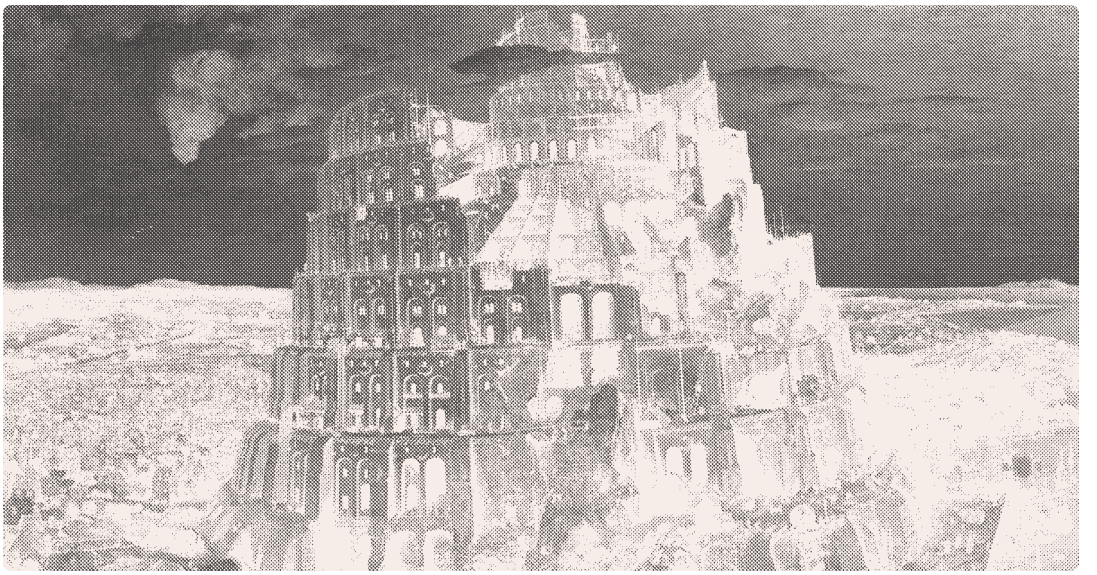
THE PLANK BOOK · V4.9.0

Write a language. Then read its compiler.

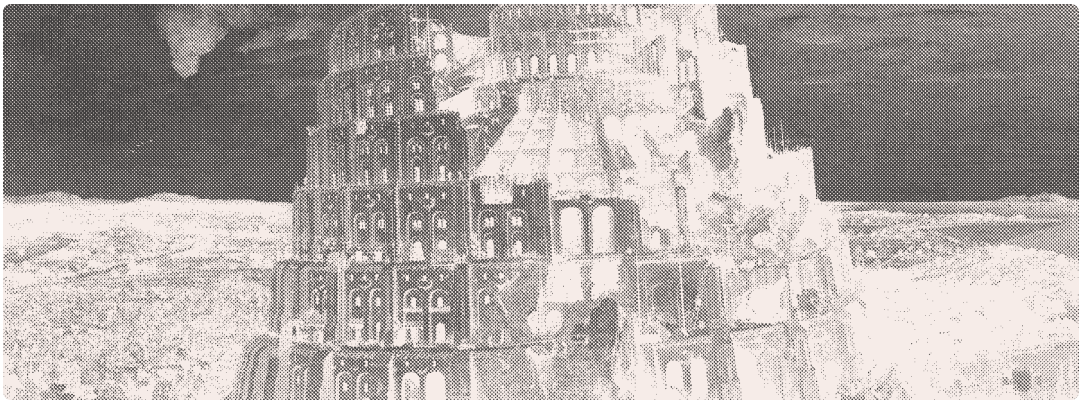
Thirteen short chapters and two appendices. Every program in them is compiled and run by the test suite, so what you read here is what Plank does.

JOSHUA TROMMEL



Contents

	Preface
1	Your first program
2	Values and names
3	Making decisions and repeating
4	Functions
5	Lists, dicts and sets
6	Your own types
7	When things are missing or wrong
8	Functions as values
9	Talking to the world
10	Programs in more than one file
11	Scripting intelligence
12	Inside the compiler
A	The language reference
B	A map of plank.py



Preface

This book teaches Plank from the first line to the last stage of the compiler. You do not need to have written a compiler before. You need to have written a little code in something, Python or JavaScript or Swift, and to be curious about what happens after you press run.

Plank is a small language that compiles to real machine code. Its whole compiler is one Python file, `plank.py`. That fact shapes the book. The first half teaches you to write Plank. The second half opens the compiler and shows you how it turns your programs into binaries, so that by the end there is nothing about the language you have to take on trust.

Every program in this book is real. The test suite pulls each one out, compiles it, runs it and checks that it prints exactly what the book says it prints. If a page and the compiler ever disagree, the build goes red.

How to read it

Read the chapters in order the first time. Each one builds on the last and ends with something you can run. Type the programs in yourself rather than copying them; you learn a language through your fingers.

NOTE

The book is built from the same files as the docs. Appendix A is the language reference and Appendix B is the map of the compiler, pulled in fresh every time the book is built, so neither can fall behind.

When you see a block like this one, it is a program:

```
fn main() {
  print("hello, reader")
}
```

and the block under it is what it prints:

```
OUTPUT
hello, reader
```

Getting Plank

On a Mac with Homebrew:

```
brew install nulljosh/tap/plank-lang
```

Or clone the repository and run `./plank` from inside it. Either way you need a C compiler on the machine, because Plank hands its object files to your system linker. On a Mac that comes with the Xcode command line tools; on Linux, install `gcc` or `clang`. The first run fetches LLVM's Python bindings by itself.

Check it worked:

```
plank --version
```



CHAPTER 1

1 Your first program

Make a file called `hello.pk`:

```
fn main() {
  print("hello, plank")
}
```

Run it:

```
plank run hello.pk
```

```
OUTPUT
hello, plank
```

That one command did a lot. Plank read the file, checked it, turned it into LLVM's intermediate language, asked LLVM to optimize that into machine code, and asked your C compiler to link it into a program. Then it ran the program. Figure 1-1 shows the trip your one line of code takes.

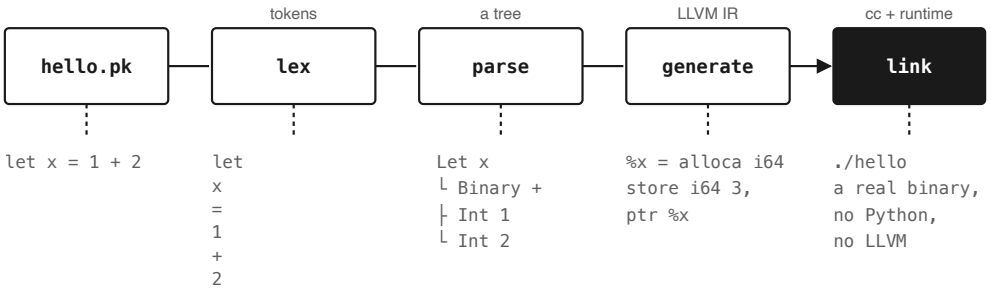


Figure 1-1: *The four steps from a source file to a binary. Every Plank program takes this road.*

If you want to keep the program, build it instead:

```
plank build hello.pk
./hello
```

Now there is a file called `hello` next to your source. It is a native binary. It does not need Plank, Python or LLVM to run. You can copy it to another machine of the same kind and it will work.

What the pieces mean

`fn main()` is where every program starts. `fn` starts a function, `main` is its name, and the empty parentheses say it takes nothing. The braces hold its body.

`print` writes its arguments to the terminal with spaces between them and a newline at the end:

```
fn main() {
  print("one", 2, 3.5, true)
  print()
  print("after a blank line")
}
```

```
OUTPUT
one 2 3.5 true

after a blank line
```

Statements end at the end of a line. There are no semicolons. Comments start with `#` and run to the end of the line.

Trying things quickly

For small experiments, start the REPL:

```
plank repl
```

Type a line and Plank shows you what it does. A bare expression prints its value. Behind the scenes the REPL rebuilds a real binary after every line, so what you see there is exactly what a compiled program would print.

When something is wrong

Make a mistake on purpose:

```
fn main() {
  print(1 + "two")
}
```

Plank refuses to compile it and tells you where and why:

```
hello.pk:2: int + str: types must match, use str()
```

NOTE

Plank checks types before anything runs. A mistake like this one never becomes a crash at 3 a.m.; it is a message at your desk.

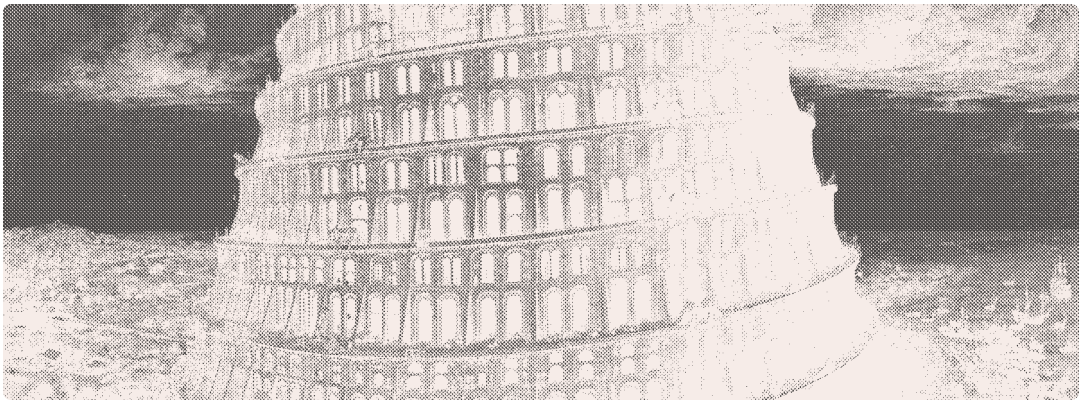
Every error in Plank names the file and the line. Most of them also tell you what to do next. Read them; they were written to be read.

Summary

You wrote, ran and built your first Plank program, and saw the road it takes from text to a native binary. You met `print`, comments and the REPL, and you saw that Plank's errors name the file and line and usually say what to do. Next you will give values names.

TRY IT YOURSELF

1. Make `print` say your name, then build the program and run the binary without `plank`.
2. Print three values of different kinds on one line. What separates them?
3. Open the REPL and type `2 + 3 * 4`. Is the answer what you expected?
4. Break the program on purpose three different ways and read each error. Which one helps the most?



CHAPTER 2

Values and names

Numbers, text and truth

Plank has four simple kinds of value:

- `int`, a whole number: `42`, `-7`, `0`
- `float`, a number with a fractional part: `3.14`, `2.0`, `1.5e3`
- `bool`, `true` or `false`
- `str`, a piece of text: `"hello"`, `"世界"`

Plank never turns one kind into another behind your back. `1 + 2.0` is an error, because one side is an `int` and the other a `float`. When you want a conversion, ask for it:

```
fn main() {
  print(1 + int(2.9))
  print(float(1) + 2.5)
  print("age " + str(36))
  print(int("42") * 2)
}
```

```
OUTPUT
3
3.5
age 36
84
```

`int` on a float drops the fraction. `int` on text reads a number out of it, and stops the program with the line number if the text is not a number. Chapter 7 shows how to catch that instead.

let and var

Give a value a name with `let`:

```
fn main() {
  let name = "plank"
  let year = 2026
  print(name, year)
}
```

```
OUTPUT
plank 2026
```

A `let` name can never point at anything else. If you want a name you can change, use `var`:

```
fn main() {
  var count = 0
  count = count + 1
  count += 10
  print(count)
}
```

```
OUTPUT
11
```

Plank works out the type from the value. You can write it yourself when you want to be clear, or when there is no value to look at yet:

```
fn main() {
  var total: float = 0.0
  total += 1.5
  print(total)
}
```

```
OUTPUT
1.5
```

Prefer `let`. When a reader sees `let`, they know the name means one thing for the rest of the block, and that is one less thing to keep in their head.

Text

Strings join with `+`. Better, drop values straight into a string with `\( )`:

```
fn main() {
  let lang = "plank"
  let n = len(lang)
  print("\(lang) has \(n) letters, \(n * 2) if you say it twice")
}
```

```
OUTPUT
plank has 5 letters, 10 if you say it twice
```

Anything can go inside `\( )`, even another call. Strings come with tools:

```
fn main() {
  let s = "  Hello, Plank  "
  print(s.trim())
  print(s.trim().upper(), s.trim().lower())
  print("a,b,c".split(","))
  print(["x", "y"].join(" and "))
  print("plank".starts_with("pl"), "lan" in "plank")
  print("plank"[0], "plank"[1..3], "plank"[-1])
}
```

```
OUTPUT
Hello, Plank
HELLO, PLANK hello, plank
["a", "b", "c"]
x and y
true true
p la k
```

Indexes count characters, not bytes, so `"héllo"[1]` is `"é"`. A slice like `[1..3]` takes characters 1 and 2: the start is included, the end is not. Negative numbers count from the end.

NOTE

Strings never change in place. Every tool hands back a new string and leaves the old one alone.

Doing arithmetic

`+ - * / %` work on ints and floats. Integer division drops the remainder, and `%` gives it to you:

```
fn main() {
  print(17 / 5, 17 % 5)
  print(17.0 / 5.0)
  print(sqrt(2.0), pow(2.0, 10.0), abs(-4))
  print(fixed(3.14159, 2))
}
```

```
OUTPUT
3 2
3.4
1.4142135623731 1024 4
3.14
```

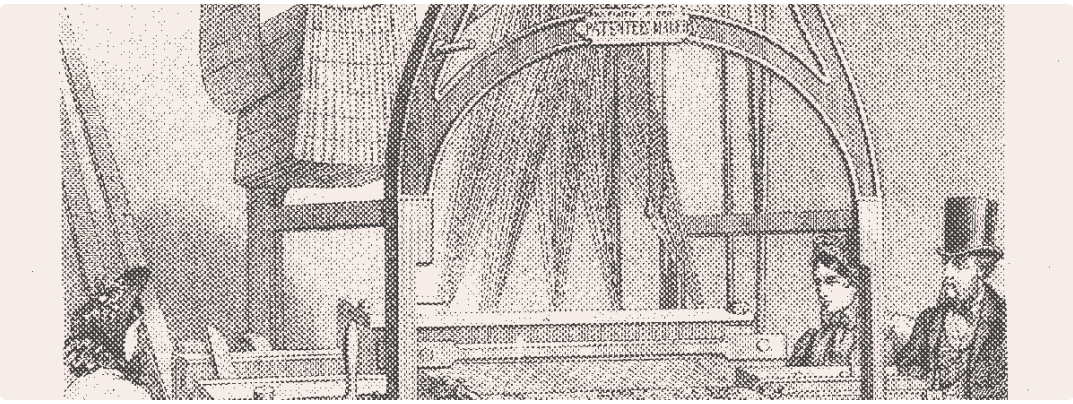
Floats print with up to 15 significant digits, so `0.1 + 0.2` shows as `0.3`. `fixed` gives you exactly as many digits as you ask for.

Summary

Plank has four simple kinds of value, and it never converts one into another unless you ask. `let` names a value for good; `var` names one you can change. Strings interpolate with `\( )`, count characters rather than bytes, and never change in place. Next you will make your programs choose and repeat.

TRY IT YOURSELF

1. Make a `var` counter, add to it twice and print it. Now change `var` to `let`. What does Plank say?
2. Build a sentence out of three variables with `\( )` interpolation.
3. Divide 7 by 2 as ints and as floats. Explain the difference in one sentence.



CHAPTER 3

3

Making decisions and repeating

if

```
fn main() {
  let temp = 18
  if temp > 25 {
    print("hot")
  } else if temp > 15 {
    print("mild")
  } else {
    print("cold")
  }
}
```

```
OUTPUT
mild
```

The condition must be a `bool`. Plank will not treat `0` or an empty string as false; say what you mean with `n == 0` or `s == ""`.

`if` can also produce a value. Each branch holds one expression, and both sides must have the same type:

```
fn main() {
  let n = 140
  let size = if n > 100 { "big" } else if n > 5 { "medium" } else {
"small" }
  print(size)
}
```

```
OUTPUT
big
```

Combine conditions with `and`, `or` and `not`. They are words, not symbols, so they read out loud the way they work.

for

`for` walks over a range of numbers, a list, the keys of a dict or the characters of a string:

```
fn main() {
  for i in 0..3 {
    print("lap", i)
  }
  for name in ["ada", "grace"] {
    print("hi", name)
  }
  for i, c in "abc" {
    print(i, c)
  }
}
```

```
OUTPUT
lap 0
lap 1
lap 2
hi ada
hi grace
0 a
1 b
2 c
```

`0..3` counts 0, 1 and 2. It stops before the end, the same as a slice, so `0..len(xs)` visits every index of `xs`. Figure 3-1 draws it.

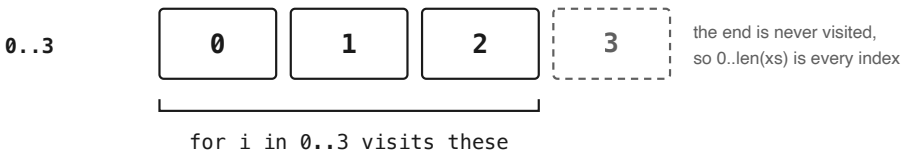


Figure 3-1: *A range includes its start and stops just before its end.*

while

`while` repeats as long as its condition holds. `break` leaves the loop early, `continue` skips to the next time round:

```
fn main() {
  var n = 27
  var steps = 0
  while n != 1 {
    if n % 2 == 0 {
      n = n / 2
    } else {
      n = 3 * n + 1
    }
    steps += 1
  }
  print("27 reaches 1 in", steps, "steps")
}
```

```
OUTPUT
27 reaches 1 in 111 steps
```

A whole program

Here is FizzBuzz, the program every language has to be able to write:

```
fn main() {
  for i in 1..16 {
    if i % 15 == 0 {
      print("FizzBuzz")
    } else if i % 3 == 0 {
      print("Fizz")
    } else if i % 5 == 0 {
      print("Buzz")
    } else {
      print(i)
    }
  }
}
```

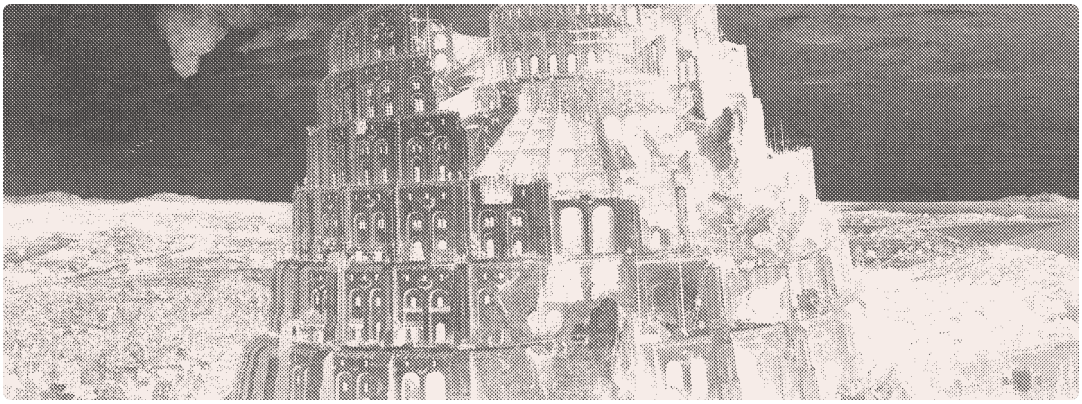
```
OUTPUT
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
```

Summary

`if` takes a real `bool` and can produce a value. `for` walks ranges, lists, dict keys and strings, and a range stops just before its end. `while` repeats with `break` and `continue` for the exits. That was enough to write FizzBuzz. Next you will package code into functions.

TRY IT YOURSELF

1. Print the numbers 1 to 20, but print `even` in place of every even one.
2. Write a `while` loop that halves 1000 until it drops below 1 and counts the steps.
3. Change the whole program at the end of the chapter so it stops early with `break`.



CHAPTER 4

4

Functions

A function takes values in and may hand one back. Every parameter has a type, and so does the result:

```
fn area(w: float, h: float) -> float {
  return w * h
}

fn main() {
  print(area(3.0, 4.0))
}
```

```
OUTPUT
12
```

A function with a result type must return on every path. If one branch could fall off the end, Plank refuses to compile it and tells you which function. That rules out a whole family of bugs where a function quietly returns nothing.

Functions can be declared in any order. `main` can call a function written below it.

Defaults and labels

Parameters can have default values, and any call can name its arguments:

```
fn greet(name: str, greeting: str = "hello", times: int = 1) {
  for i in 0..times {
    print(greeting + ", " + name)
  }
}

fn main() {
  greet("ada")
  greet("grace", times: 2, greeting: "hi")
}
```

```
OUTPUT
hello, ada
hi, grace
hi, grace
```

Labels make a call read like a sentence and let you skip the defaults you are happy with. Once one argument has a label, the ones after it need labels too.

More than one result

Return a tuple to hand back several values, and take it apart where you call:

```
fn divmod(a: int, b: int) -> (int, int) {
  return (a / b, a % b)
}

fn main() {
  let (q, r) = divmod(17, 5)
  print(q, r)
}
```

```
OUTPUT
3 2
```

Recursion

A function can call itself. This is the Fibonacci function, the one the landing page times:

```
fn fib(n: int) -> int {
  if n < 2 {
    return n
  }
  return fib(n - 1) + fib(n - 2)
}

fn main() {
  print(fib(20))
}
```

```
OUTPUT
6765
```

Every call that is not a base case makes two more. Figure 4-1 shows the whole tree for `fib(4)`.

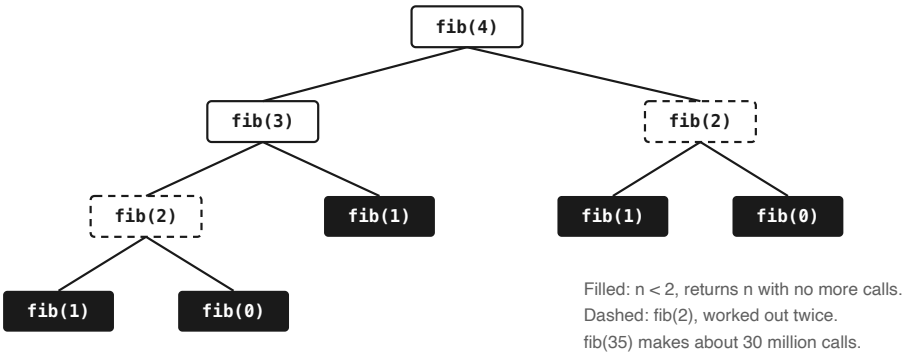


Figure 4-1: *The nine calls behind fib(4). The filled ones are base cases; the dashed fib(2) is worked out twice.*

NOTE

Because Plank compiles to machine code, `fib(35)` and its 30 million calls take a few hundredths of a second. The same function in a tree-walking interpreter takes seconds.

Summary

Every parameter and result has a type, and a function with a result must return on every path. Defaults and labels make calls read like sentences, tuples hand back more than one value, and recursion lets a function call itself. Next you will hold many values at once.

TRY IT YOURSELF

1. Write `max3(a, b, c)` that returns the biggest of three ints.
2. Give one of its parameters a default and call it both ways.
3. Write a recursive `sum_to(n)` and check it against `n * (n + 1) / 2`.



CHAPTER 5

5

Lists, dicts and sets

Lists

A list holds any number of values of one type, in order:

```
fn main() {
  let xs = [3, 1, 4]
  xs.append(1)
  print(xs, len(xs))
  print(xs[0], xs[-1])
  xs[1] += 10
  print(xs.pop(), xs)
  print(xs.sorted(), xs.sum(), xs.max())
  print(xs[1..], 4 in xs)
}
```

```
OUTPUT
[3, 1, 4, 1] 4
3 1
1 [3, 11, 4]
[3, 4, 11] 18 11
[11, 4] true
```

`max()` gives an `int?`, an int that might be missing, because an empty list has no largest item. Chapter 7 is about those. Here it prints as the number.

An empty list needs its type written once, because there is nothing in it to look at:

```
fn main() {
  var names: [str] = []
  names.append("ada")
  print(names)
}
```

```
OUTPUT
["ada"]
```

Lists are shared. Hand one to a function and the function sees and changes the same list, not a copy. `let` stops the name from pointing at a different list; it does not freeze the list. Figure 5-1 shows why.

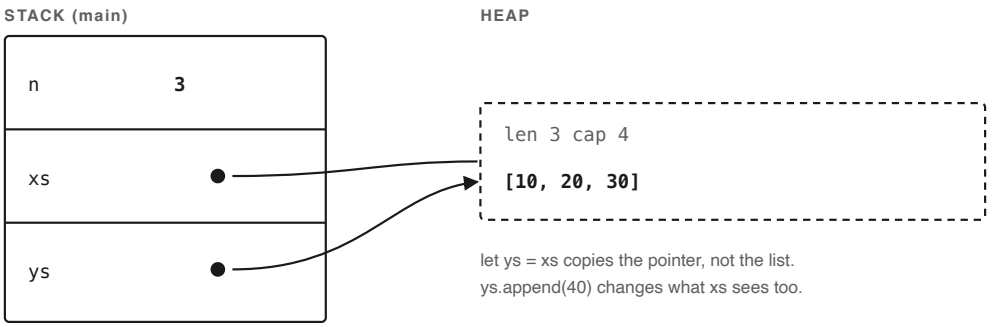


Figure 5-1: *Two names, one list. The names live on the stack and hold a pointer; the list itself lives on the heap.*

Doing something to every item

`map` makes a new list by running a function on each item. `filter` keeps the items a function says yes to. `reduce` folds them into one value:

```
fn main() {
  let xs = [1, 2, 3, 4, 5, 6]
  let evens = xs.filter(fn(x) => x % 2 == 0)
  let squares = evens.map(fn(x) => x * x)
  let total = squares.reduce(0, fn(acc, x) => acc + x)
  print(evens, squares, total)
}
```

```
OUTPUT
[2, 4, 6] [4, 16, 36] 56
```

`fn(x) => x % 2 == 0` is a function written in place. Chapter 8 covers them properly; for now, read `=>` as "gives".

Dicts

A dict maps keys to values. Keys are ints or strings, and the dict remembers the order they went in:

```
fn main() {
  let ages = ["ada": 36, "linus": 28]
  ages["grace"] = 85
  ages["linus"] += 1
  print(ages["ada"], "bob" in ages, ages.get("bob", 0))
  for name, age in ages {
    print(name, age)
  }
  print(ages.keys(), len(ages))
}
```

```
OUTPUT
36 false 0
ada 36
linus 29
grace 85
["ada", "linus", "grace"] 3
```

Reading a key that is not there stops the program and names the key. When a key might be missing, check with `in` or use `get` with a fallback.

Counting words

Lists and dicts together cover a surprising amount of everyday work. This counts words and prints the most common:

```
fn main() {
  let text = "the cat and the hat and the bat"
  let counts: [str: int] = [:]
  for w in text.split() {
    counts[w] = counts.get(w, 0) + 1
  }
  let words = counts.keys()
  words.sort(by: fn(a, b) => counts[a] > counts[b])
  for w in words[..2] {
    print(w, counts[w])
  }
}
```

```
OUTPUT
the 3
and 2
```

`sort(by:)` takes a function that says whether `a` goes before `b`. Sorting is stable, so ties keep their original order.

Sets

A `Set` holds each value once:

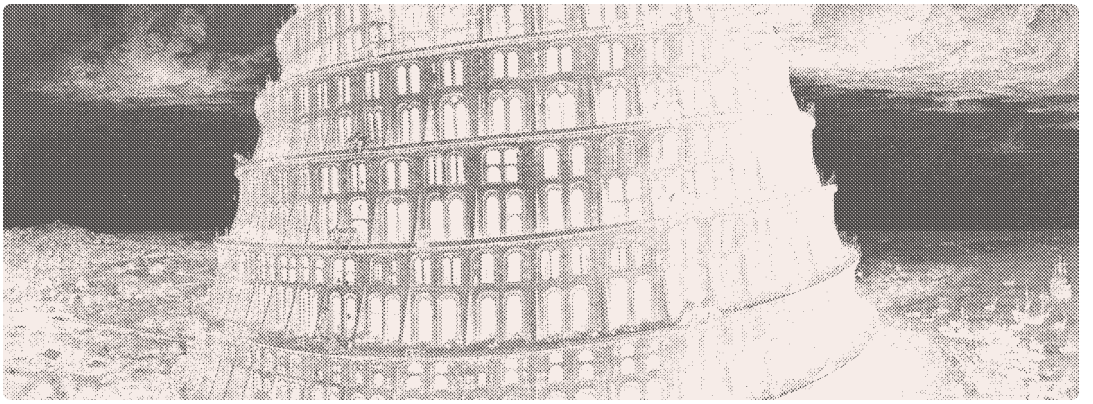
```
fn main() {
  let seen = set_of(["a", "b", "a", "c", "b"])
  seen.add("d")
  print(len(seen), "a" in seen, seen.items())
}
```

```
OUTPUT
4 true ["a", "b", "c", "d"]
```

Summary

Lists keep values in order, dicts map keys to values and remember their order, and sets hold each value once. `map`, `filter` and `reduce` work on every item, and `sort(by:)` sorts by anything. Collections are shared, not copied. Next you will make types of your own.

> **TRY IT YOURSELF**
>
> 1. Make a list of five names, sort it and print the first and last.
> 2. Change the word counter so it ignores case.
> 3. Use a set to find the letters two words have in common.



CHAPTER 6

6

Your own types

Structs

A struct gives a name to a group of fields, and can carry the functions that work on them. Struct names start with a capital letter:

```
struct Account {
  owner: str
  balance: int = 0

  fn deposit(amount: int) {
    self.balance += amount
  }

  fn can_pay(amount: int) -> bool {
    return amount <= self.balance
  }
}

fn main() {
  let a = Account(owner: "ada")
  a.deposit(50)
  print(a.balance, a.can_pay(80))
  print(a)
}
```

```
OUTPUT
50 false
Account(owner: "ada", balance: 50)
```

Build one by calling its name, with fields in order or by label. Fields with a default can be left out. Inside a method, `self` is the struct the method was called on; you never list it as a parameter.

`print` shows a struct the way you would build it. That one detail saves a lot of debugging.

Enums

An enum is a value that is exactly one of a fixed set of cases. Cases can carry values of their own:

```
enum Shape {
  circle(r: float)
  rect(w: float, h: float)
  dot

  fn area() -> float {
    match self {
      .circle(r) { return 3.14159 * r * r }
      .rect(w, h) { return w * h }
      .dot { return 0.0 }
    }
  }
}

fn main() {
  let shapes = [Shape.circle(1.0), Shape.rect(w: 2.0, h: 3.0), Shape.dot]
  for s in shapes {
    print(s, s.area())
  }
}
```

```
OUTPUT
circle(r: 1) 3.14159
rect(w: 2, h: 3) 6
dot 0
```

Under the hood every `Shape` is the same size: a number saying which case it is, called the tag, and enough room for the biggest case's values. Figure 6-1 shows the three shapes from this program.

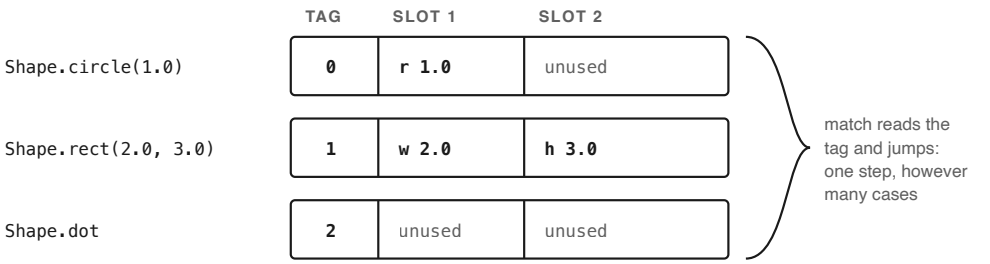


Figure 6-1: *An enum value is a tag and a row of slots. match looks at the tag and jumps straight to the right arm.*

match

`match` picks the arm that fits the value and can pull out what a case carries. The part that matters most: Plank checks that you covered every case. Add a `triangle` to `Shape` and forget to handle it, and the program will not compile:

```
shapes.pk:7: match on Shape misses .triangle
```

That is the difference between finding a bug when you write it and finding it when a user does.

`match` also works on numbers and strings, with `else` for the rest, and it can produce a value:

```
fn main() {
  for code in [200, 404, 500] {
    let word = match code {
      200 { "ok" }
      404 { "not found" }
      else { "trouble" }
    }
    print(code, word)
  }
}
```

```
OUTPUT
200 ok
404 not found
500 trouble
```

Types with parameters

A struct or function can work for any type, written as a parameter in angle brackets:

```
struct Stack<T> {
  items: [T] = []

  fn push(x: T) {
    self.items.append(x)
  }

  fn top() -> T? {
    return self.items.last()
  }
}

fn main() {
  let s = Stack<str>()
  s.push("a")
  s.push("b")
  print(s.top() ?? "empty", len(s.items))
}
```

```
OUTPUT
b 2
```

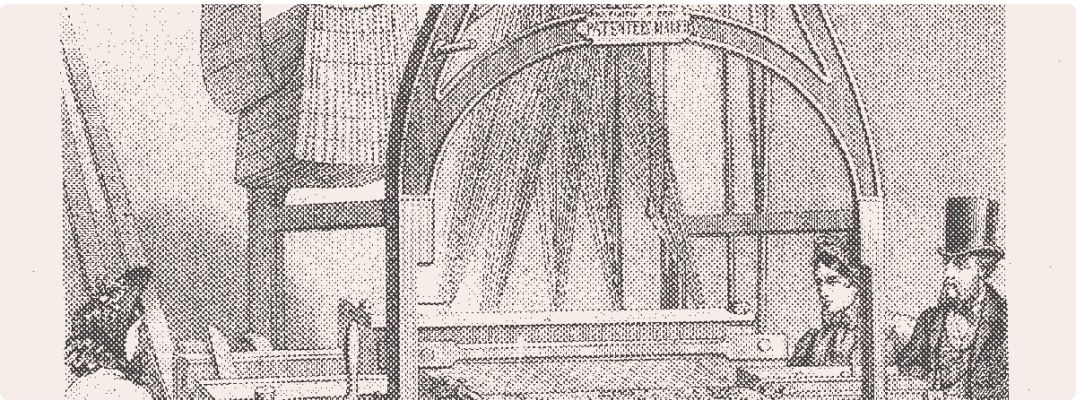
Plank compiles a separate version for each type you use, so a `Stack<str>` runs as fast as one written by hand for strings.

Summary

Structs group named fields and carry methods. Enums are exactly one of a set of cases, and `match` refuses to compile if you forget one. Type parameters let one struct or function work for any type at full speed. Next you will handle values that might not be there.

TRY IT YOURSELF

1. Add a `withdraw` method to `Account` that refuses to go below zero.
2. Add a `triangle` case to `Shape` and watch `match` refuse to compile until you handle it.
3. Give `Stack` a `pop` method, push three numbers and pop them all.



CHAPTER 7

When things are missing or wrong

Optionals

Some answers might not exist: the position of a word that is not in a list, the value of a key that is not in a dict. Plank writes that as an optional. `int?` means an int or `nil`.

```
fn find(xs: [str], want: str) -> int? {
  for i, x in xs {
    if x == want {
      return i
    }
  }
  return nil
}

fn main() {
  let langs = ["swift", "ruby", "plank"]
  if let i = find(langs, "plank") {
    print("found at", i)
  }
  print(find(langs, "cobol") ?? -1)
}
```

```
OUTPUT
found at 2
-1
```

You cannot do arithmetic on an `int?` directly; you have to say what happens when it is `nil`. There are three ways:

- `if let i = maybe { ... }` runs the block only when there is a value, with `i` as a plain `int` inside
- `maybe ?? fallback` gives the value, or the fallback
- `maybe!` gives the value, or stops the program with the line number if there is none

Nothing that is not marked with `?` can ever be `nil`. When you see a plain `int`, it is always there. Figure 7-1 shows what an optional is in memory, and the three ways to open one.

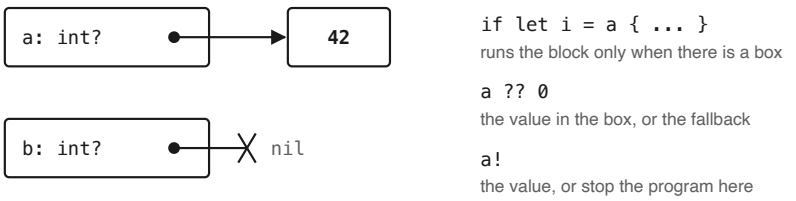


Figure 7-1: *An int? is a pointer to a box holding the int, or no box at all.*

Errors that stop the program

Some mistakes only show up when the program runs: reading past the end of a list, a missing dict key, `int("forty")`. Plank stops and tells you exactly where:

```
prog.pk:3: int() got text that is not a whole number
```

try and catch

Any of those can be caught. `throw` raises your own:

```
fn parse_age(text: str) -> int {
  let n = int(text)
  if n < 0 {
    throw "\(n) is not an age"
  }
  return n
}

fn main() {
  for t in ["36", "-4", "forty"] {
    try {
      print(parse_age(t))
    } catch err {
      print("no good:", err)
    }
  }
}
```

```
OUTPUT
36
no good: -4 is not an age
no good: int() got text that is not a whole number
```

Result

Sometimes failing is a normal answer, not a surprise. Then return a `Result<T>`: `ok(value)` or `err(message)`. A `?` after a call takes the value out, or hands the error straight back to whoever called you:

```
fn half(n: int) -> Result<int> {
  if n % 2 != 0 {
    return Result.err("\(n) is odd")
  }
  return Result.ok(n / 2)
}

fn quarter(n: int) -> Result<int> {
  return Result.ok(half(half(n)?)?)
}

fn main() {
  print(quarter(8).value())
  print(quarter(6).error() ?? "fine")
  print(quarter(6).or_else(0))
}
```

```
OUTPUT
2
3 is odd
0
```

NOTE

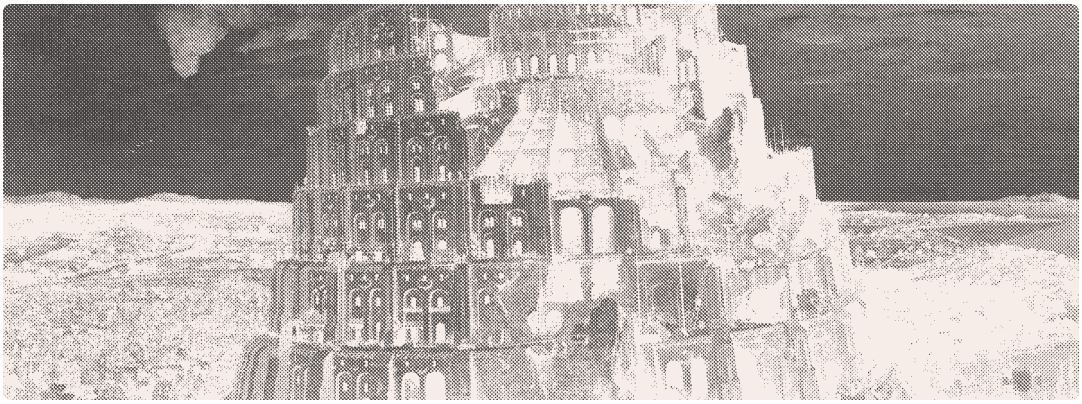
A rule of thumb: throw for bugs and broken input, return a `Result` when the caller is expected to handle the failure.

Summary

An optional `T?` is either a value or `nil`, and you must say what happens in both cases. Run-time errors stop the program with a line number unless you catch them with `try`. `Result` makes failure an ordinary answer, and `?` passes it up. Next you will pass functions around as values.

TRY IT YOURSELF

1. Write `find(xs, x) -> int?` that returns the index or `nil`.
2. Use `??` to print `none` when it finds nothing.
3. Wrap a division by zero in `try` and print the message you catch.



CHAPTER 8

8

Functions as values

A function is a value like a number or a string. You can store one in a name, pass it to another function, or return it:

```
fn twice(f: fn(int) -> int, x: int) -> int {
  return f(f(x))
}

fn double(n: int) -> int {
  return n * 2
}

fn main() {
  print(twice(double, 5))
  print(twice(fn(x) => x + 3, 1))
}
```

```
OUTPUT
20
7
```

`fn(x) => x + 3` is a closure, a function with no name written right where it is used. `=>` with one expression is the short form. Plank works out the type of `x` from where the closure goes: `twice` says it takes `fn(int) -> int`, so `x` is an `int`.

For more than one line, use a block:

```
fn main() {
  let shout = fn(s: str) -> str {
    let loud = s.upper()
    return loud + "!"
  }
  print(shout("plank"))
}
```

```
OUTPUT
PLANK!
```

Closures remember

A closure keeps the variables it uses. They are shared, not copied, so the closure sees later changes, and a closure handed out of a function keeps its variables alive:

```
fn make_counter() -> fn() -> int {
  var n = 0
  return fn() -> int {
    n += 1
    return n
  }
}

fn main() {
  let next = make_counter()
  next()
  next()
  print(next())
}
```

```
OUTPUT
3
```

How can `n` still exist after `make_counter` has returned? Plank noticed that a closure uses it and put it on the heap instead of the stack. Figure 8-1 shows what `next` really holds.

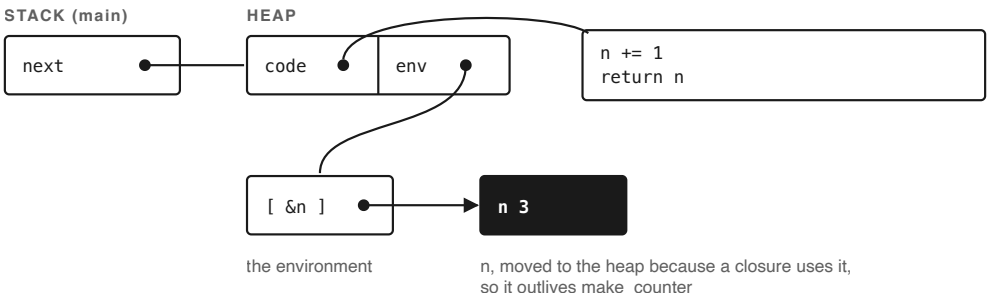


Figure 8-1: *A closure is two words: where its code is, and an environment that points at the variables it uses.*

Sorting by anything

Closures are how you tell library functions what you want. Sort people by age:

```
struct Person {
  name: str
  age: int
}

fn main() {
  let people = [Person("ada", 36), Person("linus", 28), Person("grace",
85)]
  people.sort(by: fn(a, b) => a.age < b.age)
  print(people.map(fn(p) => p.name))
}
```

```
OUTPUT
["linus", "ada", "grace"]
```

Summary

Functions are values you can store, pass and return. A closure is a function written in place that keeps the variables it uses, even after the function that made it has returned. Closures are how you tell `sort`, `map` and friends what you want. Next you will talk to files, the shell and the web.

> **TRY IT YOURSELF**
>
> 1. Write `make_adder(n)` that returns a function adding `n` to its argument.
> 2. Sort a list of words by their length, longest first.
> 3. Make a counter closure that returns 1, 2, 3 on each call.



CHAPTER 9

Talking to the world

Plank is meant for real scripts, so files, folders, the shell, JSON and HTTP are built in.

Files

```
fn main() {
  write_file("notes.txt", "milk\neggs\n")
  append_file("notes.txt", "bread\n")
  let text = read_file("notes.txt") ?? ""
  for line in text.trim().split("\n") {
    print("-", line)
  }
  print(exists("notes.txt"), remove_file("notes.txt"),
exists("notes.txt"))
}
```

```
OUTPUT
- milk
- eggs
- bread
true true false
```

`read_file` gives a `str?`: `nil` when the file cannot be read, so you decide what that means. `list_dir`, `mkdir`, `is_dir` and `cwd` cover folders.

Arguments and input

`args()` is the list of words after the program's name: `plank run greet.pk ada grace` gives `["ada", "grace"]`. `input("name? ")` asks for a line. `read_stdin()` takes everything piped in, so a Plank program fits in a shell pipeline.

The shell

`run` runs a command and gives back what it printed. `status()` is its exit code. `quote` makes any text safe to pass as one argument:

```
fn main() {
  let name = "two words"
  let out = run("echo " + quote(name)).trim()
  print(out, status())
}
```

```
OUTPUT
two words 0
```

JSON

`json_parse` reads JSON into a `Json` value. The helpers pull typed values out of it, and `.text()` writes it back:

```
fn main() {
  let doc = json_parse("{\"name\": \"plank\", \"tags\": [\"small\",
\"fast\"], \"stars\": 42}")
  print(doc.get("name").str(), doc.get("stars").int())
  for t in doc.get("tags").items() {
    print("tag:", t.str())
  }
  print(Json.object(["ok": Json.bool(true)]).text())
}
```

```
OUTPUT
plank 42
tag: small
tag: fast
{"ok":true}
```

HTTP

`http("GET", url)` sends a request and gives back the body; `status()` has the HTTP code afterwards. With a body and headers it posts:

```
let reply = http("POST", "https://example.com/api", "{\"q\": 1}",
["content-type": "application/json"])
if status() != 200 {
  print("failed:", status())
}
```

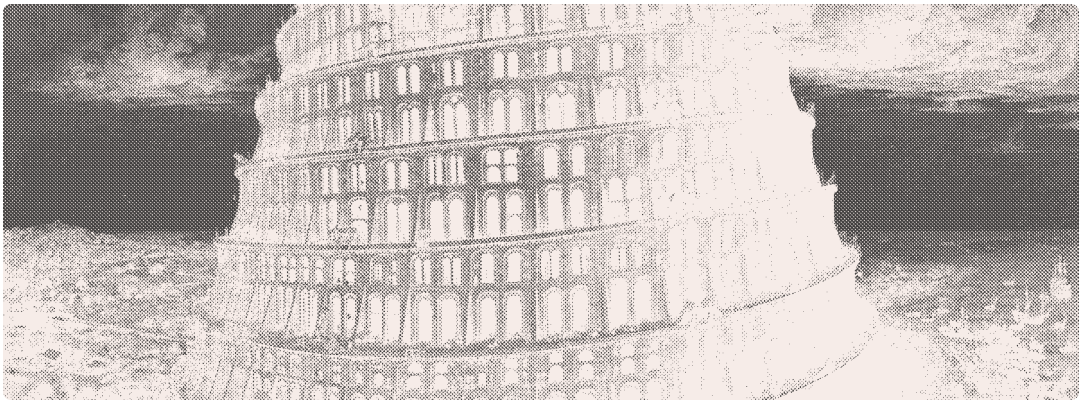
Together these cover most of what people reach for Python to do: read some files, call a command, hit an API, write a report.

Summary

Files, folders, arguments, standard input, the shell, JSON and HTTP are all built in, which covers most everyday scripting. Missing files come back as `nil`, so you decide what that means. Next you will split a program across files.

TRY IT YOURSELF

1. Write a script that counts the lines in a file named on the command line.
2. Run `ls` from Plank and print how many entries it found.
3. Read a JSON file and print one field from it.



CHAPTER 10

10

Programs in more than one file

import

`import` pulls in everything another file defines. Paths are relative to the file doing the importing:

```
import "lib/money.pk"
```

For bigger programs, give the file a name so its functions do not collide with yours:

```
import "lib/shapes.pk" as shapes

fn main() {
  print(shapes.area_of(shapes.Kind.square(3.0)))
}
```

Inside `shapes.pk` nothing changes; it uses its own names plainly. A file is read once, however many times it is imported.

Packages

An import that starts with `github.com/` is a package. Plank fetches the file the first time, keeps it under `~/.plank/pkg`, and reads it from there after that:

```
import "github.com/nulljosh/plank@v3.1.0/examples/lib/money.pk" as money
```

`@v3.1.0` pins a tag.

NOTE

There is no registry and no lock file. A package is a file in a git repository, and the tag says which version of it you mean.

Tests

A test in Plank is a program that prints `ok`. Put checks in it with `assert`:

```
fn add(a: int, b: int) -> int {
  return a + b
}

fn main() {
  assert(add(2, 2) == 4, "two and two")
  assert(add(-1, 1) == 0)
  print("ok")
}
```

```
OUTPUT
ok
```

`plank test` runs every `.pk` in `tests/` and reports any that did not print `ok`. Plank's own test suite works this way.

Tidy code

`plank fmt` lays out your files the house way: two-space indents, one statement per line, blocks opened on the same line. `plank fmt --check` only reports, which is what you want in CI. Every program in this book passes it.

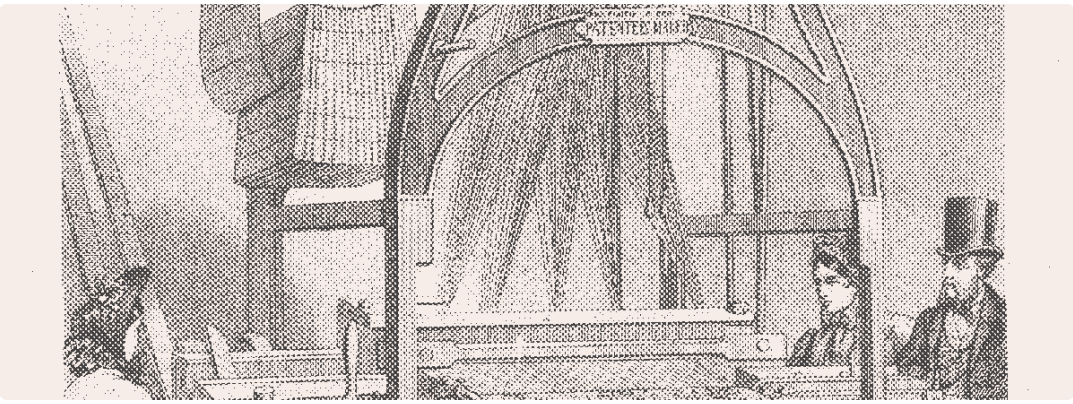
Shipping

`plank build app.pk` makes a native binary. On Linux, `plank build app.pk --static` makes one with no shared libraries at all, which you can copy to any Linux machine and run.

Summary

`import` pulls in another file, and `as` keeps its names under one prefix. A package is a file in a git repository, pinned by tag. Tests are programs that print `ok`, `plank fmt` keeps code tidy, and `plank build` ships a binary. Next you will script a model.

> **TRY IT YOURSELF**
>
> 1. Move a struct into its own file and import it.
> 2. Import the same file `as` a module and call one function through the module name.
> 3. Write a test file with three `assert`s and run `plank test` on its folder.



CHAPTER 11

11

Scripting intelligence

Python scripts computers. Plank is built to script models too. Everything here lives in `lib/ai.pk`, an ordinary Plank file you can read.

Asking a model

```
import "lib/ai.pk" as ai

fn main() {
  let m = ai.model("claude-haiku-5-5", "Answer in one line.")
  print(m.ask("What is a compiler?"))
}
```

`ai.model` talks to Anthropic, with the key in `ANTHROPIC_API_KEY`. `ai.local(name, url)` talks to any OpenAI-compatible server on your own machine, such as Ollama. `ai.mock()` needs no network at all and answers `echo:` plus your question, which is what tests use so they print the same thing every time:

```
import "lib/ai.pk" as ai

fn main() {
  let m = ai.mock()
  print(m.ask("hello"))
}
```

```
OUTPUT
echo: hello
```

`m.ask_json(question)` asks for JSON back and gives you a `Json` value, retrying once if the model's reply does not parse.

Tools

A model gets much more useful when it can call your code. Put `tool` in front of a function, and the comment right above it becomes the description the model sees:

```
# Adds two whole numbers
tool fn add(a: int, b: int) -> int {
  return a + b
}
```

Plank writes `add_tool()` for you, with a schema read off the function's signature. The schema can never drift from the code because it is the code.

Agents

An agent is a loop: ask the model, run the tools it asks for, hand back the results, repeat until it answers in plain words. Figure 11-1 draws one turn of it.

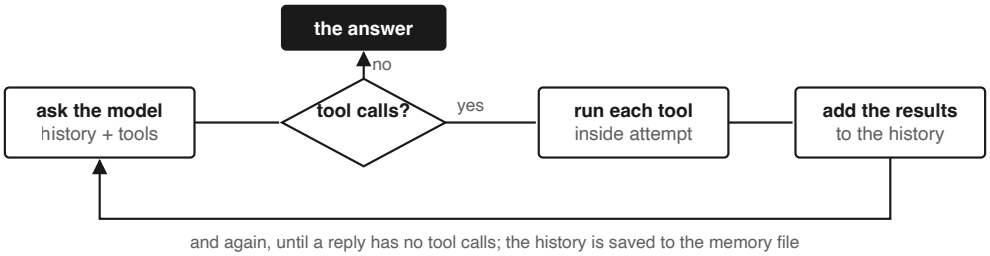


Figure 11-1: *The agent loop. Each pass through the bottom arrow is one more question to the model.*

```
let bot = ai.agent(ai.model("claude-haiku-5-5"), [add_tool()],
"memory.json")
print(bot.run("What is 20 + 22?"))
```

The third argument is a file. The agent saves the conversation there after each run and reads it back at the start of the next, so it remembers. `keep: 20` caps how much it keeps.

NOTE

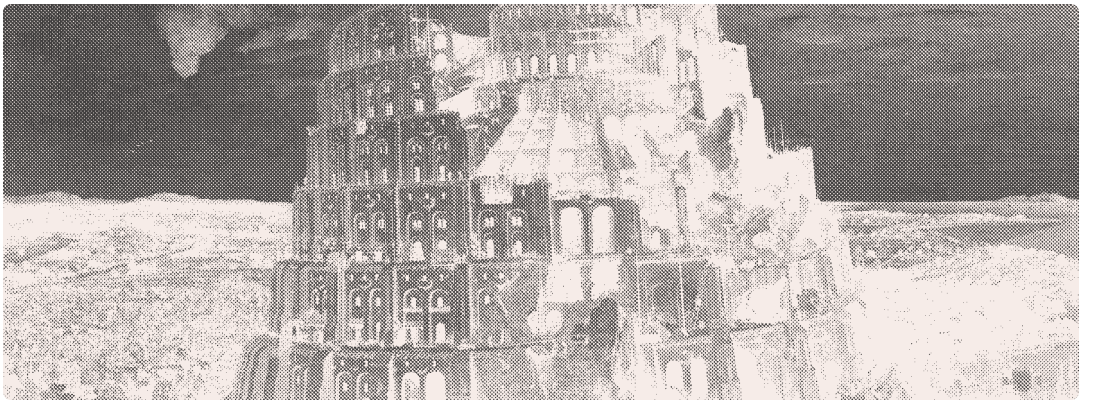
The tool's comment is the only thing telling the model when to use it. "Adds two whole numbers" works. "math" does not.

Summary

`lib/ai.pk` is plain Plank that talks to Anthropic, local models or a mock. `tool fn` turns a function and its comment into something a model can call, and an agent loops until the model answers in words, remembering the conversation in a file. Next you will open the compiler itself.

TRY IT YOURSELF

1. Turn one of your earlier functions into a `tool fn` with a comment above it.
2. Run the agent with the mock model and change what the mock says.
3. Point the program at a local model and ask it to use your tool.



CHAPTER 12

12

Inside the compiler

You can now write Plank. This last chapter is about how Plank works, following the road from Figure 1-1 one step at a time, because the point of the language is that you can understand all of it. Open `plank.py` next to this chapter. It reads top to bottom in the same order as the steps below.

Four steps

1. **Lex.** One regular expression and a small loop cut your text into tokens: names, numbers, strings, symbols. `let x = 1 + 2` becomes `let`, `x`, `=`, `1`, `+`, `2`. String interpolation is handled here too, by lexing what is inside `\( )` as code.
2. **Parse.** A recursive-descent parser turns tokens into a tree. Each piece of syntax has a small method on `Parser`: `stmt`, `if_`, `match`, `struct`, `fn` and so on. Expressions use a Pratt parser, which gets operator precedence right with one small table instead of a function per level.
3. **Generate.** One walk over the tree emits LLVM IR, a portable assembly language, and checks types as it goes. This is the biggest part of the file. When it sees `a +`

`b`, it looks at the types of both sides, refuses if they differ, and emits an integer add, a float add or a call to the string joiner.

4. **Link.** LLVM optimizes the IR and turns it into machine code for your processor. Your system's C compiler links that with a small runtime and the C library into a binary.

See it for yourself

`plank tokens` prints what the lexer made of a file, one token per line with its line number:

```
plank tokens examples/fib.pk
```

and `plank ast` the tree the parser built from them:

```
plank ast examples/fib.pk
```

`plank emit` prints the IR for any program:

```
plank emit examples/fib.pk
```

You will see `fib` as a function that compares, branches and calls itself, much like the source. Run it on a program with a list or a closure to see how those are built out of plain memory and function pointers.

Where values live

Ints, floats and bools live in registers and on the stack, like in C. Strings, lists, dicts, structs, enums and closures live on the heap and are passed around as pointers, which is why they are shared, as Figure 5-1 showed.

Nothing is ever freed by hand. A garbage collector runs now and then: it finds every object the program can still reach, starting from the stack and globals, marks them, and frees the rest. Figure 12-1 shows one collection.

!Mark and sweep. Anything a root can reach, directly or through other blocks, survives. It never moves anything, which keeps it short. The test suite runs everything a second time with the collector firing every 4KB, to shake out any place that forgets to tell it about a pointer.

Why LLVM

Writing a code generator for every processor is the part of a compiler that takes years. LLVM already did it, and does register allocation and optimization better than a small project could. Plank's job ends at producing correct IR, which is a job one file can do. That is the whole trick: pick the one piece you can borrow without hiding how anything works.

Where to go next

- Appendix A is the reference: every rule of the language on one page. It is `docs/SPEC.md`, printed here.
- Appendix B maps `plank.py`, section by section. It is `docs/ARCHITECTURE.md`, printed here.
- `apps/calc.pk` is a calculator with its own tokenizer and parser, written in Plank. It is the same four steps in miniature.
- `compiler/plank.pk` is Plank's compiler written in Plank. It already has the lexer and parser from this chapter and grows a step per release until it can compile itself.

Change something. Add a built-in, make an error message friendlier, add a case to `match`. Run `python3 test.py`. That is how every feature in this book got there.

Summary

Plank's compiler is four steps in one file: lex, parse, generate and link. `plank tokens`, `plank ast` and `plank emit` let you watch each one. Simple values live in registers and on the stack, everything else on the heap, and a mark-and-sweep collector frees what nothing reaches. LLVM does the part that takes years, so the rest fits in a file you can read.

> **TRY IT YOURSELF**
>
> 1. Run `plank tokens`, `plank ast` and `plank emit` on the same small program and match each line of output to Figure 1-1.
> 2. Find `def lex` in `plank.py` and read it start to finish. How long did it take?
> 3. Make one error message in `plank.py` friendlier and run `python3 test.py`.



APPENDIX A

The language reference

Everything the compiler accepts, in one page.

Program

A program is a list of functions, structs and enums, and any number of `import "other.pk"` lines, which pull in everything another file defines. Paths are relative to the importing file, and a file is only read once however many times it is imported. Errors and runtime messages name the file they come from.

`import "lib/shapes.pk" as shapes` keeps the file's names under its own name instead: `shapes.area_of(k)`, `shapes.Shape("box", shapes.Kind.square(3.0))`, a type written `shapes.Shape`, and `shapes.area_of` as a value. Inside the module nothing changes; it refers to its own names plainly. Two files can then both define `parse`.

A path that starts with `github.com/` is a package: `import "github.com/nulljosh/plank@v3.1.0/examples/lib/money.pk" as money` fetches that file from GitHub the first time, keeps it under `~/.plank/pkg`, and from then on reads

it from there. `@tag` pins a tag or branch; without one it is `main`. A file a package imports relatively comes from the same repo. Delete `~/.plank/pkg` to fetch everything again.

A program's top level is a list of functions. It needs a `fn main()` with no parameters and no return type. Statements end at the newline. Comments start with `#`.

```
fn add(a: int, b: int) -> int {
  return a + b
}

fn main() {
  print(add(2, 3))
}
```

Parameters can have defaults, and any call can name its arguments. Once one argument has a label, the ones after it need labels too.

```
fn greet(name: str, greeting: str = "hello", times: int = 1) { ... }

greet("plank")
greet("swift", times: 2, greeting: "hi")
```

A default is worked out fresh at every call, so `xs: [int] = []` gives each call its own list. It cannot use the caller's variables.

A function can take type parameters in angle brackets and use them anywhere a type goes. Plank works out what they are from the arguments at each call and compiles that version once.

```
fn first<T>(xs: [T]) -> T {
  return xs[0]
}
fn largest<T>(xs: [T], before: fn(T, T) -> bool) -> T { ... }

first([7, 8])            # T is int
first(["x"])             # T is str
largest(xs, fn(a, b) => a < b)   # the closure's types come from T
```

A type parameter has to appear in some parameter, so the call can pin it down; `fn make<T>() -> T` is an error at the call.

Functions and structs can be declared in any order. A function with a return type must return on every path; the compiler refuses one that can fall off the end.

Types

Type	Machine	Literals
`int`	signed 64-bit	`42`, `0`
`float`	64-bit double	`3.14`, `2.0`, `1.5e3`
`bool`	1 bit	`true`, `false`
`[T]`	list of any type, shared	`[1, 2, 3]`, `["a"]`, `[[1], [2, 3]]`
`[K: V]`	dict, int or str keys, shared	`["a": 1]`, `[1: "one"]`, `[:]`
`T?`	a `T` or `nil`	`nil`, or any `T`
`fn(A, B) -> R`	a function or closure	`fn(x: int) => x * 2`, `double`
`(A, B)`	a tuple, two or more values of any types	`(1, "one")`, `("ada", 36, true)`
`str`	C string, UTF-8	`"hello\n"`, `"世界"`, `"n is \(n)"`

String escapes are `\n`, `\t`, `\"`, `\\` and `\0`. Anything else after a backslash is an error, except `\(`, which starts an interpolation: `"\(name) has \(len(name)) letters"` drops the value of any expression into the string. Ints, floats and bools turn into text the same way `print` shows them. The expression inside can hold its own strings. A float literal needs digits on both sides of the dot.

There are no implicit conversions. `1 + 2.0` is an error. Use `int(x)`, `float(x)` and `str(x)`: float to int truncates, bool to int gives 0 or 1. `int("42")` and `float(" 2.5 ")` read numbers out of text and stop the program with the line number if the text is not one.

Variables

`let` binds once. `var` can be reassigned. Both infer the type from the value; you can state it with a colon.

```
let name = "plank"
var count: int = 0
count = count + 1
```

`x += 1` is `x = x + 1`. The same goes for `-=`, `*=`, `/=` and `%=`, and `+=` joins strings.

Blocks open a scope. A name cannot be declared twice in the same scope, but an inner block can shadow an outer one.

Tuples

A tuple groups a few values without naming a struct. Read parts with `.0`, `.1`; take it apart with `let (a, b) = ...`; a function returns several values by returning one.

```
fn divmod(a: int, b: int) -> (int, int) {
  return (a / b, a % b)
}
let (q, r) = divmod(17, 5)   # 3 and 2
let pair = ("ada", 36)
print(pair.0, pair, ((1, 2), 3).0.1)   # ada ("ada", 36) 2
```

Tuples print the way you write them and work in lists, dicts, closures and generics. `d.items()` is a dict as `[(K, V)]` in order, `xs.enumerate()` is `[(int, T)]`, and `zip(xs, ys)` pairs two lists as far as the shorter one goes.

String tools

```
"a, b ,c".split(",")          # ["a", " b ", "c"]
"the  quick fox".split()      # ["the", "quick", "fox"], on runs of
whitespace
["a", "b"].join(" & ")        # "a & b"
"  hi  ".trim()               # "hi"
"Plank".upper(), "Plank".lower()
"plank lang".replace("lang", "compiler")
"plank".find("an")            # 2, an int?, nil when it is not there
"plank".starts_with("pl"), "plank".ends_with("nk")
"héllo"[1]                    # "é", a one-character string
"héllo"[1..3], "héllo"[..2], "héllo"[-2..]
for c in "héllo" { ... }      # one character at a time
"héllo".chars()               # ["h", "é", "l", "l", "o"]
fixed(3.14159, 2)             # "3.14"
```

Regular expressions are POSIX extended, straight from the C library, so `[0-9]+` and `([a-z]+)=([a-z]+)` work and `\d` does not:

```
"plank 1.8".matches("[0-9]+\\.[0-9]+")        # true
"a1b22".find_all("[0-9]+")                    # ["1", "22"]
"key=value".captures("([a-z]+)=([a-z]+)")     # ["key=value", "key",
"value"], a [str]?, nil when nothing matches
"a-b_c".replace_all("[-_]", " ")              # "a b c"
```

A pattern that does not compile throws `bad pattern ...`. Indexes and lengths count characters, not bytes. Strings never change in place; every tool returns a new one. `upper` and `lower` change ASCII letters and leave the rest alone.

Lists

A list holds one type. Write the type as `[int]`, `[str]`, `[[float]]`. An empty list needs its type spelled out once: `let xs: [int] = []`.

```
let xs = [3, 1, 4]
xs.append(1)        # [3, 1, 4, 1]
print(xs[0], xs[-1]) # 3 1, negative counts from the end
xs[1] += 10         # [3, 11, 4, 1]
let last = xs.pop() # 1
print(len(xs), xs)  # 3 [3, 11, 4]
for x in xs {
  print(x)
}
```

Lists are shared, like Python's: pass one to a function and the function sees and changes the same list. `let` stops the name from pointing at a new list, not the list from changing. An index past either end stops the program with the line number. `print` and `str` show lists the way you would type them, strings in quotes. Lines can break freely inside `[ ]`. `xs[1..3]` is a new list of items 1 and 2; either end can be left off, negative ends count from the back, and ends past the list are clamped, so a slice never stops the program.

Dicts

A dict maps int or str keys to values of one type, and remembers the order keys went in, like Python's.

```
let ages = ["ada": 36, "linus": 28]
ages["grace"] = 85              # add or replace
ages["linus"] += 1
print(ages["ada"])              # 36
print("bob" in ages)            # false
print(ages.get("bob", 0))       # 0, the default when the key is missing
ages.remove("ada")
for name in ages {              # keys, in the order they went in
  print(name, ages[name])
}
print(ages.keys(), ages.values(), len(ages))
```

Reading a key that is not there stops the program and names the key; use `in` or `get` when it might be missing. An empty dict is `[:]` and needs its type once: `let d: [str: int] = [:]`. Dicts are shared like lists.

`in` also works on lists, `3 in [1, 2, 3]`, and strings, `"lan" in "plank"`.

Sets

`Set<T>` holds each int or str once, in the order it first arrived. It is written in Plank itself and compiled in when a program mentions it.

```
let a = set_of([3, 1, 2, 3])     # Set[3, 1, 2]
a.add(9)
a.remove(1)
print(2 in a, len(a), a.items())  # true 3 [3, 2, 9]
for x in a { ... }
a.union(b), a.intersect(b), a.minus(b), a.text()
```

Structs

A struct groups named fields and the functions that work on them. Names start with a capital letter.

```
struct Account {
  owner: str
  balance: int = 0

  fn deposit(amount: int) {
    self.balance += amount
  }
}

let a = Account(owner: "josh")  # or Account("josh", 0)
a.deposit(50)
print(a.balance, a)  # 50 Account(owner: "josh", balance: 50)
```

A struct can take type parameters too: `struct Stack<T> { items: [T] = [] ... }`. Use it as `Stack<int>()`, or let the fields say what `T` is: `Stack(items: [1, 2])`. Methods see `T` like any type, and a method can return another instance, `fn flipped() -> Pair<B, A>`. Each distinct `Stack<...>` is its own type and prints with its parameters.

Build one by calling its name with the fields in order or by label; fields with a default can be left out. Methods get `self` without asking for it and reach fields through `self.x`. Like lists, structs are shared: a function that changes `p.x` changes it for everyone holding `p`, and `let` only fixes which struct the name points at. `print` and `str` show a struct the way you would build it.

Enums and match

An enum is one of a fixed set of cases, and a case can carry values. Methods work the same as on structs.

```
enum Shape {
  circle(r: float)
  rect(w: float, h: float)
  dot

  fn area() -> float {
    match self {
      .circle(r) { return 3.14159 * r * r }
      .rect(w, h) { return w * h }
      .dot { return 0.0 }
    }
  }
}

let s = Shape.rect(w: 2.0, h: 3.0)   # a case with values is built like a
struct
print(s, s.area())                    # rect(w: 2, h: 3) 6
let d = Shape.dot                     # a case without values needs no ()
```

`match` picks the first arm that fits. On an enum, each arm names cases with a leading dot and can bind the carried values to new names. The compiler refuses a match that forgets a case, and one with an `else` that can never run. On ints, floats, bools and strings, arms list constants, several to an arm with commas, and `else` catches the rest. `==` compares enums whose cases carry nothing; for the rest, use `match`. Enums are shared and print as their case.

Optionals

`int?` is an int or `nil`. Any type can be optional, and a plain value goes wherever its optional is expected. Nothing else can be `nil`, so a plain `int` is always there.

```
fn find(xs: [str], want: str) -> int? {
  for i in 0..len(xs) {
    if xs[i] == want { return i }
  }
  return nil
}

if let i = find(names, "ada") {   # i is a plain int in here
  print(i)
} else {
  print("not found")
}
let i = find(names, "bob") ?? -1  # the value, or the fallback
let j = find(names, "ada")!       # the value, or stop the program with
the line number
print(find(names, "x") == nil)    # true
```

`p?.next?.val` reads through optionals: the result is `nil` as soon as any link is, and otherwise the value as an optional. It works on methods too, `p?.describe()`. You cannot do math on an `int?` until you take the value out, and the error says how. The fallback after `??` only runs when it is needed. `d.get(k)` with one argument gives a `V?`. Optionals print as their value or `nil`. A struct field like `next: Node? = nil` is how you build a linked list.

Functions as values

A function is a value like any other. Write one inline with `fn`, with a block body, or with `=>` and a single expression whose type Plank works out.

```
let add = fn(a: int, b: int) => a + b
let shout = fn(s: str) -> str {
  return s + "!"
}
fn twice(f: fn(int) -> int, x: int) -> int {
  return f(f(x))
}
print(twice(fn(x) => x + 3, 1))   # 7; x is an int because twice says so
print(twice(double, 5))            # a named function works too
```

A block body needs no `->`: Plank reads the return type from the first `return`, and a body with no `return` is void. Parameter types can be left off whenever the place the function goes already says them: arguments, `let` with a type, `return`, and the list methods below. A closure keeps the variables it uses, shared, not copied: change the variable later and the closure sees it, and a closure returned from a function keeps its variables alive.

```
fn make_counter() -> fn() -> int {
  var n = 0
  return fn() -> int {
    n += 1
    return n
  }
}
```

Lists also have `first()` and `last()` (a `T?`, `nil` when empty), `index_of(x)` (an `int?`), `insert(i, x)`, `remove_at(i)`, `reverse()` in place and `reversed()` as a copy, `sum()` for numbers, and `min()` and `max()` for numbers and strings, a `T?`.

Lists take functions: `xs.map(fn(x) => x * 2)`, `xs.filter(fn(x) => x > 0)`, `xs.reduce(0, fn(acc, x) => acc + x)`. `xs.sort()` sorts ints, floats and strings in place and `xs.sorted()` returns a sorted copy; both take `by: fn(a, b) => a.age < b.age` for anything else, where the function says whether `a` goes first. Sorting is stable.

Operators

`==` and `!=` look inside: two lists are equal when their items are, dicts when they hold the same keys and values in any order, tuples and structs part by part, enums when the case and its values match, optionals when both are `nil` or both hold equal values. `xs == []` asks whether a list is empty. `in` and `index_of` use the same rule. Closures compare by identity.

From loosest to tightest: `or`, `and`, `not`, comparisons (`== != < > <= >=` and `in`), `??`, `+ -`, `* / %`, unary `-`. `and` and `or` short-circuit. Integer `/` and `%` by zero stop the program with the line number, or land in the nearest `catch`; floats follow IEEE and give infinity. Integer `/` truncates toward zero and `%` follows the sign of the left operand. Floats support all of these too. Strings join with `+` and compare with all six comparisons, alphabetically by byte. Bools compare with `==` and `!=` only.

Control flow

```
if x < 0 {
  print("negative")
} else if x == 0 {
  print("zero")
} else {
  print("positive")
}

while running {
  if done { break }
  if skip { continue }
}

for i in 0..10 {   # 0 through 9, the bounds are evaluated once
  print(i)
}
```

`for x in xs` walks a list from the front; `for i, x in xs` gives the index too, `for k, v in d` the key and value of a dict, `for i, c in s` the characters of a string with

their positions. `while let top = stack.last() { ... }` keeps going while the optional has a value, with the value bound inside. `if` is also a value: `let size = if n > 100 { "big" } else if n > 5 { "medium" } else { "small" }`. Each branch holds one expression, both sides need the same type, and `nil` on one side makes the result an optional. `if let` works there too: `let label = if let top = s.peek() { "top is \(top)" } else { "empty" }`. So does `match`, with one expression per arm: `let word = match light { .red { "stop" } .green, .yellow { "go" } }`, and the same rule that every case is covered.

Conditions must be `bool`. The `for` variable is an `int` and is a `let` inside the body. `else` can sit on the same line as the closing brace or on the next one. A `while true` with no `break` counts as never finishing, so a function can end with one and still satisfy the return check.

Built-ins

Call	Does
`print(a, b, ...)`	Prints its arguments separated by spaces, then a newline. `print()` is a blank line.
`str(x)`, `int(x)`, `float(x)`	Convert, see Types.
`len(x)`	Items in a list, or characters in a string, so `len("世界")` is 2.
`input(prompt)`	Prints the optional prompt, reads one line, drops the newline. Empty at end of input.
`min(a, b)`, `max(a, b)`, `abs(x)`	Two ints or two floats in, the same type out.
`sqrt`, `floor`, `ceil`, `round`, `pow(x, y)`	Float math from libm.
`fixed(x, digits)`	A float as text with that many digits after the point.
`args()`	The command-line arguments after the program: `plank run tool.pk a b` gives `["a", "b"]`.

Call	Does
`read_file(path)`	The whole file as a `str?`, `nil` if it cannot be read.
`write_file(path, text)`	Writes the file, `true` if it worked.
`read_stdin()`	Everything on standard input, as one string.
`env(name)`	An environment variable as a `str?`.
`random(lo, hi)`, `random()`	An int from `lo` to `hi` inclusive, or a float from 0 up to 1.
`time()`	Seconds since 1970 as a float, good for timing.
`sleep(seconds)`	Waits that long, a float.
`clock()`, `clock(fmt)`, `clock(fmt, at)`	The local time as text, `2026-10-01 18:30:00`, in any `strftime` layout, or for another moment `at` in seconds since 1970.
`parse_time(text)`, `parse_time(text, layout)`	Text to seconds since 1970 as a `float?`, by `%Y-%m-%d` or any `strptime` layout; `nil` when it does not fit. A day is `86400.0`, so dates add and subtract.
`exists(path)`, `is_dir(path)`	Whether something is there, and whether it is a folder.
`list_dir(path)`	The names inside a folder, sorted; empty when it cannot be read.
`mkdir(path)`, `remove_file(path)`	Make a folder, or delete a file; `true` when it worked.
`append_file(path, text)`	Adds to the end of a file, creating it if needed.
`cwd()`	The current folder.
`url_encode(s)`	`a b/c` as `a%20b%2Fc`, for building URLs.
`exit(code)`	Stops the program with that exit code.
`run(cmd)`	Runs a shell command, returns its output; `status()` has the exit code right after.

Call	Does
`quote(s)`	Shell-quotes one argument, so `run("ls " + quote(name))` is safe with any name.
`http(method, url, body?, headers?)`	Sends the request (through curl), returns the body; `status()` has the HTTP code, or a non-zero curl code if it never connected.
`json_parse(text)`	Text to a `Json` value, or a throw naming what was wrong.
`json_quote(s)`	A string as a JSON string literal.
`assert(cond, message)`	Stops the program, or lands in the nearest `catch`, when `cond` is false. The message is optional.

Floats print with 15 significant digits, so `0.1 + 0.2` shows `0.3` and `1.0 / 3.0` shows `0.333333333333333`. `print`, `str`, `int`, `float`, `len` and `input` are reserved. The math names are not: define your own `sqrt` and yours wins.

C functions

`extern fn` names a C function and its types; Plank calls it directly. `int` is a C `long long`, `float` a `double`, `bool` an `int`, `str` a `char *`. Anything in libc or libm is there already; for another library set `PLANK_LIBS="-lfoo"` when you build.

```
extern fn strlen(s: str) -> int
extern fn getpid() -> int
print(strlen("héllo"), getpid() > 0)   # 6 true
```

A name the linker cannot find is an error that says so.

Catching errors

Anything that would stop the program at run time can be caught: an index out of range, a missing dict key, `int("x")`, `nil!`, `pop()` on an empty list, and your own `throw`.

```
fn parse_age(text: str) -> int {
  let n = int(text)
  if n < 0 { throw "\(n) is not an age" }
  return n
}

try {
  print(parse_age(input("age? ")))
} catch err {
  print("no good:", err)   # err is the message, a str
}
```

`throw` takes a str. A `try` can hold `return`, `break` and `continue`, and tries nest: a throw inside a `catch` goes to the next `try` out. An error nobody catches stops the program with the file, line and message, exit code 1.

JSON

Every program has the `Json` enum: `null`, `bool(b)`, `number(n)`, `string(s)`, `array(items)`, `object(fields)`. Build one by hand or with `json_parse`, read it with `match` or with the helpers, and turn it back into text with `.text()`.

```
let doc = json_parse("{\"name\": \"plank\", \"tags\": [\"small\"],
\"stars\": 42}")
doc.get("name").str()        # "plank"; get throws if the key is missing
doc.get("stars").int()       # 42; also .num() for a float, .truth() for a
bool
doc.get("tags").at(0).str()  # "small"; .items() is the list, .keys() the
object's keys
Json.object(["ok": Json.bool(true)]).text()   # {"ok":true}
```

Numbers are floats, as in JSON itself. Object keys keep the order they came in. Bad input throws `bad JSON: ...` with a reason.

Memory

You never free anything. A collector runs now and then, finds every list, dict, string, struct, enum and closure the program can still reach, and frees the rest. It never moves anything. Set `PLANK_GC=4000` in the environment to collect every 4,000 bytes instead, which is how the test suite shakes out bugs.

Results

When an error is part of the answer rather than a surprise, return a `Result<T>`: `ok(value)` or `err(message)`. `?` after a `Result` takes the value out, or returns that error from the function you are in, which must return a `Result` too. `attempt(fn() => ...)` runs a closure and turns a throw into an `err`.

```
fn half(n: int) -> Result<int> {
  if n % 2 != 0 { return Result.err("\(n) is odd") }
  return Result.ok(n / 2)
}
fn quarter(n: int) -> Result<int> {
  return Result.ok(half(half(n)?)?)   # the first err comes straight back
}
quarter(8).value()      # 2; value() throws the message on an err
quarter(6).error()      # "3 is odd", a str?
quarter(6).or_else(0)   # 0
quarter(8).is_ok()      # true
match quarter(6) { .ok(v) { ... } .err(m) { ... } }
```

`Result` is an enum written in Plank with a type parameter, and you can write your own: `enum Maybe<T> { some(v: T) none }`, used as `Maybe<int>.none` or `Maybe.some(5)`, with methods like any enum.

Errors

Every error names the file and line:

```
fib.pk:3: 'x' is a let, use var to reassign it
```

Exit code 1 is a Plank error, at compile time or a runtime one like a bad `int("x")`, 2 is a missing file, anything else is what your program returned.

AI (lib/ai.pk)

`import "lib/ai.pk" as ai` makes a model scriptable. It is a plain Plank module, so there is no new syntax yet.

```
let m = ai.model("claude-haiku-5-5", "Answer in one line.")   # Anthropic;
key in ANTHROPIC_API_KEY
let q = ai.prompt("Summarize {topic}", ["topic": "plank"])    # fills
{holes} in a template
print(m.ask(q))                                               # the text,
or a throw with the server's words
```

`m.ask_json(question)` returns a `Json` value instead of text: it asks for only JSON, strips a code fence if the model adds one, and if the reply still does not parse it shows the model the error and tries once more before throwing.

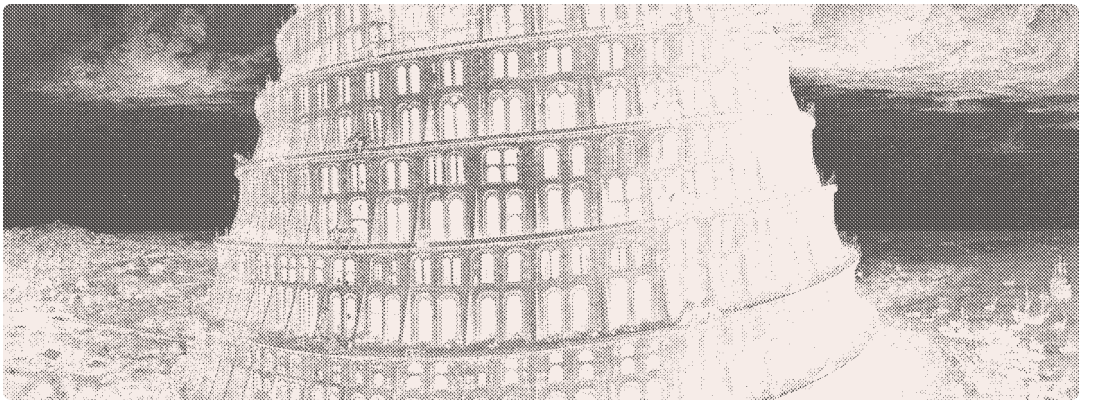
Models: `ai.model(name, system = "")` is Anthropic, `ai.local(name, url, system = "")` is any OpenAI-compatible server (oMLX, Ollama, Turing), `ai.turing(system = "")` is Turing on this Mac (`localhost:8127`, answers questions through its own fixed pipeline and tools; it does not take a tool list, so use it with `ask`, not `ai.agent`), and `ai.mock()` needs no network: it answers `echo: <question>`, `json X` answers X (for `ask_json`), and `use NAME {json}` makes it call that tool, so tests stay byte-stable.

`tool fn` writes the tool for you. Put `tool` before a top-level `fn` whose parameters and result are `str`, `int`, `float` or `bool`, and the comment lines right above it become the model's description:

```
# Adds two whole numbers
tool fn add(a: int, b: int) -> int {
  return a + b
}
let bot = ai.agent(m, [add_tool()])
```

Besides `add`, the compiler defines `add_tool()`, which returns the `ai.Tool` with its schema read off the signature, so the schema cannot drift from the code. It needs `import "lib/ai.pk"` in the same file (with or without `as`), and a tool fn takes no defaults or type parameters. Without a comment the name is the description.

Tools and agents: `ai.tool(name, about, params, run)` is a function the model may call; `params` maps each argument name to `"str"`, `"int"`, `"float"` or `"bool"`, `run` takes the arguments as `Json` and returns text, and `t.schema()` is the JSON schema the model is shown. `ai.agent(m, [tools], memory = "", keep = 0)` returns an `Agent`; `a.run(task)` asks, runs the tools the model called, hands back the results, and repeats until the model answers in plain text, or throws after `max_steps` (10). A tool that throws, or a name the agent does not have, goes back to the model as an error result instead of stopping the run. `memory` is a file: the whole conversation is saved there after each run and loaded at the start of the next. `keep = 20` caps it at about the last 20 messages, cutting only where a turn starts so a tool result never loses the call that asked for it; 0 keeps everything. Tools and agents work with all three kinds of model: Anthropic, the mock, and a local OpenAI-compatible server (the conversation is converted to its `tool_calls` format and back). The tool's comment is the only thing that tells the model when to use it, so make it specific: "Adds two whole numbers" beats "math", and rewording it changes what the model does.



APPENDIX B

A map of plank.py

One file does the work. Everything else is examples, tests and docs.

File	What it does
`plank.py`	The compiler. `lex` turns source into tokens (`string_end` and `unescape` handle strings and `\(...)` interpolation), `Parser` builds the AST (dataclasses near the top), `Codegen` walks the AST once and emits LLVM IR through llvmlite, `optimize` runs the O2 pipeline, `build` links the object with `cc`. `extern fn` registers a `Sig` whose LLVM function carries the C name itself, with bools widened to C `int` at the call; a link failure naming an unknown symbol is turned into a plain error. `RESULT_SRC` is `Result<T>` and `attempt` written in Plank; generic enums instantiate like generic structs through `instantiate_enum`, and `Result.ok(5)` infers `T` from the value or, for `Result.err`, from the function's return type. `r?` is one branch: an err returns from the function, an ok yields its value. `SET_SRC` is `Set<T>` written in Plank, a generic struct over a `[T: bool]` dict, spliced in when a program mentions `Set`; `in`, `for` and `len` on a set call its methods. `JSON_SRC` is the `Json` enum written in Plank, parsed in front of every program; `json_reader` emits the parser for it in IR against a cursor the runtime keeps. `RUNTIME` is a C string, compiled next to every program, holding what libc does not: joining strings, counting characters, number to text and back, reading a line, string tools (split, join, trim, replace, slices, all counting UTF-8 characters), `popen` for `run`, curl for `http`, a JSON lexer, `stat`, `opendir`, `strftime` and `strptime` for the file, clock and date builtins, POSIX `regcomp` with a one-pattern cache for `matches`, `captures`, `find_all` and `replace_all`, growing lists, a stable merge sort that calls back into compiled code, and dicts: insertion-ordered entry arrays under an open-addressing index, keys stored as 8 bytes (an int or a string pointer). Every list is one C struct of length, capacity and bytes; the compiler knows the element type, the runtime only its size. Bounds checks are inline IR, so the hot path never calls out. `main` is the CLI.
`plank`	Symlink to `plank.py` so `./plank file.pk` works. The `uv run --script` shebang installs llvmlite on first use.
`examples/lib/*.pk`	Files that get imported, plainly or as a module.
`examples/*.pk`	One program per feature: hello, fib, fizzbuzz, primes, sqrt, strings, text, lists, structs, dicts, enums, optionals, closures, textkit, world, errors, scripting, generics, containers, collections, tuples, interop, loops. Each has a `.out` with its expected output.

File	What it does
`apps/*.pk`	Seven real programs: a tool-using agent loop that talks to Ollama over HTTP and JSON, a calculator with its own parser, a to-do CLI that saves to a file, word frequency, a budget ledger, a maze solver, Game of Life. Each has a `.out`; `test.py` runs them like the examples and the landing shows their output.
`tests/*.pk`	The language testing itself: strings, lists, dicts, types, control. Each is a Plank program full of `assert` that prints `ok`.
`list_helper` in `plank.py`	first, last, index_of, insert, remove_at, reverse, sum, min, max on lists, mostly inline IR; min and max sort a copy and take an end.
`plank fmt`	`Fmt` prints the AST back out in the layout the examples use: two-space indent, one-line `if` and `match` arms where the source had them, lists and calls spread over lines where the source did. Comments are kept by the lexer with their line numbers and put back by line; a blank line survives where the source had one. `test.py` runs `plank fmt --check` over the repo, and `fmt` refuses any file it cannot format twice to the same text.
`plank doc`	`DOCS` in `plank.py` holds one signature and one line per built-in; `test.py` checks the table names exactly the built-ins the compiler has.
`plank repl`	Keeps the definitions and statements typed so far, rebuilds and reruns the whole program after each entry, and prints only the output past what was shown before. A bare expression is tried as `print(...)` first. A line that fails to compile or run is dropped.
`nulljosh/homebrew-plank`	The Homebrew tap: `brew install nulljosh/tap/plank-lang` installs `plank.py` as `plank` with `uv` as a dependency. Bump the formula's url and sha256 at each release.
`plank test [dir]`	Built into the CLI: compiles every `.pk` in the folder and passes it when it exits 0 printing `ok`. `test.py` is the repo's own suite; `plank test` is for programs written in Plank.
`test.py`	Compiles every example, diffs the output, and runs every `tests/*.pk`, checks 27 error messages at compile time and run time, then runs everything again with the collector firing every 4KB. CI runs it on macOS and Ubuntu.
`docs/SPEC.md`	The language reference.

File	What it does
`docs/HACKERS-AND-PAINTERS.md`	Paul Graham's book summarized, and the eight design rules Plank takes from it.
`docs/LOOP-HANDOFF.md`	Where the build loop stands and the prompt that restarts it.
`docs/COMPARE.md`	The scorecard against Python, Ruby and Swift. The roadmap works down it.
`site/index.html`, `site/icon.svg`, `wrangler.toml`	The landing page at plank.heyitsmejosh.com, deployed as Cloudflare static assets. Drifting-plank canvas, hero, and a demo window that plays the real examples.
`compiler/plank.pk`, `plank tokens`	The compiler written in Plank, growing toward 5.0 (self-hosting). Today: the lexer and parser, ports of `lex` and `Parser`. Its tree is one generic `Ast` struct (kind, line, children) in the shape of plank.py's dataclasses. `plank tokens` and `plank ast` print plank.py's tokens and tree in the same formats, and `test.py` diffs both over every `.pk` in the repo.
`book/*.md`, `tools/gen-book.py`, `site/book/`	The Plank Book, thirteen chapters in plain Markdown. `gen-book.py` turns them into one page with highlighted code at `/book/`; It also prints `docs/SPEC.md` and this file as Appendices A and B, so the book never drifts from the docs, and inlines the diagrams in `book/art/`. `--pdf` prints `site/book/plank-book.pdf` (7 x 9.25 in, gitignored) through headless Chrome; `release.sh` makes it before deploying. `--check` compiles every ``` `plank block with a` fn main`, diffs it against the` `output block under it, runs` plank fmt --check` on it, and fails if the page is stale.` test.py` runs the check.
`tools/gen-demo.py`, `site/demo.js`	The generator reads every example, its `.out`, and `plank emit` output into `demo.js`. `test.py` fails if `demo.js` is stale, so the landing can never show output the compiler did not produce.
`roadmap.md`, `MONEY.md`, `CLAUDE.md`	What is next, what it earns (nothing, on purpose), and notes for the agent.

The pipeline

Source goes through four steps, each a plain function or class:

1. **Lex.** One regex, a loop, a list of tokens. Strings get a small hand scanner instead, because `\(...)` can hold strings of its own. `unescape` decodes escapes, keeps UTF-8 intact, and hands interpolations back as source for the parser. Newlines are tokens so statements can end without semicolons; inside parentheses they are dropped.
2. **Parse.** `load` parses the main file and every file it imports, once each. An import path starting with `github.com/` maps to `~/.plank/pkg/github.com/user/repo@ref/...` and is fetched with curl from raw.githubusercontent.com the first time, including files the package imports relatively. An import with `as x` goes through `qualify`, which renames every top-level name in that file to `x.name` and rewrites the module's own references to match, so the rest of the compiler sees one flat program with no clashes; `x.f(...)` in the importer is a method call on a name the codegen knows is a module. Lines in the k-th file are numbered from k million, so a single line number carries the file too, and `locate` turns it back into a path and a line for every error, compile time or run time. Recursive descent for statements, Pratt for expressions. Precedence lives in one dict.
3. **Codegen.** `program` names every struct first, so fields can mention structs declared later, then declares every function and method (`Point.length` becomes `pk_Point.length` with `self` as its first argument), then fills in the bodies. A struct value is a pointer to a heap LLVM struct. An enum value is a pointer to a tag and enough 8-byte slots for its biggest case; `match` on an enum is one LLVM `switch` on the tag, and the compiler counts the cases it saw to refuse a match that misses one. `arrange` lines up positional and labeled arguments and fills defaults, for calls and struct building alike. A generic function is kept as its AST; `call_generic` evaluates the non-closure arguments, `unify` matches their types against the parameter types to bind each type parameter, `specialize` deep-copies the AST with the names substituted, and the copy is compiled once per distinct binding under a name like `first<int>`. A tuple type like `(int, str)` is a struct named exactly that with fields `_0` and `_1`, made by `tuple_struct` the first time it is named, so `t.0` is a field read and the formatter prints the field back as `.0`. Generic structs work the same way through `instantiate_struct`: `Stack<int>` becomes a real struct, with its methods compiled as `Stack<int>.push`, the first time the name is used in a type or a call; `Stack(items: [1])` binds `T` from the

fields with the same `unify`. An optional `T?` is a pointer to a boxed `T`, null for `nil`; `coerce` does the boxing wherever a value lands in an optional slot (let, assignment, return, arguments, append). A closure is a pointer to two words, code and environment; the environment is an array of pointers to the variables it uses. `captures` scans each function for names any lambda mentions, and `declare` puts exactly those variables on the heap so a closure can outlive the call. A top-level function used as a value gets a constant closure around a small wrapper. A closure with no `->` is compiled once as a probe, where the first `return` sets the type, then thrown away and compiled for real. `match` as a value is parsed into a closure around a `match` statement whose arms `return`, so one code path handles both, at the price of a closure call. `try` calls `setjmp` through the runtime's stack of handlers; `pk_panic`, which every runtime error goes through, `longjmp`s to the newest one instead of exiting. A function that contains a `try` keeps all its variables in memory rather than registers, so values set before a throw are still there after it, and `return`, `break` and `continue` pop the handlers they jump out of. `==` on a list, dict, tuple, struct, enum or optional calls a private `eq.<type>` function generated once per type and registered before its body, so a struct that holds itself through an optional compares fine. There is no separate type checker. Every expression returns a pair of LLVM value and Plank type, and the operator and call rules check the types as they go. Variables are `alloca` slots; LLVM's mem2reg turns them into registers at O2.

4. **Optimize and link.** llvmlite's new pass manager at O2, then the target machine emits a Mach-O or ELF object, and the system `cc` links it with `RUNTIME`, libc and libm. `RUNTIME` opens with the collector: every allocation carries a header naming its slot in a table; a collection sorts the table by address, scans the stack, the registers (spilled with `setjmp`) and the open `try` frames for words that point into a block, follows those through every non-string block, and frees what was never reached. Strings are marked atomic and never scanned. Runtime helpers that hold a caller's pointer only in a C local bump `pk_hold` so no collection can run under them; the stress mode in `test.py` is what found that this was needed. A collection runs once as many bytes have been allocated as survived the last one, so a steady program keeps a steady footprint. The generated C `main` hands `argc` and `argv` to the runtime for `args()`. `--static` adds `-static` to the link on Linux, so the

binary carries libc with it; macOS cannot do that and says so. Nothing is freed yet: programs run and exit, and the OS takes the memory back.

User functions are named `pk.name` in the IR, methods `pk.Type.name`. The dot cannot appear in a C name, so nothing you call a function can collide with libc or the runtime's `pk_` helpers. A generated C `main` calls `pk.main` and returns 0.

The AI library

`lib/ai.pk` is ordinary Plank on top of `http`, `json_quote` and `json_parse`. A `Model` has one method that matters, `step(history, tools)`: the conversation so far goes in as `[Json]` messages in Anthropic's shape, the model's next content blocks come out. `ask` is one step; `Agent.run` loops steps, runs the `tool_use` blocks through each `Tool`'s closure inside `attempt`, and appends a `tool_result` message until a reply has no tool calls. Memory is that history written to a file as a JSON array; `trim` bounds it by dropping the oldest messages and cutting only at a turn start (a user message that is plain text), falling back to the nearest earlier one. The mock model is a function of the last message, which is what keeps the examples byte-stable in CI. Requests go through `pk_http` (curl on argv), so a prompt is limited by ARG_MAX; sending the body from a file is the upgrade when that bites. `tool fn` is a parse-time rewrite in `expand_tools` (run per file, before `qualify`): the parser marks the `Fn`, the lexer hands it the comment block above as `doc`, and the loader appends a generated `name_tool()` that calls `ai.tool` with a params dict taken from the signature and a closure that unpacks the `Json` arguments by type. The formatter prints the marked `Fn` back as `tool fn`, so the generated function never shows up in `plank fmt`. `tool` is a contextual keyword (only before `fn` at the top level), so `ai.tool(...)` and variables called tool keep working. Local models speak OpenAI's tool format: `step_openai` runs `to_openai` on the history (tool_use blocks become `tool_calls`, tool_result blocks become `tool` messages) and `from_openai` on the reply, so the agent loop never knows which provider it is talking to. `ask_json` is `ask` plus `unfence` and `json_parse` with one corrective retry; it needs no provider support for a JSON mode, so it works on every model. `ai.turing()` is just `ai.local` pointed at Turing's server on port 8127. `to_openai` and `from_openai` are pure functions, which is how `tests/local.pk` covers them without a server.