time can't share a register, because writing one would destroy the other; such values **interfere**. Values whose live ranges don't overlap can share, which is why a long function can run in a handful of registers. The maximum number of values live at any one point, the **register pressure**, is a lower bound on the registers needed. In the example's block, four values are live during instruction 4.

In a block of straight-line code, liveness is just a range of instruction numbers, as the example computes. Across a function with branches and loops, it's computed by a backwards data-flow analysis on the control-flow graph: a value is live at the end of a block if it's live at the start of any successor, and live at the start of a block if it's used in the block before being redefined, or live at the end and not redefined. Iterating those two equations to a fixed point gives liveness for the whole function.



The same allocation as live ranges and as a graph. Values whose bars overlap are joined by an edge and need different registers; colouring the graph with four colours is the allocation the example found.

Graph colouring

Draw a node for each value and an edge between each pair that interfere, and register allocation becomes **graph colouring**: give each node one of k colours, the registers, so that no edge joins two nodes of the same colour. Gregory Chaitin's allocator for IBM's PL.8 compiler, published in 1981 and 1982, made this practical with a simple observation. A node with fewer than k neighbours can always be coloured, whatever colours its neighbours get, because at least one colour is left for it. So remove such nodes from the graph one at a time, pushing each onto a stack, which may leave other nodes with fewer than k neighbours. If the graph empties, pop the nodes back in reverse order, each taking a colour its already-coloured neighbours don't use. That's the example's `color` function.

If every remaining node has k or more neighbours, the method is stuck, and a value must be **spilled**: kept in a stack slot in memory, stored after it's defined and reloaded before each use. Each reload creates a new, short-lived value, so the spilled variable's long live range becomes several tiny ones, and the allocator starts again on the rewritten code. Colouring a general graph optimally is NP-complete, so allocators rely on heuristics for what to spill. The usual one weighs the cost of the spill code, uses inside loops counting much more, against how many edges removing the node would delete.

EXAMPLE

```
regalloc.py · liveness, an interference graph, Chaitin colouring with spilling, and linear scan, checked by running the allocated code
```

```
# Register allocation: give each value a machine register, so that two values alive at the same
    time
# never share one; when there aren't enough registers, keep some values in memory ("spill"
    them).
code = [("a", "load", "x"), ("b", "load", "y"), ("c", "add", "a", "b"), ("d", "mul", "a", "b"),
        ("e", "add", "c", "d"), ("f", "add", "c", "c"), ("g", "add", "e", "f"), ("h", "sub",
```

```

        "d", "a"),
        ("i", "mul", "g", "h"), (None, "store", "i", "out")]

def intervals(code):
    """A value is live from the instruction that defines it to the last instruction that uses
    it."""
    start, end = {}, {}
    for n, (dst, op, *args) in enumerate(code):
        for a in args:
            if a in start: end[a] = n
            if dst: start[dst] = n; end.setdefault(dst, n)
    return {v: (start[v], end[v]) for v in start}

def interfere(iv):
    vs = sorted(iv)
    return {v: {w for w in vs if w != v and iv[v][0] < iv[w][1] and iv[w][0] < iv[v][1]} for v
            in vs}

def color(graph, k):
    """Chaitin's method: remove nodes with fewer than k neighbours; if none is left, pick one
    to spill."""
    g = {v: set(n) for v, n in graph.items()}
    stack, spilled = [], []
    while g:
        easy = [v for v in sorted(g) if len(g[v]) < k]
        v = easy[0] if easy else max(sorted(g), key=lambda v: len(g[v])) # spill the most
                               connected
        (stack if easy else spilled).append(v)
        for n in g.pop(v): g[n].discard(v)
    regs = {}
    for v in reversed(stack): # put them back, each taking a free
        register
        regs[v] = min(set(range(k)) - {regs[n] for n in graph[v] if n in regs})
    return regs, spilled

def spill(code, var):
    """Keep var in a stack slot: store it after its definition, reload it before each use."""
    out, n = [], 0
    for dst, op, *args in code:
        if var in args:
            n += 1; tmp = f"{var}{n}"
            out.append((tmp, "load", f"[{var}]")); args = [tmp if a == var else a for a in
                args]
            out.append((dst, op, *args))
            if dst == var: out.append((None, "store", var, f"[{var}]"))
    return out

iv = intervals(code)
graph = interfere(iv)
pressure = max(sum(s <= p < e for s, e in iv.values()) for p in range(len(code)))
print("live ranges: " + ", ".join(f"{v} {s}-{e}" for v, (s, e) in iv.items()))
print(f"at most {pressure} values are live at once; {sum(map(len, graph.values())) // 2} pairs
interfere")

def execute(code, regs, x=5, y=7): # run it on a machine with k registers
    reg, mem = {}, {"x": x, "y": y}
    loc = lambda v: reg if v in regs else mem
    key = lambda v: regs.get(v, v)
    for dst, op, *args in code:
        if op == "store": mem[args[1].strip("[]")] = loc(args[0])[key(args[0])]; continue
        if op == "load": loc(dst)[key(dst)] = mem[args[0].strip("[]")]; continue
        a, b = (loc(v)[key(v)] for v in args)
        loc(dst)[key(dst)] = a + b if op == "add" else a - b if op == "sub" else a * b
    return mem["out"]

for k in (pressure, pressure - 1):
    current = code

```

```

while True:
    regs, spilled = color(interfere(intervals(current)), k)
    if not spilled: break
    print(f"\n{k} registers: not enough, spill {'', '.join(spilled)}")
    for v in spilled: current = spill(current, v)
extra = sum(str(args[-1]).startswith("(") for _, op, *args in current)
print(f"{k} registers: {len(current)} instructions, {extra} of them spill code; "
      f"out = {execute(current, regs)} for x=5, y=7")
r = lambda v: f"r{regs[v]}" if v in regs else v
for dst, op, *args in current:
    print(f"    {op:5} " + ", ".join(r(x) for x in ([dst] if dst else []) + list(args)))

# Linear scan: walk the live ranges in order of start; when registers run out, spill the one
# that ends last
def linear_scan(iv, k):
    active, regs, spilled, free = [], {}, [], list(range(k))
    for v, (s, e) in sorted(iv.items(), key=lambda kv: kv[1][0]):
        for w in [w for w in active if iv[w][1] <= s]: active.remove(w); free.append(regs[w])
        if free: regs[v] = free.pop(0); active.append(v)
        else:
            victim = max(active + [v], key=lambda w: iv[w][1])
            spilled.append(victim)
            if victim != v: regs[v] = regs.pop(victim); active.remove(victim); active.append(v)
    return regs, spilled
for k in (pressure, pressure - 1):
    regs, spilled = linear_scan(iv, k)
    print(f"\nlinear scan with {k} registers: spills {'', '.join(spilled) or 'nothing'}; "
          + ", ".join(f"{v}->r{regs[v]}" for v in sorted(regs)))

```

OUTPUT

live ranges: a 0-7, b 1-3, c 2-5, d 3-7, e 4-6, f 5-6, g 6-8, h 7-8, i 8-9
at most 4 values are live at once; 14 pairs interfere
4 registers: 10 instructions, 0 of them spill code; out = 2130 for x=5, y=7

```

load  r3, x
load  r1, y
add   r0, r3, r1
mul   r2, r3, r1
add   r1, r0, r2
add   r0, r0, r0
add   r1, r1, r0
sub   r0, r2, r3
mul   r0, r1, r0
store r0, out

```

3 registers: not enough, spill a

3 registers: 14 instructions, 4 of them spill code; out = 2130 for x=5, y=7

```

load  r0, x
store r0, [a]
load  r1, y
load  r0, [a]
add   r0, r0, r1
load  r2, [a]
mul   r2, r2, r1
add   r1, r0, r2
add   r0, r0, r0
add   r1, r1, r0
load  r0, [a]
sub   r0, r2, r0
mul   r0, r1, r0
store r0, out

```

linear scan with 4 registers: spills nothing; a->r0, b->r1, c->r2, d->r3, e->r1, f->r2, g->r1,
h->r2, i->r0

linear scan with 3 registers: spills a; b->r1, c->r2, d->r1, e->r0, f->r2, g->r0, h->r2, i->r1

With four registers the block needs no memory traffic at all. With three, the allocator found every node with three or more neighbours, spilled `a`, the most connected, and recoloured: four extra instructions, one store and three reloads, and the program still computes 2,130. Exercise 17.2 shows that the "most connected" heuristic didn't make the cheapest choice here. The executed check matters more than it looks: an allocator bug usually produces code that runs and gives wrong answers, so compilers test allocators by running their output, as the example does.

Linear scan

Graph colouring is slow to compute for large functions, which matters in a JIT compiler that must finish while the program waits. Poletto and Sarkar's **linear scan** allocator, from 1999, sorts live ranges by their start and walks them in order, keeping a list of the active ranges. At each new range it frees the registers of ranges that have ended. If a register is free, the range gets it; if not, it spills whichever range, the new one or an active one, ends last, since that frees a register for the longest time. It runs in nearly linear time and produces code almost as good as colouring. HotSpot's client compiler and V8's optimising compilers use variants of it, and the example's version found the same answers as colouring for both four and three registers.

Calling conventions

Registers also carry values between functions, by agreement. A **calling convention** fixes which registers hold arguments and results and which a function may overwrite. On x86-64 Linux and macOS, the System V convention passes the first six integer arguments in `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9` and returns the result in `rax`. On 64-bit ARM, arguments go in `x0` to `x7` and the result in `x0`. Registers are divided into **caller-saved** ones, which any call may destroy, so a caller must save them if it needs their values afterwards, and **callee-saved** ones, which a function must restore before returning if it uses them. The allocator therefore prefers callee-saved registers for values that are live across a call and caller-saved ones for values that aren't. Chapter 20 meets the same convention from the other side, when Python calls C.

Two more back-end jobs come after allocation. **Instruction scheduling** reorders independent instructions so that a slow load or multiplication overlaps with other work, which matters most on simpler processors that don't reorder instructions themselves. And the **prologue and epilogue** of each function set up and tear down its stack frame, saving the callee-saved registers it uses. Reading the output of `clang -O2 -S` for a small function shows all of these: arguments arriving in the convention's registers, values kept in registers throughout, and a stack frame only if something needed one.

COMMON TRAPS

Treating register pressure as the number of variables rather than the number live at once.

Spilling inside a hot loop when a value used outside it could have been spilled instead.

Forgetting that calls destroy caller-saved registers, so values live across a call are silently corrupted.

Testing an allocator by reading its output rather than running it on many inputs.

Using a value's last use as the end of its range across a loop, when the loop's back edge makes it live again.

EXERCISES

EXERCISE 17.1 Pressure and colouring

For the block `p = load x; q = load y; r = p + q; s = p * q; t = r - s; store t`, write the live ranges, the maximum register pressure, and the interference graph. How many registers are needed?

SOLUTION

Number the instructions 0 to 5. `p` is live from 0 to 3, its last use in `s = p * q`; `q` from 1 to 3; `r` from 2 to 4; `s` from 3 to 4; `t` from 4 to 5. Using the example's rule, where a value dying at an instruction may share a register with one defined there, the interfering pairs are `p - q`, `p - r`, `q - r` and `r - s`. At instruction 2, where `r` is defined, `p`, `q` and `r` are all live, so the pressure is 3, and `p`, `q`, `r` form a triangle, which needs three colours. Three registers suffice: `p` r0, `q` r1, `r` r2, then `s` can take r0, since `p` dies as `s` is defined, and `t` any register. Computing `s` before `r` wouldn't help, because `p` and `q` are both needed by both. In straight-line code, pressure is exactly the number of registers needed, because the interference graph of intervals is an **interval graph**, which can always be coloured with as many colours as its largest clique.

EXERCISE 17.2 A better spill

With three registers, the example spilled `a`, adding four instructions. Is there a single value whose spilling costs less and still makes three registers enough? Why didn't the heuristic choose it?

SOLUTION

Yes: `d`. It's defined at instruction 3 and used twice, at 4 and 7, so spilling it costs one store and two reloads, three instructions, and with `d` split into short pieces the rest colours with three registers. Calling the example's `spill` and `color` for every value in turn confirms that only `a` and `d` work alone, and `d` is cheaper. `a` has more uses, three, so it costs a reload more. The heuristic chose `a` because it looked only at the number of neighbours, six for `a` against five for `d`, and not at the cost of the spill code. Chaitin's actual heuristic divides an estimated spill cost, counting each use and weighting uses in loops by the loop depth, by the node's degree, and spills the node with the lowest ratio. That would favour `d` here. Modern allocators go further and **split** a live range, keeping a value in a register where it's used heavily and in memory elsewhere, rather than spilling all of it.

EXERCISE 17.3 A value across a call

In `def f(x): y = x * 3; z = g(x); return y + z`, compiled for x86-64 System V, where can `y` live while `g` runs? Compare the cost of the two options.

SOLUTION

`y` is live across the call, and the call may overwrite every caller-saved register, which includes `rax`, `rcx`, `rdx`, `rsi`, `rdi` and `r8` to `r11`. So `y` can go in a caller-saved register only if `f` stores it to the stack before the call and reloads it after: a store and a load around every call. Or it can go in a callee-saved register, `rbx`, `rbp` or `r12` to `r15`, which `g` promises to preserve. Then `f` itself must save that register's old value in its prologue and restore it in its epilogue, because `f`'s caller expects it preserved too: also a store and a load, but once per call of `f`, not per call of `g`. With one call they cost the same. If `f` called `g` in a loop, the callee-saved register would win by far, which is why allocators assign values live across calls to callee-saved registers first. `x`, by contrast, isn't needed after the call, so it can stay in `rdi`, where it arrived, and be passed straight to `g`.

CHECKLIST

- ☐ Can you compute live ranges and register pressure for a block, and liveness across a CFG?
- ☐ Can you build an interference graph and colour it with Chaitin's simplify-and-select method?

- ☐ Can you insert spill code and explain how to choose what to spill?
- ☐ Can you implement linear scan, and say why JIT compilers prefer it?
- ☐ Can you explain caller-saved and callee-saved registers and the argument registers of one ABI?

GO DEEPER

BOOK	Engineering a Compiler, chapters 11 to 13: instruction selection, scheduling and register allocation Keith Cooper & Linda Torczon
PAPER	Linear Scan Register Allocation Massimiliano Poletto & Vivek Sarkar, 1999
SPEC	System V Application Binary Interface, AMD64 supplement x86-64 psABI maintainers

JIT compilation

3 exercises

An ahead-of-time compiler must generate code that works for every input. A just-in-time compiler runs while the program does, so it can watch what actually happens, which functions are hot, which types appear, which branches are taken, and compile specialised code for that, guarded by cheap checks. When a guard fails, it throws the specialised code away and falls back to the interpreter. This chapter builds the three ideas that make dynamic languages fast in V8, HotSpot, PyPy and LuaJIT: hidden classes, inline caches, and speculative compilation with deoptimisation.

Hidden classes

In JavaScript or Python an object can gain fields at any time, so the obvious representation is a hash table per object, and reading `p.x` means hashing `"x"` and probing. That's slow and uses a lot of memory. But most programs create many objects the same way, with the same fields added in the same order, so engines share the layout. A **shape**, called a hidden class or map in V8, records which fields an object has and the slot each is stored in. Objects keep only an array of slot values and a pointer to their shape. Adding a field moves the object to a successor shape, found in a **transition** tree so that every object built the same way reaches the same shape. The idea comes from the Self language of the late 1980s, and V8, SpiderMonkey and JavaScriptCore all use it. CPython applies a similar idea to instances: since 3.3, objects of one class share their dictionaries' keys, and since 3.11 their attribute values are stored inline in the object.

Shapes turn a field read into two steps: check the object's shape, and load the slot at a known index. The check is still needed, because a different object might arrive at the same line of code. That's where caching comes in.

Inline caches

Each place in the code that reads a property, each **site**, gets a small cache recording the shapes it has seen and the slot the field had in each. On a hit, the read is one pointer comparison and one load. On a miss, the engine does the slow lookup and adds the shape to the cache. Inline caching was introduced for Smalltalk by Deutsch and Schiffman in 1984, and extended to **polymorphic** inline caches, holding several shapes, by Hölzle, Chambers and Ungar in 1991. A site that sees one shape is **monomorphic**, a site that sees a few is **polymorphic**, and one that sees many, more than the cache holds, is **megamorphic** and falls back to a slower general lookup. V8 caches up to four shapes per site.

EXAMPLE

jit.py · shapes and inline caches, then a JIT that compiles a hot bytecode loop to Python with a type guard and deoptimises when it fails

```
import time

# 1. Shapes (hidden classes): objects built the same way share one map from field name to slot
class Shape:
    def __init__(self, fields=()): self.fields, self.next = fields, {}
    def add(self, name):           # a transition: this shape plus one
        more field
        if name not in self.next: self.next[name] = Shape(self.fields + (name,))
        return self.next[name]
    def slot(self, name): return self.fields.index(name)
EMPTY = Shape()
class Obj:
    def __init__(self): self.shape, self.slots = EMPTY, []
    def set(self, name, value):
        if name in self.shape.fields: self.slots[self.shape.slot(name)] = value
        else: self.shape = self.shape.add(name); self.slots.append(value)

def point(x, y, order="xy"):
    o = Obj()
    for f in order: o.set(f, x if f == "x" else y)
    return o
p, q, r = point(1, 2), point(3, 4), point(5, 6, order="yx")
print(f"p and q share a shape: {p.shape is q.shape} {p.shape.fields}; "
      f"r was built y-first: {r.shape is p.shape} {r.shape.fields}")

# 2. An inline cache: each property-access site remembers the shapes it has seen and their
slots
class Site:
    def __init__(self, limit=4): self.cache, self.limit, self.hits, self.misses = {}, limit, 0,
    0
    def get(self, obj, name):
        slot = self.cache.get(obj.shape)
        if slot is None:
            self.misses += 1; slot = obj.shape.slot(name)           # the slow, general lookup
            if len(self.cache) < self.limit: self.cache[obj.shape] = slot
        else: self.hits += 1
        return obj.slots[slot]
    def state(self):
        n = len(self.cache)
        return ("monomorphic" if n == 1 else f"polymorphic ({n} shapes)" if self.misses < 2 *
            self.limit
            else "megamorphic")

def shapes(k):
    # k different field orders, each with
    an x
    names = ["x"] + [f"f{i}" for i in range(k - 1)]
    out = []
    for i in range(k):
        o = Obj(); o.set(f"f{i}" if i else "x", 0)
        for n in names:
            if n not in o.shape.fields: o.set(n, 0)
        o.set("x", i); out.append(o)
    return out
for k in (1, 3, 12):
    objs, site = shapes(k), Site()
    total = sum(site.get(objs[i % k], "x") for i in range(10_000))
    misses = f"{site.misses:}, miss{'es' if site.misses > 1 else ''}"
    print(f"{k:2} shape{'s' if k > 1 else ''} at one site: {site.hits:}, hits, {misses},
        {site.state()}")
```

```

# 3. A JIT for a bytecode loop. s = 0; i = 0; while i < n: s = s + xs[i] * 2; i = i + 1
LOOP = [("CONST", 0), ("STORE", "s"), ("CONST", 0), ("STORE", "i"),
        ("LOAD", "i"), ("LOAD", "n"), ("LESS", None), ("JUMP_IF_FALSE", 21),          # 4: the
        loop test
        ("LOAD", "s"), ("LOAD", "xs"), ("LOAD", "i"), ("INDEX", None), ("CONST", 2), ("MUL",
        None),
        ("ADD", None), ("STORE", "s"),
        ("LOAD", "i"), ("CONST", 1), ("ADD", None), ("STORE", "i"), ("JUMP", 4),      # 20: back
        to the test
        ("RETURN", "s")]
BINARY = {"ADD": "+", "MUL": "*", "LESS": "<"}

def jit(code, top, seen):
    """Translate the loop at `top` to Python by running its bytecode on a stack of
    expressions."""
    exit_ip = code[top + 3][1]                # where JUMP_IF_FALSE leaves the loop
    names = sorted({arg for op, arg in code[top:exit_ip] if op == "LOAD"})
    stored = ", ".join(sorted({arg for op, arg in code[top:exit_ip] if op == "STORE"}))
    def translate(start, stop):
        stack, lines = [], []
        for op, arg in code[start:stop]:
            if op == "CONST": stack.append(repr(arg))
            elif op == "LOAD": stack.append(arg)
            elif op == "STORE": lines.append(f"{arg} = {stack.pop()}")
            elif op in BINARY: b = stack.pop(); stack.append(f"{stack.pop()} {BINARY[op]} {b}")
            elif op == "INDEX":                # specialise on the element type seen
                so far
                index, seq = stack.pop(), stack.pop()
                lines += [f"v = {seq}[{index}]",
                        f"if type(v) is not {seen}: return {top + 4}, {stored}    # guard:
                        deoptimise"]
                stack.append("v")
        return stack, lines
    (test,), _ = translate(top, top + 3)        # the condition, as one expression
    _, body = translate(top + 4, exit_ip - 1)  # the body, up to the backward jump
    src = (f"def loop({'', '.join(names)}):\n    while {test}:\n        + """.join(f"        {b}\n" for
        b in body)
        + f"    return {exit_ip}, {stored}\n")
    return src, names, stored.split(", ")

def vm(code, env, hot=None):
    """Interpret; if `hot` is set, compile a loop once its backward jump has run that many
    times."""
    env, stack, ip, executed, counts, compiled = dict(env), [], 0, 0, {}, None
    while True:
        op, arg = code[ip]; ip += 1; executed += 1
        if op == "CONST": stack.append(arg)
        elif op == "LOAD": stack.append(env[arg])
        elif op == "STORE": env[arg] = stack.pop()
        elif op == "INDEX": i = stack.pop(); stack.append(stack.pop()[i])
        elif op in BINARY:
            b = stack.pop(); a = stack.pop()
            stack.append(a + b if op == "ADD" else a * b if op == "MUL" else a < b)
        elif op == "JUMP_IF_FALSE":
            if not stack.pop(): ip = arg
        elif op == "RETURN": return env[arg], executed, compiled
        elif op == "JUMP":
            ip = arg; counts[ip] = counts.get(ip, 0) + 1
            if hot and counts[ip] == hot and compiled is None:
                compiled, names, stored = jit(code, ip, type(env["xs"][env["i"]]).__name__)
                ns = {}; exec(compiled, ns)
                ip, *values = ns["loop"](*([env[v] for v in names]))          # run the machine code
                env.update(zip(stored, values))
            if ip == arg + 4:
                bad = env["xs"][env["i"]]
                print(f"    guard failed at element {env['i']:,} ({type(bad).__name__}

```

```

        {bad}): "
            f"back to the interpreter at instruction {ip}")

n = 200_000
ints = list(range(n))
t0 = time.perf_counter(); a, ran_a, _ = vm(LOOP, {"n": n, "xs": ints})
t1 = time.perf_counter(); b, ran_b, src = vm(LOOP, {"n": n, "xs": ints}, hot=1000)
t2 = time.perf_counter()
print(f"\ninterpreted: {a:,} after {ran_a:,} bytecode instructions")
print(f"with the JIT: {b:,} after {ran_b:,} interpreted; at iteration 1,000 the loop was
    compiled to")
for line in src.splitlines(): print("    " + line)
print(f"same answer: {a == b}; time with the JIT: {(t2 - t1) / (t1 - t0):.0%} of interpreting")
mixed = ints[:]; mixed[150_000] = 1.5
print("the same loop over a list with one float in it:")
c, ran_c, _ = vm(LOOP, {"n": n, "xs": mixed}, hot=1000)
print(f"    {c:,} after {ran_c:,} interpreted instructions")

```

OUTPUT

p and q share a shape: True ('x', 'y'); r was built y-first: False ('y', 'x')

1 shape at one site: 9,999 hits, 1 miss, monomorphic

3 shapes at one site: 9,997 hits, 3 misses, polymorphic (3 shapes)

12 shapes at one site: 3,332 hits, 6,668 misses, megamorphic

interpreted: 39,999,800,000 after 3,400,009 bytecode instructions

with the JIT: 39,999,800,000 after 17,005 interpreted; at iteration 1,000 the loop was compiled to

```

def loop(i, n, s, xs):
    while i < n:
        v = xs[i]
        if type(v) is not int: return 8, i, s    # guard: deoptimise
        s = s + v * 2
        i = i + 1
    return 21, i, s

```

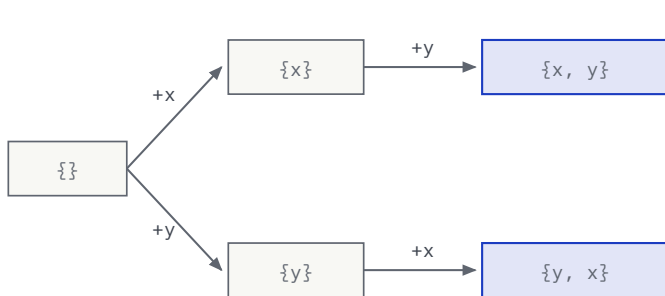
same answer: True; time with the JIT: 5% of interpreting

the same loop over a list with one float in it:

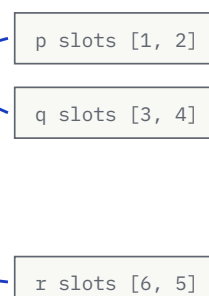
guard failed at element 150,000 (float 1.5): back to the interpreter at instruction 8

39,999,500,003.0 after 867,005 interpreted instructions

SHAPE TRANSITIONS



OBJECTS



CACHE AT o.x

{x, y}: slot 0
 {y, x}: slot 1

2 shapes seen:
 polymorphic
 a hit: compare
 the shape, load
 the slot

p and q share a shape because they were built the same way; r was built y first

Shapes and the cache at one property read. Objects built the same way share a shape, so a site that sees only such objects needs one cache entry, and one comparison per read.

The example's numbers show the three states. One shape gives 9,999 hits from 10,000 reads; three shapes fit in the cache and give 9,997. With twelve shapes cycling through the site, only the first four are ever cached, and two reads in three miss. For a JIT compiler, a site's cache is more than a speed-up: it's **type feedback**, a record of what the program has actually done, and it's what the optimising compiler reads to decide what to specialise. This is why "write monomorphic code" is standard performance advice for JavaScript: initialise all fields in the constructor, in the same order, and don't add fields later.

Compiling the hot loop

The example's virtual machine interprets a loop and counts how often its backward jump runs. At 1,000 iterations it calls the JIT compiler, which translates the loop's bytecode by running it on a stack of *expressions* instead of values: `LOAD s` pushes the string `"s"`, and `ADD` pops two expressions and pushes `"s + v * 2"`. The result is Python source for the loop, which `exec` compiles into a real function. The machine then calls it in the middle of the loop, passing the current values of the variables, which is **on-stack replacement**: switching from interpreted to compiled code without waiting for the function to be called again.

The generated code is **specialised**. The interpreter saw integers in the list, so the compiled loop assumes integers and checks the assumption with a **guard** on each element. In a real JIT, that assumption is what lets `s + v * 2` become two machine instructions on raw integers instead of a call to a generic addition routine. Here it's Python either way, so the speed-up comes only from removing the interpreter's dispatch: the run took about a twentieth of the interpreted time, executing 17,005 bytecode instructions instead of 3,400,009.

The last run shows **deoptimisation**. One float in the list made the guard fail at element 150,000. The compiled code returned the variables and the instruction at which the interpreter should resume, instruction 8, the start of the loop body, and the interpreter finished the loop correctly, more slowly. Real deoptimisation is harder, because optimised machine code keeps values in registers and may have inlined several functions, so the engine must reconstruct every interpreter frame the optimised code replaced. JITs keep metadata at each guard for exactly this purpose. Speculation is only safe because deoptimisation is always possible.

Tiers

Compiling takes time, and optimising compilation takes a lot, so engines compile in **tiers**. V8 starts every function in its Ignition bytecode interpreter, moves warm functions to Sparkplug, a fast non-optimising compiler, then to Maglev, a mid-tier optimiser, and finally the hottest to TurboFan, its optimising compiler. HotSpot moves methods from its interpreter to the C1 compiler and then to C2. Each tier is slower to compile and produces faster code, and each uses feedback gathered by the tier below. A **method JIT** like these compiles whole functions. A **tracing JIT**, as in LuaJIT and PyPy, records the operations of

one hot path through a loop, across function calls, and compiles that trace, with guards wherever the path could have diverged. The example is closer to a tracing JIT, since it compiles a loop rather than a function.

The costs are real. Programs run slowly until they warm up, which matters for short-lived processes, serverless functions and command-line tools. Memory goes to profiles, caches and several versions of the same code. A program whose types keep changing can fall into a cycle of optimising and deoptimising, which engines break by giving up on a function after too many deoptimisations. And generating executable code at run time is a security risk, so modern systems forbid memory from being writable and executable at once, and JITs must switch pages between the two. CPython has moved slowly here: 3.11's specialising interpreter collects the same kind of type feedback, and 3.13 added an experimental JIT that stitches together precompiled machine-code templates, a technique called copy-and-patch.

COMMON TRAPS

Adding fields to objects in different orders or after construction, which multiplies shapes and makes sites megamorphic.

Benchmarking before warm-up, measuring the interpreter instead of the code the program will actually run.

Mixing types in one hot array or variable, such as integers with an occasional float, which makes guards fail.

Speculating without a way back. Every assumption in compiled code needs a guard and a deoptimisation path.

Expecting a JIT to help a program that runs for a few milliseconds. It never gets hot.

EXERCISES

EXERCISE 18.1 Twelve shapes

With twelve shapes, the site had exactly 3,332 hits and 6,668 misses. Derive those numbers from the cache limit of 4 and the access pattern `objs[i % 12]` for `i` below 10,000.

SOLUTION

The cache fills with the first four shapes the site sees, shapes 0 to 3, and then never changes, because the example adds a shape only while the cache has room. Every access to shapes 4 to 11 is a miss. The accesses

cycle through the twelve shapes: $10,000 = 833 \times 12 + 4$, so shapes 0 to 3 are each accessed 834 times and the others 833 times. Shapes 0 to 3 account for $4 \times 834 = 3,336$ accesses, of which the first access to each was a miss, leaving 3,332 hits. Everything else is a miss: $10,000 - 3,332 = 6,668$. A real engine wouldn't keep probing a full cache on every access: once a site is megamorphic, V8 switches it to a shared, global cache keyed by shape and property name, which is slower than an inline cache but much faster than a full lookup.

EXERCISE 18.2 Counting the interpreted instructions

Account for the 17,005 instructions interpreted in the JIT run and the 867,005 in the run with a float, given that the loop's test is 4 instructions and its body 13.

SOLUTION

Initialisation is 4 instructions: two `CONST / STORE` pairs. Each full iteration runs the test, 4 instructions, and the body including the backward jump, 13, for 17. The 1,000th backward jump triggers compilation, so the interpreter ran 1,000 full iterations, 17,000 instructions, before handing over. The compiled loop finishes the remaining 199,000 iterations and returns instruction 21, where the interpreter executes the single `RETURN`: $4 + 17,000 + 1 = 17,005$. With the float, the compiled code ran from iteration 1,000 to the guard failure at element 150,000 and resumed the interpreter at instruction 8, the start of the body. The interpreter then ran 13 instructions for the rest of that iteration, 49,999 more full iterations at 17 each, the final test of 4 that exits, and the `RETURN`: $4 + 17,000 + 13 + 849,983 + 4 + 1 = 867,005$. It never recompiled, because the example compiles each loop at most once, which is the subject of the next exercise.

EXERCISE 18.3 After the guard fails

After the deoptimisation, the example's loop runs in the interpreter for its remaining 50,000 iterations. What should a real JIT do instead, and what must it avoid?

SOLUTION

It should record why it deoptimised, here "element type was `float`, expected `int`", update the type feedback for that site to "`int` or `float`", and let the loop become hot again. The next compilation then specialises for both types, for instance converting to float when either operand is a float, with a guard for anything else. Re-entering compiled

code through on-stack replacement means the remaining iterations run compiled, not interpreted. What it must avoid is a **deoptimisation loop**: compiling for `int`, failing on a float, compiling for `int` again, failing again. Engines count deoptimisations per function and per reason, generalise the assumption each time, and after a limit stop optimising the function, or compile a version without speculation at that site. HotSpot and V8 both do this, and their diagnostics, `-XX:+PrintCompilation` and `--trace-deopt`, show it happening.

CHECKLIST

- ☐ Can you explain shapes and transitions, and why objects built the same way share one?
- ☐ Can you implement an inline cache and say when a site is monomorphic, polymorphic or megamorphic?
- ☐ Can you explain how type feedback, specialisation and guards fit together?
- ☐ Can you explain on-stack replacement and deoptimisation, and why speculation needs the second?
- ☐ Can you describe the tiers of V8 or HotSpot, and the difference between a method and a tracing JIT?

GO DEEPER

ARTICLE [JavaScript engine fundamentals: Shapes and Inline Caches](#) Mathias Bynens & Benedikt Meurer

ARTICLE [Maglev: V8's fastest optimizing JIT](#) The V8 team

PEP [PEP 744: JIT Compilation](#) Brandt Bucher & Savannah Ostrowski

PART V

Paradigms and boundaries

Two chapters on the edges of what the rest of the volume built. The first steps back from implementing one language to compare the ways languages organise programs, imperative, object-oriented, functional and logic, and builds the core of each paradigm that's least familiar to most programmers: algebraic data types with pattern matching, persistent data structures, and a logic programming engine. The second crosses the boundary between languages, calling C from Python and C calling back, which is where everything the volume said about types, memory and calling conventions has to be done by hand.

Paradigms and pattern matching

3 exercises

A paradigm is less a set of rules than a set of defaults: what a language makes easy, and so what its programmers reach for first. Imperative programs change state step by step, object-oriented ones bundle state with the operations on it, functional ones compute new values instead of changing old ones, and logic programs state facts and rules and let a search find answers. Each paradigm also has an implementation that explains it. This chapter builds three: data types taken apart by pattern matching, a list that is never modified, and a small Prolog whose engine is chapter 7's unification plus backtracking.

Four ways to organise a program

Imperative programming is the machine's own model: variables are storage locations, and statements change them in order. Every paradigm compiles down to it eventually. **Object-oriented** programming groups data with the functions that act on it and chooses which function runs by the object's class at run time. Its central mechanism, dynamic dispatch, is implemented with a table of function pointers per class, a **vtable**, so `shape.area()` becomes "load the object's vtable, load the `area` slot, call it", or with chapter 18's inline caches in dynamic languages. **Functional** programming treats functions as values, as chapters 9 and 10 implemented, and prefers expressions that produce new values to statements that modify old ones. **Logic** programming describes relations between things and leaves the order of computation to a search.

Most languages in wide use mix all four. Python, JavaScript, Kotlin, Swift, Rust and Scala have classes, closures, immutable values and pattern matching, and C# and Java have acquired most of the functional features over the last fifteen years. The useful question about a paradigm is not "which language" but "which tool fits this problem", and knowing how each is implemented tells you what it costs.

Algebraic data types and pattern matching

An **algebraic data type** says that a value is exactly one of several cases, each with its own fields. An expression is a number, or a variable, or a sum of two expressions, or a product of two. "One of" makes it a **sum** type, and "with fields" makes each case a **product**, hence the name. OCaml, Haskell, Rust's `enum`, Swift's `enum` and Kotlin's sealed classes express this directly. Python approximates it with a dataclass per case. **Pattern matching** takes such values apart: `case Add(Num(0), x)` checks that the value is an `Add` whose left child is the number 0, and binds its right child to `x`, all in one step. Python added the `match` statement in 3.10 (PEP 634).

EXAMPLE

paradigms.py · an algebraic data type with pattern matching and an exhaustiveness check, a persistent list, and a small Prolog

```
from dataclasses import dataclass

# 1. An algebraic data type: an expression is exactly one of these, and match takes it apart
@dataclass(frozen=True)
class Num: value: int
@dataclass(frozen=True)
class Var: name: str
@dataclass(frozen=True)
class Add: left: object; right: object
@dataclass(frozen=True)
class Mul: left: object; right: object
EXPR = (Num, Var, Add, Mul)

def simplify(e):
    match e:
        case Add(Num(0), x) | Add(x, Num(0)): return simplify(x)
        case Mul(Num(1), x) | Mul(x, Num(1)): return simplify(x)
        case Mul(Num(0), _) | Mul(_, Num(0)): return Num(0)
        case Add(Num(a), Num(b)): return Num(a + b)
        case Add(l, r) | Mul(l, r):
            new = type(e)(simplify(l), simplify(r))
            return simplify(new) if new != e else new
        case _: return e

def show(e):
    match e:
        case Num(v): return str(v)
        case Var(n): return n
        case Add(l, r): return f"({show(l)} + {show(r)})"
        case Mul(l, r): return f"({show(l)} * {show(r)})"
e = Add(Mul(Num(1), Var("x")), Add(Mul(Var("y"), Num(0)), Add(Num(2), Num(3))))
print(f"{show(e)} simplifies to {show(simplify(e))}")

# An exhaustiveness check, as a compiler does it: which constructors can reach no case?
def uncovered(cases):
    covered = set()
    for c in cases:
        if c == "_": return []
        covered.add(c)
    return [t.__name__ for t in EXPR if t.__name__ not in covered]
print(f"show() covers every constructor: {not uncovered(['Num', 'Var', 'Add', 'Mul'])}; "
      f"a version without Var misses: {uncovered(['Num', 'Add', 'Mul'])}")

# 2. A persistent list: "changing" it builds a new version that shares everything it can
@dataclass(frozen=True)
class Cons: head: object; tail: object
def from_list(xs):
    out = None
    for x in reversed(xs): out = Cons(x, out)
    return out
def to_list(c):
    out = []
    while c: out.append(c.head); c = c.tail
    return out
def set_at(c, i, value):
    # copy the path to i, share the rest
    return Cons(value, c.tail) if i == 0 else Cons(c.head, set_at(c.tail, i - 1, value))
def nodes(c):
    out = []
    while c: out.append(c); c = c.tail
    return out
v1 = from_list(list(range(1000)))
```

```

v2 = Cons(-1, v1)                                # prepend: one new node
v3 = set_at(v1, 2, 99)                            # change element 2: three new nodes

for name, v in (("prepend", v2), ("change element 2", v3)):
    shared = len({id(n) for n in nodes(v)} & {id(n) for n in nodes(v1)})
    new = len(nodes(v)) - shared
    print(f"{name:17}: {new} new node{'s' if new > 1 else ''}, {shared} shared with the
          original; "
          f"the original still starts {to_list(v1)[:4]}")

# 3. Logic programming: facts, rules, and a query answered by unification and backtracking
def is_var(t): return isinstance(t, str) and t[0].isupper()
def walk(t, s):
    while is_var(t) and t in s: t = s[t]
    return t
def unify(a, b, s):
    a, b = walk(a, s), walk(b, s)
    if a == b: return s
    if is_var(a): return {**s, a: b}
    if is_var(b): return {**s, b: a}
    if isinstance(a, tuple) and isinstance(b, tuple) and len(a) == len(b):
        for x, y in zip(a, b):
            s = unify(x, y, s)
            if s is None: return None
        return s
    return None
facts = [("parent", "ada", "ben"), ("parent", "ben", "cy"), ("parent", "ben", "dee"),
         ("parent", "dee", "eve"), ("parent", "fay", "gus")]
rules = [(("ancestor", "X", "Y"), [("parent", "X", "Y")]),
         (("ancestor", "X", "Y"), [("parent", "X", "Z"), ("ancestor", "Z", "Y")])]
steps = [0]
def rename(t, n):
    if is_var(t): return f"{t}_{n}"
    return tuple(rename(x, n) for x in t) if isinstance(t, tuple) else t
def solve(goals, s, depth=0):
    """Prove the goals left to right; each fact or rule that unifies is a choice to backtrack
    to."""
    if not goals: yield s; return
    goal, rest = goals[0], goals[1:]
    for fact in facts:
        steps[0] += 1
        s2 = unify(goal, fact, s)
        if s2 is not None: yield from solve(rest, s2, depth)
    for head, body in rules:
        steps[0] += 1
        head, body = rename(head, depth), [rename(b, depth) for b in body]
        s2 = unify(goal, head, s)
        if s2 is not None: yield from solve(body + rest, s2, depth + 1)
def resolve(t, s):
    t = walk(t, s)
    return tuple(resolve(x, s) for x in t) if isinstance(t, tuple) else t
for query in (("ancestor", "ada", "Who"), ("ancestor", "Who", "eve"), ("ancestor", "fay",
"eve")):
    steps[0] = 0
    answers = [resolve(query, s)[1 if query[1][0].isupper() else 2] for s in solve([query],
    {})]
    q = f"{query[0]}{'', '.join(query[1:])}"
    print(f"?- {q:21} {'', '.join(answers) if answers else 'no'}    ({steps[0]} unification
    attempts)")

```

OUTPUT

```

(1 * x + (y * 0 + (2 + 3))) simplifies to (x + 5)
show() covers every constructor: True; a version without Var misses: ['Var']
prepend          : 1 new node, 1000 shared with the original; the original still starts [0, 1,
2, 3]
change element 2 : 3 new nodes, 997 shared with the original; the original still starts [0, 1

```

```

2, 3]
?- ancestor(ada, Who)    ben, cy, dee, eve    (105 unification attempts)
?- ancestor(Who, eve)    dee, ada, ben    (210 unification attempts)
?- ancestor(fay, eve)    no    (42 unification attempts)

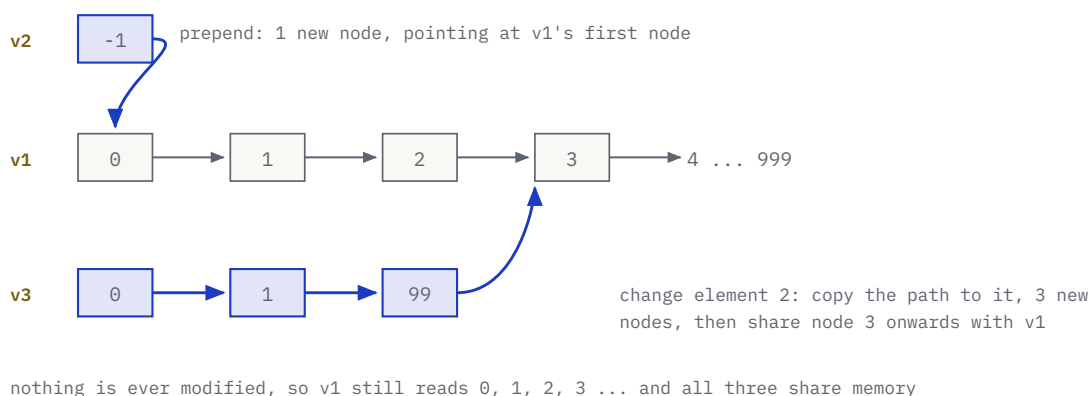
```

The simplifier is the kind of code pattern matching was made for: each rule of algebra is one **case**, written the way you'd write it on paper. A compiler turns a **match** into a **decision tree** that tests each part of the value at most once, rather than trying the cases one by one; Luc Maranget's algorithms for this are used by OCaml. And because a type's cases are known, a compiler can check **exhaustiveness**: that every possible value is handled by some case. The example's check is the simplest version, top-level constructors only. Rust and OCaml do it fully, nested patterns included, and Rust refuses to compile a **match** that misses a case. Python's **match** doesn't check at all, though type checkers such as mypy and Pyright can, for enums and unions of classes.

The trade-off between data types and classes is called the **expression problem**, a name Philip Wadler gave it in 1998. With data types and pattern matching, adding an operation is easy, one new function with a case per constructor, but adding a case means editing every function. With classes and methods it's the reverse: a new subclass is one place, but a new operation means a method in every class. Exercise 19.1 makes this concrete.

Immutability and persistent data

A functional program changes a list by building a new list. That sounds expensive, and would be if every change copied everything, but immutable structures can share. A **persistent** data structure keeps every old version intact and makes each new version mostly out of parts of old ones. Prepending to a linked list creates one node, which points to the old list: the 1,000 old nodes are shared, and both versions remain valid. Changing element 2 copies only the path to it, three nodes, and shares the rest.



Structural sharing. Each "change" builds new nodes only along the path to the change, so old versions stay valid for free and new ones cost as little as the change itself.

Immutability is what makes the sharing safe: no one can modify a shared node, so no version can be changed through another. It also makes values safe to share between threads without locks, and makes undo and history nearly free, since keeping an old version costs nothing. The costs are more allocation and more work for the garbage collector, and slow access for some operations on a plain linked list, which is why practical persistent structures use trees. Clojure's vectors and maps, based on Phil Bagwell's hash array mapped tries, branch 32 ways, so a million-element vector is only four levels deep and an update copies four small nodes. Chris Okasaki's *Purely Functional Data Structures* is the standard reference.

Logic programming

Prolog, created by Alain Colmerauer and Philippe Roussel in 1972 on logical foundations laid by Robert Kowalski, programs with **facts**, such as `parent(ada, ben)`, and **rules**, such as "`X` is an ancestor of `Y` if `X` is a parent of `Z` and `Z` is an ancestor of `Y`". A query asks which values of its variables make it true. The engine is short. To prove a goal, try each fact and rule in order: **unify** the goal with it, using the same algorithm chapter 7 used for types, and if that succeeds, prove the rule's conditions with the variables bound so far. Each fact or rule that might apply is a choice, and when a later goal fails, the search **backtracks** to the most recent choice and tries the next option. The example's `solve` is a generator, so backtracking is just resuming an earlier loop.

The three queries show what that buys. `ancestor(ada, who)` and `ancestor(who, eve)` use the same two rules in opposite directions, finding descendants in one case and ancestors in the other, because a relation has no fixed inputs and outputs. And `ancestor(fay, eve)` fails after exhausting the search. The cost of that generality is that the search order is part of the program's meaning. Clauses are tried top to bottom and goals left to right, depth first, so a rule written in a logically equivalent but different order can loop forever, as exercise 19.2 shows. **Datalog**, a restricted Prolog without function symbols, guarantees termination and evaluates bottom-up, and it has found a second life inside program-analysis tools, where analyses such as "which pointers can point to which objects" are written as a few dozen Datalog rules.

COMMON TRAPS

Forgetting a case in a `match` in a language that doesn't check exhaustiveness, so a new constructor falls through silently.

Using a catch-all `case _` that hides the missing cases a checker would otherwise report.

Treating persistent structures as free, when every update allocates and a linked list is slow to index.

Mutating a value you think is a private copy but that shares structure with another.

Writing a left-recursive Prolog rule, which the depth-first search follows forever.

EXERCISES

EXERCISE 19.1 The expression problem

In the example's style, add a constructor `Neg(e)` for negation, and separately add an operation `size(e)` that counts nodes. Which existing code must change for each? How would the answers differ if each constructor were a class with `simplify` and `show` methods?

SOLUTION

Adding `Neg` means a new dataclass and a new case in every function that matches on expressions: `simplify` needs `case Neg(Num(a))` and `case Neg(x)`, `show` needs `case Neg(x)`, and `EXPR` must list it, after which the exhaustiveness check would flag any function that still misses it. Adding `size` touches nothing existing: one new function with a case per constructor. With classes, it's the reverse. `Neg` is one new class with its own `simplify` and `show` methods, and no existing class changes. But `size` needs a new method in every class. Neither organisation makes both kinds of extension local, which is the expression problem. Languages offer partial escapes, such as type classes in Haskell, traits in Rust and the visitor pattern in object-oriented languages, each moving the cost somewhere else rather than removing it.

EXERCISE 19.2 A rule that never ends

Change the second rule to `ancestor(X, Y) :- ancestor(X, Z), parent(Z, Y)`, which means the same thing logically. What happens to the query `ancestor(ada, Who)`, and to `ancestor(fay, eve)`? Why?

SOLUTION

Making the change in the example's `rules` list, the first query prints all four correct answers, `ben`, `cy`, `dee` and `eve`, and then, asked for more, raises `RecursionError`. The second never prints anything before the error. To prove `ancestor(X, Y)` with the new rule, the engine's first subgoal is `ancestor(X, Z)`, which it proves by trying the rules again, whose first subgoal is `ancestor(X, Z')`, and so on: the depth-first search descends forever without consuming any fact. The answers that do exist are found first only because the non-recursive rule is listed first and each level tries it before recursing. Logically the two programs are equivalent; operationally, Prolog's fixed order makes one

a loop. The fixes are to keep recursive goals after the goals that bind their variables, as the original rule does, or to use a system with **tabling**, such as XSB or SWI-Prolog's `:- table`, which remembers goals in progress and stops such loops, or Datalog, which evaluates bottom-up and always terminates.

EXERCISE 19.3 The cost of a persistent update

How many new nodes does `set_at(v1, i, value)` create for the 1,000-element list, as a function of `i`? What would a persistent vector built as a 32-way tree need for the same update, and why do practical persistent collections use trees?

SOLUTION

`set_at` copies every node before position `i` and replaces the node at `i`, so it creates `i + 1` nodes and shares the remaining `999 - i`. Changing the last element copies the whole list. Reading element `i` also takes `i` steps. A persistent vector stores elements in the leaves of a tree with 32 children per node. A tree of 1,000 elements has 2 levels, and an update copies the path from the root to one leaf: 2 nodes of up to 32 entries each, whatever the index. Reads take 2 steps. Both costs are proportional to $\log_{32} n$, which is at most 7 for a collection of 34 billion elements, so in practice they're close to constant. The linked list is persistent and simple, but only its front is cheap, which is why functional languages use lists for stacks and sequences processed front to back, and trees for everything indexed.

CHECKLIST

- ☐ Can you describe the four paradigms and the implementation mechanism behind each?
- ☐ Can you define an algebraic data type and take it apart with nested patterns?
- ☐ Can you explain exhaustiveness checking and the expression problem?
- ☐ Can you explain structural sharing and compute the cost of a persistent update?
- ☐ Can you explain how a Prolog engine uses unification and backtracking, and why clause order matters?

GO DEEPER

PEP PEP 636: Structural Pattern Matching tutorial Daniel Moisset

BOOK The Power of Prolog Markus Triska

THESIS Purely Functional Data Structures Chris Okasaki

Foreign function interfaces

3 exercises

*Sooner or later every language needs to call code written in another: the operating system's libraries, a database driver, a numerical library tuned for years. Almost always the meeting point is C, because C's calling convention and data layout are the one interface every language and platform agrees on. A **foreign function interface** translates between a language's own values and C's, follows the calling convention of chapter 17, and leaves the programmer responsible for everything the type checker and garbage collector used to handle. This chapter calls C from Python with `ctypes`, lays out a struct byte by byte, lets C call back into Python, and measures what each crossing costs.*

The C ABI is the common ground

A compiled C function is machine code at an address. Nothing in it records its parameter types or its return type. The caller and callee agree on those through the source code's declarations, and on how to pass them through the platform's **application binary interface**: which registers hold which arguments, how structs are laid out, who cleans up the stack. Because the ABI is defined in terms of C, and because C has no name mangling, no exceptions and no garbage collector, a C function is something any language can call if it knows the signature. Rust exposes functions to others with `extern "C"`, C++ with `extern "C"` to turn off name mangling, Go with cgo, Java with JNI and since Java 22 the Foreign Function and Memory API, and Python with its C API, `ctypes` and `cffi`.

The missing types are the first thing an FFI user must supply, by hand. `ctypes` needs `argtypes` and `restype` for each function, and if they're wrong, nothing stops the call. The example declares `strtod`'s arguments but not, at first, its result. `ctypes` assumes every function returns a C `int` unless told otherwise, so it read the integer return register, while `strtod` had put its `double` result in a floating-point register. The result was an integer that had nothing to do with 2.5. With `restype` set, the same call returns 2.5. A wrong declaration doesn't raise an error; it returns garbage or crashes, which is the sense in which FFI code is unsafe.

EXAMPLE

ffi.py · calling the C library with ctypes, struct layout and padding, a C function calling back into Python, buffers, and the cost per call

```
import ctypes, time

libc = ctypes.CDLL(None)           # the C library already loaded in this process (Linux,
                                  macOS)
```

```

# 1. Calling C: ctypes needs the C signature, because the machine code carries no types at all
strlen = libc.strlen
strlen.argtypes, strlen.restype = [ctypes.c_char_p], ctypes.c_size_t
print(f"strlen(b'compiler') = {strlen(b'compiler')}")
strtod = libc.strtod
strtod.argtypes = [ctypes.c_char_p, ctypes.POINTER(ctypes.c_char_p)]
guess = strtod(b"2.5", None) # no restype yet: ctypes assumes the result is
                             an int
strtod.restype = ctypes.c_double
print(f"strtod(b'2.5') before restype was set: an {type(guess).__name__}, not 2.5; "
      f"after: {strtod(b'2.5', None)}")

# 2. A struct must match C's layout exactly, padding included
class Loose(ctypes.Structure):
    _fields_ = [("flag", ctypes.c_char), ("count", ctypes.c_int), ("tag", ctypes.c_char),
                ("value", ctypes.c_double)]
class Tight(ctypes.Structure): # the same fields, largest first
    _fields_ = [("value", ctypes.c_double), ("count", ctypes.c_int), ("flag", ctypes.c_char),
                ("tag", ctypes.c_char)]
for cls in (Loose, Tight):
    layout = ", ".join(f"{n}@{getattr(cls, n).offset}" for n, _ in cls._fields_)
    data = sum(ctypes.sizeof(t) for _, t in cls._fields_)
    print(f"{cls.__name__}: {ctypes.sizeof(cls)} bytes for {data} bytes of fields
          ({layout})")

# 3. C calling back into Python: qsort with a comparison function written in Python
calls = 0
CMP = ctypes.CFUNCTYPE(ctypes.c_int, ctypes.POINTER(ctypes.c_int),
                       ctypes.POINTER(ctypes.c_int))
def compare(a, b):
    global calls
    calls += 1
    return (a[0] > b[0]) - (a[0] < b[0])
values = [42, 7, 19, 3, 88, 61, 5, 23]
array = (ctypes.c_int * len(values))(*values)
callback = CMP(compare) # keep a reference: if it is collected, C calls
                        freed memory
libc.qsort.argtypes = [ctypes.c_void_p, ctypes.c_size_t, ctypes.c_size_t, CMP]
libc.qsort(array, len(array), ctypes.sizeof(ctypes.c_int), callback)
print(f"qsort with a Python comparison: {list(array)}, after {calls} calls from C into Python")

# 4. Memory C can write into: Python allocates the buffer and owns it, C fills it
buf = ctypes.create_string_buffer(8)
libc.strncpy.argtypes = [ctypes.c_char_p, ctypes.c_char_p, ctypes.c_size_t]
libc.strncpy(buf, b"interpreter", ctypes.sizeof(buf) - 1) # never more than the buffer holds
print(f"an 8-byte buffer after strncpy: {buf.value!r}; its raw bytes {bytes(buf)!r}")

# 5. Crossing the boundary has a price: convert every argument, call, convert the result
libc.abs.argtypes, libc.abs.restype = [ctypes.c_int], ctypes.c_int
n = 200_000
t0 = time.perf_counter(); total_py = sum(abs(-i) for i in range(n))
t1 = time.perf_counter(); total_c = sum(libc.abs(-i) for i in range(n))
t2 = time.perf_counter()
per_py, per_c = (t1 - t0) / n * 1e9, (t2 - t1) / n * 1e9
print(f"abs() {n:,} times: built-in {per_py:.0f} ns a call, through ctypes {per_c:.0f} ns "
      f"({per_c / per_py:.0f}x); same total: {total_py == total_c}")

```

OUTPUT

```

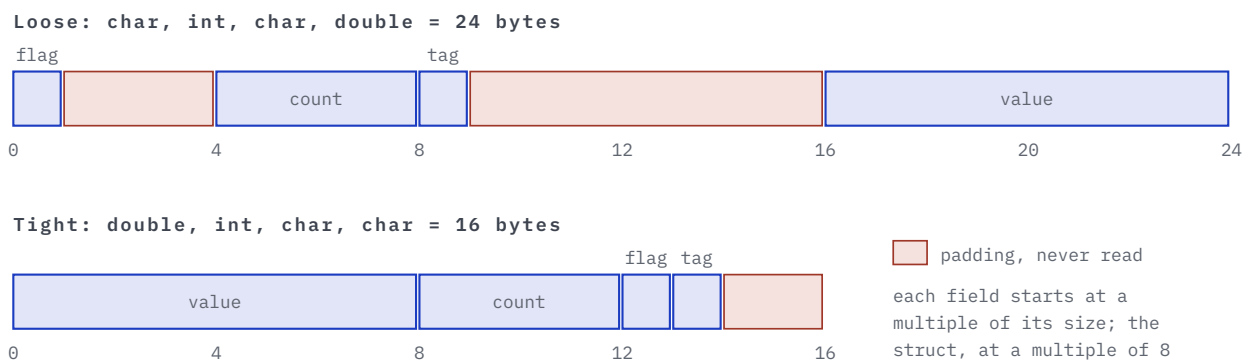
strlen(b'compiler') = 8
strtod(b'2.5') before restype was set: an int, not 2.5; after: 2.5
Loose: 24 bytes for 14 bytes of fields  (flag@0, count@4, tag@8, value@16)
Tight: 16 bytes for 14 bytes of fields  (value@0, count@8, flag@12, tag@13)
qsort with a Python comparison: [3, 5, 7, 19, 23, 42, 61, 88], after 17 calls from C into
Python
an 8-byte buffer after strncpy: b'interpr'; its raw bytes b'interpr\x00'

```

```
abs() 200,000 times: built-in 44 ns a call, through ctypes 275 ns (6x); same total: True
```

Data layout

To pass a struct to C, the FFI must lay it out exactly as the C compiler would, and C's rule is **alignment**: each field starts at an offset that's a multiple of its own size, up to the platform's maximum, and the whole struct's size is rounded up to a multiple of its largest field's alignment. Between fields that don't fit the rule, the compiler inserts **padding**. The example's `Loose` struct has 14 bytes of fields and takes 24: three bytes of padding after the first `char` so that the `int` starts at 4, and seven after the second `char` so that the `double` starts at 16. Ordering the same fields from largest to smallest leaves only two bytes of padding at the end, and 16 bytes in all.



The same four fields in two orders. Padding, shaded, keeps each field at a multiple of its size, so the order of declarations decides how much memory is wasted.

`ctypes` computes the same layout as the platform's C compiler, which is the point: a `ctypes.Structure` can be passed to C and read by C code compiled separately. The rules differ between platforms in the details, the size of `long` is 8 bytes on Linux and macOS but 4 on 64-bit Windows, which is why FFI declarations use fixed-width types such as `c_int32` and `c_uint64` where the C side does. Byte order, the layout of bit fields and packed structs are further places where two sides can silently disagree.

Calling back

Some C functions take function pointers. `qsort` sorts an array of anything, so it needs the caller's comparison function. To pass a Python function, `ctypes` creates a small piece of machine code, a **thunk**, with the C signature: when C calls it, it converts the C arguments to Python objects, calls the Python function, and converts the result back. In the example, `qsort` called back into Python 17 times to sort eight numbers. The count depends on the C library's sorting algorithm, so it will differ on another platform. The example keeps `callback` in a variable because the thunk belongs to that Python object: if it were garbage-collected while C still held the pointer, the next call would jump into freed memory. The garbage collector knows nothing about pointers stored in C.

That is the general problem of the boundary: **ownership**. When Python passes memory to C, it must keep it alive for as long as C may use it. When C returns memory to Python, someone must know whether to free it, and with which allocator. The example lets Python allocate an 8-byte buffer with `create_string_buffer`, has C write into it, and limits the copy to the buffer's size minus one. `strcpy` doesn't add a terminating zero when it runs out of room, so the last byte is left as the zero that `create_string_buffer` initialised it with. Leaving out that limit is how a buffer overflow begins.

The cost of crossing

Each call through `ctypes` converts every argument from a Python object to a C value, sets up the call according to the ABI, makes it, and converts the result back into a new Python object. In the example's measurement, calling C's `abs` through `ctypes` cost several times as much as calling Python's built-in `abs`, even though the C function does almost nothing, because nearly all the time goes on the crossing. The lesson is the design rule for every FFI: **cross the boundary rarely, and do a lot of work each time**. NumPy is fast because a single Python call hands C an entire array to process in a loop, not because each element crosses. The same rule applies to cgo in Go, where each call costs tens of nanoseconds, to JNI, and to calls between JavaScript and WebAssembly.

Runtimes add costs of their own. CPython's `ctypes` releases the global interpreter lock during a foreign call, so other Python threads can run while C works, and must take it back before calling a Python callback. Exceptions can't cross: a Python exception raised in a callback can't unwind through C's stack frames, so `ctypes` prints it and returns a default value to C, and since Rust 1.81 a Rust panic that would unwind out of an `extern "C"` function aborts the process instead. And the C API that extension modules use exposes CPython's internals, reference counts and object layouts included, which is why tools such as Cython, pybind11 and PyO3 exist: they generate the conversion and reference-counting code, which is where hand-written extensions most often go wrong.

COMMON TRAPS

Leaving out `restype`, so a function returning a pointer or a double is read as a 32-bit integer.

Letting a callback or buffer be garbage-collected while C still holds a pointer to it.

Declaring a struct whose field order or types differ from the C header, which corrupts every field after the first mismatch.

Using `c_long` where the C side means a fixed 64-bit integer, which breaks on Windows.

Crossing the boundary once per element in a hot loop instead of once per array.

EXERCISE 20.1 Ordering fields

A C struct has the fields `char a; double b; short c; char d; int e;`. What is its size on a 64-bit platform, where are the fields, and what order minimises its size? Check your answer with `ctypes.sizeof`.

SOLUTION

In the given order: `a` at 0; `b`, aligned to 8, at 8 after 7 bytes of padding; `c` at 16; `d` at 18; `e`, aligned to 4, at 20 after 1 byte of padding; ending at 24, which is already a multiple of 8. So 24 bytes for 16 bytes of fields. Sorting by alignment, largest first: `b` at 0, `e` at 8, `c` at 12, `a` at 14, `d` at 15, ending at 16 with no padding at all. A `ctypes.Structure` with each order gives `sizeof` 24 and 16. Sorting by decreasing alignment is always optimal or close to it. Compilers don't reorder fields for you in C or C++, since the order is part of the ABI, but Rust does reorder the fields of its own structs unless they're marked `#[repr(C)]`, which is required for any struct passed across an FFI boundary.

EXERCISE 20.2 Who frees it?

A C library function `char *describe(int code)` returns a string it allocated with `malloc`, and the library provides `void describe_free(char *)`. How should you declare and call it with `ctypes` so that you get the string and nothing leaks? What goes wrong if you declare `restype = c_char_p`?

SOLUTION

Declare `describe.restype = ctypes.c_void_p`, so that `ctypes` returns the raw address as an integer. Then copy the string into Python with `ctypes.string_at(ptr)`, which reads up to the terminating zero and returns `bytes`, and pass the same pointer to `describe_free`, declared with `argtypes = [ctypes.c_void_p]`, preferably in a `try/finally` so it's freed even if decoding fails. With `restype = c_char_p`, `ctypes` converts the result to a `bytes` object immediately and discards the pointer, so there's no way left to free the memory: every call leaks one string. Calling Python's or the C library's `free` instead of `describe_free` is also wrong if the library uses its own allocator, which is why well-designed C libraries always provide their own free function.

EXERCISE 20.3 Batching the crossings

A program calls a C function `double score(double)` through `ctypes` for each of ten million values, and it's slow. Without changing what is computed, how would you restructure the boundary, and roughly how much faster should it get?

SOLUTION

Ten million crossings at a few hundred nanoseconds each is several seconds of overhead, against perhaps a few milliseconds of actual work in C. Add a C function that processes a whole array, `void score_all(const double *in, double *out, size_t n)`, and call it once: allocate the arrays with `(ctypes.c_double * n)()` or, better, use NumPy arrays and pass their data pointers, so the values are converted in bulk rather than one Python object at a time. Then the crossing costs a few microseconds once and the loop runs at C speed, so the speed-up is typically a hundredfold or more, limited by the work in `score` itself. If the C function can't be changed, the next best option is to call it from compiled code, a small Cython or C extension loop, so the per-element call is C to C. The general shape is the same in every FFI: move the loop to the side where calls are cheap.

CHECKLIST

- ☐ Can you explain why C's ABI is the common interface between languages?
- ☐ Can you call a C function with `ctypes`, declaring its arguments and result correctly?
- ☐ Can you compute a struct's layout with padding and reorder it to save space?
- ☐ Can you pass a Python callback to C, and say what must keep it alive?
- ☐ Can you explain the ownership and cost rules for data crossing the boundary?

GO DEEPER

DOCS [ctypes: A foreign function library for Python](#) Python documentation

BOOK [The Rustonomicon: Foreign Function Interface](#) The Rust Project

ARTICLE [The Lost Art of Structure Packing](#) Eric S. Raymond

The stage asks you to build a complete language: Lox, from Robert Nystrom's *Crafting Interpreters*, twice. First as jlox, a tree-walking interpreter in Java, then as clox, a bytecode compiler and virtual machine in C with its own garbage collector. Then add one feature of your own that the book doesn't cover. The book is free to read online, and its chapters line up closely with this volume's, so read each pair together: the book shows one complete, careful implementation, and this volume explains the alternatives and why real languages choose among them.

jlox: the tree-walking interpreter

CRAFTING INTERPRETERS	WHAT YOU BUILD	THIS VOLUME
4	a scanner with positions and error reporting	Ch. 2
5–6	the grammar, syntax tree classes and a recursive-descent parser	Ch. 3–4
7–9	evaluating expressions, statements, variables, scopes and control flow	Ch. 5, 9
10	functions, returns and closures	Ch. 9–10
11	a resolver pass that binds every local to its scope	Ch. 5
12–13	classes, instances, methods and inheritance	Ch. 18–19

clox: the bytecode virtual machine

CRAFTING INTERPRETERS	WHAT YOU BUILD	THIS VOLUME
14–15	chunks of bytecode, a disassembler and the dispatch loop	Ch. 11–12
16–17	a scanner on demand and a single-pass Pratt compiler	Ch. 2, 4, 11
18–20	tagged values, strings and a hash table	Ch. 6, 12
21–23	global and local variables, and jumps for control flow	Ch. 5, 11, 12

24–25	call frames, functions, and closures with upvalues	Ch. 10, 12
26	a mark-sweep garbage collector	Ch. 13
27–29	classes, methods, initialisers and superclasses	Ch. 18–19
30	optimisation: faster hashing and NaN-boxing	Ch. 12, 16

Run the book's own test suite, published with its source code, against both implementations as you go: it has hundreds of small Lox programs with their expected output, and passing it is the definition of done. Keep jlox and clox in the same repository so you can compare them on the same programs, and count the work each does, nodes visited and instructions executed, the way this volume's examples do.

A feature of your own

Choose one and add it to clox, with tests and a short write-up of the design decisions. Each is a chapter of this volume turned into a project.

- 01 **Tail-call elimination**, with a `TAIL_CALL` instruction that reuses the frame, and a test that loops a million times by recursion (chapter 14).
- 02 **A static type checker** for a typed dialect of Lox, with inference for local variables and annotations at function boundaries (chapters 6 and 7).
- 03 **Algebraic data types and a `match` statement**, compiled to jumps, with an exhaustiveness check (chapter 19).
- 04 **A generational collector** with a nursery and a write barrier, measured against the book's mark-sweep collector (chapter 13).
- 05 **Inline caches and shapes** for property access, with counts of hits and misses per site (chapter 18).
- 06 **Superinstructions or a register-based instruction set**, justified by counting which instruction sequences are most frequent (chapter 12).
- 07 **A constant-folding and dead-code pass** in the compiler, with a disassembly before and after (chapter 16).
- 08 **Generators**, with `yield` implemented by keeping the frame on the heap

(chapter 14).

09 **Exceptions** with `try` and `throw`, implemented with an exception table (exercise 12.3).

10 **A foreign function interface** that lets Lox call functions from a C library, declared with their signatures (chapter 20).

One per chapter, each linked back to it. Read them when a language surprises you, when an interpreter is slow, or when you're about to write a compiler pass.

-
- 01 When a language surprises you, find the stage that made the choice: lexer, parser, checker or runtime.
 - 02 Lex by longest match, and record the line and column of every token; every later error message depends on it.
 - 03 Encode precedence in levels and associativity in the side of recursion, and check a grammar for ambiguity before parsing with it.
 - 04 Parse statements by recursive descent and expressions by binding power, and recover at statement boundaries.
 - 05 Resolve names once, before the program runs, and turn every local variable into a slot.
 - 06 A static checker examines every path and a test examines one; use both, and know where your language's types are unsound.
 - 07 Let inference fill in the inside of functions, and write the signatures at their boundaries.
 - 08 Ask for the narrowest interface: what's only read can be covariant, and what's also written must be invariant.
 - 09 Count the work an interpreter does before you try to make it faster.
 - 10 Closures capture variables, not values; give each loop iteration its own binding.
 - 11 Compile once and run many times: bytecode removes repeated decoding, not steps.
 - 12 Keep the program's calls on the virtual machine's own stack, never the host's.
-

- 13 A collector frees what is unreachable, not what is unused, and every old-to-young pointer needs the write barrier.

- 14 Only calls in tail position can be free; where the language doesn't eliminate them, write a loop or a trampoline.

- 15 Optimise on a representation in which every value has exactly one definition.

- 16 An optimisation may change anything except observable behaviour; floating point and side effects are where that breaks.

- 17 Registers are the scarcest resource: spill what's cheap to reload, and never inside the hot loop.

- 18 Keep hot code monomorphic, and let it warm up before you measure it.

- 19 Use data types and matching when operations multiply, classes when cases multiply, and let the compiler check every case.

- 20 Declare every foreign signature exactly, keep foreign memory alive on purpose, and cross the boundary rarely.

READING LIST

The books, papers and documentation behind this volume, and the chapters each one deepens. If you read only one, make it *Crafting Interpreters*, which is free and builds the stage's deliverable step by step. For the compiler half, *Engineering a Compiler* is the most practical textbook, and for the type half, *Types and Programming Languages* is the standard one.

WORK	AUTHOR	DEEPENS
Crafting Interpreters	Robert Nystrom	Ch. 1–5, 9–14
Engineering a Compiler	Keith Cooper & Linda Torczon	Ch. 2–4, 15–17
Types and Programming Languages	Benjamin C. Pierce	Ch. 6–8
Compilers: Principles, Techniques, and Tools	Aho, Lam, Sethi & Ullman	Ch. 2–4, 15–17
Modern Compiler Implementation	Andrew Appel	Ch. 3–5, 13, 15–17
Structure and Interpretation of Computer Programs	Abelson & Sussman	Ch. 9–10, 14, 19
SSA-based Compiler Design	Rastello & Bouchez Tichadou, editors	Ch. 15–17
The Garbage Collection Handbook	Jones, Hosking & Moss	Ch. 13
Purely Functional Data Structures	Chris Okasaki	Ch. 19
CPython InternalDocs	CPython developers	Ch. 5, 11–13, 18
The V8 blog	The V8 team	Ch. 12, 15, 17–18

Compiled October 2026 as the companion to stage 07, Languages, interpreters and compilers, of The Engineer's Syllabus, alongside *The Engineer's Workbench*, *Compass*, *Mill*, *Loom*, *Exchange*, *Ledger*, *Craft*, *Shipyard*, *Fleet*, *Watch*, *Citadel* and *Foundry*. It follows the order of topics in the stage and draws on the works in the reading list, and on the original sources for its named results: Backus, Naur and colleagues (1960) on the ALGOL 60 report; McCarthy (1960) on Lisp and garbage collection; Knuth (1965) on LR parsing; Cheney (1970) on copying collection; Colmerauer and Roussel (1972) on Prolog; Pratt (1973) on top-down operator precedence; Milner (1978) on type inference; Dijkstra, Lamport and colleagues (1978) on concurrent collection; Chaitin (1981, 1982) on register allocation by colouring; Ungar (1984) on generation scavenging; Deutsch and Schiffman (1984) on inline caches; Wadler (1989) on parametricity; Hölzle, Chambers and Ungar (1991) on polymorphic inline caches; Cytron, Ferrante, Rosen, Wegman and Zadeck (1991) on SSA form; Wells (1994) on System F; Poletto and Sarkar (1999) on linear scan; and Shi, Casey, Ertl and Gregg (2008) on stack and register virtual machines. Language versions, engine tiers and flags are described as of late 2026 and change; the techniques change far more slowly, and those are what to rely on.

Every example was executed before publication with Python 3.14 on macOS, and they're written to print the same output on any machine: each builds its stage of an implementation from scratch, in pure Python, and measures speed by counting work, in nodes, probes, instructions, copies and multiplications, rather than by timing it. Two examples also time real code, chapter 18's JIT and chapter 20's foreign calls, and show one recorded run on an Apple M1 Pro; chapter 20's count of `qsort` callbacks depends on the C library too. Re-run in October 2026 with Python 3.14.7, every example printed exactly the output shown, apart from the two recorded timings, which differ from run to run and support the same conclusions. Tested here: a lexer, a parser, two interpreters and a compiler for one expression; a hand-written scanner with maximal munch, error positions and a DFA; an ambiguous grammar's trees, the dangling else and an LL(1) table; a Pratt parser with error recovery; a resolver with five kinds of error; a static type checker; Hindley-Milner inference with let-polymorphism; subtyping and variance, and Java's array store check; a tree-walking interpreter counting its work; closures, the loop-variable trap and captured variables; a bytecode compiler with backpatched jumps; a stack virtual machine with call frames and fused instructions; copying and generational collectors and a missing write barrier; tail calls, a trampoline, CPS and a generator as a state machine; lowering to three-address code, dominators and SSA; six optimisation passes to a fixed point, and loop optimisations; register allocation by colouring and linear scan, with spilling; shapes, inline caches and a JIT with deoptimisation; pattern matching, a persistent list and a small Prolog; and calls into C with `ctypes`, struct layout and a callback.