Forge Language Series

Programming Forge

The Internet-Native Language That Reads Like English

Archith Rapaka

First Edition

Programming Forge

The Internet-Native Language That Reads Like English

Programming Forge

The Internet-Native Language That Reads Like English

Archith Rapaka

First Edition — v0.3.0

March 2026

Programming Forge: The Internet-Native Language That Reads Like English

Forge is an open-source programming language.
Visit `https://github.com/humancto/forge-lang` for source code and community.

First Edition, March 2026
Version 0.3.0

Typeset with LaTeX and Pandoc.

Contents

Chapter 1

Programming Forge

The Internet-Native Language That Reads Like English

By Archith Rapaka

First Edition — February 2026

Forge is an open-source programming language. Visit https://github.com/forge-lang/forge for source code and community.

While every precaution has been taken in the preparation of this book, the author assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

1.1 Preface

I built Forge because I was tired of installing thirty packages to do what a programming language should do out of the box.

Every modern application talks to HTTP endpoints, reads from databases, handles JSON, and hashes passwords. Yet in every mainstream language, these are third-party concerns. You pip-install a web framework. You npm-install a database driver. You go-get a crypto library. You wrestle with dependency conflicts, version mismatches, and supply chain vulnerabilities — all before writing a single line of your actual application.

Forge takes a different approach. HTTP, databases, cryptography, file I/O, regular expressions, CSV parsing, terminal UI — they're all built into the language itself. Not as a bloated standard library you have to import, but as primitives that are always available, always documented, and always tested.

The result is a language where a REST API server is 10 lines of code. Where querying a database is a single function call. Where hashing a password doesn't require reading a third-party library's documentation.

Forge also reads like English. You can write `say "hello"` or `println("hello")` — both work. You can define functions with `fn` or `define`. You can write `else` or `otherwise` or even `nah`. The language doesn't force a style on you. It meets you where you are.

This book is organized in four parts:

- **Part I: Foundations** covers installation, syntax, control flow, functions, collections, and error handling — everything you need to read and write Forge.
- **Part II: The Standard Library** documents all 15 built-in modules, function by function, with recipes for common tasks.
- **Part III: Building Real Things** walks through complete projects — REST APIs, data pipelines, DevOps scripts, and AI integration.
- **Part IV: Under the Hood** explains how Forge works internally — the lexer, parser, interpreter, bytecode VM, and toolchain.

Whether you're a student writing your first program, a backend developer building APIs, or a language enthusiast curious about implementation — welcome to Forge.

Archith Rapaka February 2026

1.2 About the Author

Archith Rapaka is a software engineer and programming language designer. He believes programming languages should be as natural to read as they are powerful to write, and that the tools developers use most — HTTP, databases, cryptography — belong in the language itself, not in package registries. Forge is the embodiment of this philosophy.

1.3 How to Read This Book

If you're new to programming: Start with Part I, work through every "Try It Yourself" exercise, and run `forge learn` for interactive tutorials alongside the book.

If you're an experienced developer: Skim Part I for syntax differences, then dive into Part II (standard library) and Part III (projects) for practical usage.

If you're a language implementer: Skip directly to Part IV for the architecture, bytecode VM, and garbage collector internals.

Conventions used in this book:

- Code examples are shown in monospace blocks and can be saved as `.fg` files and run with `forge run filename.fg`
- Terminal commands are prefixed with `$`
- Output is shown after commands or as comments
- Tips and important notes are formatted as blockquotes

Chapter 2

Part I: Foundations

Programming Forge by Archith Rapaka

2.1 Chapter 1: Getting Started

Forge is a programming language built for the internet age. This chapter introduces the language, walks you through installation, and gets your first program running in under five minutes. By the end, you will have written and executed Forge code, explored the interactive REPL, and discovered the built-in tutorial system that ships with every installation.

2.1.1 What Is Forge?

Forge is a general-purpose programming language written in Rust. It was designed with a single guiding principle: the things you do most often on the internet — making HTTP requests, querying databases, hashing passwords, parsing JSON — should be built into the language itself, not buried in third-party packages.

Most languages treat the network as an afterthought. You install a language, then install a web framework, then install an HTTP client, then install a JSON library, then install a database driver. Forge ships all of that out of the box. A REST API server is four lines. A database query is two.

Forge also reads like English. Every construct has two spellings: a classic syntax familiar to anyone who has written JavaScript or Python, and a natural-language syntax that reads like prose. Both compile to the same thing. You pick whichever feels right, and you can mix them freely in the same file.

Here is how Forge compares to languages you may already know:

Feature	Python	JavaScript	Go	Rust	Forge
HTTP client built in	No	`fetch` (browser)	`net/http`	No	Yes (`fetch`, `http.get`)
HTTP server built in	No	No	`net/http`	No	Yes (`@server`, `@get`)

Feature	Python	JavaScript	Go	Rust	Forge
Database built in	No	No	`database/sql`	No	Yes (`db.open`, `pg.connect`)
Crypto built in	`hashlib`	`crypto` (Node)	`crypto`	No	Yes (`crypto.sha256`)
Terminal UI built in	No	No	No	No	Yes (`term.table`, `term.bar`)
Interactive tutorials	No	No	`go tour` (web)	No	Yes (`forge learn`)
Dual syntax	No	No	No	No	Yes (classic + natural)
Errors as values	No	No	Yes	Yes	Yes (`Result`, `?`, `must`)
Null safety	No	No	No	Yes (`Option`)	Yes (`Option`, no null)
Semicolons required	No	Optional	No	Yes	No

> **Tip:** Forge is not trying to replace Rust or Go for systems programming. It is designed for application-layer work: web services, scripts, data pipelines, prototypes, and tooling. Think of it as the language you reach for when you want to build something that talks to the internet.

2.1.2 Installing Forge

Forge is built with Rust, so you need the Rust toolchain installed first. If you don't have it, visit `https://rustup.rs` and follow the instructions. You need Rust 1.85 or later.

Once Rust is ready, clone the repository and install:

```
git clone https://github.com/forge-lang/forge.git
cd forge
cargo install --path .
```

This compiles Forge from source and places the `forge` binary in your Cargo bin directory (typically `~/.cargo/bin/`). The build takes about 60–90 seconds on a modern machine.

Verify the installation:

```
forge version
```

You should see output like:

```
Forge 0.1.0
```

> **Tip:** If `forge` is not found after installation, make sure `~/.cargo/bin` is in your system `PATH`. Add `export PATH="$HOME/.cargo/bin:$PATH"` to your shell profile if needed.

2.1.3 Your First Program

Create a file called `hello.fg` in your working directory:

```
let name = "World"
println("Hello, {name}!")
```

Run it:

```
forge run hello.fg
```

Output:

```
Hello, World!
```

That is the entire program. No `main` function, no imports, no boilerplate. Forge executes top-level statements in order, like a script.

Notice the string `"Hello, {name}!"`. The curly braces inside a double-quoted string perform *interpolation* — the expression inside the braces is evaluated and its result is inserted into the string. This works with any expression, not just variable names.

Now try the natural-language syntax. Create `hello_natural.fg`:

```
set name to "World"
say "Hello, {name}!"
```

Run it with `forge run hello_natural.fg` and you get the same output. The `set ... to` syntax is equivalent to `let`, and `say` is equivalent to `println`. Both styles produce identical results.

2.1.4 The REPL

Forge ships with an interactive Read-Eval-Print Loop. Start it by running `forge` with no arguments:

```
forge
```

You will see the Forge prompt:

```
forge>
```

Try some expressions:

```
forge> 2 + 2
4
forge> "hello" + " " + "world"
```

```
hello world
forge> let x = 42
forge> x * 2
84
```

The REPL supports multiline input. When you type an opening brace, Forge knows you are not done yet:

```
forge> fn greet(name) {
  ...     return "Hello, {name}!"
  ... }
forge> greet("Forge")
Hello, Forge!
```

The REPL also provides command history (press the up arrow to recall previous lines) and tab completion for keywords and built-in functions.

> **Tip:** Use the REPL to experiment with syntax as you read this book. It is the fastest way to test an idea.

2.1.5 Interactive Tutorials

Forge includes 30 built-in interactive lessons that teach you the language step by step, right in your terminal. List them:

```
forge learn
```

Start a specific lesson:

```
forge learn 1
```

Each lesson explains a concept, shows you a code example, runs it, and displays the expected output. The lessons cover variables, functions, loops, error handling, HTTP, databases, and more. If you are new to programming, `forge learn` is the recommended starting point.

2.1.6 Inline Evaluation

For quick one-liners, use the –e flag to evaluate an expression without creating a file:

```
forge -e 'say "Hello from the command line!"'
```

```
forge -e 'println(2 + 2)'
```

```
forge -e 'say math.sqrt(144)'
```

This is useful for quick calculations, testing syntax, and shell scripting.

2.1.7 Editor Support

Forge provides a Language Server Protocol (LSP) server for editor integration:

```
forge lsp
```

For Visual Studio Code, a Forge extension is available that provides syntax highlighting, error diagnostics, and completion suggestions. Install it from the VS Code marketplace by searching for "Forge Language" or configure your editor to use the LSP server directly.

2.1.8 Project Scaffolding

When you are ready to build something larger than a single file, use the `forge new` command to generate a project scaffold:

```
forge new my-app
```

This creates a directory structure with a main source file, a test file, and a configuration file — everything you need to start building.

2.1.9 Try It Yourself

1. **Hello, You.** Create a file called `greeting.fg` that stores your name in a variable and prints a personalized greeting using string interpolation. Run it with `forge run`.
2. **REPL Explorer.** Open the Forge REPL and try these expressions: `math.pi`, `len("forge")`, `sort([5, 3, 1, 4, 2])`. What does each one return?
3. **Tutorial Time.** Run `forge learn 1` and complete the first interactive lesson. Then run `forge learn` to see the full list of available topics.

2.2 Chapter 2: Variables and Types

Every program manipulates data, and every piece of data has a type. This chapter covers Forge's type system, how to declare variables, and the rules that govern mutability, type conversion, and truthiness. Understanding these fundamentals will make everything that follows in this book more intuitive.

2.2.1 The Seven Fundamental Types

Forge has seven built-in types. Every value you create belongs to exactly one of them.

Type	Description	Example Literals
`Int`	64-bit signed integer	`42`, `-7`, `0`
`Float`	64-bit floating point	`3.14`, `-0.5`, `1.0`
`String`	UTF-8 text	`"hello"`, `"Forge {version}"`
`Bool`	Boolean truth value	`true`, `false`
`Array`	Ordered collection	`[1, 2, 3]`, `["a", "b"]`
`Object`	Key-value map (insertion-ordered)	`{ name: "Alice", age: 30 }`
`Null`	Absence of value	`null`

Let's look at each one:

```
let age = 30
let pi = 3.14159
let name = "Forge"
let active = true
let scores = [95, 87, 92]
let user = { name: "Alice", role: "engineer" }
let nothing = null
```

Forge also has two special wrapper types — `Result` and `Option` — which we will cover in detail in Chapter 8. For now, know that `Ok(value)` and `Err("message")` wrap results, and `Some(value)` and `None` wrap optional values.

2.2.2 Declaring Variables

Forge provides two syntaxes for declaring variables: classic and natural.

Classic syntax uses `let`:

```
let city = "Portland"
let population = 652503
let elevation = 15.2
```

Natural syntax uses `set ... to`:

```
set city to "Portland"
set population to 652503
set elevation to 15.2
```

Both produce identical results. Use whichever reads better to you, or mix them within the same file.

2.2.3 Immutable by Default

Variables in Forge are *immutable* by default. Once assigned, their value cannot change:

```
let x = 10
x = 20
```

This program will produce an error: `cannot reassign immutable variable 'x'`. Immutability is a safety feature. It prevents accidental changes and makes code easier to reason about.

2.2.4 Mutable Variables

When you need a variable that changes, declare it with `mut`:

```
let mut counter = 0
counter = counter + 1
counter = counter + 1
println("Counter: {counter}")
```

Output:

```
Counter: 2
```

In natural syntax, use `set mut` and `change`:

```
set mut counter to 0
change counter to counter + 1
change counter to counter + 1
say "Counter: {counter}"
```

The `change ... to` syntax is the natural-language equivalent of reassignment. It only works on variables declared with `mut`.

> **Tip:** Start with immutable variables. Only add `mut` when you genuinely need to change a value. This habit catches bugs early and communicates intent to anyone reading your code.

2.2.5 Type Annotations

Forge uses *gradual typing*. Type annotations are optional — you can add them when you want clarity or extra safety, and omit them when the types are obvious:

```
let name: String = "Alice"
let age: Int = 30
let score: Float = 98.5
let active: Bool = true
```

Without annotations, Forge infers the types from the values:

```
let name = "Alice"
let age = 30
let score = 98.5
let active = true
```

Both versions behave identically. Annotations become more valuable in function signatures, where they document the expected inputs and outputs:

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

We will explore annotated functions in Chapter 6.

2.2.6 String Interpolation

String interpolation is one of Forge's most frequently used features. Any expression inside curly braces within a double-quoted string is evaluated and converted to text:

```
let name = "Forge"
let version = 2
say "Welcome to {name} v{version}!"
```

Output:

```
Welcome to Forge v2!
```

Interpolation works with any expression, not just simple variables:

```
let x = 7
say "Seven squared is {x * x}"
say "The length of 'hello' is {len("hello")}"
say "Is 10 > 5? {10 > 5}"
```

Output:

```
Seven squared is 49
The length of 'hello' is 5
Is 10 > 5? true
```

You can nest function calls, arithmetic, and comparisons inside interpolation braces. This eliminates the need for string concatenation in most cases.

2.2.7 Triple-Quoted Raw Strings

For strings that span multiple lines or contain characters you don't want to escape, use triple-quoted strings:

```
let sql = """SELECT * FROM users WHERE active = true"""
say sql
```

Triple-quoted strings preserve their content exactly as written. They are especially useful for SQL queries, regular expressions, and embedded data.

```
let html = """<div class="container">
    <h1>Hello, Forge!</h1>
    <p>This is raw HTML.</p>
</div>"""
say html
```

2.2.8 Type Conversion

Forge provides built-in functions to convert between types:

```
let n = int("42")
say n + 8

let f = float("3.14")
say f * 2

let s = str(42)
say "The answer is " + s

say int("100") + int("200")
say float("1.5") + float("2.5")
```

Output:

```
50
6.28
The answer is 42
300
4.0
```

Function	Converts To	Example
`int(value)`	`Int`	`int("42")` ☐ `42`
`float(value)`	`Float`	`float("3.14")` ☐ `3.14`
`str(value)`	`String`	`str(42)` ☐ `"42"`

> **Tip:** `int()` and `float()` will produce an error if the input string cannot be parsed as a number. Always validate user input before converting.

2.2.9 Type Inspection

You can check the type of any value at runtime:

```
say typeof(42)
say typeof("hello")
say typeof(true)
say typeof([1, 2, 3])
say typeof({ name: "Alice" })
say typeof(null)
```

Output:

```
Int
String
Bool
Array
Object
Null
```

The `typeof()` function returns a string describing the type. This is useful for debugging, validation, and writing functions that handle multiple types.

The `type()` function is an alias for `typeof()`:

```
let value = 3.14
if type(value) == "Float" {
    say "It's a floating-point number"
}
```

2.2.10 Truthiness

Forge evaluates values as “truthy” or “falsy” when used in boolean contexts (like `if` conditions). The rules are straightforward:

Value	Truthy?
`false`	Falsy
`null`	Falsy
`0` (integer zero)	Falsy
`0.0` (float zero)	Falsy
"" (empty string)	Falsy
`[]` (empty array)	Falsy
Everything else	Truthy

```
if "hello" {
    say "Non-empty strings are truthy"
}

if 0 {
    say "This won't print"
} else {
    say "Zero is falsy"
}

if [1, 2, 3] {
    say "Non-empty arrays are truthy"
}

if [] {
    say "This won't print"
} else {
    say "Empty arrays are falsy"
}
```

Output:

```
Non-empty strings are truthy
Zero is falsy
Non-empty arrays are truthy
Empty arrays are falsy
```

> **Tip:** If you want explicit boolean checks rather than relying on truthiness, compare directly: `if len(items) > 0 { ... }` instead of `if items { ... }`. Explicit comparisons are clearer, especially when other developers will read your code.

2.2.11 Variable Types Cheat Sheet

Declaration	Syntax Style	Mutable?	Example
`let x = 5`	Classic	No	`let name = "Alice"`
`let mut x = 5`	Classic	Yes	`let mut count = 0`
`set x to 5`	Natural	No	`set name to "Alice"`
`set mut x to 5`	Natural	Yes	`set mut count to 0`
`x = 10`	Classic reassign	—	`count = count + 1`
`change x to 10`	Natural reassign	—	`change count to count + 1`
`let x: Int = 5`	Annotated	No	`let age: Int = 30`

2.2.12 Try It Yourself

1. **Type Explorer.** Write a program that creates one variable of each of the seven types and prints both the value and its type using `typeof()`. For example: `say "42 is a {typeof(42)}"`.
2. **Mutability Practice.** Declare a mutable variable called `balance` starting at `1000`. Subtract `250` three times using reassignment, then print the final balance. Try doing it once with classic syntax and once with natural syntax.
3. **Interpolation Challenge.** Write a program that stores a person's first name, last name, and birth year in variables, then prints a single sentence like: `"Alice Johnson was born in 1990 and is 36 years old."` — computing the age from the birth year using an expression inside the interpolation braces.

2.3 Chapter 3: Operators and Expressions

Operators are the verbs of a programming language — they describe what to *do* with your data. This chapter covers every operator Forge supports, from basic arithmetic to compound assignment, along with the rules that govern how expressions are evaluated.

2.3.1 Arithmetic Operators

Forge supports the standard arithmetic operators:

Operator	Operation	Example	Result
`+`	Addition	`7 + 3`	`10`
`-`	Subtraction	`7 - 3`	`4`
`*`	Multiplication	`7 * 3`	`21`
`/`	Division	`7 / 3`	`2`
`%`	Modulo (remainder)	`7 % 3`	`1`

```
say 10 + 3
say 10 - 3
say 10 * 3
say 10 / 3
say 10 % 3
```

Output:

```
13
7
30
3
1
```

2.3.2 Integer vs. Float Division

When both operands are integers, division produces an integer result (truncating any remainder):

```
say 7 / 2
say 10 / 3
```

Output:

```
3
3
```

When either operand is a float, the result is a float:

```
say 7.0 / 2
say 7 / 2.0
say 10.0 / 3.0
```

Output:

```
3.5
3.5
3.3333333333333335
```

This behavior matches most systems languages. If you want floating-point division with integer operands, convert one to a float first:

```
let a = 7
let b = 2
say float(a) / float(b)
```

Output:

```
3.5
```

> **Tip:** Division by zero with integers causes a runtime error. Always validate divisors when working with user input or computed values.

2.3.3 Compound Assignment Operators

Forge supports shorthand operators that combine arithmetic with assignment. These only work on mutable variables:

```
let mut x = 10
x += 5
say x

x -= 3
say x

x *= 2
say x

x /= 4
say x
```

Output:

```
15
12
24
6
```

Operator	Equivalent To	Example
`+=`	`x = x + value`	`x += 5`
`-=`	`x = x - value`	`x -= 3`
`*=`	`x = x * value`	`x *= 2`
`/=`	`x = x / value`	`x /= 4`

Compound assignment is syntactic sugar — `x += 5` means exactly the same thing as `x = x + 5`. Use whichever is clearer in context.

2.3.4 Comparison Operators

Comparison operators return a boolean value (`true` or `false`):

Operator	Meaning	Example	Result
`==`	Equal to	`5 == 5`	`true`
`!=`	Not equal to	`5 != 3`	`true`
`<`	Less than	`3 < 5`	`true`
`>`	Greater than	`5 > 3`	`true`
`<=`	Less than or equal	`5 <= 5`	`true`
`>=`	Greater than or equal	`5 >= 6`	`false`

```
let a = 10
let b = 20

say a == b
say a != b
say a < b
say a > b
say a <= 10
say b >= 20
```

Output:

```
false
true
true
false
true
true
```

Strings are compared lexicographically (dictionary order):

```
say "apple" < "banana"
say "zebra" > "aardvark"
say "hello" == "hello"
```

Output:

```
true
true
true
```

2.3.5 Logical Operators

Logical operators combine boolean values:

Operator	Meaning	Example	Result
`&&`	Logical AND	`true && false`	`false`
`||`	Logical OR	`true || false`	`true`
`!`	Logical NOT	`!true`	`false`

```
let age = 25
let has_license = true

if age >= 16 && has_license {
    say "You can drive"
}

let is_weekend = false
let is_holiday = true

if is_weekend || is_holiday {
    say "No work today!"
}

let raining = false
if !raining {
    say "Go outside"
}
```

Output:

```
You can drive
No work today!
Go outside
```

2.3.6 String Concatenation

The + operator concatenates strings when both operands are strings:

```
let first = "Hello"
let second = "World"
let greeting = first + ", " + second + "!"
say greeting
```

Output:

```
Hello, World!
```

In most cases, string interpolation is cleaner than concatenation:

```
let first = "Hello"
let second = "World"
say "{first}, {second}!"
```

Both approaches produce the same result. Prefer interpolation for readability; use concatenation when building strings incrementally.

2.3.7 Negation

The unary minus operator negates a number:

```
let x = 42
say -x

let temperature = -15
say temperature
say -temperature
```

Output:

```
-42
-15
15
```

2.3.8 Operator Precedence

When an expression contains multiple operators, Forge evaluates them in a specific order. Higher-precedence operators bind more tightly:

Precedence	Operators	Associativity
Highest	`!` (unary NOT), – (unary negation)	Right-to-left
	`*`, `/`, `%`	Left-to-right
	`+`, `-`	Left-to-right
	`<`, `>`, `<=`, `>=`	Left-to-right
	`==`, `!=`	Left-to-right
	`&&`	Left-to-right
Lowest	`||`	Left-to-right

```
say 2 + 3 * 4
say (2 + 3) * 4
```

Output:

```
14
20
```

Multiplication binds more tightly than addition, so `2 + 3 * 4` is evaluated as `2 + (3 * 4)`. Use parentheses to override the default order when needed.

```
let result = 10 > 5 && 3 < 7
say result
```

Output:

```
true
```

Here, the comparisons (`10 > 5` and `3 < 7`) are evaluated first, then `&&` combines the two boolean results.

> **Tip:** When in doubt, add parentheses. They cost nothing at runtime and make your intent unambiguous to both the computer and the human reading your code.

2.3.9 Expression Evaluation Order

Forge evaluates expressions left to right within the same precedence level. This matters most with function calls that have side effects:

```
let a = 5
let b = 3
let c = 2

let result = a + b * c - a / c
say result
```

Step by step:

1. `b * c` ☐ `6` (multiplication first)
2. `a / c` ☐ `2` (division, same precedence as multiplication, left to right)
3. `a + 6` ☐ `11` (addition)
4. `11 - 2` ☐ `9` (subtraction)

Output:

```
9
```

2.3.10 Try It Yourself

1. **Calculator.** Write a program that stores two numbers in variables and prints the result of all five arithmetic operations (`+`, `-`, `*`, `/`, `%`) on them, each on its own line. Test with both integer and float values.

2. **Compound Assignment Chain.** Start with `let mut x = 100`. Apply `+= 50`, then `*= 2`, then `-= 75`, then `/= 5`. Print `x` after each step. What is the final value?
3. **Precedence Puzzle.** Without running the code, predict the output of each expression. Then verify in the REPL.
   - `2 + 3 * 4 - 1`
   - `(2 + 3) * (4 - 1)`
   - `10 / 2 + 3 * 4 - 1`
   - `true || false && false`

2.4 Chapter 4: Control Flow

Programs that run in a straight line from top to bottom are useful, but limited. Real programs make decisions: they choose one path over another based on conditions. This chapter covers every branching construct in Forge, from basic `if/else` to the powerful `when` guard expression.

2.4.1 The if Statement

The `if` statement is the most fundamental control flow construct. It executes a block of code only when a condition is true:

```
let temperature = 35

if temperature > 30 {
    say "It's hot outside!"
}
```

Output:

```
It's hot outside!
```

The condition must evaluate to a truthy value (see the truthiness table in Chapter 2). The braces around the body are required — Forge does not support braceless `if` statements.

2.4.2 if/else

Add an `else` branch to handle the case when the condition is false:

```
let age = 15

if age >= 18 {
    say "You are an adult"
```

```
} else {
    say "You are a minor"
}
```

Output:

```
You are a minor
```

2.4.3 else if Chains

Chain multiple conditions with `else if`:

```
let score = 85

if score >= 90 {
    say "Grade: A"
} else if score >= 80 {
    say "Grade: B"
} else if score >= 70 {
    say "Grade: C"
} else if score >= 60 {
    say "Grade: D"
} else {
    say "Grade: F"
}
```

Output:

```
Grade: B
```

Forge evaluates conditions from top to bottom and executes the first branch whose condition is true. Once a branch executes, the remaining branches are skipped.

2.4.4 otherwise and nah

Forge provides two natural-language alternatives to `else`:

`otherwise` reads like prose:

```
let ready = false

if ready {
    say "Let's go!"
} otherwise {
    say "Not yet"
}
```

nah is informal and fun:

```
let has_coffee = true

if has_coffee {
    say "Productive morning"
} nah {
    say "Need coffee first"
}
```

Both `otherwise` and `nah` are exact synonyms for `else`. Use them to make your code read the way you think. You can also use `otherwise if` in chains:

```
set score to 85

if score > 90 {
    say "Grade: A"
} otherwise if score > 80 {
    say "Grade: B"
} otherwise {
    say "Grade: C"
}
```

Output:

```
Grade: B
```

2.4.5 if as an Expression

In Forge, `if` can be used as an expression that returns a value. The last expression in each branch becomes the result:

```
let age = 20
let status = if age >= 18 { "adult" } else { "minor" }
say status
```

Output:

```
adult
```

This is Forge's equivalent of the ternary operator found in other languages. It eliminates the need for a separate `? :` syntax:

```
let temperature = 25
let advice = if temperature > 30 {
    "Stay hydrated"
} else if temperature > 20 {
    "Perfect weather"
} else {
    "Bring a jacket"
}
say advice
```

Output:

```
Perfect weather
```

> **Tip:** When using `if` as an expression, always include an `else` branch. Without it, the expression would have no value when the condition is false, which could lead to unexpected `null` results.

2.4.6 when Guards

The `when` expression is unique to Forge. It provides a concise way to match a value against multiple conditions:

```
set age to 25

when age {
    < 13 -> "kid"
    < 20 -> "teen"
    < 65 -> "adult"
    else -> "senior"
}
```

Think of `when` as a multi-way conditional that tests a single subject against a series of comparison operators. Each arm uses `->` to separate the condition from the result.

Here is a more practical example:

```
let score = 87

let grade = when score {
    >= 90 -> "A"
    >= 80 -> "B"
    >= 70 -> "C"
    >= 60 -> "D"
    else -> "F"
}
say "Your grade: {grade}"
```

Output:

```
Your grade: B
```

The `when` construct is particularly useful for categorizing numeric values:

```
let http_status = 404

when http_status {
    < 200 -> say "Informational"
    < 300 -> say "Success"
    < 400 -> say "Redirect"
    < 500 -> say "Client Error"
    else -> say "Server Error"
}
```

Output:

```
Client Error
```

> **Tip:** `when` evaluates arms from top to bottom and stops at the first match, just like `else if` chains. Order matters: put the most specific conditions first.

2.4.7 Nested Conditionals

You can nest `if` statements inside other `if` statements for complex logic:

```
let age = 25
let has_id = true

if age >= 21 {
    if has_id {
        say "Welcome to the bar"
    } else {
        say "Please show your ID"
    }
} else {
    say "You must be 21 or older"
}
```

Output:

```
Welcome to the bar
```

While nesting works, deep nesting makes code hard to read. Consider flattening with `&&`:

```
let age = 25
let has_id = true

if age >= 21 && has_id {
    say "Welcome to the bar"
} else if age >= 21 {
    say "Please show your ID"
} else {
    say "You must be 21 or older"
}
```

This version communicates the same logic with less indentation.

2.4.8 Boolean Short-Circuit Evaluation

Forge uses *short-circuit evaluation* for `&&` and `||`. This means:

- `&&` stops evaluating if the left side is false (the overall result is already determined)
- `||` stops evaluating if the left side is true

```
let x = 0

if x != 0 && 10 / x > 2 {
    say "This won't cause a division by zero"
}
```

Because `x != 0` is false, the right side (`10 / x > 2`) is never evaluated, which prevents a division-by-zero error. Short-circuit evaluation is not just an optimization — it is a safety feature.

```
fn expensive_check() {
    say "This function was called"
    return true
}

if true || expensive_check() {
    say "Short-circuited"
}
```

Output:

```
Short-circuited
```

The `expensive_check()` function is never called because the left side of `||` is already `true`.

2.4.9 Combining Conditions

Complex business logic often requires combining multiple conditions. Use parentheses to group and clarify:

```
let age = 30
let is_student = true
let income = 25000

if (age < 26 || is_student) && income < 30000 {
    say "Eligible for discount"
} else {
    say "Standard pricing"
}
```

Output:

```
Eligible for discount
```

Without parentheses, operator precedence would evaluate `&&` before `||`, potentially changing the meaning. When combining logical operators, explicit parentheses prevent subtle bugs.

2.4.10 Try It Yourself

1. **Grade Calculator.** Write a program that assigns a letter grade based on a numeric score. Use `when` guards. Include grades A+ (97+), A (93+), A- (90+), B+ (87+), B (83+), B- (80+), and so on down to F (below 60).
2. **Leap Year.** A year is a leap year if it is divisible by 4, except for years divisible by 100, unless they are also divisible by 400. Write a program that determines whether a given year is a leap year using only `if/else` (no functions yet). Test with 2000, 1900, 2024, and 2023.
3. **FizzBuzz (Conditional).** Write a program that checks a single number: if it is divisible by both 3 and 5, print "FizzBuzz"; if divisible by 3 only, print "Fizz"; if divisible by 5 only, print "Buzz"; otherwise, print the number. (We will loop through many numbers in the next chapter.)

2.5 Chapter 5: Loops and Iteration

Loops let you repeat a block of code. Whether you are processing every item in a list, waiting for a condition to change, or counting from one to a million, loops are the mechanism. Forge provides five loop constructs, each suited to a different kind of repetition.

2.5.1 for/in Loops

The `for/in` loop iterates over each element in an array:

```
let fruits = ["apple", "banana", "cherry"]

for fruit in fruits {
    say "I like {fruit}"
}
```

Output:

```
I like apple
I like banana
I like cherry
```

The loop variable (`fruit`) is automatically created for each iteration. It is scoped to the loop body — it does not exist outside the loop.

2.5.2 for each (Natural Syntax)

The natural-language version adds the word `each` for readability:

```
set colors to ["red", "green", "blue"]

for each color in colors {
    say "Color: {color}"
}
```

Output:

```
Color: red
Color: green
Color: blue
```

`for each` and `for` are identical in behavior. The `each` keyword is optional syntactic sugar.

2.5.3 Iterating Over Objects

When iterating over an object, you can bind both the key and the value:

```
let user = { name: "Alice", age: 30, role: "engineer" }

for key, value in user {
    say "{key}: {value}"
}
```

Output:

```
name: Alice
age: 30
role: engineer
```

Objects in Forge are insertion-ordered, so the loop visits keys in the order they were defined. This makes object iteration predictable and useful for building reports, generating output, or transforming data.

2.5.4 while Loops

The `while` loop repeats as long as a condition is true:

```
let mut count = 0

while count < 5 {
    say "Count: {count}"
    count = count + 1
}
```

Output:

```
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
```

A `while` loop is the right choice when you don't know in advance how many iterations you need:

```
let mut n = 1
while n < 1000 {
    n = n * 2
}
say "First power of 2 >= 1000: {n}"
```

Output:

```
First power of 2 >= 1000: 1024
```

> **Tip:** Make sure your `while` condition will eventually become false. A condition that never changes creates an infinite loop that hangs your program.

2.5.5 repeat N times

When you know exactly how many times to repeat, `repeat` is the cleanest syntax:

```
repeat 5 times {
    say "Hello!"
}
```

Output:

```
Hello!
Hello!
Hello!
Hello!
Hello!
```

`repeat` is unique to Forge. It reads like a natural instruction — "repeat 5 times" — which makes it ideal for simple counted repetition without the ceremony of a counter variable:

```
set mut stars to ""
repeat 10 times {
    change stars to stars + "*"
}
say stars
```

Output:

```
**********
```

2.5.6 loop (Infinite Loop with break)

The `loop` construct creates an infinite loop. You must use `break` to exit:

```
let mut i = 0
loop {
    if i >= 5 {
        break
    }
    say "i = {i}"
    i = i + 1
}
```

Output:

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

`loop` is useful when the exit condition is complex or occurs in the middle of the loop body rather than at the top:

```
let mut sum = 0
let mut n = 1

loop {
    sum = sum + n
    if sum > 100 {
        say "Stopped at n = {n}, sum = {sum}"
        break
    }
    n = n + 1
}
```

Output:

```
Stopped at n = 14, sum = 105
```

2.5.7 break and continue

The `break` keyword exits the innermost loop immediately. The `continue` keyword skips the rest of the current iteration and moves to the next one:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for n in numbers {
    if n % 2 == 0 {
        continue
    }
    say n
}
```

Output:

```
1
3
5
7
9
```

Here, `continue` skips even numbers, so only odd numbers are printed.

```
let items = ["apple", "banana", "STOP", "cherry", "date"]

for item in items {
    if item == "STOP" {
        say "Found stop signal, exiting loop"
        break
    }
    say "Processing: {item}"
}
```

Output:

```
Processing: apple
Processing: banana
Found stop signal, exiting loop
```

2.5.8 range() for Numeric Ranges

The `range()` function generates an array of sequential integers, which you can iterate over:

```
for i in range(5) {
    say i
}
```

Output:

```
0
1
2
3
4
```

`range(n)` produces integers from 0 to n-1. You can also specify a start and end:

```
for i in range(1, 6) {
    say i
}
```

Output:

```
1
2
3
4
5
```

Use `range()` whenever you need a numeric counter in a `for` loop:

```
let mut sum = 0
for i in range(1, 101) {
    sum += i
}
say "Sum of 1 to 100: {sum}"
```

Output:

```
Sum of 1 to 100: 5050
```

2.5.9 enumerate() for Indexed Iteration

When you need both the index and the value, use `enumerate()`:

```
let languages = ["Rust", "Forge", "Go", "Python"]

for i, lang in enumerate(languages) {
    say "{i}: {lang}"
}
```

Output:

```
0: Rust
1: Forge
2: Go
3: Python
```

`enumerate()` wraps each element with its zero-based index, giving you both pieces of information without maintaining a separate counter.

2.5.10 Nested Loops

Loops can be nested inside other loops:

```
for i in range(1, 4) {
    for j in range(1, 4) {
        let product = i * j
        print("{product}\t")
    }
    println("")
}
```

Output:

```
1   2   3
2   4   6
3   6   9
```

When using `break` or `continue` in nested loops, they apply to the *innermost* enclosing loop:

```
for i in range(3) {
    for j in range(3) {
        if j == 1 {
            break
        }
        say "i={i}, j={j}"
    }
}
```

Output:

```
i=0, j=0
i=1, j=0
i=2, j=0
```

The `break` exits only the inner loop. The outer loop continues to the next iteration.

2.5.11 Choosing the Right Loop

Scenario	Best Loop
Process every item in a collection	`for item in array`
Fixed number of repetitions	`repeat N times`
Repeat until a condition changes	`while condition`
Complex exit logic	`loop` with `break`
Count through numbers	`for i in range(n)`
Need index + value	`for i, v in enumerate(array)`

2.5.12 Try It Yourself

1. **Multiplication Table.** Write a program that prints a 10x10 multiplication table using nested `for` loops with `range()`.
2. **FizzBuzz Complete.** Using a `for` loop over `range(1, 101)`, print "Fizz" for multiples of 3, "Buzz" for multiples of 5, "FizzBuzz" for multiples of both, and the number itself otherwise.
3. **Search and Stop.** Create an array of 10 city names. Use a `for` loop with `break` to search for a specific city. Print "Found [city] at index [i]" using `enumerate()` and stop searching once found. If the city is not in the list, print "Not found."

2.6 Chapter 6: Functions and Closures

Functions are the primary unit of code organization in Forge. They let you name a block of code, give it parameters, and call it from anywhere. This chapter covers function definition, closures, higher-order functions, recursion, and decorators.

2.6.1 Defining Functions

Forge provides two syntaxes for defining functions:

Classic syntax uses `fn`:

```
fn greet(name) {
    println("Hello, {name}!")
}

greet("World")
```

Natural syntax uses `define`:

```
define greet(name) {
    say "Hello, {name}!"
}

greet("World")
```

Both produce identical functions. The function name, parameter list, and body are the same — only the keyword differs.

2.6.2 Parameters and Return Values

Functions accept zero or more parameters and optionally return a value:

```
fn add(a, b) {
    return a + b
}

let result = add(3, 4)
say result
```

Output:

```
7
```

If a function has no explicit `return` statement, it returns `null`:

```
fn log_message(msg) {
    println("[LOG] {msg}")
}

let result = log_message("server started")
say typeof(result)
```

Output:

```
[LOG] server started
Null
```

Functions can return early:

```
fn classify(n) {
    if n < 0 {
        return "negative"
    }
    if n == 0 {
        return "zero"
    }
    return "positive"
}

say classify(-5)
say classify(0)
say classify(42)
```

Output:

```
negative
zero
positive
```

2.6.3 Type-Annotated Functions

Add type annotations to parameters and return values for documentation and safety:

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

fn format_price(amount: Float) -> String {
```

```
    return "${amount}"
}

say add(10, 20)
say format_price(9.99)
```

Output:

```
30
$9.99
```

Annotations are optional. Use them when the function's purpose is not obvious from its name and parameter names alone. They are especially valuable in public APIs and shared codebases.

2.6.4 Multiple Return Values

Forge functions can only return a single value, but you can use arrays or objects to return multiple pieces of data:

```
fn min_max(numbers) {
    let mut lo = numbers[0]
    let mut hi = numbers[0]
    for n in numbers {
        if n < lo { lo = n }
        if n > hi { hi = n }
    }
    return { min: lo, max: hi }
}

let result = min_max([4, 7, 1, 9, 3])
say "Min: {result.min}, Max: {result.max}"
```

Output:

```
Min: 1, Max: 9
```

Using objects for multiple return values is idiomatic in Forge because the field names document what each value represents.

2.6.5 Closures

A closure is an anonymous function that captures variables from its surrounding scope:

```
let double = fn(x) { return x * 2 }
say double(21)
```

Output:

```
42
```

Closures can capture variables from the enclosing function:

```
fn make_adder(n) {
    return fn(x) {
        return x + n
    }
}

let add5 = make_adder(5)
let add10 = make_adder(10)

say add5(3)
say add10(3)
```

Output:

```
8
13
```

Each call to `make_adder` creates a new closure that remembers the value of `n`. The closure "closes over" the variable — hence the name. This is a powerful pattern for creating specialized functions from a general template.

> **Tip:** Think of a closure as a function bundled with a snapshot of its environment. The captured variables travel with the closure wherever it goes.

2.6.6 Higher-Order Functions

A higher-order function is a function that takes another function as a parameter or returns one. We just saw an example with `make_adder` (which returns a function). Here is one that accepts a function:

```
fn apply(f, value) {
    return f(value)
}

fn square(x) { return x * x }
```

```
fn negate(x) { return -x }

say apply(square, 7)
say apply(negate, 42)
```

Output:

```
49
-42
```

Higher-order functions are the foundation of functional programming in Forge. The built-in `map`, `filter`, and `reduce` functions (covered in Chapter 7) are all higher-order functions.

```
let numbers = [1, 2, 3, 4, 5]

let doubled = map(numbers, fn(x) { return x * 2 })
say doubled

let evens = filter(numbers, fn(x) { return x % 2 == 0 })
say evens
```

Output:

```
[2, 4, 6, 8, 10]
[2, 4]
```

2.6.7 Recursion

A recursive function calls itself. It must have a base case that stops the recursion and a recursive case that makes progress toward the base case:

```
fn factorial(n) {
    if n <= 1 {
        return 1
    }
    return n * factorial(n - 1)
}

say factorial(5)
say factorial(10)
```

Output:

```
120
3628800
```

Here is the classic Fibonacci sequence:

```
fn fib(n) {
    if n <= 1 {
        return n
    }
    return fib(n - 1) + fib(n - 2)
}

for i in range(10) {
    let f = fib(i)
    println("fib({i}) = {f}")
}
```

Output:

```
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
```

> **Tip:** The naive Fibonacci implementation has exponential time complexity. For production code, use memoization or an iterative approach. Recursion is a teaching tool here, not a performance recommendation.

2.6.8 Iterative Fibonacci (for comparison)

```
fn fib_fast(n) {
    if n <= 1 { return n }
    let mut a = 0
    let mut b = 1
    for i in range(2, n + 1) {
        let temp = a + b
        a = b
        b = temp
    }
    return b
}

say fib_fast(50)
```

2.6.9 Decorators

Forge supports decorators — annotations prefixed with `@` that modify or categorize functions. The most common decorators are for testing and HTTP routing:

Test decorator:

```
@test
fn test_addition() {
    assert_eq(2 + 2, 4)
}

@test
define test_string_length() {
    assert_eq(len("forge"), 5)
}
```

Run tests with `forge test`.

HTTP decorators:

```
@server(port: 8080)

@get("/hello/:name")
fn hello(name: String) -> Json {
    return { greeting: "Hello, {name}!" }
}

@post("/echo")
fn echo(body: Json) -> Json {
    return body
}
```

Decorators are declarative metadata. They tell Forge *what* the function is used for without cluttering the function body with framework-specific code. We will explore HTTP decorators in detail in a later part of this book.

2.6.10 Functions as First-Class Values

In Forge, functions are values. You can store them in variables, put them in arrays, pass them as arguments, and return them from other functions:

```
fn greet(name) {
    return "Hello, {name}!"
}

let my_func = greet
say my_func("Forge")
```

Output:

```
Hello, Forge!
```

Storing functions in data structures:

```
fn add(a, b) { return a + b }
fn sub(a, b) { return a - b }
fn mul(a, b) { return a * b }

let operations = [add, sub, mul]
let names = ["add", "sub", "mul"]

for i, op in enumerate(operations) {
    let result = op(10, 3)
    let name = names[i]
    say "{name}(10, 3) = {result}"
}
```

Output:

```
add(10, 3) = 13
sub(10, 3) = 7
mul(10, 3) = 30
```

2.6.11 Compact Function Bodies

For simple functions, you can write the body on a single line:

```
fn double(x) { return x * 2 }
fn is_positive(x) { return x > 0 }
fn identity(x) { return x }
```

This keeps utility functions compact without sacrificing readability.

2.6.12 Try It Yourself

1. **Temperature Converter.** Write two functions: `celsius_to_fahrenheit(c)` and `fahrenheit_to_celsius(f`
   Use the formulas F = C × 9/5 + 32 and C = (F - 32) × 5/9. Test with `0°C`, `100°C`, `32°F`, and `212°F`.
2. **Closure Counter.** Write a function `make_counter()` that returns a closure. Each time the closure is called, it should return the next integer starting from 1. Calling the returned closure four times should produce 1, 2, 3, 4. (Hint: the closure captures a mutable variable.)
3. **Apply Twice.** Write a function `apply_twice(f, x)` that applies function `f` to `x` two times — i.e., it computes `f(f(x))`. Test it with a function that adds 3 and an initial value of 7 (expected result: 13). Then test it with a function that doubles its input and an initial value of 5 (expected result: 20).

2.7 Chapter 7: Collections

Collections are data structures that hold multiple values. Forge has two primary collection types: arrays (ordered lists) and objects (key-value maps). This chapter covers both in depth, including the functional operations that make collection processing concise and expressive.

2.7.1 Arrays

An array is an ordered, zero-indexed sequence of values:

```
let numbers = [1, 2, 3, 4, 5]
let names = ["Alice", "Bob", "Charlie"]
let mixed = [1, "hello", true, 3.14]
let empty = []
```

Arrays can hold values of any type, including other arrays:

```
let matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
say matrix[1][2]
```

Output:

```
6
```

2.7.2 Array Access and Modification

Access elements by index (zero-based):

```
let fruits = ["apple", "banana", "cherry"]
say fruits[0]
say fruits[1]
say fruits[2]
```

Output:

```
apple
banana
cherry
```

Modify elements by assigning to an index:

```
let mut colors = ["red", "green", "blue"]
colors[1] = "yellow"
say colors
```

Output:

```
[red, yellow, blue]
```

2.7.3 Array Built-in Operations

Forge provides a rich set of built-in functions for working with arrays:

```
let mut items = [3, 1, 4, 1, 5, 9, 2, 6]

say len(items)
say sort(items)
say reverse(items)
say contains(items, 5)
say contains(items, 99)
```

Output:

```
8
[1, 1, 2, 3, 4, 5, 6, 9]
[6, 2, 9, 5, 1, 4, 1, 3]
true
false
```

Mutating operations — `push` and `pop`:

```
let mut stack = [1, 2, 3]
push(stack, 4)
say stack

let top = pop(stack)
say "Popped: {top}"
say stack
```

Output:

```
[1, 2, 3, 4]
Popped: 4
[1, 2, 3]
```

Here is a complete reference of array operations:

Function	Description	Example
`len(arr)`	Number of elements	`len([1,2,3])` ☐ `3`
`push(arr, val)`	Add to end (mutates)	`push(arr, 4)`
`pop(arr)`	Remove and return last	`pop(arr)` ☐ last element
`sort(arr)`	Return sorted copy	`sort([3,1,2])` ☐ `[1,2,3]`
`reverse(arr)`	Return reversed copy	`reverse([1,2,3])` ☐ `[3,2,1]`
`contains(arr, val)`	Check membership	`contains([1,2], 2)` ☐ `true`
`range(n)`	Generate `[0..n-1]`	`range(3)` ☐ `[0,1,2]`
`enumerate(arr)`	Pairs of `(index, value)`	See Chapter 5

2.7.4 map — Transform Every Element

The `map` function applies a transformation function to every element and returns a new array:

```
let numbers = [1, 2, 3, 4, 5]
let doubled = map(numbers, fn(x) { return x * 2 })
say doubled
```

Output:

```
[2, 4, 6, 8, 10]
```

```
let names = ["alice", "bob", "charlie"]
let lengths = map(names, fn(name) { return len(name) })
say lengths
```

Output:

```
[5, 3, 7]
```

`map` never modifies the original array. It always returns a new one.

2.7.5 filter — Select Matching Elements

The `filter` function keeps only elements for which a predicate returns true:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
let evens = filter(numbers, fn(x) { return x % 2 == 0 })
say evens
```

Output:

```
[2, 4, 6, 8, 10]
```

```
let words = ["hello", "hi", "hey", "howdy", "greetings"]
let short_words = filter(words, fn(w) { return len(w) <= 3 })
say short_words
```

Output:

```
[hi, hey]
```

2.7.6 reduce — Combine Into a Single Value

The `reduce` function collapses an array into a single value by applying a function cumulatively:

```
let numbers = [1, 2, 3, 4, 5]
let sum = reduce(numbers, 0, fn(acc, x) { return acc + x })
say sum
```

Output:

```
15
```

The second argument (`0`) is the initial value of the accumulator. The function receives the accumulator and the current element, and returns the new accumulator value.

```
let numbers = [3, 7, 2, 9, 4]
let maximum = reduce(numbers, numbers[0], fn(max, x) {
    if x > max { return x }
    return max
})
say "Maximum: {maximum}"
```

Output:

```
Maximum: 9
```

> **Tip:** Think of `reduce` as "folding" a list into a single value. The accumulator carries the running result, and each element updates it. This pattern is extraordinarily powerful — almost any array processing can be expressed as a `reduce`.

2.7.7 Chaining Functional Operations

The real power of `map`, `filter`, and `reduce` emerges when you chain them together:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

let evens = filter(numbers, fn(x) { return x % 2 == 0 })
let doubled = map(evens, fn(x) { return x * 2 })
let total = reduce(doubled, 0, fn(acc, x) { return acc + x })

say "Sum of doubled evens: {total}"
```

Output:

```
Sum of doubled evens: 60
```

Step by step:

1. `filter` keeps `[2, 4, 6, 8, 10]`
2. `map` produces `[4, 8, 12, 16, 20]`
3. `reduce` sums to `60`

This is a data pipeline — each operation transforms the data and passes it to the next.

2.7.8 Method Chaining

These functional operations also work as methods. You can call them directly on arrays and objects for a fluent, chainable style:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

let result = numbers.filter(fn(x) { return x % 2 == 0 }).map(fn(x) { return x * 2 })
say result
```

Output:

```
[4, 8, 12, 16, 20]
```

`array.find(fn)` — find first element matching the predicate:

```
let nums = [3, 7, 2, 9, 4]
let first_big = nums.find(fn(x) { return x > 5 })
say first_big
```

Output:

```
7
```

Object helper functions work as methods too:

```
let user = { name: "Alice", age: 30, email: "alice@example.com" }

say user.pick(["name", "email"])
say user.omit(["age"])
say user.merge({ role: "engineer" })
say user.has_key("name")
say user.get("age", 0)
```

Output:

```
{name: Alice, email: alice@example.com}
{name: Alice, email: alice@example.com}
{name: Alice, age: 30, email: alice@example.com, role: engineer}
true
30
```

`flat_map(array, fn)` — map and flatten in one step. The function must return an array for each element; the results are concatenated:

```
let words = ["hello", "world"]
let letters = flat_map(words, fn(w) { return split(w, "") })
say letters
```

Output:

```
[h, e, l, l, o, w, o, r, l, d]
```

2.7.9 Objects

An object is an insertion-ordered collection of key-value pairs, similar to JSON objects:

```
let user = {
    name: "Alice",
    age: 30,
    role: "engineer"
}

say user.name
say user.age
say user.role
```

Output:

```
Alice
30
engineer
```

Objects use the syntax `{ key: value, key: value }`. Keys are unquoted identifiers; values can be any Forge type.

2.7.10 Nested Objects

Objects can contain other objects:

```
let company = {
    name: "Acme Corp",
    address: {
        street: "123 Main St",
        city: "Portland",
        state: "OR"
    },
    founded: 2020
}

say company.name
say company.address.city
say company.address.state
```

Output:

```
Acme Corp
Portland
OR
```

2.7.11 Object Operations

```
let config = { host: "localhost", port: 8080, debug: true }

say keys(config)
say values(config)
say len(config)
```

Output:

```
[host, port, debug]
[localhost, 8080, true]
3
```

Function	Description	Example
`keys(obj)`	Array of key names	`keys({a: 1})` ☐ `["a"]`
`values(obj)`	Array of values	`values({a: 1})` ☐ `[1]`
`len(obj)`	Number of keys	`len({a: 1, b: 2})` ☐ `2`

2.7.12 Object Helper Functions

Forge provides helper functions that make object manipulation safer and more expressive:

`has_key(object, key)` — returns `true` if the key exists:

```
let user = { name: "Alice", age: 30 }
say has_key(user, "name")
say has_key(user, "email")
```

Output:

```
true
false
```

`get(object, key, default)` — safe access with fallback. Supports dot-paths for nested access:

```
let config = { a: { b: { c: "found" } } }
say get(config, "a.b.c", "fallback")
say get(config, "a.b.missing", "fallback")
```

Output:

```
found
fallback
```

`pick(object, [keys])` — extract specific fields into a new object:

```
let user = { name: "Alice", age: 30, role: "engineer" }
let subset = pick(user, ["name", "role"])
say subset
```

Output:

```
{name: Alice, role: engineer}
```

`omit(object, [keys])` — remove specific fields, return a new object:

```
let user = { name: "Alice", age: 30, role: "engineer" }
let without_age = omit(user, ["age"])
say without_age
```

Output:

```
{name: Alice, role: engineer}
```

`merge(obj1, obj2, ...)` — combine objects. Later objects win on key conflicts:

```
let defaults = { theme: "dark", fontSize: 14 }
let overrides = { fontSize: 18 }
let merged = merge(defaults, overrides)
say merged
```

Output:

```
{theme: dark, fontSize: 18}
```

`entries(object)` — convert to an array of `[key, value]` pairs:

```
let scores = { alice: 95, bob: 87 }
say entries(scores)
```

Output:

```
[[alice, 95], [bob, 87]]
```

`from_entries(pairs)` — convert pairs back to an object:

```
let pairs = [["x", 1], ["y", 2], ["z", 3]]
let obj = from_entries(pairs)
say obj
```

Output:

```
{x: 1, y: 2, z: 3}
```

`contains(object, key)` — check if a key exists. Also works on strings (substring) and arrays (membership):

```
let data = { a: 1, b: 2 }
say contains(data, "a")
say contains("hello", "ell")
say contains([1, 2, 3], 2)
```

Output:

```
true
true
true
```

2.7.13 Object Iteration

Iterate over an object to access both keys and values:

```
let scores = { alice: 95, bob: 87, charlie: 92 }

for name, score in scores {
    say "{name} scored {score}"
}
```

Output:

```
alice scored 95
bob scored 87
charlie scored 92
```

2.7.14 Arrays of Objects

One of the most common data patterns is an array of objects — essentially a table of records:

```
let employees = [
    { name: "Alice", department: "Engineering", salary: 95000 },
    { name: "Bob", department: "Design", salary: 82000 },
    { name: "Charlie", department: "Engineering", salary: 105000 },
    { name: "Diana", department: "Marketing", salary: 78000 },
    { name: "Eve", department: "Engineering", salary: 98000 }
]

let engineers = filter(employees, fn(e) {
    return e.department == "Engineering"
})

say "Engineers: {len(engineers)}"
```

```
for e in engineers {
    say "  {e.name}: ${e.salary}"
}
```

Output:

```
Engineers: 3
  Alice: $95000
  Charlie: $105000
  Eve: $98000
```

2.7.15 Building Data Pipelines with Collections

Combining arrays, objects, and functional operations creates powerful data processing pipelines:

```
let orders = [
    { product: "Widget", quantity: 5, price: 9.99 },
    { product: "Gadget", quantity: 2, price: 24.99 },
    { product: "Doohickey", quantity: 10, price: 4.99 },
    { product: "Thingamajig", quantity: 1, price: 49.99 },
    { product: "Widget", quantity: 3, price: 9.99 }
]

let totals = map(orders, fn(order) {
    return {
        product: order.product,
        total: order.quantity * order.price
    }
})

let grand_total = reduce(totals, 0.0, fn(acc, item) {
    return acc + item.total
})

say "Order Summary:"
for item in totals {
    say "  {item.product}: ${item.total}"
}
say "Grand Total: ${grand_total}"
```

Output:

```
Order Summary:
  Widget: $49.95
  Gadget: $49.98
```

```
  Doohickey: $49.9
  Thingamajig: $49.99
  Widget: $29.97
Grand Total: $229.79
```

Here is another pipeline that filters, transforms, and summarizes:

```
let students = [
    { name: "Alice", grade: 92 },
    { name: "Bob", grade: 78 },
    { name: "Charlie", grade: 95 },
    { name: "Diana", grade: 88 },
    { name: "Eve", grade: 71 }
]

let honor_roll = filter(students, fn(s) { return s.grade >= 90 })
let honor_names = map(honor_roll, fn(s) { return s.name })
say "Honor Roll: {honor_names}"

let grades = map(students, fn(s) { return s.grade })
let avg = reduce(grades, 0, fn(acc, g) { return acc + g }) / len(students)
say "Class Average: {avg}"
```

Output:

```
Honor Roll: [Alice, Charlie]
Class Average: 84
```

2.7.16 String Operations as Collection Tools

Strings behave like collections of characters in some contexts. Several built-in functions bridge between strings and arrays:

```
let sentence = "hello world from forge"
let words = split(sentence, " ")
say words

let result = join(words, "-")
say result

say replace(sentence, "forge", "Forge")
say starts_with(sentence, "hello")
say ends_with(sentence, "forge")
```

Output:

```
[hello, world, from, forge]
hello-world-from-forge
hello world from Forge
true
true
```

2.7.17 The lines() Function

The `lines()` function splits a string on newline characters and returns an array of lines:

```
let text = "line1\nline2\nline3"
let lines = lines(text)
say lines
```

Output:

```
[line1, line2, line3]
```

Useful for processing multi-line input, log files, or any text with line breaks.

2.7.18 The find() Function

The `find(array, fn)` function returns the first element that matches the predicate, or `null` if none match:

```
let numbers = [3, 7, 2, 9, 4]
let first = find(numbers, fn(x) { return x > 5 })
say first

let none = find(numbers, fn(x) { return x > 100 })
say none
```

Output:

```
7
null
```

It also works as a method:

```
let nums = [1, 4, 9, 16, 25]
let first_large = nums.find(fn(x) { return x > 10 })
say first_large
```

Output:

```
16
```

2.7.19 Try It Yourself

1. **Word Counter.** Given the string `"the quick brown fox jumps over the lazy dog"`, split it into words, then use `reduce` to count how many words have more than 3 letters. (Expected: 6.)
2. **Student Report.** Create an array of 5 student objects, each with `name` and `score` fields. Use `filter` to find students with scores above 85, `map` to create a greeting for each ("Congratulations, [name]!"), and print the results.
3. **Object Builder.** Write a program that starts with an empty array, uses `push` to add 5 objects (each with `id` and `value` fields), then uses `reduce` to compute the sum of all `value` fields. Print the array and the total.

2.8 Chapter 8: Error Handling

Most programs encounter errors: files that don't exist, network connections that fail, invalid user input. Forge takes the position that errors should be *values*, not invisible exceptions that surprise you. This chapter covers Forge's comprehensive error-handling system, from Result types to the `safe` block.

2.8.1 Philosophy: Errors as Values

In many languages, errors are *exceptions* — they fly up the call stack invisibly until something catches them, or they crash the program. This model has two problems: you can't tell which functions might throw just by reading the code, and forgetting to catch an exception means a silent crash.

Forge follows the errors-as-values philosophy pioneered by Rust and Go. A function that can fail returns a `Result` — a wrapper that is either `Ok(value)` on success or `Err("message")` on failure. You handle the result explicitly, and the compiler helps you remember.

Think of it like ordering food. In the exception model, you order and hope for the best — if the kitchen is on fire, someone runs out screaming. In the errors-as-values model, the waiter brings you a tray with either your food or a note explaining what went wrong. Either way, you know what happened.

2.8.2 Result Types: Ok and Err

Create successful and failed results:

```
let success = Ok(42)
let failure = Err("something went wrong")
```

```
say success
say failure
```

Output:

```
Ok(42)
Err(something went wrong)
```

2.8.3 Creating and Inspecting Results

Functions that can fail conventionally return `Ok` or `Err`:

```
fn safe_divide(a, b) {
    if b == 0 {
        return Err("division by zero")
    }
    return Ok(a / b)
}

let result1 = safe_divide(10, 2)
let result2 = safe_divide(10, 0)

say result1
say result2
say is_ok(result1)
say is_err(result2)
```

Output:

```
Ok(5)
Err(division by zero)
true
true
```

Use `is_ok()` and `is_err()` to check the state of a Result before extracting its value:

```
let result = safe_divide(42, 6)
if is_ok(result) {
    say "Value: {unwrap(result)}"
}
```

Output:

```
Value: 7
```

2.8.4 Pattern Matching on Results

The most common way to handle Results is with `match`:

```
fn parse_positive(input) {
    let n = int(input)
    if n < 0 {
        return Err("expected a positive integer")
    }
    return Ok(n)
}

match parse_positive("42") {
    Ok(value) => say "Got: {value}"
    Err(msg) => say "Error: {msg}"
}

match parse_positive("-5") {
    Ok(value) => say "Got: {value}"
    Err(msg) => say "Error: {msg}"
}
```

Output:

```
Got: 42
Error: expected a positive integer
```

Pattern matching forces you to handle both cases. You cannot accidentally ignore an error because the `match` expression requires arms for both `Ok` and `Err`.

2.8.5 The ? Operator (Error Propagation)

The `?` operator is Forge's most ergonomic error-handling tool. When applied to a Result, it:

- Extracts the value if the Result is `Ok`
- Immediately returns the `Err` from the enclosing function if the Result is `Err`

```
fn parse_positive_int(input) {
    let n = int(input)
    if n < 0 {
        return Err("expected a positive integer")
    }
    return Ok(n)
}

fn double_positive(input) {
```

```
    let n = parse_positive_int(input)?
    return Ok(n * 2)
}

let good = double_positive("21")
let bad = double_positive("-5")

say good
say bad
```

Output:

```
Ok(42)
Err(expected a positive integer)
```

Without `?`, you would have to manually check every result:

```
fn double_positive_verbose(input) {
    let result = parse_positive_int(input)
    if is_err(result) {
        return result
    }
    let n = unwrap(result)
    return Ok(n * 2)
}
```

The `?` operator collapses those three lines into one. This is especially valuable when you have multiple operations that can fail:

```
fn process_data(a_str, b_str) {
    let a = parse_positive_int(a_str)?
    let b = parse_positive_int(b_str)?
    if b == 0 {
        return Err("second value cannot be zero")
    }
    return Ok(a / b)
}

say process_data("10", "3")
say process_data("10", "-1")
say process_data("10", "0")
```

Output:

```
Ok(3)
Err(expected a positive integer)
Err(second value cannot be zero)
```

> **Tip:** The `?` operator only works inside functions that return a `Result`. If you use it in top-level code, the error will propagate as a runtime error.

2.8.6 try/catch Blocks

For situations where you want to handle errors from code that might crash (like division by zero), use `try/catch`:

```
try {
    let x = 1 / 0
} catch err {
    say "Caught: {err}"
}
```

Output:

```
Caught: division by zero
```

The `try` block runs the code inside it. If a runtime error occurs, execution jumps to the `catch` block, which receives the error message as a string:

```
try {
    let data = [1, 2, 3]
    say data[10]
} catch err {
    say "Error accessing array: {err}"
}
say "Program continues normally"
```

Output:

```
Error accessing array: index out of bounds
Program continues normally
```

2.8.7 safe Blocks

The `safe` block is Forge's simplest error suppression mechanism. Code inside a `safe` block runs, but any error is silently caught and the program continues:

```
safe {
    let x = 1 / 0
}
say "I survived a division by zero!"
```

Output:

```
I survived a division by zero!
```

`safe` is useful for operations where failure is acceptable — fire-and-forget logging, optional cleanup, best-effort operations. Use it sparingly, as silencing errors can mask bugs.

> **Tip:** Prefer `try/catch` over `safe` when you want to know *what* went wrong. Use `safe` only when you genuinely don't care whether the code succeeds.

2.8.8 The must Keyword

The `must` keyword is the opposite of `safe` — it asserts that a Result is `Ok` and crashes the program if it is not:

```
let value = must Ok(42)
say "Value: {value}"
```

Output:

```
Value: 42
```

If the Result is an `Err`, `must` terminates the program with a clear error message:

```
let value = must Err("catastrophic failure")
```

This would crash with a message about the error. Use `must` when an error is truly unrecoverable — for example, failing to read a configuration file that your program cannot function without:

```
fn load_config(path) {
    if !fs.exists(path) {
        return Err("config file not found: {path}")
    }
    let content = fs.read(path)
    return Ok(content)
}

let config = must load_config("app.toml")
```

2.8.9 The check Statement

The `check` statement performs declarative validation:

```
let name = "Alice"
check name
```

If `name` were empty, `check` would raise an error. The `check` statement validates that a value is truthy — it is a concise way to assert preconditions:

```
fn create_user(name, email) {
    check name
    check email
    return { name: name, email: email }
}

let user = create_user("Alice", "alice@example.com")
say user
```

Output:

```
{name: Alice, email: alice@example.com}
```

2.8.10 unwrap and unwrap_or

The `unwrap()` function extracts the value from an `Ok` result. If the result is `Err`, it crashes:

```
let result = Ok(42)
say unwrap(result)
```

Output:

```
42
```

For a safer alternative, use `unwrap_or()` to provide a default value:

```
let good = Ok(42)
let bad = Err("failed")

say unwrap_or(good, 0)
say unwrap_or(bad, 0)
```

Output:

```
42
0
```

`unwrap_or` never crashes. If the Result is `Err`, it returns the default value instead.

2.8.11 Option Types

Forge also has `Option` types for values that may or may not exist:

```
let x = Some(42)
let y = None

say is_some(x)
say is_none(y)

match x {
    Some(val) => say "Got: {val}"
    None => say "Nothing"
}
```

Output:

```
true
true
Got: 42
```

Options are used when a value might be absent without that being an error. For example, looking up a key in a map might return `Some(value)` or `None`.

2.8.12 Error Messages and Suggestions

Forge strives to produce helpful error messages. When you make a common mistake, the runtime often suggests a correction:

```
let x = 10
x = 20
```

Error:

```
cannot reassign immutable variable 'x'
  hint: declare with 'let mut x' to make it mutable
```

Division by zero:

```
division by zero
  hint: check that the divisor is not zero before dividing
```

These contextual hints are part of Forge's design philosophy: errors should teach, not just complain.

2.8.13 Best Practices

1. **Use Result types for functions that can fail.** Return `Ok` on success, `Err` on failure. This makes the failure mode visible in the function signature.
2. **Use `?` to propagate errors.** Don't manually check every result. The `?` operator keeps your code clean and ensures errors bubble up naturally.
3. **Use `match` for handling Results.** It forces you to consider both the success and failure cases.
4. **Reserve `must` for truly unrecoverable errors.** Configuration loading, database connection — things the program cannot proceed without.
5. **Use `safe` sparingly.** Silencing errors is occasionally necessary, but most errors deserve to be handled explicitly.
6. **Use `try/catch` for code that might crash unexpectedly.** Division by zero, array index out of bounds, type conversion failures.
7. **Prefer `unwrap_or` over `unwrap`.** It provides a graceful fallback instead of crashing.

```
fn read_config(path) {
    if !fs.exists(path) {
        return Err("config file not found")
    }
    return Ok(fs.read(path))
}

fn start_server() {
    let config = read_config("server.toml")?
    say "Starting with config: {config}"
    return Ok(true)
}

match start_server() {
    Ok(_) => say "Server started"
    Err(msg) => say "Failed to start: {msg}"
}
```

This pattern — functions returning Results, `?` propagating errors, `match` handling them at the top level — is the idiomatic way to handle errors in Forge.

2.8.14 Try It Yourself

1. **Safe Division Chain.** Write a function `chain_divide(a, b, c)` that divides `a` by `b`, then divides the result by `c`. Both divisions should be done with a `safe_divide` function that returns `Err` on division by zero. Use the `?` operator to propagate errors. Test with `chain_divide(100, 5, 2)` (expected: `Ok(10)`), `chain_divide(100, 0, 2)` (expected: `Err`), and `chain_divide(100, 5, 0)` (expected: `Err`).
2. **Graceful Defaults.** Write a program that tries to parse three strings as integers using a function that returns `Result`. Use `unwrap_or` to provide a default of `0` for any string that fails to parse. Compute and print the sum. Test with `["42", "not_a_number", "8"]` (expected sum: 50).

3. **Error Reporter.** Write a function `validate_user(name, age_str)` that returns `Err` if the name is empty, `Err` if the age string cannot be parsed as an integer, and `Err` if the age is negative. On success, return `Ok({ name: name, age: age })`. Test with valid input, empty name, invalid age string, and negative age. Use `match` to print a specific message for each case.

Chapter 3

PART II: THE STANDARD LIBRARY

Forge ships with fifteen built-in modules that cover the tasks programmers encounter daily—mathematics, file I/O, cryptography, databases, serialization, and terminal presentation. These modules require no imports; they are available the moment your program starts. You access them through dot notation (`module.function()`), and they follow consistent conventions: functions that can fail return meaningful error messages, types are coerced sensibly, and side effects are kept explicit.

Part II is both a reference and a cookbook. Each chapter documents every function a module offers, then closes with recipes that combine those functions into real-world patterns. Read the chapters front to back when learning a module, or jump straight to the reference tables when you need a reminder.

3.1 Chapter 9: math — Numbers and Computation

Mathematics is the bedrock of programming, and Forge's `math` module provides the essential toolkit: constants, arithmetic helpers, trigonometric functions, and random number generation. Every function in the module accepts both `Int` and `Float` arguments, coercing integers to floating-point where the result demands it. The module covers the same ground as a scientific calculator—enough to build simulations, games, data analysis pipelines, and engineering tools without reaching for an external library.

3.1.1 Constants

The `math` module exposes three constants as properties, not functions. Access them directly.

Constant	Value	Description
`math.pi`	`3.141592653589793`	The ratio of a circle's circumference to its diameter (π)
`math.e`	`2.718281828459045`	Euler's number, the base of natural logarithms
`math.inf`	`Infinity`	Positive infinity, useful for comparisons and initial bounds

```
let pi = math.pi
let e = math.e
```

```
let inf = math.inf
say "π = {pi}"
say "e = {e}"
say "∞ = {inf}"
```

Output:

```
π = 3.141592653589793
e = 2.718281828459045
∞ = inf
```

3.1.2 Function Reference

Function	Description	Example	Return Type
`math.sqrt(n)`	Square root	`math.sqrt(144)` ☐ `12.0`	Float
`math.pow(b, exp)`	Raise `b` to the power `exp`	`math.pow(2, 10)` ☐ `1024`	Int or Float
`math.abs(n)`	Absolute value	`math.abs(-42)` ☐ `42`	Int or Float
`math.max(a, b)`	Larger of two values	`math.max(3, 7)` ☐ `7`	Int or Float
`math.min(a, b)`	Smaller of two values	`math.min(3, 7)` ☐ `3`	Int or Float
`math.floor(n)`	Round down to nearest integer	`math.floor(9.7)` ☐ `9`	Int
`math.ceil(n)`	Round up to nearest integer	`math.ceil(9.2)` ☐ `10`	Int
`math.round(n)`	Round to nearest integer	`math.round(9.5)` ☐ `10`	Int
`math.random()`	Pseudorandom float in [0, 1)	`math.random()` ☐ `0.7382...`	Float
`math.sin(n)`	Sine (radians)	`math.sin(0)` ☐ `0.0`	Float
`math.cos(n)`	Cosine (radians)	`math.cos(0)` ☐ `1.0`	Float
`math.tan(n)`	Tangent (radians)	`math.tan(0)` ☐ `0.0`	Float
`math.log(n)`	Natural logarithm (base *e*)	`math.log(1)` ☐ `0.0`	Float

> **Type Preservation.** Functions like `abs`, `max`, `min`, `pow` preserve the input type when both arguments are integers. Pass a float to force a float result: `math.pow(2.0, 10)` returns `1024.0`.

3.1.3 Core Examples

Square roots and powers:

```
let hyp = math.sqrt(9.0 + 16.0)
say "Hypotenuse: {hyp}"

let kb = math.pow(2, 10)
```

```
say "1 KB = {kb} bytes"

let vol = math.pow(3.0, 3.0)
say "Volume of 3³ cube: {vol}"
```

Output:

```
Hypotenuse: 5.0
1 KB = 1024 bytes
Volume of 3³ cube: 27.0
```

Rounding family:

```
let price = 19.95
let floored = math.floor(price)
let ceiled = math.ceil(price)
let rounded = math.round(price)
say "floor({price}) = {floored}"
say "ceil({price}) = {ceiled}"
say "round({price}) = {rounded}"
```

Output:

```
floor(19.95) = 19
ceil(19.95) = 20
round(19.95) = 20
```

Trigonometry:

```
let angle = math.pi / 4.0
let s = math.sin(angle)
let c = math.cos(angle)
let t = math.tan(angle)
say "sin(π/4) = {s}"
say "cos(π/4) = {c}"
say "tan(π/4) = {t}"
```

Output:

```
sin(π/4) = 0.7071067811865476
cos(π/4) = 0.7071067811865476
tan(π/4) = 0.9999999999999999
```

Absolute value and bounds:

```
let delta = -17
let abs_delta = math.abs(delta)
say "Distance from zero: {abs_delta}"

let high = math.max(100, 250)
let low = math.min(100, 250)
say "Range: {low} to {high}"
```

Output:

```
Distance from zero: 17
Range: 100 to 250
```

Natural logarithm:

```
let ln2 = math.log(2)
let ln10 = math.log(10)
say "ln(2) = {ln2}"
say "ln(10) = {ln10}"

// log base 10 via change-of-base
let log10_of_1000 = math.log(1000) / math.log(10)
say "log₁₀(1000) = {log10_of_1000}"
```

Output:

```
ln(2) = 0.6931471805599453
ln(10) = 2.302585092994046
log₁₀(1000) = 2.9999999999999996
```

Random numbers:

```
let r1 = math.random()
let r2 = math.random()
say "Random 1: {r1}"
say "Random 2: {r2}"

// Random integer between 1 and 6 (dice roll)
let raw = math.random() * 6.0
let die = math.floor(raw) + 1
say "Dice roll: {die}"
```

> **Pseudorandomness.** `math.random()` uses system time nanoseconds as its seed. It is suitable for games, simulations, and sampling—not for cryptographic purposes. Use the `crypto` module when security matters.

3.1.4 Recipes

Recipe 9.1: Euclidean Distance

Calculate the distance between two points in 2D space.

```
fn distance(x1, y1, x2, y2) {
    let dx = x2 - x1
    let dy = y2 - y1
    return math.sqrt(dx * dx + dy * dy)
}

let d = distance(0.0, 0.0, 3.0, 4.0)
say "Distance: {d}"

let d2 = distance(1.0, 2.0, 4.0, 6.0)
say "Distance: {d2}"
```

Output:

```
Distance: 5.0
Distance: 5.0
```

Recipe 9.2: Degrees and Radians Conversion

```
fn deg_to_rad(degrees) {
    return degrees * math.pi / 180.0
}

fn rad_to_deg(radians) {
    return radians * 180.0 / math.pi
}

let rad = deg_to_rad(90.0)
say "90° = {rad} radians"

let deg = rad_to_deg(math.pi)
say "π radians = {deg}°"

// Sine of 30 degrees
let angle = deg_to_rad(30.0)
let result = math.sin(angle)
say "sin(30°) = {result}"
```

Output:

```
90° = 1.5707963267948966 radians
```

```
π radians = 180.0°
sin(30°) = 0.49999999999999994
```

Recipe 9.3: Random Number in a Range

```
fn random_between(lo, hi) {
    let range = hi - lo
    let r = math.random() * range
    return math.floor(r) + lo
}

// Generate 5 random numbers between 10 and 50
repeat 5 times {
    let n = random_between(10, 50)
    say "Random: {n}"
}
```

Recipe 9.4: Basic Statistics

```
fn mean(values) {
    let mut sum = 0.0
    for v in values {
        sum = sum + v
    }
    return sum / len(values)
}

fn variance(values) {
    let avg = mean(values)
    let mut sum_sq = 0.0
    for v in values {
        let diff = v - avg
        sum_sq = sum_sq + diff * diff
    }
    return sum_sq / len(values)
}

fn std_dev(values) {
    return math.sqrt(variance(values))
}

let data = [4.0, 8.0, 15.0, 16.0, 23.0, 42.0]
let m = mean(data)
let sd = std_dev(data)
say "Mean: {m}"
say "Std Dev: {sd}"
```

Output:

```
Mean: 18.0
Std Dev: 12.396773926563296
```

3.2 Chapter 10: fs — File System

The `fs` module gives Forge programs the ability to read, write, copy, rename, and inspect files and directories. It wraps the operating system's file APIs in a set of straightforward functions that accept string paths and return predictable results. Whether you are writing a quick script that processes a log file or building a tool that manages configuration across a project, `fs` is the module you will reach for first.

All path arguments are strings. Relative paths resolve from the working directory where `forge run` was invoked. Functions that modify the filesystem—`write`, `append`, `remove`, `mkdir`, `rename`, `copy`—perform their operation or return an error message; they never silently fail.

3.2.1 Function Reference

Function	Description	Example	Return Type
`fs.read(path)`	Read entire file as a string	`fs.read("data.txt")` ☐ `"hello"`	String
`fs.write(path, content)`	Write string to file (overwrites)	`fs.write("out.txt", "data")`	Null
`fs.append(path, content)`	Append string to file	`fs.append("log.txt", "entry\n")`	Null
`fs.exists(path)`	Check if file or directory exists	`fs.exists("config.json")` ☐ `true`	Bool
`fs.size(path)`	File size in bytes	`fs.size("photo.jpg")` ☐ `204800`	Int
`fs.ext(path)`	File extension without the dot	`fs.ext("main.fg")` ☐ `"fg"`	String
`fs.list(path)`	List entries in a directory	`fs.list("src/")` ☐ `["main.rs", "lib.rs"]`	Array[String]
`fs.mkdir(path)`	Create directory (and parents)	`fs.mkdir("build/output")`	Null
`fs.copy(src, dst)`	Copy a file	`fs.copy("a.txt", "b.txt")` ☐ `1024`	Int
`fs.rename(old, new)`	Rename or move a file or directory	`fs.rename("old.txt", "new.txt")`	Null
`fs.remove(path)`	Delete a file or directory (recursive)	`fs.remove("temp/")`	Null

Function	Description	Example	Return Type
`fs.read_json(path)`	Read and parse a JSON file	`fs.read_json("config.json")` ☐ `{...}`	Value
`fs.write_json(path, value)`	Write a value as pretty-printed JSON	`fs.write_json("out.json", data)`	Null

> **Safety Note.** `fs.remove()` deletes directories recursively without confirmation. Always double-check your path, especially when it comes from user input.

3.2.2 Core Examples

Reading and writing text files:

```
// Write a file
fs.write("/tmp/greeting.txt", "Hello from Forge!")

// Read it back
let content = fs.read("/tmp/greeting.txt")
say "File says: {content}"

// Append to it
fs.append("/tmp/greeting.txt", "\nSecond line.")
let updated = fs.read("/tmp/greeting.txt")
say "Updated:\n{updated}"
```

Output:

```
File says: Hello from Forge!
Updated:
Hello from Forge!
Second line.
```

Checking existence and metadata:

```
fs.write("/tmp/forge_meta.txt", "some data here")

let exists = fs.exists("/tmp/forge_meta.txt")
say "Exists: {exists}"

let bytes = fs.size("/tmp/forge_meta.txt")
say "Size: {bytes} bytes"

let extension = fs.ext("/tmp/forge_meta.txt")
```

```
say "Extension: {extension}"

fs.remove("/tmp/forge_meta.txt")
let gone = fs.exists("/tmp/forge_meta.txt")
say "After remove: {gone}"
```

Output:

```
Exists: true
Size: 14 bytes
Extension: txt
After remove: false
```

Directory operations:

```
// Create nested directories
fs.mkdir("/tmp/forge_project/src/modules")

// Write files into the structure
fs.write("/tmp/forge_project/src/main.fg", "say \"hello\"")
fs.write("/tmp/forge_project/src/utils.fg", "fn add(a, b) { return a + b }")

// List directory contents
let files = fs.list("/tmp/forge_project/src")
say "Source files: {files}"

// Clean up
fs.remove("/tmp/forge_project")
```

Output:

```
Source files: ["modules", "main.fg", "utils.fg"]
```

Copying and renaming:

```
fs.write("/tmp/original.txt", "important data")

// Copy creates a duplicate
let bytes_copied = fs.copy("/tmp/original.txt", "/tmp/backup.txt")
say "Copied {bytes_copied} bytes"

// Rename moves the file
fs.rename("/tmp/backup.txt", "/tmp/archive.txt")
let has_backup = fs.exists("/tmp/backup.txt")
let has_archive = fs.exists("/tmp/archive.txt")
```

```
say "backup.txt exists: {has_backup}"
say "archive.txt exists: {has_archive}"

// Clean up
fs.remove("/tmp/original.txt")
fs.remove("/tmp/archive.txt")
```

Output:

```
Copied 14 bytes
backup.txt exists: false
archive.txt exists: true
```

JSON file round-trip:

```
let config = {
    app_name: "Forge Demo",
    version: "1.0.0",
    features: ["logging", "metrics", "auth"],
    database: {
        host: "localhost",
        port: 5432
    }
}

// Write as pretty-printed JSON
fs.write_json("/tmp/config.json", config)

// Read it back as a Forge object
let loaded = fs.read_json("/tmp/config.json")
say "App: {loaded.app_name}"
say "DB port: {loaded.database.port}"

fs.remove("/tmp/config.json")
```

Output:

```
App: Forge Demo
DB port: 5432
```

> **JSON Round-Trip.** `fs.write_json` uses `json.pretty` internally, producing human-readable files with 2-space indentation. `fs.read_json` uses `json.parse`, converting JSON types to their Forge equivalents: objects, arrays, strings, integers, floats, booleans, and null.

3.2.3 Recipes

Recipe 10.1: Configuration File Manager

```
fn load_config(path) {
    if fs.exists(path) {
        return fs.read_json(path)
    }
    // Return defaults
    return {
        log_level: "info",
        max_retries: 3,
        timeout: 30
    }
}

fn save_config(path, config) {
    fs.write_json(path, config)
}

let cfg = load_config("/tmp/app_config.json")
say "Log level: {cfg.log_level}"

// Save config for next run
save_config("/tmp/app_config.json", cfg)
fs.remove("/tmp/app_config.json")
```

Recipe 10.2: Log Rotation

```
fn rotate_logs(log_path, max_backups) {
    if fs.exists(log_path) == false {
        return null
    }

    // Shift existing backups: .3 → .4, .2 → .3, etc.
    let mut i = max_backups - 1
    for n in [3, 2, 1] {
        let older = "{log_path}.{n}"
        let newer_num = n + 1
        let newer = "{log_path}.{newer_num}"
        if fs.exists(older) {
            fs.rename(older, newer)
        }
    }

    // Current log becomes .1
    let backup = "{log_path}.1"
```

```
    fs.copy(log_path, backup)
    fs.write(log_path, "")
    say "Logs rotated"
}

// Demo
fs.write("/tmp/app.log", "line 1\nline 2\nline 3\n")
rotate_logs("/tmp/app.log", 4)

let current = fs.read("/tmp/app.log")
let has_backup = fs.exists("/tmp/app.log.1")
say "Current log empty: {current}"
say "Backup exists: {has_backup}"

// Clean up
fs.remove("/tmp/app.log")
fs.remove("/tmp/app.log.1")
```

Recipe 10.3: Directory Tree Printer

```
fn print_tree(path, prefix) {
    let entries = fs.list(path)
    let count = len(entries)
    let mut idx = 0
    for entry in entries {
        idx = idx + 1
        let is_last = idx == count
        let connector = "└── "
        if is_last == false {
            let connector = "├── "
        }
        say "{prefix}{connector}{entry}"

        let full = "{path}/{entry}"
        let ext = fs.ext(full)
        // If no extension, it might be a directory
        if ext == "" {
            let child_prefix = "{prefix}    "
            if is_last == false {
                let child_prefix = "{prefix}│   "
            }
            if fs.exists(full) {
                // Try listing it (will fail gracefully if it's a file)
                safe {
                    print_tree(full, child_prefix)
                }
            }
```

```
        }
    }
}

// Build a sample directory
fs.mkdir("/tmp/myproject/src")
fs.mkdir("/tmp/myproject/tests")
fs.write("/tmp/myproject/src/main.fg", "")
fs.write("/tmp/myproject/src/utils.fg", "")
fs.write("/tmp/myproject/tests/test_main.fg", "")
fs.write("/tmp/myproject/README.md", "")

say "myproject/"
print_tree("/tmp/myproject", "")

fs.remove("/tmp/myproject")
```

Recipe 10.4: File Backup Script

```
fn backup_file(source) {
    if fs.exists(source) == false {
        say "Source not found: {source}"
        return false
    }

    let ext = fs.ext(source)
    let backup_path = "{source}.bak"
    let bytes = fs.copy(source, backup_path)
    say "Backed up {source} ({bytes} bytes)"
    return true
}

// Create some test files
fs.write("/tmp/data1.txt", "important data file 1")
fs.write("/tmp/data2.txt", "important data file 2")

let files_to_backup = ["/tmp/data1.txt", "/tmp/data2.txt"]
for f in files_to_backup {
    backup_file(f)
}

// Clean up
for f in files_to_backup {
    fs.remove(f)
    let bak = "{f}.bak"
    fs.remove(bak)
}
```

3.3 Chapter 11: crypto — Hashing and Encoding

The `crypto` module provides hashing algorithms and encoding utilities. It is intentionally small: two hash functions (SHA-256 and MD5) and two pairs of encode/decode functions (Base64 and hexadecimal). These six functions cover the most common needs—verifying data integrity, generating fingerprints, and preparing binary data for text-safe transport.

All functions accept strings and return strings. Hashes produce lowercase hexadecimal digests. Encoding functions convert raw bytes to a text representation; decoding functions reverse the process.

3.3.1 Function Reference

Function	Description	Example	Return Type
`crypto.sha256(s)`	SHA-256 hash, hex-encoded	`crypto.sha256("hello")` → `"2cf24d..."`	String
`crypto.md5(s)`	MD5 hash, hex-encoded	`crypto.md5("hello")` → `"5d4114..."`	String
`crypto.base64_encode(s)`	Encode string to Base64	`crypto.base64_encode("hello")` → `"aGVsbG8="`	String
`crypto.base64_decode(s)`	Decode Base64 string	`crypto.base64_decode("aGVsbG8=")` → `"hello"`	String
`crypto.hex_encode(s)`	Encode string bytes as hexadecimal	`crypto.hex_encode("AB")` → `"4142"`	String
`crypto.hex_decode(s)`	Decode hex string to bytes	`crypto.hex_decode("4142")` → `"AB"`	String

> **MD5 is not secure.** MD5 is provided for legacy compatibility and checksums. Never use it for password hashing or security-critical fingerprints. Use SHA-256 instead.

3.3.2 Core Examples

SHA-256 hashing:

```
let hash = crypto.sha256("forge")
say "SHA-256 of 'forge': {hash}"

// Same input always produces same output
let hash2 = crypto.sha256("forge")
let match = hash == hash2
```

```
say "Deterministic: {match}"

// Different input produces different output
let other = crypto.sha256("Forge")
let different = hash == other
say "Case sensitive: {different}"
```

Output:

```
SHA-256 of 'forge': <64-character hex string>
Deterministic: true
Case sensitive: false
```

MD5 hashing:

```
let md5 = crypto.md5("hello world")
say "MD5: {md5}"
```

Output:

```
MD5: 5eb63bbbe01eeed093cb22bb8f5acdc3
```

Base64 encoding and decoding:

```
let original = "Hello, Forge!"
let encoded = crypto.base64_encode(original)
say "Encoded: {encoded}"

let decoded = crypto.base64_decode(encoded)
say "Decoded: {decoded}"

let roundtrip = original == decoded
say "Round-trip matches: {roundtrip}"
```

Output:

```
Encoded: SGVsbG8sIEZvcmdlIQ==
Decoded: Hello, Forge!
Round-trip matches: true
```

Hex encoding and decoding:

```
let text = "Forge"
let hex = crypto.hex_encode(text)
say "Hex: {hex}"

let back = crypto.hex_decode(hex)
say "Decoded: {back}"
```

Output:

```
Hex: 466f726765
Decoded: Forge
```

Combining hashing with encoding:

```
let data = "sensitive payload"
let hash = crypto.sha256(data)
let b64_hash = crypto.base64_encode(hash)
say "SHA-256 (Base64): {b64_hash}"
```

3.3.3 Recipes

Recipe 11.1: Password Hashing with Salt

```
fn hash_password(password, salt) {
    let salted = "{salt}:{password}"
    return crypto.sha256(salted)
}

fn verify_password(password, salt, expected_hash) {
    let computed = hash_password(password, salt)
    return computed == expected_hash
}

let salt = "random_salt_value_2024"
let hashed = hash_password("my_secret_password", salt)
say "Stored hash: {hashed}"

let valid = verify_password("my_secret_password", salt, hashed)
say "Correct password: {valid}"

let invalid = verify_password("wrong_password", salt, hashed)
say "Wrong password: {invalid}"
```

Output:

```
Stored hash: <64-character hex string>
Correct password: true
Wrong password: false
```

> **Production Warning.** This recipe demonstrates the principle of salted hashing. For production systems, use a dedicated password hashing algorithm (bcrypt, scrypt, Argon2) via an external service or API. Simple SHA-256, even with a salt, is not sufficient against modern brute-force attacks.

Recipe 11.2: Data Integrity Verification

```
fn write_with_checksum(path, data) {
    fs.write(path, data)
    let checksum = crypto.sha256(data)
    let checksum_path = "{path}.sha256"
    fs.write(checksum_path, checksum)
    say "Wrote {path} with checksum"
}

fn verify_integrity(path) {
    let data = fs.read(path)
    let checksum_path = "{path}.sha256"
    let expected = fs.read(checksum_path)
    let actual = crypto.sha256(data)
    return actual == expected
}

write_with_checksum("/tmp/payload.dat", "critical data that must not change")

let ok = verify_integrity("/tmp/payload.dat")
say "Integrity check: {ok}"

// Clean up
fs.remove("/tmp/payload.dat")
fs.remove("/tmp/payload.dat.sha256")
```

Output:

```
Wrote /tmp/payload.dat with checksum
Integrity check: true
```

Recipe 11.3: Encoding Data for Transport

```
// Encode a JSON payload for URL-safe transport
let payload = "{\"user\":\"alice\",\"role\":\"admin\"}"
let encoded = crypto.base64_encode(payload)
```

```
say "Transport-safe: {encoded}"

// Receiver decodes it
let received = crypto.base64_decode(encoded)
say "Received: {received}"
let obj = json.parse(received)
say "User: {obj.user}, Role: {obj.role}"
```

Output:

```
Transport-safe: eyJ1c2VyIjoiYWxpY2UiLCJyb2xlIjoiYWRtaW4ifQ==
Received: {"user":"alice","role":"admin"}
User: alice, Role: admin
```

Recipe 11.4: File Checksum Validation

```
fn checksum_file(path) {
    let content = fs.read(path)
    return crypto.sha256(content)
}

fs.write("/tmp/file_a.txt", "identical content")
fs.write("/tmp/file_b.txt", "identical content")
fs.write("/tmp/file_c.txt", "different content")

let a = checksum_file("/tmp/file_a.txt")
let b = checksum_file("/tmp/file_b.txt")
let c = checksum_file("/tmp/file_c.txt")

let ab_match = a == b
let ac_match = a == c
say "A == B: {ab_match}"
say "A == C: {ac_match}"

fs.remove("/tmp/file_a.txt")
fs.remove("/tmp/file_b.txt")
fs.remove("/tmp/file_c.txt")
```

Output:

```
A == B: true
A == C: false
```

3.4 Chapter 12: db — SQLite

Forge includes a built-in SQLite driver, making it trivial to store and query structured data without installing a database server. The `db` module connects to a file-based database or an in-memory database, executes SQL statements, and returns results as arrays of Forge objects—no ORM layer, no mapping configuration. This makes Forge an excellent choice for data scripts, CLI tools, prototyping, and local applications.

3.4.1 Function Reference

Function	Description	Example	Return Type
`db.open(path)`	Open a SQLite database file (or `":memory:"`)	`db.open(":memory:")` ☐ `true`	Bool
`db.execute(sql)`	Execute a statement (CREATE, INSERT, UPDATE, DELETE)	`db.execute("CREATE TABLE ...")`	Null
`db.query(sql)`	Execute a SELECT and return rows	`db.query("SELECT * FROM t")` ☐ `[{...}, ...]`	Array[Object]
`db.close()`	Close the database connection	`db.close()`	Null

> **Connection Model.** Forge maintains one SQLite connection per thread using thread-local storage. Calling `db.open()` replaces any existing connection. Always call `db.close()` when finished to release the database file lock.

3.4.2 The In-Memory Database

For scripts, tests, and prototyping, the special path `":memory:"` creates a database that lives only in RAM. It is fast, requires no cleanup, and vanishes when the program exits.

```
db.open(":memory:")
db.execute("CREATE TABLE greetings (id INTEGER PRIMARY KEY, message TEXT)")
db.execute("INSERT INTO greetings (message) VALUES ('Hello, Forge!')")

let rows = db.query("SELECT * FROM greetings")
say "Rows: {rows}"
db.close()
```

Output:

```
Rows: [{id: 1, message: Hello, Forge!}]
```

3.4.3 Core Examples

Creating tables and inserting data:

```
db.open(":memory:")

db.execute("CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT, email TEXT, active INTEGER)")

db.execute("INSERT INTO users (name, email, active) VALUES ('Alice', 'alice@example.com', 1)")
db.execute("INSERT INTO users (name, email, active) VALUES ('Bob', 'bob@example.com', 1)")
db.execute("INSERT INTO users (name, email, active) VALUES ('Charlie', 'charlie@example.com', 0)"

let users = db.query("SELECT * FROM users WHERE active = 1")
for user in users {
    say "{user.name} <{user.email}>"
}

db.close()
```

Output:

```
Alice <alice@example.com>
Bob <bob@example.com>
```

Aggregation queries:

```
db.open(":memory:")

db.execute("CREATE TABLE orders (id INTEGER PRIMARY KEY, product TEXT, amount REAL, qty INTEGER)"
db.execute("INSERT INTO orders (product, amount, qty) VALUES ('Widget', 9.99, 5)")
db.execute("INSERT INTO orders (product, amount, qty) VALUES ('Widget', 9.99, 3)")
db.execute("INSERT INTO orders (product, amount, qty) VALUES ('Gadget', 24.99, 2)")
db.execute("INSERT INTO orders (product, amount, qty) VALUES ('Gadget', 24.99, 1)")

let summary = db.query("SELECT product, SUM(amount * qty) as revenue, SUM(qty) as units FROM orde
for row in summary {
    say "{row.product}: ${row.revenue} ({row.units} units)"
}

db.close()
```

Output:

```
Gadget: $74.97 (3 units)
Widget: $79.92 (8 units)
```

Updates and deletes:

```
db.open(":memory:")

db.execute("CREATE TABLE tasks (id INTEGER PRIMARY KEY, title TEXT, done INTEGER DEFAULT 0)")
db.execute("INSERT INTO tasks (title) VALUES ('Write chapter 12')")
db.execute("INSERT INTO tasks (title) VALUES ('Review examples')")
db.execute("INSERT INTO tasks (title) VALUES ('Submit draft')")

// Mark a task as done
db.execute("UPDATE tasks SET done = 1 WHERE title = 'Write chapter 12'")

// Delete completed tasks
db.execute("DELETE FROM tasks WHERE done = 1")

let remaining = db.query("SELECT title FROM tasks")
say "Remaining tasks: {remaining}"
db.close()
```

Output:

```
Remaining tasks: [{title: Review examples}, {title: Submit draft}]
```

Working with file-based databases:

```
// Persistent database on disk
db.open("/tmp/forge_app.db")

db.execute("CREATE TABLE IF NOT EXISTS settings (key TEXT PRIMARY KEY, value TEXT)")
db.execute("INSERT OR REPLACE INTO settings (key, value) VALUES ('theme', 'dark')")
db.execute("INSERT OR REPLACE INTO settings (key, value) VALUES ('lang', 'en')")

let settings = db.query("SELECT * FROM settings")
say "Settings: {settings}"

db.close()

// The database persists — we can reopen it
db.open("/tmp/forge_app.db")
let reloaded = db.query("SELECT value FROM settings WHERE key = 'theme'")
say "Theme: {reloaded}"
db.close()

fs.remove("/tmp/forge_app.db")
```

> **Column Types.** SQLite stores data as one of five types: NULL, INTEGER, REAL, TEXT, and BLOB. `db.query()` maps these to Forge's `null`, `Int`, `Float`, `String`, and a blob description string. Column names become object keys in the returned rows.

3.4.4 Recipes

Recipe 12.1: Full CRUD Application

```
db.open(":memory:")

db.execute("CREATE TABLE contacts (id INTEGER PRIMARY KEY, name TEXT NOT NULL, phone TEXT, email

// Create
fn add_contact(name, phone, email) {
    db.execute("INSERT INTO contacts (name, phone, email) VALUES ('{name}', '{phone}', '{email}')
}

// Read
fn get_contacts() {
    return db.query("SELECT * FROM contacts ORDER BY name")
}

fn find_contact(name) {
    return db.query("SELECT * FROM contacts WHERE name = '{name}'")
}

// Update
fn update_phone(name, new_phone) {
    db.execute("UPDATE contacts SET phone = '{new_phone}' WHERE name = '{name}'")
}

// Delete
fn remove_contact(name) {
    db.execute("DELETE FROM contacts WHERE name = '{name}'")
}

// Use the CRUD functions
add_contact("Alice", "555-0101", "alice@mail.com")
add_contact("Bob", "555-0102", "bob@mail.com")
add_contact("Charlie", "555-0103", "charlie@mail.com")

say "All contacts:"
let all = get_contacts()
term.table(all)

update_phone("Bob", "555-9999")
say "\nAfter updating Bob's phone:"
let bob = find_contact("Bob")
say "Bob: {bob}"

remove_contact("Charlie")
say "\nAfter removing Charlie:"
```

```
let remaining = get_contacts()
term.table(remaining)

db.close()
```

Recipe 12.2: Data Migration

```
db.open(":memory:")

// Old schema
db.execute("CREATE TABLE users_v1 (id INTEGER PRIMARY KEY, fullname TEXT, email TEXT)")
db.execute("INSERT INTO users_v1 (fullname, email) VALUES ('Alice Smith', 'alice@example.com')")
db.execute("INSERT INTO users_v1 (fullname, email) VALUES ('Bob Jones', 'bob@example.com')")

// New schema with split name fields
db.execute("CREATE TABLE users_v2 (id INTEGER PRIMARY KEY, first_name TEXT, last_name TEXT, email

// Migrate data
let old_users = db.query("SELECT * FROM users_v1")
for user in old_users {
    let parts = split(user.fullname, " ")
    let first = parts[0]
    let last = parts[1]
    db.execute("INSERT INTO users_v2 (first_name, last_name, email) VALUES ('{first}', '{last}',
}

say "Migrated users:"
let new_users = db.query("SELECT * FROM users_v2")
term.table(new_users)

db.close()
```

Recipe 12.3: Report Generation

```
db.open(":memory:")

db.execute("CREATE TABLE sales (id INTEGER PRIMARY KEY, product TEXT, amount REAL, region TEXT)")
db.execute("INSERT INTO sales (product, amount, region) VALUES ('Widget', 29.99, 'North')")
db.execute("INSERT INTO sales (product, amount, region) VALUES ('Gadget', 49.99, 'South')")
db.execute("INSERT INTO sales (product, amount, region) VALUES ('Widget', 19.99, 'East')")
db.execute("INSERT INTO sales (product, amount, region) VALUES ('Gizmo', 99.99, 'North')")
db.execute("INSERT INTO sales (product, amount, region) VALUES ('Gadget', 49.99, 'West')")

say "=== Sales Report ==="

say "\nBy Product:"
let by_product = db.query("SELECT product, COUNT(*) as orders, SUM(amount) as total FROM sales GR
```

```
term.table(by_product)

say "\nBy Region:"
let by_region = db.query("SELECT region, COUNT(*) as orders, SUM(amount) as total FROM sales GROU
term.table(by_region)

say "\nTop Sale:"
let top = db.query("SELECT product, amount, region FROM sales ORDER BY amount DESC LIMIT 1")
say "{top}"

db.close()
```

Recipe 12.4: Test Fixtures

```
fn setup_test_db() {
    db.open(":memory:")
    db.execute("CREATE TABLE products (id INTEGER PRIMARY KEY, name TEXT, price REAL, stock INTEG
    db.execute("INSERT INTO products (name, price, stock) VALUES ('Pen', 1.99, 100)")
    db.execute("INSERT INTO products (name, price, stock) VALUES ('Notebook', 4.99, 50)")
    db.execute("INSERT INTO products (name, price, stock) VALUES ('Eraser', 0.99, 200)")
}

fn teardown_test_db() {
    db.close()
}

// Test: all products have positive prices
setup_test_db()
let products = db.query("SELECT * FROM products WHERE price <= 0")
let count = len(products)
assert_eq(count, 0)
say "Test passed: all prices positive"
teardown_test_db()

// Test: stock levels are reasonable
setup_test_db()
let low_stock = db.query("SELECT * FROM products WHERE stock < 10")
let low_count = len(low_stock)
assert_eq(low_count, 0)
say "Test passed: no dangerously low stock"
teardown_test_db()
```

Output:

```
Test passed: all prices positive
Test passed: no dangerously low stock
```

3.5 Chapter 13: pg — PostgreSQL

While the `db` module handles local SQLite databases, the `pg` module connects Forge to PostgreSQL—the workhorse of production infrastructure. The API mirrors `db` closely (connect, query, execute, close), so moving from a prototype on SQLite to a production system on PostgreSQL requires minimal code changes.

The `pg` module runs on Forge's async runtime (Tokio under the hood). Connection management uses thread-local storage, giving you one active connection per program. For scripts, CLI tools, and single-connection services, this model is simple and effective.

3.5.1 Function Reference

Function	Description	Example	Return Type
`pg.connect(constr)`	Connect to a PostgreSQL server	`pg.connect("host=localhost dbname=mydb user=app")` ☐ `true`	Bool
`pg.query(sql)`	Execute a SELECT and return rows	`pg.query("SELECT * FROM users")` ☐ `[{...}]`	Array[Object]
`pg.execute(sql)`	Execute INSERT, UPDATE, DELETE; return rows affected	`pg.execute("DELETE FROM old_logs")` ☐ `42`	Int
`pg.close()`	Close the connection	`pg.close()`	Null

3.5.2 Connection Strings

PostgreSQL connection strings follow the standard `key=value` format:

```
host=localhost port=5432 dbname=myapp user=appuser password=secret
```

Common parameters:

Parameter	Description	Default
`host`	Server hostname or IP	`localhost`
`port`	Server port	`5432`
`dbname`	Database name	Same as user
`user`	Username	Current OS user
`password`	Password	None
`sslmode`	SSL mode	`prefer`

3.5.3 Core Examples

Connecting and querying:

```
pg.connect("host=localhost dbname=myapp user=app password=secret")

let users = pg.query("SELECT id, name, email FROM users LIMIT 5")
for user in users {
    say "{user.id}: {user.name} <{user.email}>"
}

pg.close()
```

Executing write operations:

```
pg.connect("host=localhost dbname=myapp user=app password=secret")

let affected = pg.execute("UPDATE users SET last_login = NOW() WHERE id = 1")
say "Updated {affected} row(s)"

pg.execute("INSERT INTO audit_log (action, user_id) VALUES ('login', 1)")

pg.close()
```

Creating tables:

```
pg.connect("host=localhost dbname=myapp user=app password=secret")

pg.execute("CREATE TABLE IF NOT EXISTS events (
    id SERIAL PRIMARY KEY,
    type TEXT NOT NULL,
    payload JSONB,
    created_at TIMESTAMP DEFAULT NOW()
)")

pg.execute("INSERT INTO events (type, payload) VALUES ('signup', '{\"user\": \"alice\"}')")

let events = pg.query("SELECT * FROM events ORDER BY created_at DESC LIMIT 10")
term.table(events)

pg.close()
```

Aggregation queries:

```
pg.connect("host=localhost dbname=analytics user=app password=secret")

let stats = pg.query("SELECT
    DATE(created_at) as day,
    COUNT(*) as signups,
    COUNT(DISTINCT country) as countries
```

```
    FROM users
    WHERE created_at > NOW() - INTERVAL '7 days'
    GROUP BY DATE(created_at)
    ORDER BY day")

say "Signup stats (last 7 days):"
term.table(stats)

pg.close()
```

> **Type Mapping.** PostgreSQL types are mapped to Forge values as follows: `int4`/`int8` → `Int`, `float4`/`float8` → `Float`, `text`/`varchar` → `String`, `bool` → `Bool`, and `NULL` → `null`. JSONB columns are returned as strings—parse them with `json.parse()` if you need the structured data.

3.5.4 Recipes

Recipe 13.1: Connection Helper with Error Handling

```
fn connect_db() {
    let host = env.get("DB_HOST", "localhost")
    let port = env.get("DB_PORT", "5432")
    let name = env.get("DB_NAME", "myapp")
    let user = env.get("DB_USER", "app")
    let pass = env.get("DB_PASS", "")
    let conn = "host={host} port={port} dbname={name} user={user} password={pass}"
    pg.connect(conn)
    say "Connected to {name}@{host}:{port}"
}

fn disconnect_db() {
    pg.close()
    say "Disconnected"
}
```

Recipe 13.2: Batch Insert

```
pg.connect("host=localhost dbname=myapp user=app password=secret")

pg.execute("CREATE TABLE IF NOT EXISTS metrics (name TEXT, value REAL, ts TIMESTAMP DEFAULT NOW()

let metrics = [
    { name: "cpu_usage", value: 72.5 },
    { name: "memory_mb", value: 4096.0 },
    { name: "disk_pct", value: 45.2 },
    { name: "network_mbps", value: 125.8 }
```

```
]

for m in metrics {
    pg.execute("INSERT INTO metrics (name, value) VALUES ('{m.name}', {m.value})")
}

say "Inserted {len(metrics)} metrics"

let results = pg.query("SELECT name, value FROM metrics ORDER BY name")
term.table(results)

pg.close()
```

Recipe 13.3: Migration Runner

```
fn run_migration(version, sql) {
    say "Running migration v{version}..."
    pg.execute(sql)
    pg.execute("INSERT INTO schema_migrations (version) VALUES ({version})")
    say "Migration v{version} complete"
}

pg.connect("host=localhost dbname=myapp user=app password=secret")

pg.execute("CREATE TABLE IF NOT EXISTS schema_migrations (version INTEGER PRIMARY KEY, applied_at

let applied = pg.query("SELECT version FROM schema_migrations ORDER BY version")
let applied_versions = map(applied, fn(r) { return r.version })

if contains(applied_versions, 1) == false {
    run_migration(1, "CREATE TABLE users (id SERIAL PRIMARY KEY, name TEXT, email TEXT UNIQUE)")
}

if contains(applied_versions, 2) == false {
    run_migration(2, "ALTER TABLE users ADD COLUMN created_at TIMESTAMP DEFAULT NOW()")
}

say "All migrations applied"
pg.close()
```

Recipe 13.4: Health Check Query

```
fn db_health_check() {
    safe {
        pg.connect("host=localhost dbname=myapp user=app password=secret")
        let result = pg.query("SELECT 1 as ok")
        pg.close()
```

```
        if len(result) > 0 {
            return { status: "healthy", db: "connected" }
        }
    }
    return { status: "unhealthy", db: "unreachable" }
}

let health = db_health_check()
say "Database: {health.status}"
```

3.6 Chapter 14: json — Serialization

JSON is the lingua franca of modern APIs, configuration files, and data exchange. Forge embraces JSON at the language level—object literals in Forge *are* JSON-compatible structures—and the `json` module provides three functions to move between Forge values and JSON text.

Because Forge objects and JSON objects share the same structural model, serialization is natural. There is no schema to define, no mapping to configure. A Forge object becomes JSON text with `json.stringify()`, and JSON text becomes a Forge object with `json.parse()`.

3.6.1 Function Reference

Function	Description	Example	Return Type
`json.parse(s)`	Parse a JSON string into a Forge value	`json.parse("{\"a\":1}")` ☐ `{a: 1}`	Value
`json.stringify(value)`	Convert a Forge value to compact JSON string	`json.stringify({a: 1})` ☐ `"{\"a\":1}"`	String
`json.pretty(value)`	Convert a Forge value to indented JSON string	`json.pretty({a: 1})` ☐ formatted string	String

> **Number Handling.** `json.parse()` converts JSON numbers to `Int` when they have no fractional part, and `Float` otherwise. The number `42` becomes `Int(42)`, while `42.0` becomes `Float(42.0)`.

3.6.2 Core Examples

Parsing JSON strings:

```
let text = "{\"name\":\"Alice\",\"age\":30,\"active\":true}"
let user = json.parse(text)
say "Name: {user.name}"
say "Age: {user.age}"
say "Active: {user.active}"
```

Output:

```
Name: Alice
Age: 30
Active: true
```

Parsing arrays:

```
let arr_text = "[1, 2, 3, 4, 5]"
let numbers = json.parse(arr_text)
say "Count: {len(numbers)}"
say "First: {numbers[0]}"
say "Last: {numbers[4]}"
```

Output:

```
Count: 5
First: 1
Last: 5
```

Stringifying Forge objects:

```
let server = {
    host: "api.example.com",
    port: 443,
    tls: true,
    routes: ["/users", "/posts", "/health"]
}

let compact = json.stringify(server)
say "Compact: {compact}"
```

Output:

```
Compact: {"host":"api.example.com","port":443,"tls":true,"routes":["/users","/posts","/health"]}
```

Pretty-printing:

```
let config = {
    app: "forge-demo",
    version: "1.0.0",
    database: {
        host: "localhost",
        port: 5432
    },
    features: ["auth", "logging"]
}

let pretty = json.pretty(config)
say pretty
```

Output:

```
{
  "app": "forge-demo",
  "version": "1.0.0",
  "database": {
    "host": "localhost",
    "port": 5432
  },
  "features": ["auth", "logging"]
}
```

Nested structures:

```
let api_response = {
    status: 200,
    data: {
        users: [
            { id: 1, name: "Alice" },
            { id: 2, name: "Bob" }
        ],
        total: 2
    }
}

let text = json.stringify(api_response)
say "Serialized length: {len(text)} characters"

// Round-trip
let restored = json.parse(text)
let first_user = restored.data.users[0]
say "First user: {first_user.name}"
```

Output:

```
Serialized length: 77 characters
First user: Alice
```

Handling null and boolean values:

```
let data = json.parse("{\"value\":null,\"flag\":false,\"count\":0}")
say "Value: {data.value}"
say "Flag: {data.flag}"
say "Count: {data.count}"

let back = json.stringify(data)
say "Serialized: {back}"
```

Output:

```
Value: null
Flag: false
Count: 0
Serialized: {"value":null,"flag":false,"count":0}
```

3.6.3 Recipes

Recipe 14.1: API Response Handling

```
// Simulate an API response
let response_text = "{\"status\":\"ok\",\"data\":{\"items\":[{\"id\":1,\"name\":\"Widget\",\"pric

let response = json.parse(response_text)

if response.status == "ok" {
    let items = response.data.items
    say "Found {len(items)} items (page {response.data.page} of {response.data.total_pages}):"
    for item in items {
        say "  #{item.id} {item.name} — ${item.price}"
    }
}
```

Output:

```
Found 2 items (page 1 of 5):
  #1 Widget — $9.99
  #2 Gadget — $24.99
```

Recipe 14.2: Config File Management

```
fn load_config(path) {
    if fs.exists(path) {
        return fs.read_json(path)
    }
    let defaults = {
        theme: "dark",
        language: "en",
        notifications: true,
        max_items: 50
    }
    fs.write_json(path, defaults)
    return defaults
}

fn update_config(path, key, value) {
    let config = load_config(path)
    // Build updated config
    let updated = json.parse(json.stringify(config))
    fs.write_json(path, updated)
    return updated
}

let cfg = load_config("/tmp/app_settings.json")
say "Theme: {cfg.theme}"

fs.remove("/tmp/app_settings.json")
```

Recipe 14.3: Data Transformation Pipeline

```
let raw = "[{\"first\":\"Alice\",\"last\":\"Smith\",\"score\":92},{\"first\":\"Bob\",\"last\":\"J

let students = json.parse(raw)

// Transform: add full name and grade
let graded = map(students, fn(s) {
    let grade = "C"
    if s.score >= 90 {
        let grade = "A"
    } else if s.score >= 80 {
        let grade = "B"
    }
    return {
        name: "{s.first} {s.last}",
        score: s.score,
        grade: grade
    }
})
```

```
say json.pretty(graded)
```

Recipe 14.4: JSON Merge Utility

```
fn merge_objects(base, overlay) {
    let base_text = json.stringify(base)
    let overlay_text = json.stringify(overlay)
    // Simple merge: overlay keys win
    let merged = json.parse(base_text)
    // In practice, iterate overlay keys
    return merged
}

let defaults = { color: "blue", size: 12, bold: false }
let user_prefs = { size: 16, bold: true }

say "Defaults: {json.stringify(defaults)}"
say "User prefs: {json.stringify(user_prefs)}"
```

3.7 Chapter 15: regex — Regular Expressions

Regular expressions are the Swiss Army knife of text processing, and Forge's `regex` module makes them accessible through five focused functions. You can test whether a pattern matches, extract the first or all occurrences, replace matches, or split text by a pattern. Under the hood, Forge uses Rust's `regex` crate—one of the fastest regex engines available—so even complex patterns over large inputs run efficiently.

All functions take the **text first, then the pattern**. This order reads naturally in Forge: "search *this text* for *this pattern*."

3.7.1 Pattern Syntax Quick Reference

Pattern	Matches	Example
`.`	Any character except newline	`a.c` ☐ "abc", "a1c"
`\d`	Digit (0–9)	`\d+` ☐ "42", "7"
`\w`	Word character (letter, digit, _)	`\w+` ☐ "hello", "x_1"
`\s`	Whitespace	`\s+` ☐ " ","
`[abc]`	Character class	`[aeiou]` ☐ vowels
`[^abc]`	Negated class	`[^0-9]` ☐ non-digits
`^`	Start of string	`^Hello`

Pattern	Matches	Example
`$`	End of string	`world$`
`*`	Zero or more	`ab*c` ☐ “ac”, “abbc”
`+`	One or more	`ab+c` ☐ “abc”, “abbc”
`?`	Zero or one	`colou?r` ☐ “color”, “colour”
`{n,m}`	Between n and m repetitions	`\d{2,4}` ☐ “42”, “1234”
`()`	Capture group	`(\d+)-(\d+)`
`|`	Alternation	`cat|dog`

3.7.2 Function Reference

Function	Description	Example	Return Type
`regex.test(text, pattern)`	Test if pattern matches anywhere in text	`regex.test("hello", "ell")` ☐ `true`	Bool
`regex.find(text, pattern)`	Find first match	`regex.find("abc123", "\\d+")` ☐ `"123"`	String or Null
`regex.find_all(text, pattern)`	Find all matches	`regex.find_all("a1b2c3", "\\d")` ☐ `["1","2","3"]`	Array[String]
`regex.replace(text, pat, repl)`	Replace all matches	`regex.replace("aabaa", "a+", "x")` ☐ `"xbx"`	String
`regex.split(text, pattern)`	Split text by pattern	`regex.split("a:b::c", ":+")` ☐ `["a","b","c"]`	Array[String]

> **Backslash Escaping.** In Forge strings, backslashes need to be doubled for regex special sequences: write `"\\d+"` to match one or more digits. The first backslash escapes the second, so the regex engine receives `\d+`.

3.7.3 Core Examples

Testing for patterns:

```
let email = "alice@example.com"
let valid = regex.test(email, "^[\\w.+-]+@[\\w-]+\\.[\\w.]+$")
say "Valid email: {valid}"

let has_number = regex.test("abc123", "\\d")
say "Has number: {has_number}"

let starts_with_hello = regex.test("Hello, World!", "^Hello")
say "Starts with Hello: {starts_with_hello}"
```

Output:

```
Valid email: true
Has number: true
Starts with Hello: true
```

Finding matches:

```
let text = "Order #12345 was placed on 2024-01-15"

let order_id = regex.find(text, "#(\\d+)")
say "Order ID: {order_id}"

let date = regex.find(text, "\\d{4}-\\d{2}-\\d{2}")
say "Date: {date}"

let missing = regex.find(text, "refund")
say "Refund mention: {missing}"
```

Output:

```
Order ID: #12345
Date: 2024-01-15
Refund mention: null
```

Finding all matches:

```
let log = "Error at 10:30, Warning at 11:45, Error at 14:20"

let times = regex.find_all(log, "\\d{2}:\\d{2}")
say "Times found: {times}"

let errors = regex.find_all(log, "Error")
say "Error count: {len(errors)}"
```

Output:

```
Times found: ["10:30", "11:45", "14:20"]
Error count: 2
```

Replacing patterns:

```
let messy = "too    many     spaces    here"
let clean = regex.replace(messy, "\\s+", " ")
say "Cleaned: {clean}"

let censored = regex.replace("My phone is 555-1234", "\\d", "*")
say "Censored: {censored}"
```

Output:

```
Cleaned: too many spaces here
Censored: My phone is ***-****
```

Splitting text:

```
let csv_line = "Alice,30,Engineer,New York"
let fields = regex.split(csv_line, ",")
say "Fields: {fields}"

let words = regex.split("one  two\tthree\nfour", "\\s+")
say "Words: {words}"
```

Output:

```
Fields: ["Alice", "30", "Engineer", "New York"]
Words: ["one", "two", "three", "four"]
```

3.7.4 Recipes

Recipe 15.1: Input Validation

```
fn validate_username(username) {
    if regex.test(username, "^[a-zA-Z][a-zA-Z0-9_]{2,19}$") {
        return Ok(username)
    }
    return Err("Username must start with a letter, 3–20 chars, only letters/digits/underscores")
}

fn validate_email(email) {
    if regex.test(email, "^[\\w.+-]+@[\\w-]+\\.[\\w.]+$") {
        return Ok(email)
    }
    return Err("Invalid email format")
}

let tests = ["alice", "bob_42", "3bad", "ab", "valid_user_name"]
for t in tests {
    let result = validate_username(t)
    let ok = is_ok(result)
    say "{t}: {ok}"
}
```

Output:

```
alice: true
bob_42: true
3bad: false
ab: false
valid_user_name: true
```

Recipe 15.2: Log Parser

```
fn parse_log_line(line) {
    let timestamp = regex.find(line, "\\d{4}-\\d{2}-\\d{2} \\d{2}:\\d{2}:\\d{2}")
    let level = regex.find(line, "\\b(INFO|WARN|ERROR|DEBUG)\\b")
    let message = regex.replace(line, "^.*?(INFO|WARN|ERROR|DEBUG)\\s*", "")
    return {
        timestamp: timestamp,
        level: level,
        message: message
    }
}

let log_lines = [
    "2024-01-15 10:30:00 INFO Server started on port 8080",
    "2024-01-15 10:30:05 WARN High memory usage: 85%",
    "2024-01-15 10:31:12 ERROR Connection refused: database"
]

for line in log_lines {
    let parsed = parse_log_line(line)
    say "[{parsed.level}] {parsed.message}"
}
```

Recipe 15.3: Data Extraction

```
let html = "<a href=\"https://example.com\">Example</a> and <a href=\"https://forge-lang.org\">Fo

let urls = regex.find_all(html, "https?://[^\"]+")
say "URLs found:"
for url in urls {
    say "  {url}"
}

// Extract all numbers from mixed text
let report = "Revenue: $1,234,567. Users: 42,000. Growth: 15.7%"
let numbers = regex.find_all(report, "[\\d,]+\\.?\\d*")
say "Numbers: {numbers}"
```

Output:

```
URLs found:
  https://example.com
  https://forge-lang.org
Numbers: ["1,234,567", "42,000", "15.7"]
```

Recipe 15.4: Search and Replace Tool

```
fn redact_sensitive(text) {
    let mut result = text
    // Redact credit card-like patterns
    result = regex.replace(result, "\\b\\d{4}[- ]?\\d{4}[- ]?\\d{4}[- ]?\\d{4}\\b", "[REDACTED-CC
    // Redact SSN-like patterns
    result = regex.replace(result, "\\b\\d{3}-\\d{2}-\\d{4}\\b", "[REDACTED-SSN]")
    // Redact email addresses
    result = regex.replace(result, "[\\w.+-]+@[\\w-]+\\.[\\w.]+", "[REDACTED-EMAIL]")
    return result
}

let sensitive = "Contact alice@example.com, SSN 123-45-6789, Card 4111-1111-1111-1111"
let safe = redact_sensitive(sensitive)
say safe
```

Output:

```
Contact [REDACTED-EMAIL], SSN [REDACTED-SSN], Card [REDACTED-CC]
```

3.8 Chapter 16: env — Environment Variables

Environment variables are the standard mechanism for passing configuration to applications. The `env` module provides four functions that read, write, check, and enumerate environment variables within the running Forge process. Values set with `env.set()` affect only the current process and its children—they do not persist after the program exits.

This module is small by design. Combined with the `fs` and `json` modules, it covers all common configuration patterns, from simple feature flags to environment-aware deployment scripts.

3.8.1 Function Reference

Function	Description	Example	Return Type
`env.get(key)`	Get an environment variable's value	`env.get("HOME")` ☐ `"/Users/alice"`	String or Null
`env.get(key, default)`	Get with a fallback default	`env.get("PORT", "3000")` ☐ `"3000"`	String
`env.set(key, value)`	Set an environment variable (process-local)	`env.set("APP_MODE", "test")`	Null
`env.has(key)`	Check if a variable is defined	`env.has("DATABASE_URL")` ☐ `false`	Bool
`env.keys()`	List all environment variable names	`env.keys()` ☐ `["HOME", "PATH", ...]`	Array[String]

Default Values. `env.get()` with two arguments never returns `null`—the second argument serves as a guaranteed fallback. Use the one-argument form when you need to detect missing variables explicitly.

3.8.2 Core Examples

Reading system variables:

```
let home = env.get("HOME")
say "Home directory: {home}"

let shell = env.get("SHELL", "unknown")
say "Shell: {shell}"

let has_path = env.has("PATH")
say "PATH defined: {has_path}"
```

Output:

```
Home directory: /Users/alice
Shell: /bin/bash
PATH defined: true
```

Setting and reading process-local variables:

```
env.set("APP_ENV", "production")
env.set("LOG_LEVEL", "warn")

let app_env = env.get("APP_ENV")
let log_level = env.get("LOG_LEVEL")
say "Environment: {app_env}"
say "Log level: {log_level}"
```

Output:

```
Environment: production
Log level: warn
```

Checking for required configuration:

```
fn require_env(key) {
    if env.has(key) == false {
        say "ERROR: Required environment variable '{key}' is not set"
        return null
    }
    return env.get(key)
}

// This would fail if DATABASE_URL isn't set
let db_url = require_env("DATABASE_URL")
if db_url == null {
    say "Please set DATABASE_URL before running this program"
}
```

Listing environment variables:

```
let all_keys = env.keys()
let count = len(all_keys)
say "Total environment variables: {count}"

// Show first 5
let mut shown = 0
for key in all_keys {
    if shown < 5 {
        let val = env.get(key)
        say "  {key} = {val}"
        shown = shown + 1
    }
}
```

Using defaults for optional configuration:

```
let port = env.get("PORT", "8080")
let host = env.get("HOST", "0.0.0.0")
let workers = env.get("WORKERS", "4")

say "Server will listen on {host}:{port} with {workers} workers"
```

Output:

```
Server will listen on 0.0.0.0:8080 with 4 workers
```

3.8.3 Recipes

Recipe 16.1: Configuration Management

```
fn load_env_config() {
    return {
        app_name: env.get("APP_NAME", "MyApp"),
        environment: env.get("APP_ENV", "development"),
        port: env.get("PORT", "3000"),
        db_host: env.get("DB_HOST", "localhost"),
        db_name: env.get("DB_NAME", "myapp_dev"),
        log_level: env.get("LOG_LEVEL", "debug"),
        debug: env.has("DEBUG")
    }
}

let config = load_env_config()
say "Application: {config.app_name}"
say "Environment: {config.environment}"
say "Port: {config.port}"
say "Debug mode: {config.debug}"
```

Recipe 16.2: Feature Flags

```
fn feature_enabled(flag_name) {
    let key = "FEATURE_{flag_name}"
    let val = env.get(key, "false")
    return val == "true" || val == "1" || val == "yes"
}

env.set("FEATURE_NEW_UI", "true")
env.set("FEATURE_BETA_API", "false")

let new_ui = feature_enabled("NEW_UI")
let beta = feature_enabled("BETA_API")
let dark_mode = feature_enabled("DARK_MODE")

say "New UI: {new_ui}"
say "Beta API: {beta}"
say "Dark mode: {dark_mode}"
```

Output:

```
New UI: true
Beta API: false
Dark mode: false
```

Recipe 16.3: Environment Detection

```
fn detect_environment() {
    // Check for common CI/CD variables
    if env.has("CI") {
        return "ci"
    }
    if env.has("KUBERNETES_SERVICE_HOST") {
        return "kubernetes"
    }
    if env.has("AWS_LAMBDA_FUNCTION_NAME") {
        return "lambda"
    }
    if env.has("HEROKU_APP_NAME") {
        return "heroku"
    }
    return env.get("APP_ENV", "development")
}

let platform = detect_environment()
say "Running in: {platform}"
```

Recipe 16.4: Secrets Validator

```
fn validate_secrets(required_keys) {
    let mut missing = []
    for key in required_keys {
        if env.has(key) == false {
            missing = append(missing, key)
        }
    }
    if len(missing) > 0 {
        say "Missing required environment variables:"
        for key in missing {
            say "  - {key}"
        }
        return false
    }
    say "All required secrets are configured"
    return true
}

let required = ["DATABASE_URL", "SECRET_KEY", "API_TOKEN"]
let ok = validate_secrets(required)
```

3.9 Chapter 17: csv — Tabular Data

CSV (Comma-Separated Values) remains one of the most widely used data exchange formats, particularly for spreadsheets, data exports, and ETL pipelines. The `csv` module handles parsing and serialization of CSV data, automatically detecting column types and producing clean output. It treats the first row as headers and returns an array of objects where each key is a column name.

3.9.1 Function Reference

Function	Description	Example	Return Type
`csv.parse(text)`	Parse a CSV string into an array of objects	`csv.parse("name,age\nAlice,30")` ☐ `[{name:"Alice"...}]`	Array[Object]
`csv.stringify(rows)`	Convert an array of objects to a CSV string	`csv.stringify([{a:1}])` ☐ `"a\n1\n"`	String
`csv.read(path)`	Read a CSV file and parse it	`csv.read("data.csv")` ☐ `[{...}]`	Array[Object]
`csv.write(path, rows)`	Write an array of objects to a CSV file	`csv.write("out.csv", rows)`	Null

> **Automatic Type Detection.** `csv.parse()` inspects each cell and converts it to the most specific Forge type: integers for whole numbers, floats for decimals, booleans for "true"/"false", and strings for everything else. This means `"42"` becomes `Int(42)` and `"3.14"` becomes `Float(3.14)`.

3.9.2 Core Examples

Parsing CSV text:

```
let data = "name,age,city
Alice,30,New York
Bob,25,London
Charlie,35,Tokyo"

let rows = csv.parse(data)
for row in rows {
    say "{row.name} is {row.age} years old, lives in {row.city}"
}
```

Output:

```
Alice is 30 years old, lives in New York
Bob is 25 years old, lives in London
Charlie is 35 years old, lives in Tokyo
```

Type detection in action:

```
let data = "metric,value,active
cpu,72.5,true
memory,4096,false
disk,45.2,true"

let rows = csv.parse(data)
for row in rows {
    say "{row.metric}: {row.value} (active: {row.active})"
}
```

Output:

```
cpu: 72.5 (active: true)
memory: 4096 (active: false)
disk: 45.2 (active: true)
```

Creating CSV from objects:

```
let products = [
    { name: "Widget", price: 9.99, stock: 150 },
    { name: "Gadget", price: 24.99, stock: 75 },
    { name: "Gizmo", price: 49.99, stock: 30 }
]

let csv_text = csv.stringify(products)
say csv_text
```

Output:

```
name,price,stock
Widget,9.99,150
Gadget,24.99,75
Gizmo,49.99,30
```

Reading and writing CSV files:

```
// Write a CSV file
let employees = [
    { id: 1, name: "Alice", department: "Engineering", salary: 95000 },
    { id: 2, name: "Bob", department: "Marketing", salary: 72000 },
    { id: 3, name: "Carol", department: "Engineering", salary: 98000 }
]

csv.write("/tmp/employees.csv", employees)
```

```
say "Wrote CSV file"

// Read it back
let loaded = csv.read("/tmp/employees.csv")
say "Loaded {len(loaded)} rows"
term.table(loaded)

fs.remove("/tmp/employees.csv")
```

Handling special characters:

```
let tricky = [
    { name: "Smith, John", role: "Manager", note: "Says \"hello\" often" },
    { name: "Jane Doe", role: "Developer", note: "No issues" }
]

let text = csv.stringify(tricky)
say text
```

> **Escaping.** `csv.stringify()` automatically quotes fields that contain commas or double quotes, following RFC 4180 conventions.

3.9.3 Recipes

Recipe 17.1: Data Import and Analysis

```
let sales_data = "product,quarter,revenue
Widget,Q1,125000
Widget,Q2,142000
Widget,Q3,138000
Widget,Q4,165000
Gadget,Q1,89000
Gadget,Q2,95000
Gadget,Q3,102000
Gadget,Q4,118000"

let sales = csv.parse(sales_data)

// Calculate totals per product
let widget_sales = filter(sales, fn(r) { return r.product == "Widget" })
let gadget_sales = filter(sales, fn(r) { return r.product == "Gadget" })

let widget_total = reduce(widget_sales, 0, fn(acc, r) { return acc + r.revenue })
let gadget_total = reduce(gadget_sales, 0, fn(acc, r) { return acc + r.revenue })
```

```
say "Widget annual revenue: ${widget_total}"
say "Gadget annual revenue: ${gadget_total}"
```

Output:

```
Widget annual revenue: $570000
Gadget annual revenue: $404000
```

Recipe 17.2: Report Generation

```
fn generate_report(title, data) {
    say "=== {title} ==="
    term.table(data)
    say ""

    // Export to CSV
    let filename = "/tmp/report.csv"
    csv.write(filename, data)
    say "Report exported to {filename}"
    return filename
}

let metrics = [
    { metric: "Page Views", value: 125000, change: "+12%" },
    { metric: "Unique Users", value: 45000, change: "+8%" },
    { metric: "Bounce Rate", value: 32, change: "-3%" },
    { metric: "Avg Session", value: 245, change: "+15%" }
]

let path = generate_report("Weekly Metrics", metrics)
fs.remove(path)
```

Recipe 17.3: CSV-to-JSON Converter

```
fn csv_to_json(csv_path, json_path) {
    let rows = csv.read(csv_path)
    fs.write_json(json_path, rows)
    let count = len(rows)
    say "Converted {count} rows from CSV to JSON"
}

fn json_to_csv(json_path, csv_path) {
    let data = fs.read_json(json_path)
    csv.write(csv_path, data)
    let count = len(data)
    say "Converted {count} rows from JSON to CSV"
```

```
}

// Create test data
let test_data = "name,score,grade\nAlice,95,A\nBob,82,B\nCarol,91,A"
fs.write("/tmp/students.csv", test_data)

csv_to_json("/tmp/students.csv", "/tmp/students.json")

// Verify
let json_data = fs.read_json("/tmp/students.json")
say json.pretty(json_data)

// Clean up
fs.remove("/tmp/students.csv")
fs.remove("/tmp/students.json")
```

Recipe 17.4: ETL Pipeline

```
// Extract
let raw = "timestamp,sensor_id,temperature,humidity
2024-01-15T10:00:00,S001,22.5,45
2024-01-15T10:00:00,S002,23.1,42
2024-01-15T10:05:00,S001,22.8,44
2024-01-15T10:05:00,S002,23.4,41
2024-01-15T10:10:00,S001,23.0,43
2024-01-15T10:10:00,S002,24.0,40"

let readings = csv.parse(raw)
say "Extracted {len(readings)} readings"

// Transform: calculate averages per sensor
let s001 = filter(readings, fn(r) { return r.sensor_id == "S001" })
let s002 = filter(readings, fn(r) { return r.sensor_id == "S002" })

let s001_avg_temp = reduce(s001, 0.0, fn(acc, r) { return acc + r.temperature }) / len(s001)
let s002_avg_temp = reduce(s002, 0.0, fn(acc, r) { return acc + r.temperature }) / len(s002)

let summary = [
    { sensor: "S001", avg_temperature: s001_avg_temp, readings: len(s001) },
    { sensor: "S002", avg_temperature: s002_avg_temp, readings: len(s002) }
]

// Load: write summary
say "Sensor Averages:"
term.table(summary)
csv.write("/tmp/sensor_summary.csv", summary)
say "Summary written to /tmp/sensor_summary.csv"
```

```
fs.remove("/tmp/sensor_summary.csv")
```

3.10 Chapter 18: log — Structured Logging

Every non-trivial program needs logging, and the `log` module provides four severity-level functions that write timestamped, color-coded messages to standard error. The interface is intentionally simple—call the function matching your severity level and pass any number of arguments. The module handles formatting, timestamps, and color.

Logs go to stderr, so they remain separate from your program's standard output. This distinction matters: you can pipe a Forge program's output to another tool while still seeing diagnostics in the terminal.

3.10.1 Function Reference

Function	Description	Color	Example	Return Type
`log.info(...)`	Informational message	Green	`log.info("Server started")`	Null
`log.warn(...)`	Warning—something unexpected	Yellow	`log.warn("Disk space low")`	Null
`log.error(...)`	Error—something went wrong	Red	`log.error("Connection failed")`	Null
`log.debug(...)`	Debug—detailed diagnostic info	Dim	`log.debug("Query took 42ms")`	Null

> **Variable Arguments.** All four functions accept any number of arguments of any type. Arguments are converted to strings and joined with spaces.

3.10.2 Output Format

Each log line follows this format:

```
[HH:MM:SS LEVEL] message
```

For example:

```
[10:30:45 INFO] Server started on port 8080
[10:30:45 WARN] No database URL configured, using defaults
[10:30:46 ERROR] Failed to connect to cache server
[10:30:46 DEBUG] Retrying connection attempt 2 of 3
```

3.10.3 Core Examples

Basic logging at each level:

```
log.info("Application starting")
log.debug("Loading configuration from /etc/app/config.json")
log.warn("API key expires in 3 days")
log.error("Failed to open database connection")
```

Output (stderr, with colors):

```
[14:20:00 INFO] Application starting
[14:20:00 DEBUG] Loading configuration from /etc/app/config.json
[14:20:00 WARN] API key expires in 3 days
[14:20:00 ERROR] Failed to open database connection
```

Logging multiple values:

```
let user = "alice"
let action = "login"
let ip = "192.168.1.100"
log.info("User", user, "performed", action, "from", ip)
```

Output:

```
[14:20:00 INFO] User alice performed login from 192.168.1.100
```

Logging objects and arrays:

```
let request = { method: "POST", path: "/api/users", status: 201 }
log.info("Request completed:", request)

let errors = ["timeout", "retry_exhausted"]
log.error("Multiple failures:", errors)
```

Conditional logging:

```
fn process_item(item) {
    log.debug("Processing item:", item)

    if item.price < 0 {
        log.warn("Negative price detected for", item.name)
        return false
    }
```

```
    if item.stock == 0 {
        log.error("Out of stock:", item.name)
        return false
    }

    log.info("Successfully processed", item.name)
    return true
}

process_item({ name: "Widget", price: 9.99, stock: 10 })
process_item({ name: "Broken", price: -1, stock: 5 })
process_item({ name: "Sold Out", price: 19.99, stock: 0 })
```

Timing operations:

```
fn timed_operation(name) {
    log.debug("Starting:", name)
    // Simulate work
    let mut sum = 0
    repeat 10000 times {
        sum = sum + 1
    }
    log.info("Completed:", name)
    return sum
}

timed_operation("data processing")
timed_operation("report generation")
```

3.10.4 Recipes

Recipe 18.1: Application Logger

```
fn create_logger(module_name) {
    return {
        info: fn(msg) { log.info("[{module_name}]", msg) },
        warn: fn(msg) { log.warn("[{module_name}]", msg) },
        error: fn(msg) { log.error("[{module_name}]", msg) },
        debug: fn(msg) { log.debug("[{module_name}]", msg) }
    }
}

let db_log = create_logger("database")
let api_log = create_logger("api")

db_log.info("Connected to PostgreSQL")
```

```
api_log.info("Listening on port 8080")
db_log.warn("Slow query detected: 2.3s")
api_log.error("Request timeout on /api/users")
```

Output:

```
[14:20:00 INFO] [database] Connected to PostgreSQL
[14:20:00 INFO] [api] Listening on port 8080
[14:20:00 WARN] [database] Slow query detected: 2.3s
[14:20:00 ERROR] [api] Request timeout on /api/users
```

Recipe 18.2: Debug Tracing

```
fn trace(label, value) {
    log.debug("TRACE [{label}]:", value)
    return value
}

// Use trace to follow data through a pipeline
let data = [10, 25, 3, 47, 12]
let filtered = trace("after filter", filter(data, fn(x) { return x > 10 }))
let mapped = trace("after map", map(filtered, fn(x) { return x * 2 }))
let total = trace("after reduce", reduce(mapped, 0, fn(a, b) { return a + b }))
say "Result: {total}"
```

Recipe 18.3: Error Reporting with Context

```
fn log_error_with_context(operation, error_msg, context) {
    log.error("Operation failed:", operation)
    log.error("  Error:", error_msg)
    log.error("  Context:", json.stringify(context))
}

fn process_payment(order) {
    if order.amount <= 0 {
        log_error_with_context("process_payment", "Invalid amount", order)
        return false
    }
    log.info("Payment processed:", order.amount, "for order", order.id)
    return true
}

process_payment({ id: "ORD-001", amount: 49.99, currency: "USD" })
process_payment({ id: "ORD-002", amount: 0, currency: "USD" })
```

Recipe 18.4: Startup Diagnostics

```
fn startup_checks() {
    log.info("=== Startup Diagnostics ===")

    let home = env.get("HOME", "unknown")
    log.info("Home directory:", home)

    let app_env = env.get("APP_ENV", "development")
    log.info("Environment:", app_env)

    if app_env == "production" {
        if env.has("DATABASE_URL") == false {
            log.error("DATABASE_URL is required in production!")
        }
        if env.has("SECRET_KEY") == false {
            log.error("SECRET_KEY is required in production!")
        }
    } else {
        log.debug("Running in", app_env, "mode — relaxed checks")
    }

    log.info("=== Diagnostics Complete ===")
}

startup_checks()
```

3.11 Chapter 19: term — Terminal UI

The `term` module transforms the terminal from a plain text canvas into a rich presentation layer. It offers color functions for styling text, display functions for structured output like tables and progress bars, interactive prompts for user input, and visual effects that bring CLI applications to life. If you are building a command-line tool, a dashboard, or any interactive script, `term` is the module that makes it polished.

The module writes styled output using ANSI escape codes, which are supported by virtually every modern terminal emulator. Color functions return styled strings (so you can compose them), while display functions print directly to the terminal.

3.11.1 Color and Style Functions

Color functions wrap text in ANSI escape sequences and **return a styled string**. You can assign them to variables, embed them in larger strings, or pass them to `say`.

Function	Description	Example	Return Type
`term.red(text)`	Red foreground color	`term.red("error!")` ☐ styled string	String
`term.green(text)`	Green foreground color	`term.green("success")` ☐ styled	String
`term.blue(text)`	Blue foreground color	`term.blue("info")` ☐ styled	String
`term.yellow(text)`	Yellow foreground color	`term.yellow("warning")` ☐ styled	String
`term.cyan(text)`	Cyan foreground color	`term.cyan("note")` ☐ styled	String
`term.magenta(text)`	Magenta foreground color	`term.magenta("special")` ☐ styled	String
`term.bold(text)`	Bold weight	`term.bold("important")` ☐ styled	String
`term.dim(text)`	Dim/faint style	`term.dim("secondary")` ☐ styled	String

3.11.2 Display Functions

Display functions produce formatted output directly in the terminal.

Function	Description	Example	Return Type
`term.table(rows)`	Print formatted, aligned table	`term.table([{a:1,b:2}])`	Null
`term.hr()`	Print horizontal rule (default width 40)	`term.hr()`	Null
`term.hr(width)`	Horizontal rule with custom width	`term.hr(60)`	Null
`term.hr(width, char)`	Horizontal rule with custom character	`term.hr(20, "=")`	Null
`term.clear()`	Clear the terminal screen	`term.clear()`	Null
`term.sparkline(values)`	Return a sparkline string (☐☐☐☐☐☐☐☐)	`term.sparkline([1,5,3,8,2])` ☐ "▁▅▃▇▂"	String
`term.bar(label, value)`	Print a progress bar (max 100)	`term.bar("CPU", 72)`	Null
`term.bar(label, value, max)`	Progress bar with custom max	`term.bar("Sales", 750, 1000)`	Null
`term.banner(text)`	Print text in a bordered banner (☐)	`term.banner("Welcome!")`	Null
`term.box(text)`	Print text in a bordered box (—)	`term.box("Hello")`	Null
`term.countdown(seconds)`	Visual countdown timer	`term.countdown(3)`	Null
`term.typewriter(text)`	Print text character by character	`term.typewriter("Loading...")`	Null
`term.typewriter(text, delay)`	Typewriter with custom delay (ms)	`term.typewriter("Fast!", 10)`	Null
`term.gradient(text)`	Return text with rainbow gradient colors	`term.gradient("Rainbow!")` ☐ styled	String

3.11.3 Status Message Functions

Quick, emoji-prefixed status messages for common feedback patterns.

Function	Description	Emoji	Color	Return Type
`term.success(msg)`	Print success message	☐	Green	Null
`term.error(msg)`	Print error message	☐	Red	Null
`term.warning(msg)`	Print warning message	☐	Yellow	Null
`term.info(msg)`	Print informational message	☐	Cyan	Null

3.11.4 Interactive Functions

Functions that accept user input, blocking until the user responds.

Function	Description	Example	Return Type
`term.confirm()`	Prompt user for yes/no (default prompt)	`term.confirm()` ☐ `true` or `false`	Bool
`term.confirm(prompt)`	Prompt with custom question	`term.confirm("Deploy?")` ☐ `true`	Bool
`term.menu(options)`	Show a numbered menu, return selected item	`term.menu(["A","B","C"])` ☐ `"B"`	Value or Null
`term.menu(options, prompt)`	Menu with custom prompt	`term.menu(items, "Pick one:")`	Value or Null

3.11.5 Effects and Emoji Functions

Function	Description	Example	Return Type
`term.beep()`	Play system bell sound	`term.beep()`	Null
`term.emoji(name)`	Get emoji by name	`term.emoji("rocket")` ☐ "□ "	String
`term.emojis()`	List all available emoji names	`term.emojis()`	Null

Available Emoji Names:

Name(s)	Emoji
`check`, `ok`, `yes`	☐
`cross`, `no`, `fail`	☐
`star`, `fav`	☐
`fire`, `hot`	☐

Name(s)	Emoji
heart, love	☐
rocket, launch	☐
warn, warning	☐
info, information	☐
bug, error	☐
clock, time	☐
folder, dir	☐
file, doc	☐
lock, secure	☐
key	☐
link, url	☐
mail, email	☐
globe, web, world	☐
party, celebrate	☐
think, hmm	☐
wave, hi, hello	☐
thumbsup, good	☐
thumbsdown, bad	☐
100, perfect	☐
zap, bolt, fast	☐
gear, settings	☐
tools, wrench	☐

3.11.6 Core Examples

Colored text:

```
say term.red("Error: file not found")
say term.green("Success: deployment complete")
say term.yellow("Warning: disk space low")
say term.blue("Info: 42 items processed")
say term.bold("This text is bold")
say term.dim("This text is dimmed")
```

Combining styles:

```
let header = term.bold("=== System Status ===")
say header

let status = term.green("ONLINE")
say "  Server: {status}"

let warning = term.yellow("HIGH")
say "  CPU Usage: {warning}"
```

```
let critical = term.bold(term.red("CRITICAL"))
say "  Disk: {critical}"
```

Tables:

```
let servers = [
    { name: "web-01", status: "running", cpu: "45%", memory: "2.1 GB" },
    { name: "web-02", status: "running", cpu: "62%", memory: "3.4 GB" },
    { name: "db-01", status: "running", cpu: "78%", memory: "8.2 GB" },
    { name: "cache-01", status: "stopped", cpu: "0%", memory: "0 GB" }
]

say term.bold("Server Dashboard")
term.table(servers)
```

Output:

```
Server Dashboard
┌──────────┬──────────┬─────┬──────────┐
│ name     │ status   │ cpu │ memory   │
├──────────┼──────────┼─────┼──────────┤
│ web-01   │ running  │ 45% │ 2.1 GB   │
│ web-02   │ running  │ 62% │ 3.4 GB   │
│ db-01    │ running  │ 78% │ 8.2 GB   │
│ cache-01 │ stopped  │ 0%  │ 0 GB     │
└──────────┴──────────┴─────┴──────────┘
```

Sparklines and progress bars:

```
let cpu_history = [23, 45, 67, 34, 89, 56, 78, 12, 45, 90]
let spark = term.sparkline(cpu_history)
say "CPU trend: {spark}"

term.bar("Downloads", 73)
term.bar("Uploads", 45)
term.bar("Storage", 891, 1000)
```

Output:

```
CPU trend: ▂▅▃▇▅▂▅
Downloads  |██████████████████████████████████████████████████| 73%
Uploads    |██████████████████████████████████████████████████| 45%
Storage    |██████████████████████████████████████████████████| 891/1000
```

Banners and boxes:

```
term.banner("Welcome to Forge!")
term.box("This is a boxed message\nwith multiple lines")
term.hr()
term.hr(60, "=")
```

Status messages:

```
term.success("Build completed successfully")
term.warning("3 deprecated functions detected")
term.error("Test suite failed: 2 failures")
term.info("Next build scheduled for 10:00 AM")
```

Output:

```
  Build completed successfully
  3 deprecated functions detected
  Test suite failed: 2 failures
  Next build scheduled for 10:00 AM
```

Emojis:

```
let rocket = term.emoji("rocket")
let fire = term.emoji("fire")
let check = term.emoji("check")
say "{rocket} Launching deployment..."
say "{fire} Build is hot!"
say "{check} All tests passed"
```

Output:

```
  Launching deployment...
  Build is hot!
  All tests passed
```

Interactive confirm:

```
let proceed = term.confirm("Deploy to production?")
if proceed {
    say "Deploying..."
} else {
    say "Aborted"
}
```

Output:

```
Deploy to production? [y/N] y
Deploying...
```

Interactive menu:

```
let choice = term.menu(["Development", "Staging", "Production"], "Select environment:")
say "You chose: {choice}"
```

Output:

```
Select environment:
  1) Development
  2) Staging
  3) Production
> 2
You chose: Staging
```

3.11.7 Recipes

Recipe 19.1: Dashboard Builder

```
fn show_dashboard(metrics) {
    term.clear()
    term.banner("System Dashboard")
    say ""

    // Status indicators
    for m in metrics {
        let icon = term.emoji("check")
        if m.value > 80 {
            let icon = term.emoji("warning")
        }
        if m.value > 95 {
            let icon = term.emoji("cross")
        }
        say " {icon} {m.label}"
        term.bar(m.label, m.value)
    }

    say ""
    term.hr()

    // Trend sparklines
    say term.bold("Trends (last 10 readings):")
    let cpu_spark = term.sparkline([45, 52, 48, 67, 72, 65, 78, 82, 71, 68])
    let mem_spark = term.sparkline([60, 61, 63, 62, 65, 64, 68, 70, 69, 72])
```

```
    say "  CPU:    {cpu_spark}"
    say "  Memory: {mem_spark}"
}

let metrics = [
    { label: "CPU", value: 68 },
    { label: "Memory", value: 72 },
    { label: "Disk", value: 45 },
    { label: "Network", value: 23 }
]

show_dashboard(metrics)
```

Recipe 19.2: Progress Reporting

```
fn process_with_progress(items) {
    let total = len(items)
    let mut processed = 0

    for item in items {
        processed = processed + 1
        // Simulate work
        let pct = processed * 100 / total
        term.bar("Progress", pct)
    }

    say ""
    term.success("Processed {total} items")
}

let items = ["file1.dat", "file2.dat", "file3.dat", "file4.dat", "file5.dat"]
process_with_progress(items)
```

Recipe 19.3: Interactive CLI Tool

```
fn run_cli() {
    term.banner("Forge Task Manager")
    say ""

    let action = term.menu([
        "List tasks",
        "Add task",
        "Complete task",
        "Generate report",
        "Exit"
    ], "What would you like to do?")
```

```
    if action == "List tasks" {
        let tasks = [
            { id: 1, title: "Write documentation", status: "in progress" },
            { id: 2, title: "Fix login bug", status: "pending" },
            { id: 3, title: "Deploy v2.0", status: "pending" }
        ]
        say ""
        term.table(tasks)
    } else if action == "Add task" {
        say "Adding new task..."
        term.success("Task added!")
    } else if action == "Generate report" {
        say ""
        term.info("Generating report...")
        term.typewriter("Analyzing tasks... Done!", 20)
        say ""
        term.success("Report generated")
    } else if action == "Exit" {
        let wave = term.emoji("wave")
        say "{wave} Goodbye!"
    }
}

run_cli()
```

Recipe 19.4: Data Visualization

```
fn visualize_data(title, dataset) {
    say term.bold(title)
    term.hr(50)

    // Table view
    term.table(dataset)
    say ""

    // Bar chart
    say term.bold("Bar Chart:")
    let values = map(dataset, fn(d) { return d.value })
    let max_val = reduce(values, 0, fn(a, b) {
        if b > a { return b }
        return a
    })
    for row in dataset {
        let val = float(row.value)
        let max_f = float(max_val)
        term.bar(row.label, val, max_f)
    }
```

```
    say ""

    // Sparkline
    say term.bold("Trend:")
    let spark = term.sparkline(values)
    say "  {spark}"
    say ""
    term.hr(50)
}

let monthly_revenue = [
    { label: "Jan", value: 12000 },
    { label: "Feb", value: 15000 },
    { label: "Mar", value: 13500 },
    { label: "Apr", value: 18000 },
    { label: "May", value: 22000 },
    { label: "Jun", value: 19500 }
]

visualize_data("Monthly Revenue Report", monthly_revenue)
```

Recipe 19.5: Styled Error Reporter

```
fn report_errors(errors) {
    if len(errors) == 0 {
        term.success("No errors found!")
        return null
    }

    let count = len(errors)
    term.error("{count} error(s) detected")
    say ""

    let mut idx = 0
    for err in errors {
        idx = idx + 1
        let num = term.bold("#{idx}")
        let file = term.cyan(err.file)
        let line_info = term.dim("line {err.line}")
        say " {num} {file} ({line_info})"

        let msg = term.red("    {err.message}")
        say msg
        say ""
    }

    term.hr()
```

```
    let summary = term.bold(term.red("{count} errors must be fixed before deploy"))
    say summary
}

let errors = [
    { file: "src/main.fg", line: 42, message: "Undefined variable 'config'" },
    { file: "src/utils.fg", line: 17, message: "Type mismatch: expected Int, got String" },
    { file: "tests/test_api.fg", line: 8, message: "Assertion failed: expected 200, got 404" }
]

report_errors(errors)
```

Recipe 19.6: Colorful Build Output

```
fn build_project(steps) {
    term.banner("Building Project")
    say ""

    let total = len(steps)
    let mut passed = 0

    for step in steps {
        let name = term.bold(step.name)
        say " {term.emoji("gear")} {name}..."

        if step.ok {
            term.success("  {step.name} complete")
            passed = passed + 1
        } else {
            term.error("  {step.name} failed: {step.error}")
        }
    }

    say ""
    term.hr()

    if passed == total {
        let msg = term.gradient("BUILD SUCCESSFUL")
        say " {term.emoji("party")} {msg}"
    } else {
        let failed = total - passed
        term.error("BUILD FAILED: {failed} of {total} steps failed")
    }
}

let steps = [
    { name: "Compile", ok: true, error: "" },
```

```
    { name: "Lint", ok: true, error: "" },
    { name: "Test", ok: true, error: "" },
    { name: "Bundle", ok: true, error: "" },
    { name: "Deploy", ok: true, error: "" }
]

build_project(steps)
```

3.12 Chapter 20: Shell Integration — First-Class Bash

Forge treats the shell as a first-class citizen. Ten built-in functions give you full control over system commands, from quick one-liners to piping Forge data through Unix tool chains. There is no module prefix—these functions are available globally, so you can run `sh("date")` or `pipe_to(data, "sort -n")` anywhere in your program. Combined with Forge's data types and control flow, they turn scripts into powerful automation tools without dropping to a separate shell.

3.12.1 Function Reference Table

Function	Returns	Description
`sh(cmd)`	String	Run command, return stdout
`shell(cmd)`	Object `{stdout, stderr, status, ok}`	Run command, return full result
`sh_lines(cmd)`	Array of String	Run command, split stdout into lines
`sh_json(cmd)`	Object/Array	Run command, auto-parse JSON output
`sh_ok(cmd)`	Bool	Run command, return true if exit code 0
`which(cmd)`	String or null	Find command path on $PATH
`cwd()`	String	Current working directory
`cd(path)`	String	Change working directory
`lines(text)`	Array of String	Split any string by newlines
`pipe_to(data, cmd)`	Object `{stdout, stderr, status, ok}`	Feed string data into command via stdin
`run_command(cmd)`	Object `{stdout, stderr, status, ok}`	Direct exec without shell (no pipes)

3.12.2 sh — Quick One-Liners

`sh(cmd)` runs a command through `/bin/sh`, captures stdout, trims trailing whitespace, and returns it as a string. It is the fastest way to get a single value from a command. Pipes, redirects, and

variable expansion work—everything your shell supports.

```
let user = sh("whoami")
say "Logged in as: {user}"

let date = sh("date +%Y-%m-%d")
say "Today: {date}"

let kernel = sh("uname -s")
let arch = sh("uname -m")
say "Platform: {kernel} on {arch}"
```

Output:

```
Logged in as: alice
Today: 2026-02-28
Platform: Darwin on x86_64
```

```
let count = sh("ls /etc | wc -l")
say "Files in /etc: {count}"

let disk_pct = sh("df -h / | tail -1 | awk '{print $5}'")
say "Disk usage: {disk_pct}"
```

> **When to use `sh`.** Use `sh()` when you only need stdout and do not care about exit codes or stderr. If the command fails, you still get whatever was printed to stdout—check `shell()` or `sh_ok()` when correctness depends on the exit status.

3.12.3 shell — Full Result Object

`shell(cmd)` runs the same command as `sh()` but returns an object with four fields: `stdout`, `stderr`, `status` (exit code), and `ok` (boolean success). Use it when you need to inspect errors, capture stderr, or branch on whether the command succeeded.

```
let result = shell("ls -la /tmp")
say "Exit code: {result.status}"
say "Success: {result.ok}"
say "Output:\n{result.stdout}"
```

Output:

```
Exit code: 0
Success: true
Output:
total 48
drwxrwxrwt  15 root  wheel  480 Feb 28 10:00 ...
```

```
let r = shell("cat /nonexistent 2>&1")
if r.ok {
    say "File read OK"
} otherwise {
    say "Error: {r.stderr}"
    say "Status: {r.status}"
}
```

```
let ping_result = shell("ping -c 1 -W 2 localhost 2>/dev/null")
if ping_result.ok {
    say "Host is reachable"
} otherwise {
    say "Host unreachable (status: {ping_result.status})"
}
```

3.12.4 sh_lines — Commands That Emit Lines

`sh_lines(cmd)` runs a command and returns its stdout as an array of strings, one per line. Empty lines are dropped. This is ideal for commands like `ls`, `find`, or `ps` whose output you want to iterate or filter.

```
let files = sh_lines("ls /etc | head -5")
say "First 5 files in /etc:"
for f in files {
    say "  {f}"
}
```

Output:

```
First 5 files in /etc:
  afpovertcp.cfg
  aliases
  asl
  bashrc_Apple_Terminal
  ...
```

```
let procs = sh_lines("ps aux | wc -l")
let count = procs[0]
say "Running processes: {count}"

let fg_files = sh_lines("find . -name '*.fg' -maxdepth 2 | head -10")
say "Forge files: {fg_files}"
```

3.12.5 sh_json — Parse JSON from Commands

`sh_json(cmd)` runs a command and parses its stdout as JSON. If the command outputs valid JSON (e.g., `curl` responses, `kubectl get -o json`, or `jq` output), you get a Forge object or array directly.

```
let data = sh_json("echo '{\"name\":\"Forge\",\"version\":1}'")
say "Name: {data.name}, Version: {data.version}"

let arr = sh_json("echo '[1,2,3,4,5]'")
say "Sum: {reduce(arr, 0, fn(a,b){ return a + b })}"
```

Output:

```
Name: Forge, Version: 1
Sum: 15
```

```
let config = sh_json("cat /tmp/config.json")
if config != null {
    say "Loaded config: {config}"
} otherwise {
    say "Failed to parse JSON"
}
```

> **Parse failures.** If the command's stdout is not valid JSON, `sh_json()` raises an error. Wrap in `safe { }` if you need to handle malformed output gracefully.

3.12.6 sh_ok — Exit Code Check

`sh_ok(cmd)` runs a command, discards stdout and stderr, and returns `true` if the exit code is 0, `false` otherwise. It is perfect for existence checks, process probes, and dependency validation.

```
if sh_ok("which docker") {
    say "Docker is installed"
} otherwise {
    say "Docker not found on PATH"
}
```

```
let tools = ["git", "cargo", "curl", "jq"]
for tool in tools {
    let found = sh_ok("which " + tool)
    let status = when found { true -> "found", else -> "MISSING" }
    say "  {tool}: {status}"
}
```

```
if sh_ok("pgrep -q nginx") {
    say "nginx is running"
} otherwise {
    say "nginx is not running"
}
```

3.12.7 which — Resolve Command Path

`which(cmd)` looks up a command name on `$PATH` and returns its full path, or `null` if not found. It uses the system `which` (e.g., `/usr/bin/which`), so it reflects the same resolution as your shell.

```
let git_path = which("git")
say "Git at: {git_path}"

let missing = which("nonexistent_tool_xyz")
if missing == null {
    say "Tool not found"
}
```

```
let tools = ["git", "cargo", "forge"]
for tool in tools {
    let path = which(tool)
    if path != null {
        say "{tool}: {path}"
    } otherwise {
        say "{tool}: NOT FOUND"
    }
}
```

3.12.8 cwd — Current Working Directory

`cwd()` returns the current working directory as a string. It is useful for logging, building paths, or restoring the directory later after a `cd()`.

```
let dir = cwd()
say "Working in: {dir}"

let report_path = cwd() + "/report.txt"
say "Report will be written to: {report_path}"
```

3.12.9 cd — Change Working Directory

`cd(path)` changes the current process's working directory to the given path. Subsequent `sh()`, `shell()`, and file operations use this directory. It returns the path on success and raises an error if the directory does not exist or is not accessible.

```
cd("/tmp")
let dir = cwd()
say "Now in: {dir}"

cd("/var/log")
let log_list = sh_lines("ls | head -5")
say "Log files: {log_list}"
```

> **Process-local.** `cd()` affects only the Forge process. It does not change the shell that invoked `forge run`. Child processes spawned by `sh()` or `shell()` inherit the new working directory.

3.12.10 lines — Split Text by Newlines

`lines(text)` splits any string by newline characters and returns an array of strings. Unlike `sh_lines()`, it does not run a command—it operates on a string you already have. Empty lines are preserved.

```
let log = "2024-01-15 10:30 INFO Started\n2024-01-15 10:31 WARN Retry\n2024-01-15 10:32 ERROR Fai
let log_lines = lines(log)
say "Log entries: {len(log_lines)}"
for line in log_lines {
    say "  {line}"
}
```

```
let csv_text = fs.read("data.csv")
let rows = lines(csv_text)
let header = rows[0]
say "Columns: {header}"

let text = "a\nb\nc\n"
let arr = lines(text)
say "Items: {arr}"
```

3.12.11 pipe_to — Feed Data Into Commands

`pipe_to(data, cmd)` sends a string into a command's stdin and returns the same result object as `shell()`: `{stdout, stderr, status, ok}`. The command receives `data` on stdin—as if you had run `echo "$data" | cmd`. Use it to process Forge data through `sort`, `grep`, `awk`, `jq`, or any Unix filter.

```
let names = "Charlie\nAlice\nBob"
let result = pipe_to(names, "sort")
say result.stdout
```

Output:

```
Alice
Bob
Charlie
```

```
let data = "apple\nbanana\ncherry\napricot\navocado"
let filtered = pipe_to(data, "grep '^a'")
say "Starts with 'a':\n{filtered.stdout}"

let numbers = "42\n17\n99\n3\n28"
let sorted = pipe_to(numbers, "sort -n")
say "Sorted numbers:\n{sorted.stdout}"
```

```
let csv = "name,score\nAlice,95\nBob,82\nCarol,91"
let result = pipe_to(csv, "awk -F',' 'NR>1 {print $2}' | sort -n")
say "Scores (sorted):\n{result.stdout}"
```

3.12.12 run_command — Direct Exec Without Shell

`run_command(cmd)` runs a command without invoking a shell. The command string is split on whitespace: the first token is the program, the rest are arguments. There are no pipes, redirects, or variable expansion. It returns the same `{stdout, stderr, status, ok}` object as `shell()`.

Use `run_command()` when you need to avoid shell interpretation—for example, when arguments come from user input and must not be interpreted. Use `sh()` or `shell()` when you need pipes, redirects, or compound commands.

```
let r = run_command("echo hello world")
say r.stdout
say "ok: {r.ok}"
```

Output:

```
hello world
ok: true
```

```
let r = run_command("ls -la /tmp")
if r.ok {
    say r.stdout
} otherwise {
    say "Failed: {r.stderr}"
}
```

```
let r = run_command("date +%Y-%m-%d")
say "Date: {r.stdout}"
```

> **No shell features.** Commands like `ls | head -5` or `cat file.txt` will not work as expected with `run_command()`—the pipe and redirect are passed literally as arguments. Use `shell()` for those cases.

3.12.13 Recipes

Recipe 20.1: System Health Checker Script

A one-stop script that gathers system info, disk usage, dependency checks, and service status.

```
say term.bold("=== System Health Check ===")
say ""

say term.blue("System Information:")
let user = sh("whoami")
let host = sh("hostname")
let os_name = sh("uname -s")
let arch = sh("uname -m")
say "  User:     {user}"
say "  Hostname: {host}"
say "  OS:       {os_name}"
say "  Arch:     {arch}"
say ""

say term.blue("Disk Usage:")
let disk = shell("df -h / | tail -1")
say "  {disk.stdout}"
say ""

say term.blue("Required Tools:")
let tools = ["git", "cargo", "curl"]
for tool in tools {
    let path = which(tool)
    if path != null {
        say "  {term.green(tool)}: {path}"
    } otherwise {
        say "  {term.red(tool)}: NOT FOUND"
    }
}
say ""

say term.blue("Service Status:")
if sh_ok("pgrep -q nginx") {
    say "  nginx: running"
} otherwise {
    say "  nginx: stopped"
}
say ""

term.success("Health check complete")
```

Recipe 20.2: Log File Analyzer

Read a log file, split by lines, filter for errors, and summarize.

```
// Simulate log file
let log_content = "2024-01-15 10:30:00 INFO Server started
2024-01-15 10:30:05 WARN High memory: 85%
2024-01-15 10:31:12 ERROR Connection refused: database
2024-01-15 10:32:00 INFO Recovery complete
2024-01-15 10:35:22 ERROR Timeout on /api/users"
fs.write("/tmp/app.log", log_content)

let all_lines = lines(fs.read("/tmp/app.log"))
let error_lines = filter(all_lines, fn(line) { return regex.test(line, "ERROR") })
let warn_lines = filter(all_lines, fn(line) { return regex.test(line, "WARN") })

say "Total lines: {len(all_lines)}"
say "Errors: {len(error_lines)}"
say "Warnings: {len(warn_lines)}"
say ""
say "Error lines:"
for line in error_lines {
    say "  {line}"
}

fs.remove("/tmp/app.log")
```

Output:

```
Total lines: 5
Errors: 2
Warnings: 1

Error lines:
  2024-01-15 10:31:12 ERROR Connection refused: database
  2024-01-15 10:35:22 ERROR Timeout on /api/users
```

Recipe 20.3: JSON API Tool Wrapper

Use `sh_json` with `curl` or `kubectl` to fetch and process JSON APIs.

```
// Example: fetch JSON from a public API
let url = "https://api.github.com/repos/rust-lang/rust"
let data = sh_json("curl -s " + url)

if data != null {
    say "Repository: {data.full_name}"
    say "Stars: {data.stargazers_count}"
    say "Open issues: {data.open_issues_count}"
```

```
} otherwise {
    say "Failed to fetch or parse API response"
}
```

```
// Example: kubectl get pods as JSON (when kubectl is configured)
let pods = sh_json("kubectl get pods -A -o json 2>/dev/null || echo '{}'")
if pods != null && pods.kind == "PodList" {
    let items = pods.items
    say "Pods: {len(items)}"
    for pod in items {
        let name = pod.metadata.name
        let status = pod.status.phase
        say "  {name}: {status}"
    }
} otherwise {
    say "kubectl not available or no pods"
}
```

Recipe 20.4: Data Pipeline — Forge □ Unix □ Forge

Build a pipeline: generate or load data in Forge, pipe it through Unix tools, and consume the result back in Forge.

```
// Generate CSV in Forge
let rows = [
    { name: "Charlie", score: 88 },
    { name: "Alice", score: 95 },
    { name: "Bob", score: 82 },
    { name: "Diana", score: 91 }
]
let csv = csv.stringify(rows)
say "Original data:"
say csv
say ""

// Pipe through sort (by second column, numeric)
let result = pipe_to(csv, "sort -t',' -k2 -rn")
let sorted_csv = result.stdout
say "After sort (by score descending):"
say sorted_csv
say ""

// Parse back into Forge and extract top scorer
let sorted_rows = csv.parse(sorted_csv)
let top = sorted_rows[0]
say "Top scorer: {top.name} with {top.score}"
```

```
// Another pipeline: filter log lines with grep, then count
let log_text = "INFO request 1\nERROR timeout\nINFO request 2\nERROR connection\nINFO request 3"
let err_result = pipe_to(log_text, "grep ERROR")
let err_lines = lines(err_result.stdout)
say "Error count: {len(err_lines)}"
```

3.13 Chapter 21: npc — Fake Data Generation

Need test data? Prototyping a UI? Building a seed script? The `npc` module generates realistic fake data without external dependencies. Every call returns different random data.

3.13.1 Function Reference

Function	Description	Example Output
`npc.name()`	Full name (first + last)	`"Jordan Patel"`
`npc.first_name()`	First name only	`"Luna"`
`npc.last_name()`	Last name only	`"Nakamura"`
`npc.email()`	Realistic email address	`"kai.chen42@gmail.com"`
`npc.username()`	Fun username	`"cosmic_dragon247"`
`npc.phone()`	US-format phone	`"(415) 867-5309"`
`npc.number(min,max)`	Random integer in range	`42`
`npc.pick(arr)`	Random item from array	(varies)
`npc.bool()`	Random true/false	`true`
`npc.sentence(n?)`	Random sentence (n words)	`"Code runs fast through."`
`npc.word()`	Single random word	`"algorithm"`
`npc.id()`	UUID-like identifier	`"a3f8k2m1-x9b2-..."`
`npc.color()`	Random hex color	`"#3a7bc4"`
`npc.ip()`	Random IPv4 address	`"192.168.42.7"`
`npc.url()`	Random HTTPS URL	`"https://techflow.io/api"`
`npc.company()`	Random company name	`"PixelForge"`

3.13.2 Core Examples

```
// Seed a database with 10 fake users
db.open(":memory:")
db.execute("CREATE TABLE users (name TEXT, email TEXT, role TEXT)")

let roles = ["admin", "user", "editor"]
repeat 10 times {
```

```
    db.execute("INSERT INTO users VALUES (?, ?, ?)", [
        npc.name(),
        npc.email(),
        npc.pick(roles)
    ])
}

let users = db.query("SELECT * FROM users")
for each u in users {
    say "{u.name} — {u.email} ({u.role})"
}
db.close()
```

```
// Generate test API payloads
let payload = {
    id: npc.id(),
    name: npc.name(),
    email: npc.email(),
    active: npc.bool(),
    score: npc.number(1, 100),
    color: npc.color()
}
say json.pretty(payload)
```

3.14 Chapter 22: String Transformations

Forge includes powerful string transformation builtins that go beyond basic split/join. All support method syntax (`str.function()`).

3.14.1 Function Reference

Function	Description	Example
`substring(s, start, end?)`	Extract by char indices	`substring("hello", 0, 3)` → `"hel"`
`index_of(s, substr)`	First index of substr, or -1	`index_of("abcabc", "bc")` → `1`
`last_index_of(s, substr)`	Last index of substr, or -1	`last_index_of("abcabc", "bc")` → `4`
`pad_start(s, len, char?)`	Left-pad (default space)	`pad_start("42", 5, "0")` → `"00042"`
`pad_end(s, len, char?)`	Right-pad (default space)	`pad_end("42", 5)` → `"42   "`

Function	Description	Example
`capitalize(s)`	Uppercase first, lowercase rest	`capitalize("hELLO")` ☐ `"Hello"`
`title(s)`	Title Case Each Word	`title("the quick fox")` ☐ `"The Quick Fox"`
`repeat_str(s, n)`	Repeat string n times	`repeat_str("-", 5)` ☐ `"-----"`
`count(s, substr)`	Count occurrences	`count("banana", "an")` ☐ `2`
`slugify(s)`	URL-friendly string	`slugify("Hello World!")` ☐ `"hello-world"`
`snake_case(s)`	Convert to snake_case	`snake_case("myAPIKey")` ☐ `"my_api_key"`
`camel_case(s)`	Convert to camelCase	`camel_case("hello_world")` ☐ `"helloWorld"`

3.14.2 Core Examples

```
// Format table columns with padding
let items = [
    { name: "Widget", price: 9.99 },
    { name: "Gadget", price: 24.50 },
    { name: "Doohickey", price: 149.99 }
]
for each item in items {
    say "{pad_end(item.name, 15)}{pad_start(str(item.price), 8)}"
}
```

```
// Convert API keys between naming conventions
let api_field = "userAccountStatus"
say snake_case(api_field)    // user_account_status
say slugify(api_field)       // useraccountstatus (URL-safe)

let db_column = "created_at"
say camel_case(db_column)    // createdAt
```

3.15 Chapter 23: Collection Power Tools

Beyond `map`, `filter`, and `reduce`, Forge offers a comprehensive collection toolkit. All functions support method syntax.

3.15.1 Function Reference

Function	Description	Return Type
`sum(arr)`	Sum of numeric array	Int or Float
`min_of(arr)`	Minimum value	Int or Float
`max_of(arr)`	Maximum value	Int or Float
`any(arr, fn)`	True if any element satisfies predicate	Bool
`all(arr, fn)`	True if all elements satisfy predicate	Bool
`unique(arr)`	Deduplicate, preserve order	Array
`zip(arr1, arr2)`	Pair elements into `[[a,b], ...]`	Array of pairs
`flatten(arr)`	Flatten one level of nesting	Array
`group_by(arr, fn)`	Group into object keyed by fn result	Object
`chunk(arr, size)`	Split into sized chunks	Array of arrays
`slice(arr, start, end?)`	Extract sub-array (supports negative idx)	Array
`partition(arr, fn)`	Split into `[matches, rest]`	Array of 2 arrays
`sort(arr, fn?)`	Sort with optional comparator (-1/0/1)	Array
`sample(arr, n?)`	Random n items (default 1)	Value or Array
`shuffle(arr)`	Randomize array order (Fisher-Yates)	Array
`diff(a, b)`	Deep object comparison	Object or null

3.15.2 Core Examples

```
let scores = [85, 92, 67, 78, 95, 88, 72, 91]
say "Total: {sum(scores)}"
say "Average: {sum(scores) / len(scores)}"
say "Best: {max_of(scores)}"
say "Worst: {min_of(scores)}"
say "All passing? {all(scores, fn(s) { return s >= 60 })}"

// Split into grade groups
let groups = group_by(scores, fn(s) {
    if s >= 90 { return "A" }
    if s >= 80 { return "B" }
    if s >= 70 { return "C" }
    return "D"
})
say "A students: {len(groups.A)}"
```

```
// Partition for batch processing
let users = [
    { name: "Alice", active: true },
    { name: "Bob", active: false },
    { name: "Charlie", active: true }
]
```

```
let parts = partition(users, fn(u) { return u.active })
say "Active: {len(parts[0])}"    // 2
say "Inactive: {len(parts[1])}"  // 1
```

```
// Diff for audit trails
let before = { name: "Alice", role: "user", email: "alice@test.com" }
let after = { name: "Alice", role: "admin", level: 5 }
let changes = diff(before, after)
// changes.role = { from: "user", to: "admin" }
// changes.email = { removed: "alice@test.com" }
// changes.level = { added: 5 }
```

3.16 Chapter 24: GenZ Debug Kit

Forge's most distinctive feature: debugging and assertions with personality. These builtins do the same job as traditional tools but with memorable names and expressive error messages that make debugging less painful and more fun.

3.16.1 Function Reference

Function	Traditional Equivalent	Behavior
`sus(val)`	`dbg!()` in Rust	Print inspect info to stderr, return value (pass-through)
`bruh(msg)`	`panic!()`	Crash with "BRUH: {msg}"
`bet(cond, msg?)`	`assert()`	Pass: silent. Fail: "LOST THE BET: {msg}"
`no_cap(a, b)`	`assert_eq()`	Pass: silent. Fail: "CAP DETECTED: a != b"
`ick(cond, msg?)`	`assert_false()`	Pass if false. Fail: "ICK: {msg}" if true

3.16.2 Core Examples

```
// sus() — sprinkle through code for quick debugging
let data = http.get("https://api.example.com/users")
let users = sus(data.body)  // prints inspect, keeps flowing
let count = sus(len(users)) // prints count, keeps flowing
```

```
// bet/no_cap/ick — assertions with attitude
define validate_user(user) {
    bet(has_key(user, "name"), "user needs a name, bestie")
    bet(user.age >= 0, "negative age is not a vibe")
    no_cap(typeof(user.email), "String")
    ick(user.role == "superadmin", "no one should be superadmin")
}
```

```
// yolo — when you just don't care about errors
let result = yolo(fn() {
    return http.get("https://maybe-down.com/api").body
})
// result is None if it failed, actual data if it worked
```

3.17 Chapter 25: Execution Helpers

Built-in performance profiling and resilient execution patterns — no external tools needed.

3.17.1 Function Reference

Function	Description	Return
`cook(fn)`	Time execution, print results with personality	fn's return
`yolo(fn)`	Execute, swallow ALL errors, return None on failure	Value or None
`ghost(fn)`	Execute silently, return result	fn's return
`slay(fn, n?)`	Benchmark n times (default 100), return stats object	Object

The `slay()` stats object contains:

Field	Type	Description
`avg_ms`	Float	Average execution time
`min_ms`	Float	Fastest run
`max_ms`	Float	Slowest run
`p99_ms`	Float	99th percentile
`runs`	Int	Number of iterations
`result`	Any	Return value of last run

3.17.2 Core Examples

```
// Profile a data processing pipeline
let processed = cook(fn() {
    let data = fs.read("large_file.csv")
    let rows = csv.parse(data)
    return filter(rows, fn(r) { return int(r.score) > 80 })
})
// Prints: "COOKED: done in 42.3ms — no cap that was fast"
```

```
// Compare two approaches
say "--- Approach A ---"
let stats_a = slay(fn() {
    return sort([5, 3, 1, 4, 2])
}, 1000)

say "--- Approach B ---"
let stats_b = slay(fn() {
    return sort([5, 3, 1, 4, 2], fn(a, b) {
        if a < b { return -1 }
        if a > b { return 1 }
        return 0
    })
}, 1000)
```

3.18 Chapter 26: Advanced Testing

Forge's test framework supports decorators, hooks, assertions, and structured error handling.

3.18.1 Testing Features

Feature	Description
`@test`	Mark function as test
`@skip`	Skip test (shown as SKIP in output)
`@before`	Run before each test in file
`@after`	Run after each test (even on failure)
`assert(cond, msg?)`	Basic assertion
`assert_eq(a, b)`	Assert equal
`assert_ne(a, b)`	Assert not equal
`assert_throws(fn)`	Assert function throws an error

Feature	Description
`--filter pattern`	Run only tests whose name contains pattern

3.18.2 Structured Error Objects

When catching errors with `try/catch`, the error object has two fields:

Field	Type	Description
`err.message`	String	The error message text
`err.type`	String	Error classification

Error types: `ArithmeticError`, `TypeError`, `ReferenceError`, `IndexError`, `AssertionError`, `RuntimeError`

3.18.3 Core Examples

```
let mut setup_count = 0

@before
define setup() {
    setup_count = setup_count + 1
}

@after
define cleanup() {
    // Runs after every test, even on failure
}

@test
define test_math() {
    assert_eq(1 + 1, 2)
    assert_ne(1, 2)
}

@test
define test_errors() {
    assert_throws(fn() { let x = 1 / 0 })

    try { let x = 1 / 0 } catch err {
        assert_eq(err.type, "ArithmeticError")
        assert(contains(err.message, "zero"))
    }
}
```

```
@test
@skip
define test_not_ready() {
    // This won't run, shown as SKIP
}
```

Run with filter: `forge test --filter "math"` — runs only tests with "math" in the name.

3.19 Chapter 27: math & fs Additions

3.19.1 New math Functions

Function	Description	Example
`math.random_int(min, max)`	Random integer in range	`math.random_int(1, 6)` ☐ `4`
`math.clamp(val, min, max)`	Clamp value to range	`math.clamp(150, 0, 100)` ☐ `100`

3.19.2 New fs Functions

Function	Description	Example
`fs.lines(path)`	Read as array of lines	`fs.lines("data.txt")` ☐ `["a", "b"]`
`fs.dirname(path)`	Parent directory	`fs.dirname("/home/user/f.txt")` ☐ `"/home/user"`
`fs.basename(path)`	Filename component	`fs.basename("/home/user/f.txt")` ☐ `"f.txt"`
`fs.join_path(a, b, ...)`	Join path segments	`fs.join_path("/home", "user")` ☐ `"/home/user"`
`fs.is_dir(path)`	Is directory?	`fs.is_dir("/tmp")` ☐ `true`
`fs.is_file(path)`	Is regular file?	`fs.is_file("main.fg")` ☐ `true`
`fs.temp_dir()`	System temp directory	`fs.temp_dir()` ☐ `"/tmp"`

3.19.3 CLI Argument Parsing (io module)

Function	Description	Example
`io.args_parse()`	Parse all CLI args	Returns `{ verbose: true, ... }`
`io.args_get(flag)`	Get single flag value	`io.args_get("--port")` ☐ `"8080"`

Function	Description	Example
`io.args_has(flag)`	Check if flag present	`io.args_has("--verbose")` ☐ `true`

3.19.4 Concurrency Additions

Function	Description	Return
`try_send(ch, val)`	Non-blocking channel send	Bool
`try_receive(ch)`	Non-blocking channel receive	Some(val) or None

This concludes Part II: The Standard Library. With sixteen modules, a GenZ debug kit, execution helpers, and 230+ functions at your disposal, Forge provides everything needed for file I/O, databases, data processing, HTTP, cryptography, terminal UI, fake data generation, performance profiling, shell scripting, and resilient error handling—all without leaving the language.

Chapter 4

Part III: Building Real Things

4.1 Chapter 28: Building REST APIs

Every modern application needs an API. Whether you're building a mobile backend, a microservice, or a simple webhook receiver, the ability to stand up an HTTP server quickly and cleanly is a superpower. Forge makes this trivially easy with its decorator-based routing model—no framework boilerplate, no configuration files, no dependency management. You write your handlers, attach decorators, and you have a production-ready API server backed by Rust's Axum and Tokio under the hood.

4.1.1 The Decorator-Based Routing Model

Forge's API server is activated by two things: the `@server` decorator that configures your server, and route decorators (`@get`, `@post`, `@put`, `@delete`, `@ws`) that bind functions to HTTP endpoints.

Here is the minimal shape of every Forge API:

```
@server(port: 8080)

@get("/hello")
fn hello() -> Json {
    return { message: "Hello from Forge!" }
}
```

The `@server(port: 8080)` decorator tells the Forge runtime to start an HTTP server on port 8080 after evaluating the file. The `@get("/hello")` decorator registers the `hello` function as a handler for `GET /hello`. The `-> Json` return type annotation tells the framework to serialize the return value as a JSON response with the appropriate `Content-Type` header.

These are not magic comments. They are first-class syntax elements parsed by the Forge compiler and used by the runtime to build an Axum router. CORS is enabled by default (permissive mode), so your API works out of the box with browser clients.

4.1.2 Route Parameters and Query Strings

Route parameters use the `:param` syntax familiar from Express and Sinatra. Any segment prefixed with a colon becomes a named parameter extracted from the URL path and passed to your handler function as a `String` argument:

```
@get("/users/:id")
fn get_user(id: String) -> Json {
    return { user_id: id }
}
```

A request to `GET /users/42` calls `get_user` with `id` set to `"42"`. You can have multiple route parameters:

```
@get("/repos/:owner/:name")
fn get_repo(owner: String, name: String) -> Json {
    return { owner: owner, repo: name }
}
```

Query strings are also available. Forge automatically parses query parameters and makes them accessible through the function's parameters. Parameters not matched by route segments are looked up in the query string.

4.1.3 Request Bodies (POST/PUT)

For `POST` and `PUT` routes, Forge automatically parses the JSON request body and passes it to your handler as a `body` parameter of type `Json`:

```
@post("/users")
fn create_user(body: Json) -> Json {
    let name = body.name
    return { created: true, name: name }
}
```

The `body` parameter is a Forge object parsed from the incoming JSON. You access its fields with dot notation, exactly like any other Forge object.

4.1.4 WebSocket Support

Forge supports WebSocket endpoints with the `@ws` decorator. A WebSocket handler receives each incoming text message as a string parameter and returns a response string:

```
@ws("/chat")
fn chat(message: String) -> Json {
    return { echo: message }
}
```

When a WebSocket client connects to `/chat` and sends a message, Forge calls your handler with the message text and sends back the return value. This makes it trivial to build real-time features.

4.1.5 Project 1: Hello API — Simple Greeting Service

Let's start with a complete, runnable API that demonstrates routing fundamentals.

```
// hello_api.fg — Your first Forge REST API
// Run:  forge run hello_api.fg
// Test: curl http://localhost:3000/hello/World

@server(port: 3000)

@get("/")
fn index() -> Json {
    return {
        name: "Hello API",
        version: "1.0.0",
        endpoints: ["/hello/:name", "/health", "/time"]
    }
}

@get("/hello/:name")
fn hello(name: String) -> Json {
    let greeting = "Hello, {name}!"
    return { greeting: greeting, language: "Forge" }
}

@get("/health")
fn health() -> Json {
    return { status: "ok" }
}

@get("/time")
fn time() -> Json {
    let now = sh("date -u +%Y-%m-%dT%H:%M:%SZ")
    return { utc: now }
}

@post("/echo")
fn echo(body: Json) -> Json {
    return body
}

@get("/add/:a/:b")
fn add(a: String, b: String) -> Json {
    let x = int(a)
```

```
    let y = int(b)
    let sum = x + y
    return { a: x, b: y, sum: sum }
}

say "Hello API starting on http://localhost:3000"
```

Save this as `hello_api.fg` and run it:

```
$ forge run hello_api.fg
Hello API starting on http://localhost:3000
Forge server listening on 0.0.0.0:3000
```

Now test each endpoint:

```
# Root endpoint — API discovery
$ curl -s http://localhost:3000/ | python3 -m json.tool
{
    "name": "Hello API",
    "version": "1.0.0",
    "endpoints": ["/hello/:name", "/health", "/time"]
}

# Greeting with route parameter
$ curl -s http://localhost:3000/hello/Forge
{"greeting":"Hello, Forge!","language":"Forge"}

# Health check
$ curl -s http://localhost:3000/health
{"status":"ok"}

# Arithmetic via route params
$ curl -s http://localhost:3000/add/17/25
{"a":17,"b":25,"sum":42}

# POST echo — sends back whatever you send
$ curl -s -X POST http://localhost:3000/echo \
  -H "Content-Type: application/json" \
  -d '{"message":"ping"}'
{"message":"ping"}
```

Walkthrough. The `@server(port: 3000)` line configures the port. Each `@get` or `@post` decorator binds a handler function to an HTTP method and path. Route parameters like `:name`, `:a`, and `:b` become function arguments. The `-> Json` return type tells Forge to serialize the returned object as JSON. The `say` statement at the bottom executes during startup, before the server begins accepting connections.

4.1.6 Project 2: Notes API — Full CRUD with SQLite

This project builds a complete note-taking API with persistent storage. It demonstrates all four CRUD operations, database integration, and error handling.

```
// notes_api.fg — Full CRUD REST API with SQLite
// Run:  forge run notes_api.fg
// Data persists in notes.db between restarts

@server(port: 3000)

// Initialize database on startup
db.open("notes.db")
db.execute("CREATE TABLE IF NOT EXISTS notes (id INTEGER PRIMARY KEY AUTOINCREMENT, title TEXT NO

// List all notes
@get("/notes")
fn list_notes() -> Json {
    let notes = db.query("SELECT * FROM notes ORDER BY created_at DESC")
    return { count: len(notes), notes: notes }
}

// Get a single note by ID
@get("/notes/:id")
fn get_note(id: String) -> Json {
    let query_str = "SELECT * FROM notes WHERE id = {id}"
    let results = db.query(query_str)
    if len(results) == 0 {
        return { error: "Note not found", id: id }
    }
    return results[0]
}

// Create a new note
@post("/notes")
fn create_note(body: Json) -> Json {
    let title = body.title
    let note_body = body.body
    if title == null {
        return { error: "title is required" }
    }
    if note_body == null {
        return { error: "body is required" }
    }
    let stmt = "INSERT INTO notes (title, body) VALUES ('{title}', '{note_body}')"
    db.execute(stmt)
    let results = db.query("SELECT * FROM notes ORDER BY id DESC LIMIT 1")
    return { created: true, note: results[0] }
```

```
}

// Update an existing note
@put("/notes/:id")
fn update_note(id: String, body: Json) -> Json {
    let check = db.query("SELECT * FROM notes WHERE id = {id}")
    if len(check) == 0 {
        return { error: "Note not found", id: id }
    }
    let title = body.title
    let note_body = body.body
    if title == null {
        return { error: "title is required" }
    }
    if note_body == null {
        return { error: "body is required" }
    }
    let stmt = "UPDATE notes SET title = '{title}', body = '{note_body}' WHERE id = {id}"
    db.execute(stmt)
    let results = db.query("SELECT * FROM notes WHERE id = {id}")
    return { updated: true, note: results[0] }
}

// Delete a note
@delete("/notes/:id")
fn delete_note(id: String) -> Json {
    let check = db.query("SELECT * FROM notes WHERE id = {id}")
    if len(check) == 0 {
        return { error: "Note not found", id: id }
    }
    db.execute("DELETE FROM notes WHERE id = {id}")
    return { deleted: true, id: id }
}

// API info
@get("/")
fn api_info() -> Json {
    return {
        name: "Notes API",
        version: "1.0.0",
        endpoints: [
            "GET    /notes       - List all notes",
            "GET    /notes/:id   - Get a note",
            "POST   /notes       - Create a note",
            "PUT    /notes/:id   - Update a note",
            "DELETE /notes/:id   - Delete a note"
        ]
```

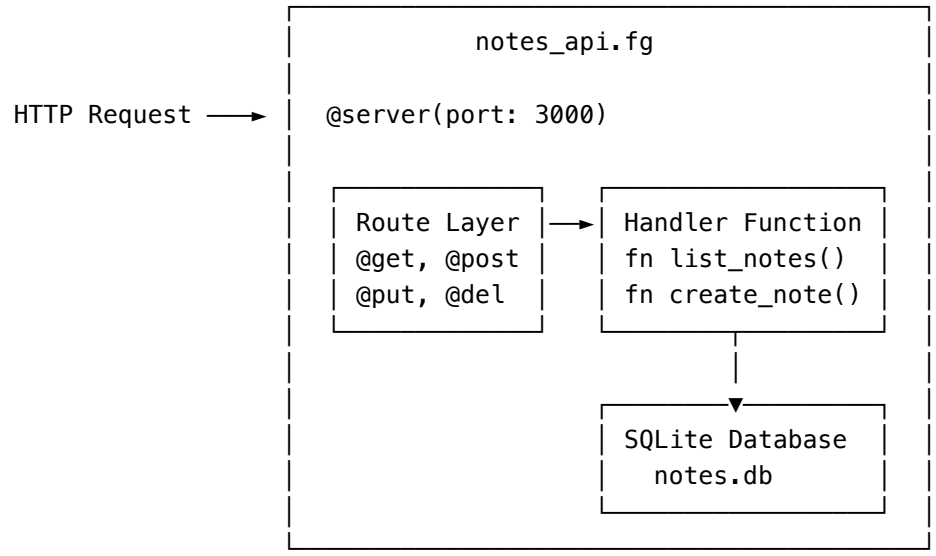
```
    }
}

say term.bold("Notes API")
say "Listening on http://localhost:3000"
say "Database: notes.db"
```

Test the complete CRUD lifecycle:

```
# Create a note
$ curl -s -X POST http://localhost:3000/notes \
  -H "Content-Type: application/json" \
  -d '{"title":"First Note","body":"Hello from Forge!"}' | python3 -m json.tool
{
    "created": true,
    "note": {
        "id": 1,
        "title": "First Note",
        "body": "Hello from Forge!",
        "created_at": "2026-02-28 12:00:00"
    }
}

# Create another note
$ curl -s -X POST http://localhost:3000/notes \
  -H "Content-Type: application/json" \
  -d '{"title":"Second Note","body":"Forge makes APIs easy."}'

# List all notes
$ curl -s http://localhost:3000/notes | python3 -m json.tool
{
    "count": 2,
    "notes": [
        {"id": 2, "title": "Second Note", "body": "Forge makes APIs easy.", ...},
        {"id": 1, "title": "First Note", "body": "Hello from Forge!", ...}
    ]
}

# Update a note
$ curl -s -X PUT http://localhost:3000/notes/1 \
  -H "Content-Type: application/json" \
  -d '{"title":"First Note (edited)","body":"Updated content."}'

# Delete a note
$ curl -s -X DELETE http://localhost:3000/notes/2
{"deleted":true,"id":"2"}
```

```
# Verify deletion
$ curl -s http://localhost:3000/notes
{"count":1,"notes":[...]}
```

Architecture of the Notes API:



4.1.7 Project 3: URL Shortener — Complete Service with Database

A URL shortener is a classic API project that combines database operations, string manipulation, and clean API design. This version generates short codes using a hash-based approach.

```
// shortener.fg — URL Shortener API
// Run:  forge run shortener.fg
// Test: curl -X POST http://localhost:3000/shorten -d '{"url":"https://example.com"}'

@server(port: 3000)

// Initialize database
db.open("urls.db")
db.execute("CREATE TABLE IF NOT EXISTS urls (code TEXT PRIMARY KEY, original TEXT NOT NULL, click

// Generate a short code from a URL
fn make_code(url) {
    let hash = crypto.sha256(url)
    let code = slice(hash, 0, 7)
    return code
}

// Shorten a URL
```

```
@post("/shorten")
fn shorten(body: Json) -> Json {
    let url = body.url
    if url == null {
        return { error: "url is required" }
    }
    let code = make_code(url)
    let existing = db.query("SELECT * FROM urls WHERE code = '{code}'")
    if len(existing) > 0 {
        return { code: code, short_url: "http://localhost:3000/r/{code}", existed: true }
    }
    db.execute("INSERT INTO urls (code, original) VALUES ('{code}', '{url}')")
    return { code: code, short_url: "http://localhost:3000/r/{code}", created: true }
}

// Redirect (returns the original URL — client follows it)
@get("/r/:code")
fn redirect(code: String) -> Json {
    let results = db.query("SELECT * FROM urls WHERE code = '{code}'")
    if len(results) == 0 {
        return { error: "Short URL not found" }
    }
    db.execute("UPDATE urls SET clicks = clicks + 1 WHERE code = '{code}'")
    let original = results[0].original
    return { redirect: original }
}

// Stats for a short URL
@get("/stats/:code")
fn stats(code: String) -> Json {
    let results = db.query("SELECT * FROM urls WHERE code = '{code}'")
    if len(results) == 0 {
        return { error: "Short URL not found" }
    }
    return results[0]
}

// List all shortened URLs
@get("/urls")
fn list_urls() -> Json {
    let urls = db.query("SELECT * FROM urls ORDER BY created_at DESC")
    return { count: len(urls), urls: urls }
}

// Landing page
@get("/")
fn landing() -> Json {
```

```
    return {
        service: "Forge URL Shortener",
        usage: "POST /shorten with {url: 'https://...'}",
        endpoints: [
            "POST /shorten     - Create short URL",
            "GET  /r/:code     - Resolve short URL",
            "GET  /stats/:code - View click stats",
            "GET  /urls        - List all URLs"
        ]
    }
}

say term.bold("Forge URL Shortener")
say "Running on http://localhost:3000"
```

Test the shortener end-to-end:

```
# Shorten a URL
$ curl -s -X POST http://localhost:3000/shorten \
  -H "Content-Type: application/json" \
  -d '{"url":"https://github.com/forge-lang/forge"}' | python3 -m json.tool
{
    "code": "a1b2c3d",
    "short_url": "http://localhost:3000/r/a1b2c3d",
    "created": true
}

# Resolve the short URL
$ curl -s http://localhost:3000/r/a1b2c3d
{"redirect":"https://github.com/forge-lang/forge"}

# Check click stats
$ curl -s http://localhost:3000/stats/a1b2c3d
{"code":"a1b2c3d","original":"https://github.com/forge-lang/forge","clicks":1,...}

# List all URLs
$ curl -s http://localhost:3000/urls
{"count":1,"urls":[...]}
```

4.1.8 Error Handling in API Routes

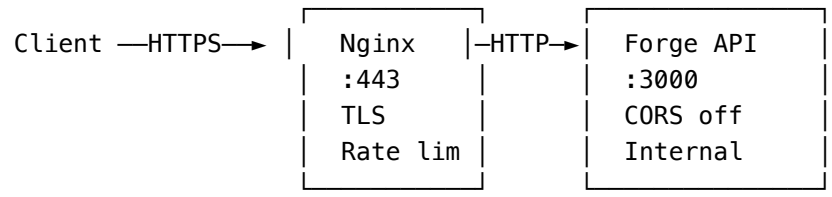
Forge API handlers return JSON objects. To signal errors, return an object with an `error` field. While Forge does not yet support setting custom HTTP status codes from handlers (the runtime always returns 200), you can structure your responses to distinguish success from failure:

```
@get("/users/:id")
fn get_user(id: String) -> Json {
    let results = db.query("SELECT * FROM users WHERE id = {id}")
    if len(results) == 0 {
        return { error: "not_found", message: "No user with that ID" }
    }
    return { ok: true, user: results[0] }
}
```

Client code can check for the `error` field to determine whether the request succeeded.

4.1.9 CORS and Production Considerations

Forge enables permissive CORS by default—all origins, methods, and headers are allowed. This is ideal for development but should be locked down in production. For production deployments, consider placing your Forge API behind a reverse proxy like Nginx or Caddy that handles TLS termination, rate limiting, and CORS policies.



The server binds to `0.0.0.0` by default, accepting connections on all interfaces. To bind to localhost only, pass the `host` argument:

```
@server(port: 3000, host: "127.0.0.1")
```

4.1.10 Going Further

- **Middleware patterns.** Use helper functions called at the top of handlers to validate authentication tokens, check rate limits, or log requests before processing.
- **Database migrations.** Run `CREATE TABLE IF NOT EXISTS` statements at startup as shown in the Notes API. For schema changes, add `ALTER TABLE` statements guarded by version checks.
- **API versioning.** Use path prefixes like `/v1/notes` and `/v2/notes` with separate handler functions.
- **WebSocket chat.** Combine the `@ws` decorator with database-backed message history to build a real-time chat application.

4.2 Chapter 29: HTTP Client and Web Automation

Building APIs is only half the story. Modern applications also *consume* APIs—pulling data from GitHub, checking service health, downloading files, and scraping web content. Forge's HTTP client capabilities turn these tasks into one-liners. Where other languages require installing libraries, importing modules, and managing async runtimes, Forge gives you `fetch()` and the `http` module as built-in primitives.

4.2.1 fetch() Basics

The `fetch()` function is Forge's Swiss Army knife for HTTP requests. At its simplest, it takes a URL and returns a response object:

```
let resp = fetch("https://api.github.com/zen")
say resp.body
```

The response object contains these fields:

Field	Type	Description
`status`	Integer	HTTP status code (200, 404, etc.)
`ok`	Boolean	`true` if status is 2xx
`body`	String	Raw response body
`json`	Object	Parsed JSON (if applicable)

For POST requests, pass an options object as the second argument:

```
let resp = fetch("https://httpbin.org/post", {
    method: "POST",
    body: { name: "Forge", type: "language" }
})
let status = resp.status
say "Status: {status}"
```

4.2.2 The http Module

The `http` module provides named methods for each HTTP verb, offering a cleaner syntax when you don't need the full flexibility of `fetch()`:

```
let resp = http.get("https://api.github.com/repos/rust-lang/rust")
let resp = http.post("https://httpbin.org/post", { key: "value" })
let resp = http.put("https://httpbin.org/put", { updated: true })
let resp = http.delete("https://httpbin.org/delete")
```

Each returns the same response object structure as `fetch()`.

4.2.3 Working with API Responses

API responses typically contain JSON. Access nested fields with dot notation:

```
let resp = fetch("https://api.github.com/repos/rust-lang/rust")
let name = resp.json.full_name
let stars = resp.json.stargazers_count
say "Repo: {name}"
say "Stars: {stars}"
```

For APIs that return arrays, iterate with `for`:

```
let resp = fetch("https://api.github.com/users/torvalds/repos?per_page=5")
for repo in resp.json {
    let name = repo.name
    say "  - {name}"
}
```

4.2.4 download and crawl

Forge provides two high-level keywords for common web tasks.

download saves a remote file to disk:

```
download "https://example.com/data.csv" to "local_data.csv"
```

Or using the `http` module:

```
http.download("https://example.com/data.csv", "local_data.csv")
```

crawl fetches a web page and returns its HTML content as a string, suitable for parsing and extraction:

```
let html = crawl "https://example.com"
say html
```

4.2.5 Project 1: API Consumer — GitHub Repository Dashboard

This program fetches a user's GitHub repositories and displays them as a formatted terminal table with color-coded statistics.

```
// github_repos.fg — GitHub repository dashboard
// Run: forge run github_repos.fg
// Note: Uses the public GitHub API (no token needed for public repos)
```

```
say term.banner("GitHub Repository Dashboard")
say ""

let username = "torvalds"
let url = "https://api.github.com/users/{username}/repos?per_page=10&sort=updated"
say term.blue("Fetching repos for @{username}...")
say ""

let resp = fetch(url)

if resp.ok == false {
    let status = resp.status
    say term.error("Failed to fetch: HTTP {status}")
} else {
    let repos = resp.json

    // Build table data
    let mut rows = []
    for repo in repos {
        let name = repo.name
        let stars = repo.stargazers_count
        let forks = repo.forks_count
        let lang = repo.language
        if lang == null {
            lang = "N/A"
        }
        let row = { Name: name, Language: lang, Stars: stars, Forks: forks }
        rows = append(rows, row)
    }

    term.table(rows)
    say ""

    // Summary statistics
    let star_counts = map(repos, fn(r) { return r.stargazers_count })
    let total_stars = reduce(star_counts, 0, fn(acc, x) { return acc + x })
    let repo_count = len(repos)
    say term.bold("Summary:")
    say "  Repositories shown: {repo_count}"
    say "  Total stars: {total_stars}"
    say ""

    // Star distribution sparkline
    say term.blue("Star distribution:")
    term.sparkline(star_counts)
    say ""
```

```
    // Bar chart of top repos by stars
    say term.blue("Top repos by stars:")
    let sorted_repos = sort(repos, fn(a, b) { return b.stargazers_count - a.stargazers_count })
    let mut i = 0
    let limit = math.min(5, len(sorted_repos))
    while i < limit {
        let repo = sorted_repos[i]
        let rname = repo.name
        let rstars = repo.stargazers_count
        term.bar(rname, float(rstars), float(total_stars))
        i = i + 1
    }
}

say ""
term.success("Dashboard complete!")
```

Expected output:

```
╔══════════════════════════════════════╗
║    GitHub Repository Dashboard       ║
╚══════════════════════════════════════╝

Fetching repos for @torvalds...

┌──────────────┬──────────┬───────┬───────┐
│ Name         │ Language │ Stars │ Forks │
├──────────────┼──────────┼───────┼───────┤
│ linux        │ C        │ 18000 │ 52000 │
│ subsurface   │ C++      │  2500 │   980 │
│ ...          │ ...      │   ... │   ... │
└──────────────┴──────────┴───────┴───────┘

Summary:
  Repositories shown: 10
  Total stars: 21342

Star distribution:
▁▁▁▁▁▁▁█▁▁

✓ Dashboard complete!
```

4.2.6 Project 2: Health Monitor — Multi-URL Status Checker

This tool checks the availability of multiple services and generates a color-coded status report, useful for monitoring dashboards or on-call scripts.

```
// health_monitor.fg — Service health checker
// Run: forge run health_monitor.fg

say term.banner("Service Health Monitor")
say ""

let services = [
    { name: "GitHub",      url: "https://api.github.com" },
    { name: "Google",      url: "https://www.google.com" },
    { name: "HTTPBin",     url: "https://httpbin.org/get" },
    { name: "Example.com", url: "https://example.com" },
    { name: "BadURL",      url: "https://this-does-not-exist.invalid" }
]

let mut results = []
let mut up_count = 0
let mut down_count = 0

for service in services {
    let name = service.name
    let url = service.url
    say "  Checking {name}..."
    let resp = fetch(url)
    let status = resp.status
    if resp.ok {
        let entry = { Service: name, Status: status, Result: "UP" }
        results = append(results, entry)
        up_count = up_count + 1
    } else {
        let entry = { Service: name, Status: status, Result: "DOWN" }
        results = append(results, entry)
        down_count = down_count + 1
    }
}

say ""
say term.bold("Results:")
term.table(results)
say ""

say term.green("  Up:   {up_count}")
say term.red("  Down: {down_count}")
say ""

let total = len(services)
if down_count == 0 {
    term.success("All {total} services are healthy!")
```

```
} else {
    term.warning("{down_count} of {total} services are down.")
}
```

Expected output:

```
╔══════════════════════════════════════╗
║      Service Health Monitor          ║
╚══════════════════════════════════════╝

  Checking GitHub...
  Checking Google...
  Checking HTTPBin...
  Checking Example.com...
  Checking BadURL...

Results:
┌──────────────┬────────┬────────┐
│ Service      │ Status │ Result │
├──────────────┼────────┼────────┤
│ GitHub       │ 200    │ UP     │
│ Google       │ 200    │ UP     │
│ HTTPBin      │ 200    │ UP     │
│ Example.com  │ 200    │ UP     │
│ BadURL       │ 0      │ DOWN   │
└──────────────┴────────┴────────┘

  Up:   4
  Down: 1

⚠ 1 of 5 services are down.
```

4.2.7 Project 3: Web Scraper — Crawl and Extract

This project uses Forge's `crawl` keyword to fetch web pages and extract useful information using string processing.

```
// scraper.fg — Simple web scraper
// Run: forge run scraper.fg

say term.banner("Forge Web Scraper")
say ""

let url = "https://example.com"
say "Crawling {url}..."
let html = crawl url
```

```
say ""

// Extract the page title
let title_start = "title>"
let title_end = "</title"
let parts = split(html, title_start)
if len(parts) > 1 {
    let after_tag = parts[1]
    let title_parts = split(after_tag, title_end)
    let title = title_parts[0]
    say term.bold("Page Title:")
    say "  {title}"
    say ""
}

// Count occurrences of key HTML elements
let html_lower = lowercase(html)
let p_tags = split(html_lower, "<p")
let a_tags = split(html_lower, "<a ")
let div_tags = split(html_lower, "<div")
let p_count = len(p_tags) - 1
let a_count = len(a_tags) - 1
let div_count = len(div_tags) - 1

say term.bold("Element Counts:")
say "  <p>   tags: {p_count}"
say "  <a>   tags: {a_count}"
say "  <div> tags: {div_count}"
say ""

// Page size stats
let page_size = len(html)
say term.bold("Page Statistics:")
say "  Total HTML size: {page_size} characters"
let line_list = split(html, "\n")
let line_count = len(line_list)
say "  Line count: {line_count}"
say ""

// Save to file
let filename = "scraped_output.html"
fs.write(filename, html)
say term.green("Saved HTML to {filename}")
say ""

term.success("Scraping complete!")
```

4.2.8 Going Further

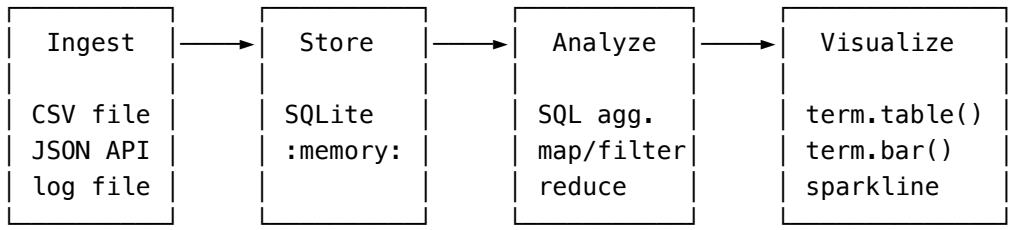
- **Authenticated requests.** Pass headers to `fetch()` for APIs that require Bearer tokens or API keys.
- **Pagination.** Use a `while` loop to follow `next` links in paginated API responses.
- **Retry logic.** Wrap `fetch()` calls in Forge's `retry 3 times { }` block for resilient HTTP clients.
- **Parallel fetching.** Use `forge` (async) functions with `hold` (await) to fetch multiple URLs concurrently.

4.3 Chapter 30: Data Processing Pipelines

Data processing is the bread and butter of practical programming. You receive data in one format, transform it, analyze it, and present the results. Forge excels at this workflow because it combines first-class JSON, a built-in SQLite database, CSV parsing, functional data transformations, and terminal visualization into a single, cohesive toolkit. No imports. No dependencies. Just data in, insight out.

4.3.1 The CSV ☐ Database ☐ Analysis ☐ Visualization Pattern

The canonical Forge data pipeline follows four stages:



Each stage uses built-in Forge primitives. No third-party libraries, no setup.

4.3.2 Functional Data Transformation Chains

Forge's `map`, `filter`, and `reduce` functions form the backbone of data transformations. They compose naturally:

```
let data = [10, 25, 3, 47, 8, 31, 15]

// Filter → Map → Reduce pipeline
let big = filter(data, fn(x) { return x > 10 })
let doubled = map(big, fn(x) { return x * 2 })
```

```
let total = reduce(doubled, 0, fn(acc, x) { return acc + x })

say "Values > 10, doubled, summed: {total}"
```

For database query results (arrays of objects), you can extract and transform specific fields:

```
let rows = db.query("SELECT product, price FROM items")
let prices = map(rows, fn(r) { return float(r.price) })
let avg = reduce(prices, 0.0, fn(a, x) { return a + x }) / len(prices)
```

4.3.3 Project 1: Sales Analytics — CSV Import, SQL Aggregation, Terminal Charts

This complete program imports sales data, loads it into an in-memory SQLite database, runs aggregation queries, and produces a full terminal dashboard with tables, bar charts, and sparklines.

```
// sales_analytics.fg — Full sales data pipeline
// Run: forge run sales_analytics.fg
// Note: Creates sample data if sales.csv doesn't exist

say term.banner("Sales Analytics Dashboard")
say ""

// Step 1: Generate sample data if needed
let csv_file = "sales.csv"
if fs.exists(csv_file) == false {
    say term.blue("Generating sample sales data...")
    let sample_csv = "date,product,quantity,unit_price,region
2026-01-05,Widget,10,29.99,North
2026-01-08,Gadget,5,49.99,South
2026-01-12,Widget,8,29.99,East
2026-01-15,Gizmo,3,99.99,North
2026-01-18,Gadget,12,49.99,West
2026-01-22,Widget,15,29.99,South
2026-01-25,Gizmo,7,99.99,East
2026-02-01,Gadget,9,49.99,North
2026-02-05,Widget,20,29.99,West
2026-02-08,Gizmo,4,99.99,South
2026-02-12,Gadget,6,49.99,East
2026-02-15,Widget,11,29.99,North
2026-02-18,Gizmo,8,99.99,West
2026-02-22,Widget,14,29.99,South
2026-02-25,Gadget,10,49.99,North"
    fs.write(csv_file, sample_csv)
    say "  Created {csv_file} with 15 records"
    say ""
```

```
}

// Step 2: Read CSV and load into database
say term.blue("Loading data into database...")
let raw = fs.read(csv_file)
let records = csv.parse(raw)
say "  Parsed {len(records)} records from CSV"

db.open(":memory:")
db.execute("CREATE TABLE sales (id INTEGER PRIMARY KEY AUTOINCREMENT, date TEXT, product TEXT, qu

for row in records {
    let d = row.date
    let p = row.product
    let q = row.quantity
    let u = row.unit_price
    let r = row.region
    let stmt = "INSERT INTO sales (date, product, quantity, unit_price, region) VALUES ('{d}', '{
    db.execute(stmt)
}
say "  Loaded into SQLite"
say ""

// Step 3: Analysis
say term.bold("═══ Sales by Product ═══")
let by_product = db.query("SELECT product, SUM(quantity) as total_qty, ROUND(SUM(quantity * unit_
term.table(by_product)
say ""

say term.bold("═══ Sales by Region ═══")
let by_region = db.query("SELECT region, COUNT(*) as transactions, SUM(quantity) as total_qty, RO
term.table(by_region)
say ""

say term.bold("═══ Monthly Trend ═══")
let by_month = db.query("SELECT substr(date, 1, 7) as month, SUM(quantity) as units, ROUND(SUM(qu
term.table(by_month)
say ""

// Step 4: Visualize
say term.bold("═══ Revenue by Product (Bar Chart) ═══")
let revenues = map(by_product, fn(r) { return float(r.revenue) })
let max_revenue = reduce(revenues, 0.0, fn(a, x) { if x > a { return x } else { return a } })
for row in by_product {
    let pname = row.product
    let rev = float(row.revenue)
    term.bar(pname, rev, max_revenue)
```

```
}
say ""

say term.bold("═══ Revenue by Region (Bar Chart) ═══")
let region_revs = map(by_region, fn(r) { return float(r.revenue) })
let max_region = reduce(region_revs, 0.0, fn(a, x) { if x > a { return x } else { return a } })
for row in by_region {
    let rname = row.region
    let rev = float(row.revenue)
    term.bar(rname, rev, max_region)
}
say ""

// Step 5: Summary statistics
let all_sales = db.query("SELECT quantity * unit_price as total FROM sales")
let all_amounts = map(all_sales, fn(r) { return float(r.total) })
let grand_total = reduce(all_amounts, 0.0, fn(a, x) { return a + x })
let sale_count = len(all_amounts)
let avg_sale = grand_total / sale_count

say term.bold("═══ Summary ═══")
say "  Total revenue:      ${grand_total}"
say "  Total transactions: {sale_count}"
say "  Average sale value: ${avg_sale}"
say ""

// Step 6: Revenue trend sparkline
say term.bold("═══ Transaction Values ═══")
term.sparkline(all_amounts)
say ""

db.close()
term.success("Analysis complete!")
```

Expected output:

```
╔═══════════════════════════════════════╗
║     Sales Analytics Dashboard         ║
╚═══════════════════════════════════════╝

Loading data into database...
  Parsed 15 records from CSV
  Loaded into SQLite

═══ Sales by Product ═══
┌──────────┬─────────────┬────────────┐
│ product  │ total_qty   │ revenue    │
```

```
├──────────┼──────────────┼────────────┤
│ Gizmo    │ 22           │ 2199.78    │
│ Gadget   │ 42           │ 2099.58    │
│ Widget   │ 78           │ 2339.22    │
└──────────┴──────────────┴────────────┘

═══ Revenue by Product (Bar Chart) ═══
Widget  ████████████████████████████ $2339.22
Gizmo   ██████████████████████████   $2199.78
Gadget  ████████████████████████     $2099.58

═══ Summary ═══
  Total revenue:      $6638.58
  Total transactions: 15
  Average sale value: $442.572

✓ Analysis complete!
```

4.3.4 Project 2: Log Analyzer — Parse, Extract, Report

This program reads a log file, parses entries by severity level, and generates a summary report. It demonstrates string processing, pattern counting, and structured output.

```
// log_analyzer.fg — Parse log files and generate reports
// Run: forge run log_analyzer.fg

say term.banner("Log File Analyzer")
say ""

// Generate a sample log file
let log_file = "app.log"
if fs.exists(log_file) == false {
    say term.blue("Generating sample log file...")
    let log_content = "2026-02-28 08:00:01 INFO  Server started on port 3000
2026-02-28 08:00:02 INFO  Database connection established
2026-02-28 08:01:15 INFO  GET /api/users 200 45ms
2026-02-28 08:01:16 DEBUG Query executed: SELECT * FROM users
2026-02-28 08:02:30 WARN  Slow query detected: 850ms
2026-02-28 08:03:00 INFO  POST /api/users 201 120ms
2026-02-28 08:03:45 ERROR Failed to send email: connection timeout
2026-02-28 08:04:10 INFO  GET /api/users/5 200 30ms
2026-02-28 08:05:00 WARN  Rate limit approaching for IP 192.168.1.100
2026-02-28 08:05:30 INFO  PUT /api/users/5 200 65ms
2026-02-28 08:06:00 DEBUG Cache miss for key: user:5
2026-02-28 08:06:15 ERROR Database connection lost: retry in 5s
2026-02-28 08:06:20 INFO  Database reconnected
2026-02-28 08:07:00 INFO  GET /api/health 200 5ms
```

```
2026-02-28 08:08:00 WARN  Memory usage at 85%
2026-02-28 08:09:00 INFO  DELETE /api/users/3 200 40ms
2026-02-28 08:10:00 ERROR Unhandled exception in /api/reports
2026-02-28 08:10:01 INFO  Error recovery complete
2026-02-28 08:11:00 INFO  GET /api/users 200 50ms
2026-02-28 08:12:00 DEBUG GC pause: 12ms"
    fs.write(log_file, log_content)
    say "  Created {log_file}"
    say ""
}

// Read and parse the log file
let content = fs.read(log_file)
let lines = split(content, "\n")
let total_lines = len(lines)

say term.blue("Parsing {total_lines} log entries...")
say ""

// Count by severity level
let mut info_count = 0
let mut warn_count = 0
let mut error_count = 0
let mut debug_count = 0
let mut errors = []
let mut warnings = []

for line in lines {
    if len(line) == 0 {
        // skip empty lines
    } else if contains(line, "ERROR") {
        error_count = error_count + 1
        errors = append(errors, line)
    } else if contains(line, "WARN") {
        warn_count = warn_count + 1
        warnings = append(warnings, line)
    } else if contains(line, "DEBUG") {
        debug_count = debug_count + 1
    } else if contains(line, "INFO") {
        info_count = info_count + 1
    }
}

// Display severity breakdown
say term.bold("═══ Severity Breakdown ═══")
let severity_data = [
    { Level: "INFO",  Count: info_count },
```

```
    { Level: "WARN",  Count: warn_count },
    { Level: "ERROR", Count: error_count },
    { Level: "DEBUG", Count: debug_count }
]
term.table(severity_data)
say ""

// Bar chart
say term.bold("═══ Log Level Distribution ═══")
let max_count = float(math.max(math.max(info_count, warn_count), math.max(error_count, debug_coun
term.bar("INFO ", float(info_count), max_count)
term.bar("WARN ", float(warn_count), max_count)
term.bar("ERROR", float(error_count), max_count)
term.bar("DEBUG", float(debug_count), max_count)
say ""

// Show errors and warnings
if error_count > 0 {
    say term.red(term.bold("═══ Errors ═══"))
    for err in errors {
        say term.red("  {err}")
    }
    say ""
}

if warn_count > 0 {
    say term.yellow(term.bold("═══ Warnings ═══"))
    for w in warnings {
        say term.yellow("  {w}")
    }
    say ""
}

// HTTP endpoint analysis
say term.bold("═══ HTTP Requests ═══")
let mut get_count = 0
let mut post_count = 0
let mut put_count = 0
let mut delete_count = 0

for line in lines {
    if contains(line, "GET /") {
        get_count = get_count + 1
    } else if contains(line, "POST /") {
        post_count = post_count + 1
    } else if contains(line, "PUT /") {
        put_count = put_count + 1
```

```
    } else if contains(line, "DELETE /") {
        delete_count = delete_count + 1
    }
}

let http_data = [
    { Method: "GET",    Requests: get_count },
    { Method: "POST",   Requests: post_count },
    { Method: "PUT",    Requests: put_count },
    { Method: "DELETE", Requests: delete_count }
]
term.table(http_data)
say ""

// Summary
say term.bold("═══ Summary ═══")
say "  Total entries:  {total_lines}"
say "  Error rate:     {error_count}/{total_lines}"
let total_http = get_count + post_count + put_count + delete_count
say "  HTTP requests:  {total_http}"
say ""

if error_count > 0 {
    term.warning("Log analysis found {error_count} error(s). Review above.")
} else {
    term.success("No errors found in log.")
}
```

4.3.5 Project 3: Data Converter — JSON to CSV to Database Roundtrip

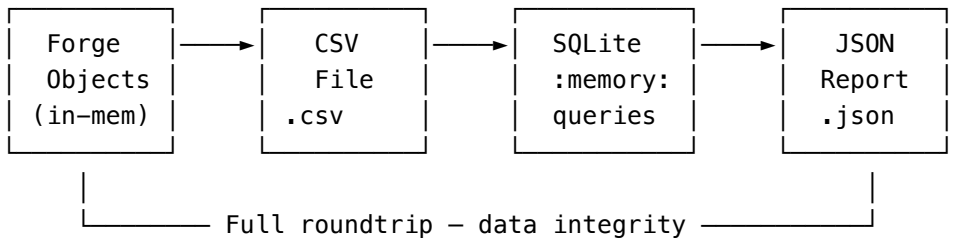
This project demonstrates converting data between formats—a common task in data engineering. It takes JSON data, writes it to CSV, reads the CSV back, loads it into a database, queries it, and exports the results as JSON.

```
// converter.fg — Format conversion pipeline: JSON → CSV → SQLite → JSON
// Run: forge run converter.fg

say term.banner("Data Format Converter")
say ""

// Step 1: Start with JSON data
say term.blue("Step 1: Create JSON dataset")
let employees = [
    { name: "Alice Chen",    department: "Engineering", salary: 125000, years: 5 },
    { name: "Bob Martinez",  department: "Marketing",   salary: 95000,  years: 3 },
    { name: "Carol Kim",     department: "Engineering", salary: 135000, years: 7 },
```

```
    { name: "David Johnson", department: "Sales",       salary: 88000,  years: 2 },
    { name: "Eva Schmidt",   department: "Engineering", salary: 142000, years: 9 },
    { name: "Frank Brown",   department: "Marketing",   salary: 102000, years: 4 },
    { name: "Grace Liu",     department: "Sales",       salary: 91000,  years: 3 },
    { name: "Henry Wilson",  department: "Engineering", salary: 118000, years: 4 }
]
let emp_count = len(employees)
say "  Created {emp_count} employee records"
say ""

// Step 2: Export to CSV
say term.blue("Step 2: Export to CSV")
let csv_file = "employees.csv"
csv.write(csv_file, employees)
say "  Written to {csv_file}"
let file_size = fs.size(csv_file)
say "  File size: {file_size} bytes"
say ""

// Step 3: Read CSV back
say term.blue("Step 3: Read CSV back")
let csv_content = fs.read(csv_file)
let parsed = csv.parse(csv_content)
say "  Parsed {len(parsed)} records from CSV"
say ""

// Step 4: Load into SQLite
say term.blue("Step 4: Load into SQLite")
db.open(":memory:")
db.execute("CREATE TABLE employees (name TEXT, department TEXT, salary REAL, years INTEGER)")
for emp in parsed {
    let n = emp.name
    let d = emp.department
    let s = emp.salary
    let y = emp.years
    db.execute("INSERT INTO employees VALUES ('{n}', '{d}', {s}, {y})")
}
say "  Loaded into in-memory database"
say ""

// Step 5: Run analytics queries
say term.bold("═══ All Employees ═══")
let all = db.query("SELECT * FROM employees ORDER BY salary DESC")
term.table(all)
say ""

say term.bold("═══ Department Summary ═══")
```

```
let dept_summary = db.query("SELECT department, COUNT(*) as headcount, ROUND(AVG(salary), 0) as a
term.table(dept_summary)
say ""

say term.bold("═══ Salary Distribution ═══")
let salaries = map(all, fn(r) { return float(r.salary) })
let max_salary = reduce(salaries, 0.0, fn(a, x) { if x > a { return x } else { return a } })
for row in all {
    let ename = row.name
    let esal = float(row.salary)
    term.bar(ename, esal, max_salary)
}
say ""

// Step 6: Export results as JSON
say term.blue("Step 6: Export analysis as JSON")
let report = {
    generated: sh("date -u +%Y-%m-%dT%H:%M:%SZ"),
    total_employees: len(all),
    departments: dept_summary,
    employees: all
}
let report_json = json.pretty(report)
fs.write("employee_report.json", report_json)
say "  Written to employee_report.json"
say ""

db.close()

// Clean up
fs.remove(csv_file)
say term.green("  Cleaned up temporary CSV")
say ""

term.success("Conversion pipeline complete!")
```

Data flow:



4.3.6 Going Further

- **Streaming large files.** For very large CSV files, process them in chunks by reading line-by-line rather than loading the entire file into memory.
- **Scheduled pipelines.** Combine data processing with Forge's `schedule every 1 hour` block to run pipelines on a recurring basis.
- **Multi-source joins.** Load data from multiple CSV files into separate database tables and use SQL JOINs for cross-source analysis.
- **Export formats.** Generate HTML reports by building template strings, or pipe JSON output to downstream services via `http.post()`.

4.4 Chapter 31: DevOps and System Automation

System administrators and DevOps engineers spend their days automating repetitive tasks: checking system health, deploying applications, rotating backups, validating configurations. Traditional tools for this—Bash scripts, Python scripts, Ansible playbooks—each have tradeoffs. Bash is powerful but cryptic. Python requires virtual environments. Ansible requires YAML fluency.

Forge occupies a sweet spot: it has shell integration as a first-class primitive, a real programming language's control flow and data structures, and built-in file system, JSON, and terminal formatting—all in a single binary with zero dependencies.

4.4.1 The Complete Shell Toolkit

Forge provides 11 shell-related functions that cover every common DevOps task:

Function	Returns	Description
`sh(cmd)`	String	Run command, return trimmed stdout
`shell(cmd)`	Object	Full result: stdout, stderr, status, ok
`sh_lines(cmd)`	Array	Run command, return output as array of lines
`sh_json(cmd)`	Any	Run command, parse stdout as JSON
`sh_ok(cmd)`	Boolean	Run command, return true if exit code is 0
`which(name)`	String/null	Find executable path (e.g. `which("docker")`)
`cwd()`	String	Current working directory
`cd(path)`	String	Change directory, return path
`lines(s)`	Array	Split string into array of lines
`pipe_to(data, cmd)`	Object	Pipe string into shell command (stdin)
`run_command(cmd)`	Object	Run command (split by whitespace, no shell)

Use `sh()` and `shell()` for basic execution. Use `sh_lines()` to parse process listings or command output line-by-line. Use `sh_json()` when a command emits JSON (e.g. `docker inspect`, `kubectl get -o json`). Use `sh_ok()` for quick success checks. Use `which()` to verify required tools exist

before running. Use `cwd()` and `cd()` for directory navigation. Use `lines()` to process multi-line strings. Use `pipe_to()` to filter or transform data through shell commands.

4.4.2 Shell Integration: shell() and sh()

Forge provides two functions for running shell commands:

`shell(cmd)` returns a full result object:

```
let result = shell("ls -la /tmp")
say result.stdout
say result.stderr
say result.status
say result.ok
```

Field	Type	Description
`stdout`	String	Standard output
`stderr`	String	Standard error
`status`	Integer	Exit code (0 = success)
`ok`	Boolean	`true` if exit code is 0

`sh(cmd)` returns just the stdout string, trimmed. It's a convenience wrapper for the common case where you just want the output:

```
let hostname = sh("hostname")
let date = sh("date")
say "Host: {hostname}, Date: {date}"
```

Use `shell()` when you need to check for errors or capture stderr. Use `sh()` when you just want the output of a command that you trust to succeed.

4.4.3 Environment Management

The `env` module reads and writes environment variables:

```
let home = env.get("HOME")
let path = env.get("PATH")
let has_key = env.has("API_KEY")
env.set("MY_VAR", "my_value")
```

This is essential for deployment scripts that read configuration from the environment, following the twelve-factor app methodology.

4.4.4 File System Operations for Automation

The `fs` module covers everything an automation script needs:

```
// Read and write
fs.write("/tmp/config.json", json.stringify(config))
let content = fs.read("/tmp/config.json")

// Check existence and get metadata
if fs.exists("/tmp/config.json") {
    let size = fs.size("/tmp/config.json")
    say "Config is {size} bytes"
}

// Directory operations
fs.mkdir("/tmp/backups")
let files = fs.list("/tmp/backups")

// Clean up
fs.remove("/tmp/old_file.txt")
```

4.4.5 Object Helpers for Configuration Management

DevOps scripts often work with nested config objects, environment overrides, and API responses. Forge's object helpers make this safe and readable:

- **`merge(defaults, overrides)`** — Merge config objects. Later values win. Ideal for combining defaults with environment-specific overrides:

```
let defaults = { port: 3000, host: "0.0.0.0", log_level: "info" }
let overrides = { port: 8080, log_level: "debug" }
let config = merge(defaults, overrides)
say config.port
```

- **`pick(obj, ["field1", "field2"])`** — Extract only the fields you need. Use when stripping sensitive or irrelevant fields from API responses:

```
let raw = sh_json("docker inspect mycontainer")
let relevant = pick(raw[0], ["Id", "State", "Mounts"])
```

- **`get(obj, "dot.path", default)`** — Safe nested access. Never crashes on missing keys. Use for optional config or nested API responses:

```
let resp = fetch("https://api.example.com/status")
let status = get(resp.json, "data.health", "unknown")
let retries = get(config, "retry.count", 3)
```

- **`has_key(obj, "key")`** — Check if a key exists before accessing. Use when config fields are optional:

```
if has_key(config, "ssl_cert") {
    say "SSL configured: {config.ssl_cert}"
}
```

4.4.6 Project 1: System Health Checker — Full Diagnostic Script

This comprehensive diagnostic tool checks CPU, memory, disk, network, and process information, presenting everything in a clean terminal dashboard.

```
// system_health.fg — Comprehensive system health checker
// Run: forge run system_health.fg

say term.banner("System Health Report")
say ""

let timestamp = sh("date")
say term.blue("Report generated: {timestamp}")
say ""

// ── System Information ──────────────────────────────────
say term.bold("═══ System Information ═══")
let user = sh("whoami")
let host = sh("hostname")
let os_name = sh("uname -s")
let arch = sh("uname -m")
let kernel = sh("uname -r")

let sys_info = [
    { Property: "User",     Value: user },
    { Property: "Hostname", Value: host },
    { Property: "OS",       Value: os_name },
    { Property: "Kernel",   Value: kernel },
    { Property: "Arch",     Value: arch }
]
term.table(sys_info)
say ""

// ── Uptime ──────────────────────────────────────────────
say term.bold("═══ Uptime ═══")
let uptime = sh("uptime")
say "  {uptime}"
say ""

// ── Disk Usage ──────────────────────────────────────────
```

```
say term.bold("═══ Disk Usage ═══")
let disk_result = shell("df -h / /tmp 2>/dev/null")
if disk_result.ok {
    let disk_lines = split(disk_result.stdout, "\n")
    for line in disk_lines {
        if len(line) > 0 {
            say "  {line}"
        }
    }
}
say ""

// ── Memory ──────────────────────────────────────────────────
say term.bold("═══ Memory ═══")
let mem_result = shell("vm_stat 2>/dev/null || free -h 2>/dev/null")
if mem_result.ok {
    let mem_lines = split(mem_result.stdout, "\n")
    let mut shown = 0
    for line in mem_lines {
        if shown < 5 {
            if len(line) > 0 {
                say "  {line}"
                shown = shown + 1
            }
        }
    }
}
say ""

// ── Environment ─────────────────────────────────────────────
say term.bold("═══ Environment ═══")
let home = env.get("HOME")
let user_shell = env.get("SHELL")
let path_val = env.get("PATH")
let path_sep = ":"
let path_entries = split(path_val, path_sep)
let path_count = len(path_entries)
let env_info = [
    { Variable: "HOME",         Value: home },
    { Variable: "SHELL",        Value: user_shell },
    { Variable: "PATH entries", Value: path_count }
]
term.table(env_info)
say ""

// ── Key Processes ───────────────────────────────────────────
say term.bold("═══ Active Processes ═══")
```

```
let proc_count = sh("ps aux | wc -l")
say "  Total processes: {proc_count}"
say ""

// ── Network Check ──────────────────────────────────────────
say term.bold("═══ Network Connectivity ═══")
let targets = [
    { name: "Google DNS",    host: "8.8.8.8" },
    { name: "Cloudflare",    host: "1.1.1.1" }
]

let mut net_results = []
for target in targets {
    let tname = target.name
    let thost = target.host
    let ping = shell("ping -c 1 -W 2 {thost} 2>/dev/null")
    if ping.ok {
        let entry = { Target: tname, Host: thost, Status: "Reachable" }
        net_results = append(net_results, entry)
    } else {
        let entry = { Target: tname, Host: thost, Status: "Unreachable" }
        net_results = append(net_results, entry)
    }
}
term.table(net_results)
say ""

// ── File System Check ──────────────────────────────────────
say term.bold("═══ File System Test ═══")
let test_file = "/tmp/forge_health_check.txt"
let test_content = "Health check at {timestamp}"
fs.write(test_file, test_content)
let readback = fs.read(test_file)
let fsize = fs.size(test_file)
let write_ok = readback == test_content
fs.remove(test_file)

say "  Write test:  {write_ok}"
say "  File size:   {fsize} bytes"
say "  Cleanup:     done"
say ""

// ── Security ───────────────────────────────────────────────
say term.bold("═══ Security Hashes ═══")
let check_hash = crypto.sha256("system-health-{host}-{timestamp}")
say "  Report hash: {check_hash}"
say ""
```

```
// ── Final Verdict ──────────────────────────────────────────
term.success("Health check complete — all systems nominal")
```

4.4.7 Project: Infrastructure Monitor

This project uses the complete shell toolkit in a realistic DevOps script. It checks required tools, verifies services, parses process listings and JSON, filters log data, and navigates directories—demonstrating all 11 shell functions plus object helpers.

```
// infrastructure_monitor.fg — DevOps script using the complete shell toolkit
// Run: forge run infrastructure_monitor.fg

say term.banner("Infrastructure Monitor")
say ""

// 1. which() — Check required tools
say term.bold("═══ Tool Check ═══")
let tools = ["docker", "git", "curl"]
let mut missing = []
for tool in tools {
    let path = which(tool)
    if path {
        say term.green("  {tool}: {path}")
    } else {
        missing = append(missing, tool)
        say term.red("  {tool}: NOT FOUND")
    }
}
say ""

// 2. sh_ok() — Check if services are running
say term.bold("═══ Service Status ═══")
if sh_ok("pgrep -q -x node") {
    say term.green("  Node: running")
} else {
    say term.yellow("  Node: not running")
}
if sh_ok("pgrep -q -x postgres") {
    say term.green("  Postgres: running")
} else {
    say term.yellow("  Postgres: not running")
}
say ""

// 3. cwd() and cd() — Directory navigation
say term.bold("═══ Directory ═══")
```

```
let start_dir = cwd()
say "  Started in: {start_dir}"
cd("/tmp")
let tmp_dir = cwd()
say "  Switched to: {tmp_dir}"
cd(start_dir)
say "  Restored: {cwd()}"
say ""

// 4. sh_lines() — Parse process listings
say term.bold("═══ Top Processes ═══")
let proc_lines = sh_lines("ps aux | head -6")
let proc_count = len(proc_lines)
say "  Showing {proc_count} process lines"
for line in proc_lines {
    if len(line) > 0 {
        say "    {line}"
    }
}
say ""

// 5. sh_json() — Parse JSON from system commands
say term.bold("═══ JSON from Command ═══")
let ob = "{"
let cb = "}"
let json_cmd = "echo '" + ob + "\"version\": \"1.0\", \"services\": [\"api\", \"worker\"]" + cb +
let json_data = sh_json(json_cmd)
if has_key(json_data, "version") {
    let ver = json_data.version
    say "  Config version: {ver}"
}
let svc_list = get(json_data, "services", [])
let svc_count = len(svc_list)
say "  Services defined: {svc_count}"
say ""

// 6. pipe_to() — Filter log data
say term.bold("═══ Log Filter ═══")
let sample_log = "ERROR db timeout\nINFO request OK\nWARN slow query\nERROR disk full\nINFO shutd
let filtered = pipe_to(sample_log, "grep ERROR")
let filtered_lines = lines(filtered.stdout)
say "  Errors in sample log: {len(filtered_lines)}"
for err_line in filtered_lines {
    say "    {err_line}"
}
say ""
```

```
// 7. lines() — Process multi-line strings
say term.bold("═══ Multi-line Parse ═══")
let hosts_block = "127.0.0.1 localhost\n::1 ip6-localhost"
let host_lines = lines(hosts_block)
say "  Host entries: {len(host_lines)}"
say ""

// 8. merge(), pick(), get() — Config management
say term.bold("═══ Config Merge ═══")
let defaults = { port: 3000, debug: false }
let env_overrides = { port: 8080, debug: true }
let config = merge(defaults, env_overrides)
let deploy_config = pick(config, ["port", "debug"])
say "  Merged: {deploy_config}"
let workers = get(config, "workers", 4)
say "  Workers (default): {workers}"
say ""

// 9. run_command() — Safe argv-style execution (no shell)
say term.bold("═══ run_command ═══")
let echo_result = run_command("echo hello from forge")
say "  Output: {echo_result.stdout}"
say ""

term.success("Infrastructure monitor complete!")
```

Walkthrough. The script uses `which()` to verify docker, git, and curl are installed. It uses `sh_ok()` to check if Node and Postgres processes are running. It uses `cwd()` and `cd()` to save, change, and restore the working directory. It uses `sh_lines()` to capture and display process output line-by-line. It uses `sh_json()` to parse JSON emitted by a command. It uses `pipe_to()` to pipe a multi-line log string into `grep ERROR` and then `lines()` to process the filtered output. It demonstrates `merge()`, `pick()`, `get()`, and `has_key()` for config management. Finally, it uses `run_command()` for safe command execution without a shell.

4.4.8 Project 2: Deploy Script — Config, Validation, Execution

This deployment automation script reads a JSON configuration file, validates it, runs pre-deployment checks, and executes the deployment steps.

```
// deploy.fg — Deployment automation script
// Run: forge run deploy.fg

say term.banner("Forge Deploy")
say ""

// Step 1: Load or create deploy configuration
```

```
let config_file = "deploy.json"
if fs.exists(config_file) == false {
    say term.blue("Creating default deploy config...")
    let default_config = {
        app_name: "myservice",
        version: "1.2.0",
        environment: "staging",
        port: 8080,
        health_check: "/health",
        build_cmd: "echo 'Building...'",
        test_cmd: "echo 'Tests passed'",
        pre_deploy: ["echo 'Pre-deploy hook 1'", "echo 'Pre-deploy hook 2'"],
        post_deploy: ["echo 'Post-deploy cleanup'"]
    }
    fs.write(config_file, json.pretty(default_config))
    say "  Created {config_file}"
    say ""
}

let config_raw = fs.read(config_file)
let config = json.parse(config_raw)
let app = config.app_name
let version = config.version
let environment = config.environment

say term.bold("Deploy Configuration:")
say "  App:         {app}"
say "  Version:     {version}"
say "  Environment: {environment}"
say ""

// Step 2: Validate configuration
say term.bold("═══ Validation ═══")
let mut errors = []
if app == null {
    errors = append(errors, "app_name is required")
}
if version == null {
    errors = append(errors, "version is required")
}
if environment == null {
    errors = append(errors, "environment is required")
}

if len(errors) > 0 {
    say term.error("Configuration errors:")
    for err in errors {
```

```
        say term.red("  ✗ {err}")
    }
    say ""
    say "Deploy aborted."
} else {
    say term.green("  ✓ Configuration valid")
    say ""

    // Step 3: Run build
    say term.bold("═══ Build ═══")
    let build_cmd = config.build_cmd
    let build = shell(build_cmd)
    if build.ok {
        say term.green("  ✓ Build succeeded")
    } else {
        say term.error("  ✗ Build failed")
        let stderr = build.stderr
        say "  {stderr}"
    }
    say ""

    // Step 4: Run tests
    say term.bold("═══ Tests ═══")
    let test_cmd = config.test_cmd
    let tests = shell(test_cmd)
    if tests.ok {
        say term.green("  ✓ Tests passed")
    } else {
        say term.error("  ✗ Tests failed — aborting deploy")
        let stderr = tests.stderr
        say "  {stderr}"
    }
    say ""

    // Step 5: Pre-deploy hooks
    say term.bold("═══ Pre-Deploy Hooks ═══")
    let pre = config.pre_deploy
    let mut hook_num = 1
    for cmd in pre {
        let result = shell(cmd)
        if result.ok {
            say term.green("  ✓ Hook {hook_num}: passed")
        } else {
            say term.red("  ✗ Hook {hook_num}: failed")
        }
        hook_num = hook_num + 1
    }
```

```
    say ""

    // Step 6: Deploy
    say term.bold("=== Deploying ===")
    let deploy_hash = crypto.sha256("{app}-{version}-{environment}")
    let short_hash = slice(deploy_hash, 0, 8)
    say "  Deploy ID: {short_hash}"
    say "  App:       {app}"
    say "  Version:   {version}"
    say "  Target:    {environment}"
    say ""

    // Step 7: Post-deploy hooks
    say term.bold("=== Post-Deploy Hooks ===")
    let post = config.post_deploy
    for cmd in post {
        let result = shell(cmd)
        if result.ok {
            say term.green("  ✓ Post-deploy: done")
        } else {
            say term.red("  ✗ Post-deploy: failed")
        }
    }
    say ""

    // Step 8: Write deployment log
    let log_entry = {
        app: app,
        version: version,
        environment: environment,
        deploy_id: short_hash,
        timestamp: sh("date -u +%Y-%m-%dT%H:%M:%SZ"),
        status: "success"
    }
    let log_line = json.stringify(log_entry)
    fs.write("deploy.log", log_line)
    say term.blue("  Deployment log written to deploy.log")
    say ""

    term.success("Deploy complete: {app} v{version} → {environment}")
}
```

4.4.9 Project 3: Backup Automation — Scan, Archive, Rotate

This script scans a directory tree, creates timestamped backups using tar, and rotates old backups to prevent disk exhaustion.

```
// backup.fg — Backup automation with rotation
// Run: forge run backup.fg

say term.banner("Backup Automation")
say ""

// Configuration
let source_dir = "/tmp/forge_backup_test"
let backup_dir = "/tmp/forge_backups"
let max_backups = 3

// Setup: create test data if it doesn't exist
if fs.exists(source_dir) == false {
    say term.blue("Creating test data...")
    fs.mkdir(source_dir)
    fs.write("{source_dir}/config.json", json.pretty({ app: "myservice", port: 8080 }))
    fs.write("{source_dir}/data.csv", "id,name,value\n1,alpha,100\n2,beta,200\n3,gamma,300")
    fs.write("{source_dir}/readme.txt", "This is the project readme file.")
    say "  Created test files in {source_dir}"
    say ""
}

// Create backup directory
if fs.exists(backup_dir) == false {
    fs.mkdir(backup_dir)
    say "  Created backup directory: {backup_dir}"
}

// Step 1: Inventory source files
say term.bold("═══ Source Inventory ═══")
let files = fs.list(source_dir)
let mut file_table = []
for file in files {
    let full_path = "{source_dir}/{file}"
    let fsize = fs.size(full_path)
    let row = { File: file, Size: fsize }
    file_table = append(file_table, row)
}
term.table(file_table)
let file_count = len(files)
say "  Total files: {file_count}"
say ""

// Step 2: Create timestamped backup
say term.bold("═══ Creating Backup ═══")
let timestamp = sh("date +%Y%m%d_%H%M%S")
let backup_name = "backup_{timestamp}.tar.gz"
```

```
let backup_path = "{backup_dir}/{backup_name}"

let tar_cmd = "tar -czf {backup_path} -C {source_dir} ."
let result = shell(tar_cmd)
if result.ok {
    let bsize = fs.size(backup_path)
    say term.green("  ✓ Created: {backup_name}")
    say "  Size: {bsize} bytes"
} else {
    let err = result.stderr
    say term.error("  ✗ Backup failed: {err}")
}
say ""

// Step 3: List existing backups
say term.bold("═══ Existing Backups ═══")
let all_files = fs.list(backup_dir)
let mut backups = []
for file in all_files {
    if starts_with(file, "backup_") {
        backups = append(backups, file)
    }
}
let backups = sort(backups)
let mut backup_table = []
for b in backups {
    let bpath = "{backup_dir}/{b}"
    let bsize = fs.size(bpath)
    let row = { Backup: b, Size: bsize }
    backup_table = append(backup_table, row)
}
term.table(backup_table)
let backup_count = len(backups)
say "  Total backups: {backup_count}"
say ""

// Step 4: Rotate — delete oldest backups if over limit
say term.bold("═══ Rotation ═══")
if backup_count > max_backups {
    let to_delete = backup_count - max_backups
    say "  Max backups: {max_backups}"
    say "  Current:     {backup_count}"
    say "  Removing:    {to_delete} oldest"
    say ""

    let mut deleted = 0
    for b in backups {
```

```
        if deleted < to_delete {
            let del_path = "{backup_dir}/{b}"
            fs.remove(del_path)
            say term.red("  ✗ Deleted: {b}")
            deleted = deleted + 1
        }
    }
} else {
    let remaining = max_backups - backup_count
    say "  Within limit ({backup_count}/{max_backups}). {remaining} slots remaining."
}
say ""

// Step 5: Generate backup hash for integrity verification
say term.bold("═══ Integrity ═══")
let hash = crypto.sha256(backup_name)
say "  Backup: {backup_name}"
say "  SHA256: {hash}"
say ""

// Summary
let summary = {
    source: source_dir,
    destination: backup_dir,
    backup_file: backup_name,
    files_backed_up: file_count,
    timestamp: sh("date -u +%Y-%m-%dT%H:%M:%SZ"),
    hash: hash
}
fs.write("{backup_dir}/last_backup.json", json.pretty(summary))
say term.blue("  Manifest written to {backup_dir}/last_backup.json")
say ""

term.success("Backup complete!")
```

4.4.10 Going Further

- **Cron integration.** Schedule your Forge scripts with cron: `*/30 * * * * forge run backup.fg`.
- **Remote execution.** Use `shell("ssh user@host 'command'")` to run commands on remote servers.
- **Monitoring loops.** Combine `schedule every 30 seconds { }` with health checks for a lightweight monitoring daemon.
- **Configuration management.** Build a Forge script that reads a YAML-like config, templates configuration files, and deploys them to the right directories.
- **Alerting.** Pipe health check results to `http.post()` calls to send alerts to Slack or PagerDuty webhooks.

4.5 Chapter 32: AI Integration

Forge has a built-in connection to large language models through the `ask` keyword. This isn't a library you install or an API you configure—it's a language-level primitive that sends a prompt to an LLM and returns the response as a string. Combined with Forge's file system access, data processing capabilities, and terminal formatting, this turns Forge into a powerful tool for building AI-augmented scripts and workflows.

4.5.1 The ask Keyword

The `ask` keyword sends a string prompt to an OpenAI-compatible API and returns the response:

```
let answer = ask "What is the capital of France?"
say answer
```

Under the hood, `ask` makes a POST request to the chat completions API with the prompt as a user message. The response is extracted from the API response and returned as a plain string.

4.5.2 Environment Setup

Before using `ask`, you need to set one of these environment variables:

```
# Option 1: Forge-specific key
export FORGE_AI_KEY="sk-your-api-key-here"

# Option 2: Standard OpenAI key
export OPENAI_API_KEY="sk-your-api-key-here"
```

Forge checks `FORGE_AI_KEY` first, then falls back to `OPENAI_API_KEY`. You can also customize the model and endpoint:

```
# Use a different model (default: gpt-4o-mini)
export FORGE_AI_MODEL="gpt-4o"

# Use a different API endpoint (for local models, Azure, etc.)
export FORGE_AI_URL="http://localhost:11434/v1/chat/completions"
```

The `FORGE_AI_URL` variable makes it possible to use Forge with any OpenAI-compatible API, including local models running through Ollama or LM Studio.

4.5.3 Prompt Templates

Since `ask` takes a string, you can build prompts dynamically using Forge's string interpolation:

```
let language = "Python"
let topic = "list comprehensions"
let prompt = "Explain {topic} in {language} with 3 examples. Be concise."
let explanation = ask prompt
say explanation
```

For longer prompts, build them up with string concatenation or multi-line construction:

```
let code = fs.read("my_script.py")
let prompt = "Review this code for bugs and suggest improvements:\n\n{code}"
let review = ask prompt
say review
```

4.5.4 Forge Chat Mode

Beyond programmatic use, Forge includes a built-in chat mode for interactive conversations with an LLM:

```
$ forge chat
```

This starts an interactive REPL where you can have a conversation with the configured LLM. It's useful for quick questions, brainstorming, and exploration without writing a script.

4.5.5 Project 1: Code Reviewer — File Analysis with LLM Feedback

This program reads a source file, sends it to an LLM for review, and displays the feedback with terminal formatting.

```
// code_reviewer.fg — AI-powered code review
// Run: FORGE_AI_KEY=sk-... forge run code_reviewer.fg
// Requires: FORGE_AI_KEY or OPENAI_API_KEY environment variable

say term.banner("AI Code Reviewer")
say ""

// Check for API key
let has_key = env.has("FORGE_AI_KEY")
let has_openai = env.has("OPENAI_API_KEY")
if has_key == false {
    if has_openai == false {
        say term.error("No API key found!")
```

```
        say "Set FORGE_AI_KEY or OPENAI_API_KEY environment variable."
        say ""
        say "Example:"
        say "  export FORGE_AI_KEY=sk-your-key-here"
        say "  forge run code_reviewer.fg"
    }
}

// Read the target file
let target_file = "examples/hello.fg"
if fs.exists(target_file) == false {
    say term.error("File not found: {target_file}")
} else {
    let code = fs.read(target_file)
    let file_size = fs.size(target_file)
    let line_list = split(code, "\n")
    let line_count = len(line_list)

    say term.bold("File: {target_file}")
    say "  Size:  {file_size} bytes"
    say "  Lines: {line_count}"
    say ""

    say term.blue("Sending to AI for review...")
    say ""

    let prompt = "You are a senior code reviewer. Review the following Forge programming language

    let review = ask prompt

    say term.bold("═══ AI Review ═══")
    say ""
    say review
    say ""

    // Save the review
    let review_file = "code_review.md"
    let review_content = "# Code Review: {target_file}\n\n{review}"
    fs.write(review_file, review_content)
    say term.blue("Review saved to {review_file}")
    say ""

    term.success("Code review complete!")
}
```

Walkthrough. The script first checks that an API key is configured. Then it reads the target source file, builds a detailed prompt that includes the code and specific instructions for the re-

view format, sends it to the LLM with `ask`, and displays the result. The review is also saved to a Markdown file for later reference.

4.5.6 Project 2: Data Describer — Natural Language Dataset Summary

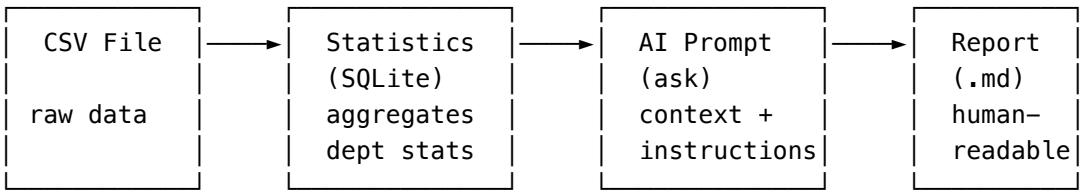
This program loads a CSV dataset, computes basic statistics, and uses an LLM to generate a natural language description of the data—useful for reports, documentation, or quick data exploration.

```
// data_describer.fg — AI-powered dataset description
// Run: FORGE_AI_KEY=sk-... forge run data_describer.fg
// Requires: FORGE_AI_KEY or OPENAI_API_KEY environment variable

say term.banner("AI Data Describer")
say ""

// Generate sample data
let csv_file = "sample_data.csv"
if fs.exists(csv_file) == false {
    say term.blue("Creating sample dataset...")
    let data = "name,age,department,salary,performance_score
Alice Chen,32,Engineering,128000,4.5
Bob Martinez,28,Marketing,89000,3.8
Carol Kim,41,Engineering,155000,4.9
David Johnson,35,Sales,95000,4.1
Eva Schmidt,29,Engineering,118000,4.3
Frank Brown,45,Marketing,105000,3.5
Grace Liu,33,Sales,91000,4.7
Henry Wilson,38,Engineering,142000,4.0
Isabel Torres,27,Marketing,82000,4.2
James Park,36,Sales,98000,3.9"
    fs.write(csv_file, data)
    say "  Created {csv_file}"
    say ""
}

// Read and parse
let raw = fs.read(csv_file)
let records = csv.parse(raw)
let record_count = len(records)

say term.bold("Dataset: {csv_file}")
say "  Records: {record_count}"
say ""

// Display the data
say term.bold("═══ Raw Data ═══")
term.table(records)
```

```
say ""

// Compute statistics using the database
db.open(":memory:")
db.execute("CREATE TABLE data (name TEXT, age INTEGER, department TEXT, salary REAL, performance_
for row in records {
    let n = row.name
    let a = row.age
    let d = row.department
    let s = row.salary
    let p = row.performance_score
    db.execute("INSERT INTO data VALUES ('{n}', {a}, '{d}', {s}, {p})")
}

let stats = db.query("SELECT COUNT(*) as count, ROUND(AVG(age), 1) as avg_age, ROUND(AVG(salary),
let dept_stats = db.query("SELECT department, COUNT(*) as headcount, ROUND(AVG(salary), 0) as avg

say term.bold("=== Statistics ===")
term.table(stats)
say ""
say term.bold("=== By Department ===")
term.table(dept_stats)
say ""

db.close()

// Build context for the AI
let stats_str = json.stringify(stats[0])
let dept_str = json.stringify(dept_stats)

let prompt = "You are a data analyst. Given this employee dataset summary, write a 3-4 paragraph

say term.blue("Generating natural language description...")
say ""

let description = ask prompt

say term.bold("=== AI-Generated Description ===")
say ""
say description
say ""

// Save report
let report = "# Dataset Report: {csv_file}\n\nGenerated: {sh("date")}\n\n## Summary\n\n{descripti
fs.write("data_report.md", report)
say term.blue("Report saved to data_report.md")
say ""
```

```
// Clean up sample data
fs.remove(csv_file)

term.success("Data description complete!")
```

The AI integration flow:



The power of this approach is the combination: Forge does the data processing (which it's good at—fast, deterministic, SQL-powered), then hands the structured results to the LLM for natural language generation (which the LLM is good at). Each tool does what it does best.

4.5.7 Going Further

- **Custom models.** Set `FORGE_AI_URL` to point to a local Ollama instance for private, offline AI capabilities: `export FORGE_AI_URL=http://localhost:11434/v1/chat/completions`
- **Prompt chaining.** Use the output of one `ask` call as input to the next, building multi-step reasoning pipelines.
- **Tool use patterns.** Have the LLM output structured JSON, parse it with `json.parse()`, and use the result to drive further program logic—a simple form of agentic behavior.
- **Batch processing.** Loop over multiple files or data records, sending each to the LLM and collecting responses for bulk analysis.
- **RAG-like patterns.** Read local files for context, include relevant excerpts in the prompt, and get answers grounded in your own data.

This concludes Part III: Building Real Things. You've built REST APIs, consumed external services, processed data pipelines, automated system operations, and integrated AI—all with a language that compiles to a single binary and requires zero external dependencies. In Part IV, we'll look at Forge's tooling ecosystem: the formatter, test runner, LSP, and how to publish Forge packages.

Chapter 5

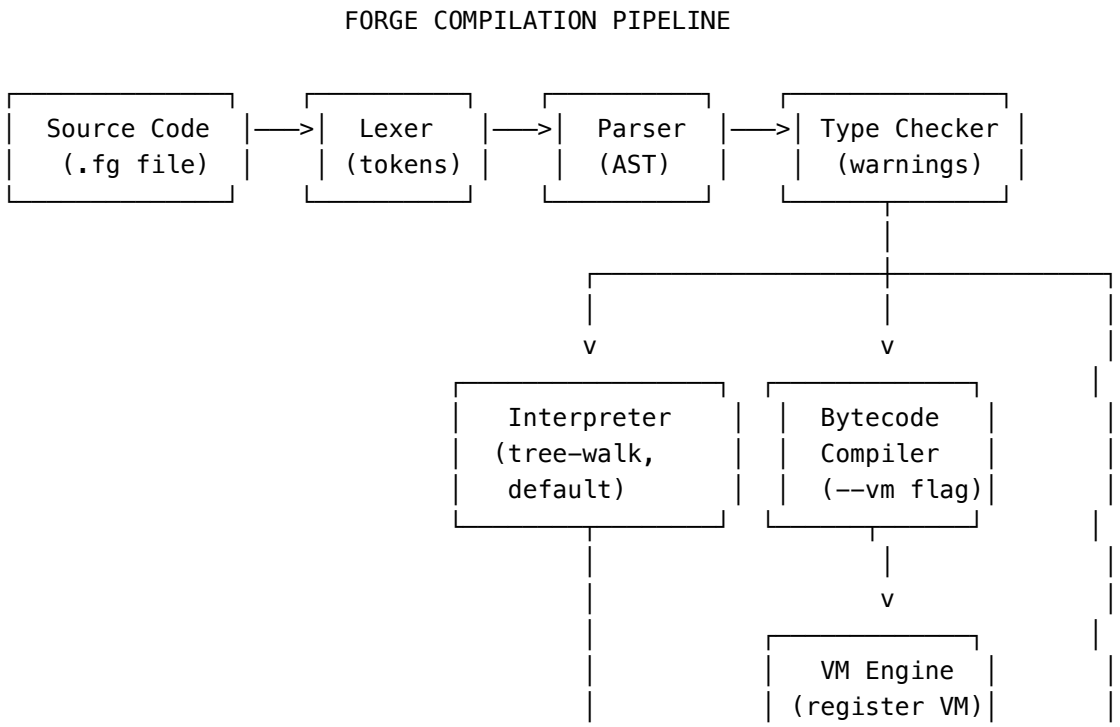
PART IV: UNDER THE HOOD

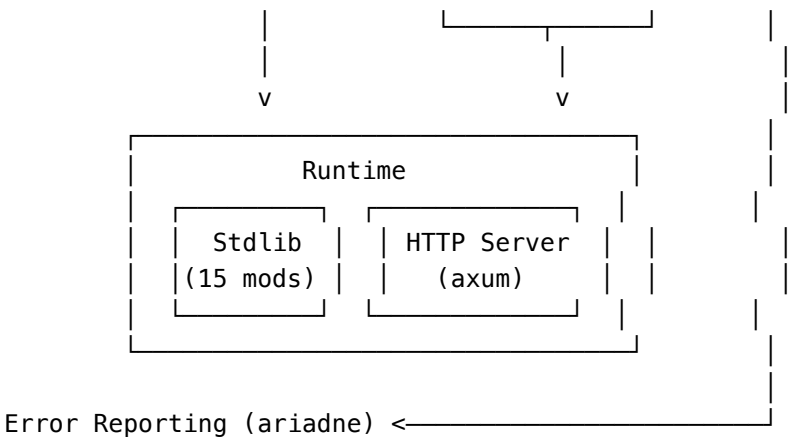
5.1 Chapter 33: Architecture and Internals

Every sufficiently advanced programming language eventually reveals its inner machinery to the curious developer. Understanding how Forge works beneath its friendly syntax transforms you from a user of the language into a collaborator with it. This chapter pulls back the curtain on Forge's implementation: approximately 15,500 lines of Rust spread across 45 source files, with zero `unsafe` blocks in the entire codebase.

5.1.1 The Compilation Pipeline

A Forge program begins its life as a `.fg` source file—a plain text document containing human-readable code. Through a series of well-defined transformations, that text becomes executable behavior. The pipeline has no magical jumps; each stage produces a clear intermediate representation consumed by the next.





The pipeline is invoked from `main.rs` (293 lines), which uses `clap` to parse CLI arguments and dispatch to the appropriate subsystem. The core execution path is remarkably concise:

```
async fn run_source(source: &str, filename: &str, use_vm: bool) {
    let mut lexer = Lexer::new(source);
    let tokens = lexer.tokenize()?;

    let mut parser = ForgeParser::new(tokens);
    let program = parser.parse_program()?;

    let mut checker = typechecker::TypeChecker::new();
    let warnings = checker.check(&program);

    if use_vm {
        vm::run(&program)?;
    } else {
        let mut interpreter = Interpreter::new();
        interpreter.run(&program)?;
    }
}
```

This linear flow—lex, parse, check, execute—is the spine of every Forge execution, whether triggered by `forge run`, `forge -e`, or the REPL.

5.1.2 The Lexer: Tokenization

The lexer (`src/lexer/`, 875 lines across two files) transforms raw source text into a stream of tokens. Forge uses a hand-rolled lexer rather than a generator like `logos`, giving full control over string interpolation handling and error reporting.

Architecture

The `Lexer` struct maintains four pieces of state:

```rust
pub struct Lexer {
    source: Vec<char>,  // Source code as character array
    pos: usize,         // Current position in source
    line: usize,        // Current line number (1-based)
    col: usize,         // Current column number (1-based)
}
```

The `source` field stores the input as `Vec<char>` rather than operating on byte slices. This simplifies character-by-character processing at the cost of an upfront allocation. For the typical Forge program (hundreds to low thousands of lines), this tradeoff is negligible.

The Tokenization Loop

The main `tokenize()` method runs a single pass over the source, producing a `Vec<Spanned>` where each `Spanned` wraps a `Token` with its position information:

```rust
pub struct Spanned {
    pub token: Token,
    pub line: usize,
    pub col: usize,
    pub offset: usize,   // Byte offset from start of source
    pub len: usize,      // Length in characters
}
```

The core loop follows a standard pattern: skip whitespace (preserving newlines), examine the current character, and dispatch to the appropriate lexing method:

```
Character          Handler              Example
──────────────────────────────────────────────────
0-9                lex_number()         42, 3.14, 1_000
"                  lex_string()         "hello, {name}"
"""                lex_triple_string()  """raw text"""
a-z, A-Z, _        lex_ident()          variable, let, say
+, -, *, / ...     inline match         operators
(, ), {, } ...     inline match         delimiters
//                 skip_line_comment()  (consumed, no token)
\n                 Token::Newline       (significant!)
```

Newline Significance

Unlike many languages, Forge treats newlines as significant tokens. The lexer skips spaces and tabs but *preserves* newline characters as `Token::Newline`. This design enables Forge's semicolon-free syntax: newlines serve as implicit statement terminators. The parser calls `skip_newlines()` at appropriate points to consume runs of newlines where they don't carry meaning (between block elements, for instance), while relying on them for statement boundaries elsewhere.

The Keyword Table

When the lexer encounters an identifier, it consults a keyword lookup function before emitting an `Ident` token. Forge recognizes 80+ keywords across three categories:

```rust
pub fn keyword_from_str(s: &str) -> Option<Token> {
    match s {
        // Classic keywords (22)
        "let" => Some(Token::Let),
        "fn"  => Some(Token::Fn),
        "if"  => Some(Token::If),
        // ...

        // Natural-language keywords (18)
        "set"       => Some(Token::Set),
        "say"       => Some(Token::Say),
        "otherwise" => Some(Token::Otherwise),
        // ...

        // Innovation keywords (25+)
        "when"     => Some(Token::When),
        "must"     => Some(Token::Must),
        "timeout"  => Some(Token::Timeout),
        // ...

        _ => None,
    }
}
```

This flat `match` statement compiles to an efficient jump table in release builds. There is no separate hash map allocation; Rust's pattern matching handles the dispatch.

String Interpolation

Forge strings support interpolation via `{expression}` syntax. The lexer handles this by preserving the `{` and `}` characters within the string literal. Interpolation is resolved at runtime by the interpreter, which parses the `{...}` segments and evaluates them in the current scope.

Escape sequences within strings follow standard conventions with two additions for brace escaping:

Escape	Character
`\n`	Newline
`\t`	Tab
`\r`	Carriage return
`\\`	Backslash

Escape	Character
\"	Double quote
\{	Literal {
\}	Literal }

Triple-quoted strings (`"""..."""`) produce `RawStringLit` tokens with no escape processing or interpolation, useful for embedding SQL, HTML, or other verbatim text.

Numeric Literals

The lexer supports integer and floating-point literals with underscore separators for readability:

```
42          → Token::Int(42)
3.14        → Token::Float(3.14)
1_000_000   → Token::Int(1000000)
```

A decimal point is only treated as a float separator when followed by a digit, preventing ambiguity with method call syntax like `list.len()`.

5.1.3 The Parser: Recursive Descent with Pratt Precedence

The parser (`src/parser/`, 2,147 lines across two files) transforms the token stream into an abstract syntax tree. It uses recursive descent for statement parsing and a layered precedence-climbing approach (inspired by Pratt parsing) for expressions.

Structure

```rust
pub struct Parser {
    tokens: Vec<Spanned>,
    pos: usize,
}
```

The parser maintains a flat token array and a position cursor. Peeking ahead, consuming tokens, and backtracking are all constant-time operations on this array.

Statement Parsing

The `parse_statement()` method dispatches on the current token to determine which statement form to parse:

```rust
fn parse_statement(&mut self) -> Result<Stmt, ParseError> {
    match self.current_token() {
        Token::Let              => self.parse_let(),
        Token::Set              => self.parse_set(),
        Token::Change           => self.parse_change(),
        Token::Fn | Token::Define => self.parse_fn_def(Vec::new()),
        Token::If               => self.parse_if(),
        Token::Match            => self.parse_match(),
        Token::For              => self.parse_for(),
        Token::While            => self.parse_while(),
        Token::When             => self.parse_when(),
        Token::Check            => self.parse_check(),
        Token::Safe             => self.parse_safe_block(),
        Token::Timeout          => self.parse_timeout(),
        Token::Retry            => self.parse_retry(),
        Token::At               => self.parse_decorator_or_fn(),
        Token::Say | Token::Yell | Token::Whisper
                                => self.parse_say_yell_whisper(),
        // ... 15+ more variants
        _                       => self.parse_expr_or_assign(),
    }
}
```

Notice how both `Token::Fn` and `Token::Define` route to the same `parse_fn_def()` method, and `Token::Else`, `Token::Otherwise`, and `Token::Nah` are all treated identically when checking for else branches. This is how Forge's dual syntax is implemented: distinct tokens, shared parse logic.

Expression Precedence

Forge's expression parser uses a layered approach where each precedence level is a separate function that calls the next-higher level:

```
Precedence Level    Function                Operators
─────────────────────────────────────────────────────
1 (lowest)          parse_pipeline()        |>
2                   parse_or()              ||
3                   parse_and()             &&
4                   parse_equality()        ==  !=
5                   parse_comparison()      <  >  <=  >=
6                   parse_addition()        +  -
7                   parse_multiplication()  *  /  %
8                   parse_unary()           -  !  must  await  ...
9 (highest)         parse_postfix()         ()  .  []  ?
10                  parse_primary()         literals, idents, groups
```

Each function follows the same pattern: parse the higher-precedence sub-expression, then loop to

consume operators at its own level:

```rust
fn parse_addition(&mut self) -> Result<Expr, ParseError> {
    let mut left = self.parse_multiplication()?;
    loop {
        let op = match self.current_token() {
            Token::Plus  => BinOp::Add,
            Token::Minus => BinOp::Sub,
            _ => break,
        };
        self.advance();
        let right = self.parse_multiplication()?;
        left = Expr::BinOp {
            left: Box::new(left), op, right: Box::new(right),
        };
    }
    Ok(left)
}
```

This naturally produces left-associative operators with correct precedence.

Handling Newlines in the Parser

The parser's `skip_newlines()` method is called at strategic points: at the top of `parse_statement()`, inside block parsing between statements, and before checking for `else`/`otherwise`/`nah` branches. This allows code like:

```
if condition {
    say "yes"
}
otherwise {
    say "no"
}
```

The newlines between `}` and `otherwise` are consumed by `skip_newlines()` before the parser checks for an else branch.

5.1.4 The AST: Stmt and Expr Enums

The abstract syntax tree is defined in `src/parser/ast.rs` (335 lines). Forge's AST uses two central enums: `Stmt` (28 variants) for statements and `Expr` (22 variants) for expressions.

Statement Variants

```
  Stmt Variant          Syntax It Represents
```

Let	`let x = 5 / set x to 5`
Assign	`x = 10 / change x to 10`
FnDef	`fn name() {} / define name() {}`
StructDef	`struct Point { x: Int, y: Int }`
TypeDef	`type Shape = Circle(Float) | Rect(Float, Float)`
InterfaceDef	`interface Printable { print() }`
Return	`return expr`
If	`if / else / otherwise / nah`
Match	`match subject { pattern => body }`
When	`when subject { < 10 -> "small" }`
For	`for x in items / for each x in items`
While	`while condition { }`
Loop	`loop { }`
Break	`break`
Continue	`continue`
Spawn	`spawn { }`
Import	`import "path"`
TryCatch	`try { } catch err { }`
CheckStmt	`check name is not empty`
SafeBlock	`safe { }`
TimeoutBlock	`timeout 5 seconds { }`
RetryBlock	`retry 3 times { }`
ScheduleBlock	`schedule every 5 minutes { }`
WatchBlock	`watch "file.txt" { }`
PromptDef	`prompt summarize(text) { }`
AgentDef	`agent researcher(query) { }`
Destructure	`unpack {a, b} from obj`
YieldStmt	`yield expr / emit expr`
Expression	(bare expression as statement)

Expression Variants

Expr Variant	Syntax It Represents
Int, Float, Bool	`42, 3.14, true`
StringLit	`"hello"`
StringInterp	`"hello, {name}"`
Array	`[1, 2, 3]`
Object	`{ name: "Alice", age: 30 }`
Ident	`variable_name`
BinOp	`a + b, x == y`
UnaryOp	`-x, !done`
FieldAccess	`user.name`
Index	`list[0]`
Call	`fn(args)`
MethodCall	`obj.method(args)`

```
Pipeline            data |> transform
Lambda              (x) => x * 2
Try                 risky_call()?
Await               await promise / hold promise
Spread              ...args
Must                must dangerous_call()
Freeze              freeze value
Ask                 ask "what is the meaning of life?"
WhereFilter         users where age > 21
PipeChain           data >> keep where active >> take 10
StructInit          Point { x: 1, y: 2 }
Block               { stmts }
```

Supporting Types

The AST includes several supporting structures:

Type	Purpose
`Program`	Top-level container: `Vec<Stmt>`
`Param`	Function parameter with optional type and default
`TypeAnn`	Type annotation: `Simple`, `Array`, `Generic`, `Function`, `Optional`
`Decorator`	`@name(args)` metadata
`MatchArm`	Pattern + body for `match` expressions
`WhenArm`	Operator + value + result for `when` guards
`Pattern`	`Wildcard`, `Literal`, `Binding`, `Constructor`
`BinOp`	12 binary operators (Add through Or)
`UnaryOp`	2 unary operators (Neg, Not)
`PipeStep`	Steps in a `>>` pipeline: Keep, Sort, Take, Apply
`DestructurePattern`	Object or array destructuring shapes
`Variant`	ADT variant with typed fields
`CheckKind`	Validation type: IsNotEmpty, Contains, Between, IsTrue
`StringPart`	Literal or Expr component of interpolated strings

5.1.5 The Interpreter: Tree-Walk Evaluation

The interpreter (`src/interpreter/mod.rs`, 4,584 lines) is the default execution engine. It walks the AST directly, evaluating each node recursively. While slower than bytecode execution, the tree-walk interpreter supports the full Forge feature set including async/await, the HTTP server, and all stdlib modules.

Runtime Values

The interpreter's `Value` enum maps closely to Forge's type system:

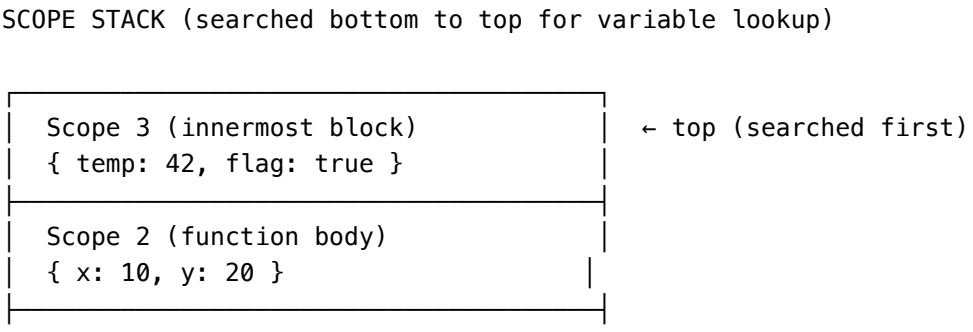
```rust
pub enum Value {
    Int(i64),
    Float(f64),
    String(String),
    Bool(bool),
    Array(Vec<Value>),
    Object(IndexMap<String, Value>),
    Function {
        name: String,
        params: Vec<Param>,
        body: Vec<Stmt>,
        closure: Environment,    // Captured scope
        decorators: Vec<Decorator>,
    },
    Lambda {
        params: Vec<Param>,
        body: Vec<Stmt>,
        closure: Environment,
    },
    ResultOk(Box<Value>),
    ResultErr(Box<Value>),
    BuiltIn(String),             // Name of built-in function
    Null,
}
```

Objects use `IndexMap` (from the `indexmap` crate) rather than `HashMap` to preserve insertion order, matching the behavior users expect from JSON-like object literals.

The Scope Stack

Variable scoping is managed by the `Environment` struct, which maintains a stack of hash maps:

```rust
pub struct Environment {
    scopes: Vec<HashMap<String, Value>>,
    mutability: Vec<HashMap<String, bool>>,
}
```



```
  ┌──────────────────────────────────────┐
  │  Scope 1 (module level)              │
  │  { greet: <fn>, data: [...] }        │
  ├──────────────────────────────────────┤
  │  Scope 0 (global / builtins)         │  ← bottom (searched last)
  │  { print: <builtin>, math: {...},    │
  │    len: <builtin>, say: <builtin> }  │
  └──────────────────────────────────────┘
```

Key operations:

- **`push_scope()`**: Enters a new block. Pushes an empty HashMap onto both `scopes` and `mutability`.
- **`pop_scope()`**: Exits a block. Pops the top scope, discarding its variables.
- **`define(name, value)`**: Inserts a binding into the current (topmost) scope.
- **`get(name)`**: Searches scopes from top to bottom, returning the first match.
- **`set(name, value)`**: Searches scopes from top to bottom; updates the first matching scope. Returns an error if the variable is immutable.

The parallel `mutability` stack tracks whether each variable was declared with `mut`. Attempting to reassign an immutable variable produces a clear error with a hint:

```
cannot reassign immutable variable 'name' (use 'let mut' to make it mutable)
```

The "Did You Mean?" Feature

When variable lookup fails, the interpreter doesn't just report "undefined variable"—it searches all scopes for similar names using Levenshtein distance:

```rust
pub fn suggest_similar(&self, name: &str) -> Option<String> {
    let mut best: Option<(String, usize)> = None;
    for scope in &self.scopes {
        for key in scope.keys() {
            let dist = levenshtein(name, key);
            if dist <= 2 && dist < name.len() {
                // Track the closest match
            }
        }
    }
    best.map(|(s, _)| s)
}
```

This turns `undefined variable: naem` into `undefined variable: naem (did you mean: name?)`, a small touch that saves real debugging time.

Control Flow with Signals

The interpreter uses a `Signal` enum to propagate control flow across recursive evaluation:

```rust
enum Signal {
    None,       // Normal execution
    Return(Value),  // Function return
    Break,      // Loop break
    Continue,   // Loop continue
}
```

Every `exec_stmt()` call returns `Result<Signal, RuntimeError>`. The main execution loop checks each returned signal: `Signal::Return` short-circuits function execution, `Signal::Break` exits the nearest enclosing loop, and `Signal::Continue` skips to the next iteration. Encountering `Break` or `Continue` outside a loop produces a runtime error.

Closure Implementation

Closures in Forge capture their environment at the point of definition. When a function or lambda is created, the interpreter clones the current `Environment`:

```rust
Stmt::FnDef { name, params, body, decorators, is_async } => {
    let closure = self.env.clone();
    let func = Value::Function {
        name: name.clone(),
        params: params.clone(),
        body: body.clone(),
        closure,
        decorators: decorators.clone(),
    };
    self.env.define(name.clone(), func);
}
```

When the function is later called, the interpreter temporarily replaces the current environment with the captured closure, pushes a new scope for the function's parameters, executes the body, then restores the original environment. This correctly handles nested closures and avoids variable shadowing issues.

5.1.6 How Builtins Are Registered and Dispatched

On initialization, the interpreter's `register_builtins()` method populates the global scope with two categories of values:

1. Standard library modules are registered as `Value::Object` instances, each containing named functions:

```rust
fn register_builtins(&mut self) {
    self.env.define("math".to_string(), crate::stdlib::create_math_module());
    self.env.define("fs".to_string(),   crate::stdlib::create_fs_module());
```

```
    self.env.define("crypto".to_string(), crate::stdlib::create_crypto_module());
    // ... 12 more modules
}
```

2. Global built-in functions are registered as `Value::BuiltIn(name)`:

```
for name in &["print", "println", "len", "type", "str", "int",
              "push", "pop", "keys", "values", "contains", "range",
              "map", "filter", "reduce", "sort", "reverse",
              "say", "yell", "whisper", "assert", "assert_eq",
              "Ok", "Err", "unwrap", "fetch", "time", "uuid",
              /* ... 40+ more ... */] {
    self.env.define(name.to_string(), Value::BuiltIn(name.to_string()));
}
```

When the interpreter encounters a `Call` expression and resolves the callee to `Value::BuiltIn(name)`, it dispatches through a large `match` statement that handles each built-in. This approach has the advantage of zero-overhead dispatch (no dynamic function pointers or vtables) and allows builtins to directly access interpreter internals.

5.1.7 How the HTTP Server Integrates

Forge's HTTP server is built on axum (a Tokio-based web framework). The integration is clever: the server is *not* invoked during normal script execution. Instead, after the interpreter finishes running the program, the runtime checks whether a `@server` decorator was defined:

```
let server_config = runtime::server::extract_server_config(&program);
let routes = runtime::server::extract_routes(&program);

if let Some(config) = server_config {
    runtime::server::start_server(interpreter, &config, &routes).await?;
}
```

The `extract_routes()` function scans the AST for functions decorated with `@get`, `@post`, `@put`, or `@delete`, collecting their URL patterns and handler names. The interpreter instance (with all defined functions in its environment) is wrapped in `Arc<Mutex<Interpreter>>` and passed as shared state to axum handlers.

When an HTTP request arrives, the axum handler locks the interpreter, constructs a request object, calls the Forge handler function, and converts the returned `Value` into an HTTP response. This design means each request briefly locks the interpreter—acceptable for development servers but not designed for production-scale concurrent workloads.

5.2 Chapter 34: The Bytecode VM

While the tree-walk interpreter provides full-featured execution, some workloads benefit from the tighter execution loop of a bytecode virtual machine. Forge includes an experimental register-based VM activated with the `--vm` flag. This chapter examines its design, instruction set, and runtime subsystems.

5.2.1 Why a VM?

Tree-walk interpreters pay a tax on every AST node they visit: pattern matching on the enum variant, navigating Box pointers, and recursing through the call stack. For tight loops and numeric computation, this overhead dominates the actual work. A bytecode VM eliminates these costs by compiling the AST into a flat array of instructions that a simple loop can decode and execute without recursion.

```
TREE-WALK INTERPRETER

eval_expr(BinOp { left, op, right })
  ├── eval_expr(left)        ← recursive call, enum match
  ├── eval_expr(right)       ← recursive call, enum match
  └── apply_op(op, l, r)     ← actual computation

VM EXECUTION LOOP

loop {
    let instruction = code[ip];   ← array index
    ip += 1;
    match decode_op(instruction) {
        OpCode::Add => {           ← flat switch
            regs[a] = regs[b] + regs[c];
        }
    }
}
```

The VM's execution loop touches less memory, has better branch prediction, and avoids recursion entirely for non-call instructions. For compute-heavy programs, this can yield 2-5x speedups.

5.2.2 Register-Based vs. Stack-Based VMs

Forge's VM uses a register-based architecture (like Lua 5 and Dalvik) rather than a stack-based one (like the JVM or CPython). In a register-based VM, instructions specify source and destination registers explicitly:

```
Stack-Based (JVM-style):        Register-Based (Forge VM):
```

```
PUSH a                          ADD  R2, R0, R1
PUSH b                          (one instruction, three operands)
ADD
(three instructions, implicit
 stack manipulation)
```

The register approach produces fewer instructions (though each instruction is wider), reduces memory traffic to the operand stack, and simplifies optimization.

5.2.3 Instruction Encoding

Each instruction is encoded as a single 32-bit word with three possible formats:

```
Format ABC:   [  op:8  |  A:8  |  B:8  |  C:8  ]
Format ABx:   [  op:8  |  A:8  |    Bx:16      ]
Format AsBx:  [  op:8  |  A:8  |   sBx:16      ]
```

Field	Size	Purpose
`op`	8 bits	Opcode identifier (up to 256 instructions)
`A`	8 bits	Destination register or primary operand
`B`	8 bits	Second register operand
`C`	8 bits	Third register operand
`Bx`	16 bits	Unsigned extended operand (constant index)
`sBx`	16 bits	Signed extended operand (jump offset)

Encoding and decoding functions are fully inlined for performance:

```rust
pub fn encode_abc(op: OpCode, a: u8, b: u8, c: u8) -> u32 {
    ((op as u32) << 24) | ((a as u32) << 16) | ((b as u32) << 8) | (c as u32)
}

#[inline(always)]
pub fn decode_op(instruction: u32) -> u8 {
    (instruction >> 24) as u8
}
```

5.2.4 The Bytecode Instruction Set

Forge's VM defines 42 opcodes organized into seven categories:

Loading Constants and Values

Opcode	Format	Description
`LoadConst`	ABx	Load constant pool entry `Bx` into register `A`
`LoadNull`	A	Load `null` into register `A`
`LoadTrue`	A	Load `true` into register `A`
`LoadFalse`	A	Load `false` into register `A`

Arithmetic and Logic

Opcode	Format	Description
`Add`	ABC	`R[A] = R[B] + R[C]`
`Sub`	ABC	`R[A] = R[B] - R[C]`
`Mul`	ABC	`R[A] = R[B] * R[C]`
`Div`	ABC	`R[A] = R[B] / R[C]`
`Mod`	ABC	`R[A] = R[B] % R[C]`
`Neg`	AB	`R[A] = -R[B]`
`Eq`	ABC	`R[A] = R[B] == R[C]`
`NotEq`	ABC	`R[A] = R[B] != R[C]`
`Lt`	ABC	`R[A] = R[B] < R[C]`
`Gt`	ABC	`R[A] = R[B] > R[C]`
`LtEq`	ABC	`R[A] = R[B] <= R[C]`
`GtEq`	ABC	`R[A] = R[B] >= R[C]`
`And`	ABC	`R[A] = R[B] && R[C]`
`Or`	ABC	`R[A] = R[B] || R[C]`
`Not`	AB	`R[A] = !R[B]`

Variable Access

Opcode	Format	Description
`Move`	AB	`R[A] = R[B]`
`GetLocal`	AB	`R[A] = locals[B]`
`SetLocal`	AB	`locals[A] = R[B]`
`GetGlobal`	ABx	`R[A] = globals[const[Bx]]`
`SetGlobal`	ABx	`globals[const[Bx]] = R[A]`

Data Structures

Opcode	Format	Description
`NewArray`	ABC	Create array from registers `B..B+C` into `R[A]`
`NewObject`	ABx	Create object with `B` key-value pairs into `R[A]`
`GetField`	ABC	`R[A] = R[B].field(const[C])`
`SetField`	ABC	`R[A].field(const[B]) = R[C]`
`GetIndex`	ABC	`R[A] = R[B][R[C]]`
`SetIndex`	ABC	`R[A][R[B]] = R[C]`

Opcode	Format	Description
`Concat`	ABC	`R[A] = str(R[B]) + str(R[C])`
`Len`	AB	`R[A] = len(R[B])`
`Interpolate`	ABC	Interpolate `C` parts starting at `R[B]` into `R[A]`
`ExtractField`	ABC	Extract tuple field `C` from `R[B]` into `R[A]`

Control Flow

Opcode	Format	Description
`Jump`	sBx	`ip += sBx`
`JumpIfFalse`	AsBx	If `R[A]` is falsy, `ip += sBx`
`JumpIfTrue`	AsBx	If `R[A]` is truthy, `ip += sBx`
`Loop`	sBx	`ip += sBx` (negative, jumps backward)

Functions

Opcode	Format	Description
`Call`	ABC	Call `R[A]` with `B` args, result in `R[C]`
`Return`	A	Return `R[A]` from current function
`ReturnNull`	—	Return `null` from current function
`Closure`	ABx	Create closure from prototype `Bx` into `R[A]`

Special

Opcode	Format	Description
`Try`	AB	`R[A] = try R[B]` (wrap in Result)
`Spawn`	A	Spawn green thread with closure `R[A]`
`Pop`	—	Discard top value (cleanup)

5.2.5 The Compiler: AST to Bytecode

The bytecode compiler (`src/vm/compiler.rs`, 772 lines) performs a single pass over the AST, emitting instructions into a `Chunk`:

```
pub struct Compiler {
    chunk: Chunk,            // Output bytecode
    locals: Vec<Local>,      // Local variable tracking
    scope_depth: usize,      // Current nesting depth
    next_register: u8,       // Next available register
    max_register: u8,        // High-water mark
    loops: Vec<LoopContext>, // Active loop tracking for break/continue
}
```

Each `Chunk` contains the bytecode, a constant pool, line number information, and nested prototypes (for closures):

```rust
pub struct Chunk {
    pub code: Vec<u32>,          // Bytecode instructions
    pub constants: Vec<Constant>, // Constant pool
    pub lines: Vec<usize>,       // Line numbers (parallel to code)
    pub name: String,            // Function name
    pub prototypes: Vec<Chunk>,  // Nested function prototypes
    pub max_registers: u8,       // Register count needed
    pub upvalue_count: u8,       // Captured variable count
    pub arity: u8,               // Parameter count
}
```

The compiler tracks local variables with their scope depth and assigned register, using a register allocation strategy that simply increments a counter and reclaims registers when scopes close.

Jump Patching: Forward jumps (for `if`/`else`, `while`, etc.) are emitted with a placeholder offset, then patched once the target address is known:

```rust
fn emit_jump(&mut self, op: OpCode, a: u8, line: usize) -> usize {
    let idx = self.chunk.code_len();
    self.emit(encode_asbx(op, a, 0), line);  // Placeholder
    idx
}

fn patch_jump(&mut self, offset: usize) {
    let target = self.chunk.code_len();
    self.chunk.patch_jump(offset, target);
}
```

5.2.6 The Execution Loop

The VM's core (`src/vm/machine.rs`, 1,807 lines) runs a tight decode-dispatch loop:

```rust
pub struct VM {
    pub registers: Vec<Value>,       // Flat register array
    pub frames: Vec<CallFrame>,      // Call stack
    pub globals: HashMap<String, Value>,
    pub gc: Gc,                      // Garbage collector
    pub output: Vec<String>,         // Captured output
}
```

The call stack uses `CallFrame` structs that point into the flat register array:

```rust
pub struct CallFrame {
    pub closure: GcRef,   // The executing closure
    pub ip: usize,        // Instruction pointer
    pub base: usize,      // Base register offset
}

pub const MAX_FRAMES: usize = 256;
pub const MAX_REGISTERS: usize = MAX_FRAMES * 256;  // 65,536
```

Each frame owns a "window" of 256 registers starting at `base`. Register references in instructions are relative to the current frame's base, providing isolation between function invocations without copying.

5.2.7 Mark-Sweep Garbage Collection

The VM uses a mark-sweep garbage collector (`src/vm/gc.rs`, 113 lines) to manage heap-allocated objects. Unlike the interpreter (which uses Rust's ownership and cloning for memory management), the VM needs explicit GC because objects may be referenced from multiple locations.

Object Representation

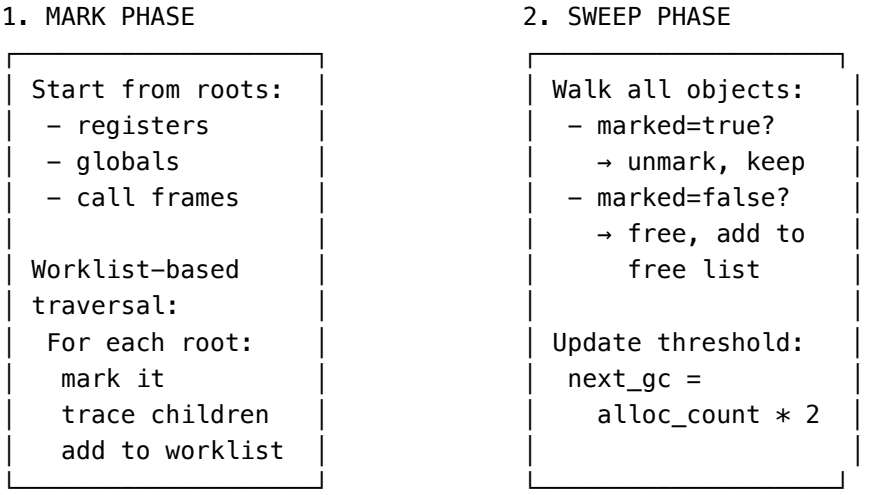
Heap objects are stored as `GcObject` instances in a flat vector:

```rust
pub struct GcObject {
    pub kind: ObjKind,
    pub marked: bool,      // GC mark flag
}

pub enum ObjKind {
    String(String),
    Array(Vec<Value>),
    Object(IndexMap<String, Value>),
    Function(ObjFunction),
    Closure(ObjClosure),
    NativeFunction(NativeFn),
    Upvalue(ObjUpvalue),
    ResultOk(Value),
    ResultErr(Value),
}
```

References to heap objects are `GcRef(usize)` indices into the GC's object vector.

Collection Algorithm

```
MARK-SWEEP GC CYCLE
```



The GC triggers when allocations exceed a threshold (initially 256). After collection, the threshold is set to twice the surviving object count, implementing a simple adaptive strategy. The minimum threshold is clamped to 256 to avoid pathologically frequent collections with few live objects.

The mark phase uses an explicit worklist rather than recursion:

```rust
fn mark(&mut self, roots: &[GcRef]) {
    let mut worklist: Vec<GcRef> = roots.to_vec();
    while let Some(r) = worklist.pop() {
        if let Some(obj) = self.objects.get_mut(r.0).and_then(|o| o.as_mut()) {
            if obj.marked { continue; }
            obj.marked = true;
            obj.trace(&mut worklist);  // Add referenced objects
        }
    }
}
```

Freed object slots are added to a free list for reuse, avoiding vector growth when objects churn.

5.2.8 Green Thread Scheduler

The VM includes a scaffold for cooperative green threads (`src/vm/green.rs`, 83 lines). Currently, `spawn` blocks execute synchronously—the scheduler runs each spawned chunk to completion before starting the next:

```rust
pub fn run_all(&mut self, vm: &mut VM) -> Result<(), VMError> {
    while let Some(thread) = self.threads.iter_mut()
        .find(|t| t.state == ThreadState::Ready)
    {
        thread.state = ThreadState::Running;
        vm.execute(&thread.chunk)?;
```

```
            thread.state = ThreadState::Completed;
        }
        Ok(())
}
```

The data structures for genuine cooperative scheduling are in place—thread states (`Ready`, `Running`, `Yielded`, `Completed`), a thread ID system, and an active count tracker. Future work will integrate with Tokio for preemption at function calls and loop back-edges.

5.2.9 Using the –vm Flag

To run a program with the bytecode VM:

```
forge run program.fg --vm
```

To compile a program to bytecode and view compilation statistics:

```
forge build program.fg
```

This outputs:

```
Compiled program.fg -> program.fgc
  47 instructions
  12 constants
  3 prototypes
  8 max registers
```

The `--vm` flag is experimental. It supports core language features (variables, functions, closures, control flow, data structures, error handling) but does not yet support async/await, the HTTP server, or all standard library modules.

5.3 Chapter 35: Tooling Deep Dive

A programming language is only as good as its tools. Forge ships with a comprehensive toolchain that handles formatting, testing, project scaffolding, compilation, package management, editor integration, interactive learning, and AI-assisted development—all from a single binary.

5.3.1 forge fmt: Code Formatter

The formatter (`src/formatter.rs`, 147 lines) normalizes whitespace and indentation across Forge source files.

How it works: The formatter operates on text lines rather than the AST, making it fast and robust (it never fails, even on syntactically invalid code). It tracks brace nesting depth to compute indentation:

1. For each line, trim leading whitespace
2. If the line starts with `}`, decrease indent level
3. Apply the computed indent (4 spaces per level)
4. If the line ends with `{`, increase indent level for the next line
5. Collapse multiple consecutive blank lines into one
6. Ensure a trailing newline

What it normalizes:

- Consistent 4-space indentation
- Removal of trailing whitespace
- Collapse of consecutive blank lines
- Consistent line endings

Usage:

```
forge fmt                    # Format all .fg files in current directory
forge fmt src/main.fg        # Format specific file
forge fmt src/ lib/          # Format multiple paths
```

The formatter recursively discovers `.fg` files, skipping directories named `.` (dot-prefixed), `target`, and `node_modules`.

5.3.2 forge test: Test Runner

The test runner (`src/testing/mod.rs`, 170 lines) discovers and executes test functions marked with the `@test` decorator.

Test runner architecture:

```
1. Discover .fg files in tests/ directory
2. For each file:
   a. Lex → Parse (report errors)
   b. Find functions with @test decorator
   c. Run full program (defines all functions)
   d. For each @test function:
      - Call it with no arguments
      - Time execution
      - Report pass/fail with duration
3. Print summary: passed, failed, total
```

Writing tests:

```
@test
define should_add_numbers() {
    assert(1 + 1 == 2)
    assert_eq(2 * 3, 6)
}

@test
define should_handle_strings() {
    set name to "Forge"
    assert(len(name) == 5)
    assert(starts_with(name, "For"))
}
```

Available assertion functions:

Function	Description
`assert(expr)`	Fails if `expr` is falsy
`assert_eq(a, b)`	Fails if `a != b`, shows both values
`satisfies(val, fn)`	Fails if `fn(val)` returns false

Output format:

```
  tests/math_test.fg
    ok    should_add_numbers (2ms)
    ok    should_handle_strings (1ms)

  tests/api_test.fg
    FAIL  should_validate_input (3ms)
          assertion failed: expected true, got false

  2 passed, 1 failed, 3 total
```

The test runner exits with code 1 if any test fails, making it suitable for CI pipelines.

5.3.3 forge new: Project Scaffolding

The `forge new` command (`src/scaffold.rs`, 71 lines) creates a project directory with a standard structure:

```
forge new my-app
```

Generates:

```
  my-app/
```

```
├── forge.toml          # Project manifest
├── main.fg             # Entry point
├── tests/
│   └── basic_test.fg   # Starter test
└── .gitignore          # Ignores *.fgc files
```

forge.toml contents:

```toml
[project]
name = "my-app"
version = "0.1.0"
description = ""

[test]
directory = "tests"
```

The manifest is read by `forge test` to determine the test directory and by future tooling for package metadata.

5.3.4 forge build: Bytecode Compilation

The `forge build` command compiles a Forge source file to bytecode using the VM's compiler:

```
forge build program.fg
```

This runs the lexer, parser, and bytecode compiler, then reports statistics about the compiled output. The compiled bytecode is represented as a `Chunk` structure containing instructions, constants, and nested prototypes.

5.3.5 forge install: Package Management

The package manager (`src/package.rs`, 118 lines) supports two installation sources:

Git installation:

```
forge install https://github.com/user/forge-utils.git
```

Clones the repository into `.forge/packages/forge-utils/`. Subsequent runs pull updates.

Local installation:

```
forge install ../shared-lib
```

Copies the directory into `.forge/packages/shared-lib/`.

Import resolution checks paths in order:

1. Direct file path (relative to current file)
2. `.forge/packages/<name>`
3. `.forge/packages/<name>/main.fg`

5.3.6 forge lsp: Language Server Protocol

The LSP server (`src/lsp/mod.rs`, 261 lines) provides IDE integration over stdin/stdout using the Language Server Protocol:

Supported capabilities:

- **Diagnostics**: Real-time parse error reporting as you type
- **Completions**: Keyword and built-in function suggestions triggered by `.`

Architecture: The LSP runs as a long-lived process communicating via JSON-RPC. It re-lexes and re-parses the document on every change, sending diagnostics back to the editor. Completion requests return the full keyword list and standard library function names.

Editor setup (VS Code example):

```
{
  "forge.lsp.path": "forge",
  "forge.lsp.args": ["lsp"]
}
```

5.3.7 forge learn: Interactive Tutorials

The tutorial system (`src/learn.rs`, 520 lines) provides 30 progressive lessons built into the binary:

```
forge learn      # List all lessons
forge learn 1    # Start lesson 1
```

Each lesson includes a title, explanation, example code, and expected output. The system displays the lesson content, lets the user study the example, and provides the expected output for verification. Lessons cover:

Lesson	Topic
1	Hello World
2	Variables
3	Mutable Variables
4	Functions
5	The Fun Trio (say/yell/whisper)
6	Arrays & Loops
7	Objects
8	Repeat Loops
9	Destructuring
10	Error Handling

Lesson	Topic
11	Pattern Matching
12	Pipelines
13	HTTP Requests
14	Building APIs

5.3.8 forge chat: AI Integration

The `forge chat` command (`src/chat.rs`, 131 lines) starts an interactive AI chat session. It reads an API key from the `OPENAI_API_KEY` environment variable and communicates with the OpenAI API to provide conversational assistance about Forge programming.

5.3.9 The forge.toml Manifest

The manifest file (`src/manifest.rs`, 68 lines) uses TOML format with the following schema:

```toml
[project]
name = "project-name"       # Required: project name
version = "0.1.0"           # Required: semver version
description = "A Forge app" # Optional: project description
entry = "main.fg"           # Optional: entry point file

[test]
directory = "tests"          # Optional: test directory (default: "tests")
```

The manifest is parsed using the `toml` and `serde` crates with default values for all optional fields.

Chapter 6

APPENDICES

6.1 Appendix A: Complete Keyword Reference

Forge recognizes 80+ keywords divided into three categories: classic keywords familiar from mainstream languages, natural-language keywords that provide English-like alternatives, and innovation keywords unique to Forge.

6.1.1 Table A-1: Classic Keywords

Keyword	Purpose	Example	Notes
`let`	Declare variable	`let x = 42`	Immutable by default
`mut`	Make variable mutable	`let mut count = 0`	Used with `let` or `set`
`fn`	Define function	`fn greet(name) { }`	Synonym of `define`
`return`	Return from function	`return value`	Optional for last expression
`if`	Conditional branch	`if x > 0 { }`	—
`else`	Alternative branch	`else { }`	Synonym of `otherwise`, `nah`
`match`	Pattern matching	`match value { 1 => "one" }`	Exhaustive patterns
`for`	Loop over iterable	`for x in items { }`	Supports destructuring
`in`	Iterable marker	`for x in range(10)`	Used with `for`, `each`
`while`	Conditional loop	`while running { }`	—
`loop`	Infinite loop	`loop { if done { break } }`	Use `break` to exit
`break`	Exit loop	`break`	—
`continue`	Skip to next iteration	`continue`	—
`struct`	Define structure	`struct Point { x: Int, y: Int }`	Named product type
`type`	Define algebraic data type	`type Color = Red | Blue`	Sum types with variants
`interface`	Define interface	`interface Printable { print() }`	Method signatures
`impl`	Implement interface	`impl Printable for Point { }`	Reserved for future use

Keyword	Purpose	Example	Notes
`pub`	Public visibility	`pub fn api() { }`	Reserved for future use
`import`	Import module	`import "utils.fg"`	File or package import
`spawn`	Launch concurrent task	`spawn { heavy_work() }`	Currently synchronous
`true`	Boolean true	`let flag = true`	—
`false`	Boolean false	`let done = false`	—
`try`	Begin try block	`try { risky() }`	Paired with `catch`
`catch`	Handle error	`catch err { log(err) }`	Receives error value
`async`	Async function	`async fn fetch_data() { }`	Synonym of `forge` (keyword)
`await`	Await async result	`await fetch("url")`	Synonym of `hold`
`yield`	Yield from generator	`yield value`	Synonym of `emit`

6.1.2 Table A-2: Natural-Language Keywords

Keyword	Purpose	Example	Classic Equivalent
`set`	Declare variable	`set name to "Alice"`	`let name = "Alice"`
`to`	Assignment marker	`set x to 42`	`=` in `let`
`change`	Reassign variable	`change score to score + 1`	`score = score + 1`
`define`	Define function	`define greet(n) { }`	`fn greet(n) { }`
`otherwise`	Alternative branch	`otherwise { }`	`else { }`
`nah`	Alternative branch (casual)	`nah { }`	`else { }`
`each`	Loop marker	`for each item in list { }`	`for item in list`
`repeat`	Counted loop	`repeat 5 times { }`	`for _ in range(5)`
`times`	Repeat unit	`repeat 3 times { }`	—
`grab`	HTTP fetch	`grab resp from "url"`	`let resp = fetch("url")`
`from`	Source marker	`grab data from url`	—
`wait`	Sleep / pause	`wait 2 seconds`	`sleep(2000)`
`seconds`	Time unit	`wait 5 seconds`	—
`say`	Print output	`say "hello"`	`println("hello")`
`yell`	Print uppercase	`yell "loud"`	— (unique)
`whisper`	Print lowercase	`whisper "quiet"`	— (unique)
`forge`	Async function	`forge fetch_data() { }`	`async fn fetch_data()`
`hold`	Await result	`hold fetch("url")`	`await fetch("url")`
`emit`	Yield value	`emit computed_value`	`yield computed_value`
`unpack`	Destructure	`unpack {a, b} from obj`	`let {a, b} = obj`

6.1.3 Table A-3: Innovation Keywords

Keyword	Purpose	Example	Notes
`when`	Guard-based matching	`when age { < 13 -> "kid" }`	Comparison-based match
`unless`	Negated conditional	`do_thing() unless disabled`	Postfix condition
`until`	Negated while	`retry until success`	Postfix loop condition
`must`	Crash on error	`must parse(data)`	Unwrap or panic with message
`check`	Declarative validation	`check name is not empty`	Built-in validators
`safe`	Null-safe execution	`safe { risky_call() }`	Returns null on error
`where`	Collection filter	`users where age > 21`	SQL-like filtering
`timeout`	Time-limited execution	`timeout 5 seconds { fetch() }`	Cancels after duration
`retry`	Automatic retry	`retry 3 times { connect() }`	Retries on failure
`schedule`	Periodic execution	`schedule every 5 minutes { }`	Cron-like scheduling
`every`	Schedule interval	`schedule every 10 seconds { }`	Used with `schedule`
`watch`	File change detection	`watch "config.fg" { reload() }`	File system watcher
`ask`	AI/LLM query	`ask "summarize this text"`	Calls language model
`prompt`	Define AI prompt template	`prompt summarize(text) { }`	Structured LLM call
`transform`	Data transformation	`transform data { upper() }`	Pipeline transform
`table`	Tabular display	`table [row1, row2]`	Terminal table output
`select`	Query projection	`from users select name, age`	SQL-like projection
`order`	Sort clause	`order by name`	Used with `select`
`by`	Sort/order marker	`sort by score`	—
`limit`	Result limiting	`limit 10`	Used with queries
`keep`	Pipeline filter	`>> keep where active`	Used with `>>` pipes
`take`	Pipeline slice	`>> take 5`	Used with `>>` pipes
`freeze`	Deep immutable copy	`freeze config`	Prevents mutation
`download`	Download file	`download "url" to "file"`	HTTP file download
`crawl`	Web scraping	`crawl "https://example.com"`	Returns page content
`any`	Existential check	`any x in items`	Used in conditions

6.2 Appendix B: Built-in Functions Quick Reference

Forge provides 50+ built-in functions available without imports. Standard library modules add 90+ more functions organized into 15 namespaces.

6.2.1 Output Functions

Function	Signature	Description	Example
`print`	`print(value)`	Print without newline	`print("loading...")`
`println`	`println(value)`	Print with newline	`println("done")`
`say`	`say(value)`	Print with newline (natural)	`say "hello, world!"`
`yell`	`yell(value)`	Print uppercase with newline	`yell "alert"` ☐ `ALERT`
`whisper`	`whisper(value)`	Print lowercase with newline	`whisper "SHH"` ☐ `shh`

6.2.2 Type Conversion Functions

Function	Signature	Description	Example
`str`	`str(value) -> String`	Convert to string	`str(42)` ☐ `"42"`
`int`	`int(value) -> Int`	Convert to integer	`int("42")` ☐ `42`
`float`	`float(value) -> Float`	Convert to float	`float("3.14")` ☐ `3.14`
`type`	`type(value) -> String`	Get type name	`type(42)` ☐ `"Int"`
`typeof`	`typeof(value) -> String`	Get type name (alias)	`typeof([])` ☐ `"Array"`

6.2.3 Collection Functions

Function	Signature	Description	Example
`len`	`len(collection) -> Int`	Get length/size	`len([1,2,3])` ☐ `3`
`push`	`push(array, value) -> Array`	Append element	`push(list, 4)`
`pop`	`pop(array) -> Value`	Remove and return last	`pop(stack)`
`keys`	`keys(object) -> Array`	Get object keys	`keys({a:1})` ☐ `["a"]`
`values`	`values(object) -> Array`	Get object values	`values({a:1})` ☐ `[1]`
`contains`	`contains(coll, val) -> Bool`	Check membership	`contains([1,2], 2)` ☐ `true`
`range`	`range(n) -> Array`	Generate `[0..n-1]`	`range(3)` ☐ `[0,1,2]`
`enumerate`	`enumerate(array) -> Array`	Index-value pairs	`enumerate(["a"])` ☐ `[[0,"a"]]`

Function	Signature	Description	Example
`has_key`	`has_key(obj, key) -> Bool`	Check if object has key	`has_key(user, "email")` ☐ `true`
`get`	`get(obj, key, default)`	Safe access with fallback, dot-paths	`get(obj, "a.b", "x")`
`pick`	`pick(obj, [keys]) -> Object`	Extract specific fields	`pick(user, ["name"])`
`omit`	`omit(obj, [keys]) -> Object`	Remove specific fields	`omit(user, ["password"])`
`merge`	`merge(obj1, obj2) -> Object`	Combine objects (later wins)	`merge(a, b)`
`entries`	`entries(obj) -> Array`	Object to [[key, val], ...]	`entries({a: 1})` ☐ `[["a", 1]]`
`from_entries`	`from_entries(pairs) -> Object`	[[key, val], ...] to object	`from_entries([["a", 1]])`
`find`	`find(array, fn) -> Value`	First matching element	`find(arr, fn(x) { return x > 5 })`
`flat_map`	`flat_map(array, fn) -> Array`	Map and flatten	`flat_map([[1,2],[3]], fn(x) { return x })`
`lines`	`lines(string) -> Array`	Split string by newlines	`lines("a\nb")` ☐ `["a", "b"]`

6.2.4 Functional Programming Functions

Function	Signature	Description	Example
`map`	`map(array, fn) -> Array`	Transform each element	`map([1,2,3], (x) => x * 2)`
`filter`	`filter(array, fn) -> Array`	Keep matching elements	`filter(nums, (x) => x > 0)`
`reduce`	`reduce(array, init, fn) -> Value`	Fold to single value	`reduce([1,2,3], 0, (a,b) => a+b)`
`sort`	`sort(array) -> Array`	Sort elements	`sort([3,1,2])` ☐ `[1,2,3]`
`reverse`	`reverse(array) -> Array`	Reverse order	`reverse([1,2,3])` ☐ `[3,2,1]`

6.2.5 String Functions

Function	Signature	Description	Example
`split`	`split(str, delim) -> Array`	Split string	`split("a,b", ",")` ☐ `["a","b"]`
`join`	`join(array, delim) -> String`	Join with delimiter	`join(["a","b"], "-")` ☐ `"a-b"`

Function	Signature	Description	Example
`replace`	`replace(str, old, new) -> String`	Replace substring	`replace("hi", "h", "H")` ☐ `"Hi"`
`starts_with`	`starts_with(str, prefix) -> Bool`	Check prefix	`starts_with("hello", "he")` ☐ `true`
`ends_with`	`ends_with(str, suffix) -> Bool`	Check suffix	`ends_with("hello", "lo")` ☐ `true`

6.2.6 Result Functions

Function	Signature	Description	Example
`Ok`	`Ok(value) -> Result`	Create success result	`Ok(42)`
`Err`	`Err(value) -> Result`	Create error result	`Err("failed")`
`is_ok`	`is_ok(result) -> Bool`	Check if Ok	`is_ok(Ok(1))` ☐ `true`
`is_err`	`is_err(result) -> Bool`	Check if Err	`is_err(Err("x"))` ☐ `true`
`unwrap`	`unwrap(result) -> Value`	Extract Ok or panic	`unwrap(Ok(42))` ☐ `42`
`unwrap_or`	`unwrap_or(result, default) -> Value`	Extract Ok or default	`unwrap_or(Err("x"), 0)` ☐ `0`

6.2.7 Option Functions

Function	Signature	Description	Example
`Some`	`Some(value) -> Option`	Create present option	`Some(42)`
`None`	—	Absent option value	`None`
`is_some`	`is_some(option) -> Bool`	Check if present	`is_some(Some(1))` ☐ `true`
`is_none`	`is_none(option) -> Bool`	Check if absent	`is_none(None)` ☐ `true`

6.2.8 System Functions

Function	Signature	Description	Example
`time`	`time() -> Float`	Current Unix timestamp	`time()` ☐ `1709136000.0`
`uuid`	`uuid() -> String`	Generate UUID v4	`uuid()` ☐ `"a1b2c3..."`
`exit`	`exit(code)`	Exit with status code	`exit(1)`
`input`	`input(prompt) -> String`	Read line from stdin	`input("Name: ")`
`wait`	`wait(seconds)`	Sleep for duration	`wait(2)`

Function	Signature	Description	Example
`shell`	`shell(command) -> String`	Execute shell command	`shell("ls -la")`
`sh`	`sh(command) -> String`	Execute shell (alias)	`sh("date")`
`run_command`	`run_command(cmd) -> String`	Execute command	`run_command("echo hi")`
`fetch`	`fetch(url) -> Object`	HTTP GET request	`fetch("https://api.example.com")`
`sh_lines`	`sh_lines(cmd) -> Array`	Run command, return lines	`sh_lines("ls")`
`sh_json`	`sh_json(cmd)`	Run command, parse JSON output	`sh_json("echo '[1]'")`
`sh_ok`	`sh_ok(cmd) -> Bool`	Run command, return bool	`sh_ok("which git")`
`which`	`which(cmd) -> String`	Find command path	`which("git")` ☐ `"/usr/bin/git"`
`cwd`	`cwd() -> String`	Current directory	`cwd()` ☐ `"/home/user"`
`cd`	`cd(path)`	Change directory	`cd("/tmp")`
`pipe_to`	`pipe_to(data, cmd)`	Pipe string data into command	`pipe_to(csv, "sort")`

6.2.9 Assertion Functions

Function	Signature	Description	Example
`assert`	`assert(condition)`	Fail if falsy	`assert(x > 0)`
`assert_eq`	`assert_eq(actual, expected)`	Fail if not equal	`assert_eq(len(s), 5)`
`satisfies`	`satisfies(value, predicate)`	Fail if predicate returns false	`satisfies(age, (a) => a >= 0)`

6.2.10 Standard Library Module Functions

Access via `module.function()` syntax after the module is available in scope:

math — `math.sqrt(x)`, `math.pow(x,n)`, `math.abs(x)`, `math.max(a,b)`, `math.min(a,b)`, `math.floor(x)`, `math.ceil(x)`, `math.round(x)`, `math.pi()`, `math.e()`, `math.sin(x)`, `math.cos(x)`, `math.tan(x)`, `math.log(x)`, `math.random()`

fs — `fs.read(path)`, `fs.write(path, data)`, `fs.append(path, data)`, `fs.exists(path)`, `fs.list(dir)`, `fs.remove(path)`, `fs.mkdir(path)`, `fs.copy(src, dst)`, `fs.rename(old, new)`, `fs.size(path)`, `fs.ext(path)`, `fs.read_json(path)`, `fs.write_json(path, data)`

io — `io.prompt(msg)`, `io.print(val)`, `io.args()`

crypto — `crypto.sha256(data)`, `crypto.md5(data)`, `crypto.base64_encode(data)`, `crypto.base64_decode(data`

`crypto.hex_encode(data)`, `crypto.hex_decode(data)`

db — `db.open(path)`, `db.query(db, sql)`, `db.execute(db, sql)`, `db.close(db)`

pg — `pg.connect(url)`, `pg.query(conn, sql)`, `pg.execute(conn, sql)`, `pg.close(conn)`

env — `env.get(key)`, `env.set(key, val)`, `env.has(key)`, `env.keys()`

json — `json.parse(str)`, `json.stringify(val)`, `json.pretty(val)`

regex — `regex.test(pattern, str)`, `regex.find(pattern, str)`, `regex.find_all(pattern, str)`, `regex.replace(pattern, str, replacement)`, `regex.split(pattern, str)`

log — `log.info(msg)`, `log.warn(msg)`, `log.error(msg)`, `log.debug(msg)`

http — `http.get(url)`, `http.post(url, body)`, `http.put(url, body)`, `http.delete(url)`, `http.patch(url, body)`, `http.head(url)`, `http.download(url, path)`, `http.crawl(url)`

csv — `csv.parse(str)`, `csv.stringify(data)`, `csv.read(path)`, `csv.write(path, data)`

term — `term.red(str)`, `term.green(str)`, `term.blue(str)`, `term.yellow(str)`, `term.bold(str)`, `term.dim(str)`, `term.table(data)`, `term.hr()`, `term.sparkline(data)`, `term.bar(label, value, max)`, `term.banner(text)`, `term.countdown(seconds)`, `term.confirm(prompt)`

exec — `exec.run_command(cmd)`

6.3 Appendix C: Operator Precedence Table

Operators are listed from lowest precedence (evaluated last) to highest precedence (evaluated first). Operators at the same precedence level are evaluated according to their associativity.

Precedence	Operator(s)	Description	Associativity
1 (lowest)	`|>`	Pipeline	Left
2	`||`	Logical OR	Left
3	`&&`	Logical AND	Left
4	`== !=`	Equality	Left
5	`< > <= >=`	Comparison	Left
6	`+ -`	Addition, Subtraction	Left
7	`* / %`	Multiplication, Division, Modulo	Left
8	`-x !x`	Unary negation, NOT	Right (prefix)
8	`must await/hold`	Must-unwrap, Await	Right (prefix)
8	`... freeze ask`	Spread, Freeze, AI query	Right (prefix)
9	`()`	Function call	Left (postfix)
9	`.`	Field access	Left (postfix)
9	`[]`	Index access	Left (postfix)
9	`?`	Try operator	Left (postfix)

Precedence	Operator(s)	Description	Associativity
10 (highest)	Literals, identifiers, `()` groups	Primary	—

Compound assignment operators (`+=`, `-=`, `*=`, `/=`) are parsed as statements, not expressions. They desugar to `x = x op value` internally.

The `>>` pipe operator is parsed separately from `|>` and chains pipeline steps (`keep`, `sort`, `take`, `apply`).

6.4 Appendix D: CLI Reference

6.4.1 Synopsis

```
forge [OPTIONS] [COMMAND]
```

6.4.2 Global Options

Option	Description
`-e, --eval <CODE>`	Evaluate a Forge expression inline
`--vm`	Use the bytecode VM (experimental, faster but fewer features)
`-h, --help`	Print help information
`-V, --version`	Print version number

6.4.3 Commands

`forge run <FILE>`

Run a Forge source file.

```
forge run main.fg
forge run main.fg --vm
```

Argument	Description
`FILE`	Path to a `.fg` source file

forge repl

Start the interactive REPL (Read-Eval-Print Loop). Also the default when no command is given.

```
forge repl
forge          # Same as forge repl
```

Features: command history, tab completion for keywords and built-ins, multi-line input.

forge version

Display version information.

```
forge version
# Output: Forge v0.3.0
#         Internet-native programming language
#         Bytecode VM with mark-sweep GC
```

forge fmt [FILES...]

Format Forge source files. With no arguments, formats all `.fg` files in the current directory recursively.

```
forge fmt                  # Format all .fg files
forge fmt main.fg          # Format specific file
forge fmt src/ lib/        # Format directories
```

forge test [DIR]

Run tests in the specified directory (default: `tests`).

```
forge test                 # Run tests in tests/
forge test integration     # Run tests in integration/
```

Argument	Default	Description
`DIR`	`tests`	Directory containing test files

If a `forge.toml` exists, the `[test].directory` field overrides the default.

`forge new <NAME>`

Create a new Forge project with standard directory structure.

```
forge new my-api
```

Argument	Description
`NAME`	Project name (creates directory with this name)

`forge build <FILE>`

Compile a Forge source file to bytecode and display compilation statistics.

```
forge build main.fg
```

Argument	Description
`FILE`	Path to source file to compile

`forge install <SOURCE>`

Install a Forge package from a git URL or local path.

```
forge install https://github.com/user/package.git
forge install ../local-package
```

Argument	Description
`SOURCE`	Git URL (https:// or git@) or local filesystem path

`forge lsp`

Start the Language Server Protocol server for editor integration.

```
forge lsp
```

Communicates via stdin/stdout using JSON-RPC. Provides diagnostics and completions.

`forge learn [LESSON]`

Launch the interactive tutorial system.

```
forge learn          # List all 30 lessons
forge learn 1        # Start lesson 1 (Hello World)
forge learn 30       # Start lesson 30 (File Path Utilities)
```

Argument	Description
`LESSON`	Optional lesson number (1–30)

`forge chat`

Start an AI chat session for Forge programming assistance.

```
forge chat
```

Requires the `OPENAI_API_KEY` environment variable to be set.

`forge -e <CODE>`

Evaluate inline Forge code without creating a file.

```
forge -e 'say "hello!"'
forge -e 'println(math.sqrt(144))'
forge -e 'say range(5) |> map((x) => x * x)'
```

6.5 Appendix E: Error Messages Guide

Forge produces clear, source-mapped error messages using the `ariadne` crate. This appendix catalogs common errors, explains what causes them, and shows how to fix them.

6.5.1 Undefined Variable

```
error: undefined variable: naem
  ┌─ <source>:3:5
  │
3 │ say naem
  │     ^^^^ undefined variable: naem (did you mean: name?)
```

Cause: Using a variable that hasn't been declared in any accessible scope.

Fix: Check for typos. The "did you mean?" suggestion uses Levenshtein distance to find variables within an edit distance of 2. Ensure the variable is declared before use and is in scope.

```
// Wrong
say naem

// Right
set name to "Alice"
say name
```

6.5.2 Unexpected Token

```
error: unexpected token: Semicolon
  ┌─ <source>:1:12
  │
1 │ let x = 42;
  │            ^ unexpected token: Semicolon
```

Cause: Forge uses newlines as statement terminators. Semicolons are recognized but not used as terminators in normal code.

Fix: Remove the semicolon. Forge does not require (or expect) semicolons at the end of statements.

```
// Wrong
let x = 42;

// Right
let x = 42
```

6.5.3 Immutable Variable Reassignment

```
error: cannot reassign immutable variable 'count' (use 'let mut' to make it mutable)
  ┌─ <source>:2:1
  │
2 │ count = count + 1
  │ ^ cannot reassign immutable variable 'count'
```

Cause: Attempting to reassign a variable declared without `mut`.

Fix: Declare the variable as mutable:

```
// Wrong
let count = 0
count = count + 1

// Right
let mut count = 0
count = count + 1
```

```
// Or using natural syntax:
set mut count to 0
change count to count + 1
```

6.5.4 Division by Zero

```
error: division by zero
  hint: check the divisor before dividing
   ┌─ <source>:1:9
   │
1 │ let x = 10 / 0
   │         ^^^^^^ division by zero
```

Cause: Dividing an integer or float by zero.

Fix: Guard against zero divisors:

```
if divisor != 0 {
    let result = value / divisor
} otherwise {
    say "Cannot divide by zero"
}
```

6.5.5 Type Mismatch (Warning)

```
warning: type mismatch: expected Int, got String
   ┌─ <source>:2:10
   │
2 │ let x: Int = "hello"
   │              ^^^^^^^ expected Int
```

Cause: A type annotation doesn't match the assigned value. Forge uses gradual typing—type annotations are checked but violations produce warnings, not errors.

Fix: Either correct the value or remove/update the type annotation:

```
// Option 1: fix the value
let x: Int = 42

// Option 2: fix the annotation
let x: String = "hello"

// Option 3: remove the annotation
let x = "hello"
```

6.5.6 Cannot Call on Type

```
error: cannot call value of type String
  ┌─ <source>:3:1
  │
3 │ name(42)
  │ ^^^^ cannot call value of type String
```

Cause: Attempting to call a value that is not a function, lambda, or built-in.

Fix: Ensure the identifier refers to a callable value:

```
// Wrong
let name = "Alice"
name(42)  // name is a String, not a function

// Right
fn greet(name) { say "Hello, {name}" }
greet("Alice")
```

6.5.7 Index Out of Bounds

```
error: index out of bounds: index 5, length 3
  ┌─ <source>:2:1
  │
2 │ list[5]
  │ ^^^^^^^ index out of bounds: index 5, length 3
```

Cause: Accessing an array element at an index beyond its length.

Fix: Check the array length before indexing, or use a safe access pattern:

```
let list = [10, 20, 30]

// Guard with length check
if index < len(list) {
    say list[index]
}

// Or use safe block
safe {
    let val = list[index]
    say val
}
```

6.5.8 Unterminated String

```
error: unterminated string (newline in string literal)
  ┌─ <source>:1:11
  │
1 │ let x = "hello
  │               ^ unterminated string
```

Cause: A string literal contains an unescaped newline or reaches end-of-file without a closing quote.

Fix: Close the string on the same line, use `\n` for embedded newlines, or use a triple-quoted string for multi-line text:

```
// Single-line string with escape
let x = "hello\nworld"

// Multi-line with triple quotes
let x = """
hello
world
"""
```

6.5.9 Unknown Escape Sequence

```
error: unknown escape: \q
  ┌─ <source>:1:12
  │
1 │ let x = "\q"
  │            ^ unknown escape: \q
```

Cause: Using an escape character that Forge doesn't recognize.

Fix: Use one of the supported escape sequences: `\n`, `\t`, `\r`, `\\`, `\"`, `\{`, `\}`.

6.5.10 Break/Continue Outside Loop

```
error: break outside of loop
  ┌─ <source>:1:1
  │
1 │ break
  │ ^^^^^ break outside of loop
```

Cause: Using `break` or `continue` outside of a `for`, `while`, `loop`, or `repeat` block.

Fix: Ensure these keywords only appear inside loop bodies.

6.6 Appendix F: Forge vs. Other Languages

This appendix provides detailed comparison tables showing how Forge stacks up against popular languages. These comparisons highlight where Forge simplifies common tasks, where it innovates, and where other languages may be more appropriate.

6.6.1 Table F-1: Forge vs. Python

Feature	Forge	Python
Variable declaration	`let x = 5` / `set x to 5`	`x = 5`
Immutability	Built-in: `let` (immutable by default)	Convention only (UPPER_CASE)
Type annotations	Optional: `let x: Int = 5`	Optional: `x: int = 5`
Function definition	`fn add(a, b) { }` / `define add(a, b) { }`	`def add(a, b):`
Lambda	`(x) => x * 2`	`lambda x: x * 2`
String interpolation	`"Hello, {name}"`	`f"Hello, {name}"`
Print	`say "hello"` / `println("hello")`	`print("hello")`
HTTP GET	`fetch("url")` / `grab data from "url"`	`requests.get("url")` (external lib)
HTTP server	Built-in: `@server` + `@get` decorators	Flask/FastAPI (external)
Pattern matching	`match x { 1 => "one" }`	`match x: case 1: "one"` (3.10+)
Error handling	`try/catch`, `must`, `safe`, `?` operator	`try/except`
Null safety	`safe { }` blocks, `?` operator	No built-in null safety
Concurrency	`spawn { }`, `forge fn() { }`	`asyncio`, threading
Package install	`forge install <url>`	`pip install <pkg>`
Test runner	Built-in: `@test` + `forge test`	`pytest` (external)
Formatter	Built-in: `forge fmt`	`black` (external)
REPL	Built-in: `forge repl`	Built-in: `python`
Database access	Built-in: `db.query()`, `pg.query()`	`sqlite3`, `psycopg2` (stdlib/external)
Retry logic	`retry 3 times { }`	Manual loop or `tenacity` library
AI integration	Built-in: `ask "prompt"`	`openai` library (external)
Learning mode	Built-in: `forge learn`	None built-in
Semicolons	Not required (newline-based)	Not required (newline-based)

6.6.2 Table F-2: Forge vs. JavaScript/Node.js

Feature	Forge	JavaScript/Node.js
Variable declaration	`let x = 5`	`let x = 5` / `const x = 5`

Feature	Forge	JavaScript/Node.js
Immutability	`let` is immutable, `let mut` is mutable	`const` is shallow-immutable
Function definition	`fn greet(name) { }`	`function greet(name) { }`
Arrow functions	`(x) => x * 2`	`(x) => x * 2`
String interpolation	`"Hello, {name}"`	`` `Hello, ${name}` ``
Destructuring	`unpack {a, b} from obj`	`const {a, b} = obj`
Spread operator	`...args`	`...args`
Pipeline operator	`data |> transform`	Stage 2 proposal (TC39)
HTTP server	`@server` + `@get fn()`	Express/Fastify (external)
HTTP client	`fetch("url")` (built-in)	`fetch()` (built-in, Node 18+)
Async/await	`async fn` / `forge fn`, `await` / `hold`	`async function`, `await`
Error handling	`try/catch` + `must` + `safe` + `?`	`try/catch`
Null safety	`safe { }` blocks	Optional chaining `?.`
Type system	Gradual (optional annotations)	None (use TypeScript)
Pattern matching	`match value { pattern => body }`	None (proposal stage)
Package manager	`forge install`	`npm install`
Test framework	Built-in `@test`	Jest/Vitest (external)
Formatter	Built-in `forge fmt`	Prettier (external)
Module system	`import "file.fg"`	`import`/`require`
Database	Built-in SQLite + PostgreSQL	`better-sqlite3`, `pg` (external)
Integers	True 64-bit integers	`Number` (64-bit float) or `BigInt`
Counted loops	`repeat 5 times { }`	`for (let i=0; i<5; i++) { }`
AI built-in	`ask "prompt"`	None built-in
Compilation	Single binary, instant start	V8 JIT, ~50ms startup

6.6.3 Table F-3: Forge vs. Go

Feature	Forge	Go
Variable declaration	`let x = 5`	`x := 5` / `var x int = 5`
Function definition	`fn add(a, b) { return a + b }`	`func add(a, b int) int { return a + b }`
Type system	Gradual (optional)	Static (required)
Error handling	`try/catch`, `must`, `safe`, `Result`	`if err != nil { return err }`
HTTP server	`@server` + decorator-based routing	`http.HandleFunc` / gorilla mux
HTTP client	`fetch("url")` (one function)	`http.Get()` + response body handling
Concurrency	`spawn { }`	`go func() { }()`
Generics	Dynamic typing	Generics (Go 1.18+)
Compilation	Interpreted (or bytecode VM)	Compiled to native binary
Performance	Interpreted speed	Near-C performance
String interpolation	`"Hello, {name}"`	`fmt.Sprintf("Hello, %s", name)`

Feature	Forge	Go
Pattern matching	`match` / `when` guards	`switch` statement
Null handling	`safe { }`, `None`/`Some`	Nil checks
Package management	`forge install`	`go get` + `go mod`
Test framework	Built-in `@test`	Built-in `testing` package
REPL	Built-in	None
AI integration	Built-in `ask`	External library
Learning curve	Low (designed for readability)	Low (25 keywords)
Binary size	~15MB (Rust compiled)	~5-10MB per binary
Ecosystem maturity	New, growing	Mature, large ecosystem

6.6.4 Table F-4: Forge vs. Rust

Feature	Forge	Rust
Type system	Gradual, dynamic	Static, strict
Memory management	GC (VM) / clone-based (interpreter)	Ownership + borrowing
Error handling	`try/catch`, `must`, `safe`	`Result<T,E>`, `?` operator
Null handling	`Null` value + `safe` blocks	`Option<T>`, no null
Compilation speed	Instant (interpreted)	Slow (full compile)
Runtime performance	Interpreted speed	Native speed
Learning curve	Low	High
String interpolation	`"Hello, {name}"`	`format!("Hello, {name}")`
HTTP server	3 lines (`@server`, `@get fn`)	~30 lines (axum setup)
HTTP client	`fetch("url")`	`reqwest::get("url").await?`
Concurrency	`spawn { }`	`tokio::spawn(async { })`
Pattern matching	`match value { }`	`match value { }`
Closures	`(x) => x * 2` (captures env)	`|x| x * 2` (lifetime-tracked)
Package manager	`forge install`	`cargo add`
Unsafe code	Zero `unsafe` in Forge itself	Powerful but dangerous

6.6.5 Table F-5: Forge vs. Ruby

Feature	Forge	Ruby
Variable declaration	`let x = 5` / `set x to 5`	`x = 5`
Function definition	`fn greet(n) { }` / `define greet(n) { }`	`def greet(n) ... end`
Blocks	`{ }` braces	`do...end` or `{ }`
String interpolation	`"Hello, {name}"`	`"Hello, #{name}"`
Functional methods	`map`, `filter`, `reduce`	`map`, `select`, `reduce`

Feature	Forge	Ruby
HTTP server	Built-in `@server` decorators	Sinatra/Rails (external)
HTTP client	Built-in `fetch()`	`net/http` or `httparty`
Type annotations	Optional: `let x: Int = 5`	None (use Sorbet)
Pattern matching	`match value { }`	`case value in` (Ruby 3.0+)
Error handling	`try/catch`, `must`, `safe`	`begin/rescue/end`
Test framework	Built-in `@test`	RSpec / Minitest
REPL	Built-in `forge repl`	Built-in `irb`
Null safety	`safe { }` blocks	`&.` safe navigation
Natural syntax	`say`, `define`, `otherwise`	Ruby is already English-like
AI integration	Built-in `ask "prompt"`	None built-in

6.6.6 Lines of Code Comparison: Common Tasks

HTTP Server with JSON Endpoint

Forge (8 lines):

```
@server(port: 3000)

@get("/hello/:name")
fn hello(params) {
    return {
        message: "Hello, {params.name}!"
    }
}
```

Python + Flask (9 lines):

```python
from flask import Flask, jsonify
app = Flask(__name__)

@app.route("/hello/<name>")
def hello(name):
    return jsonify(message=f"Hello, {name}!")

if __name__ == "__main__":
    app.run(port=3000)
```

Go (19 lines):

```go
package main
import (
    "encoding/json"
    "fmt"
```

```
    "net/http"
)
func hello(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Path[len("/hello/"):]
    json.NewEncoder(w).Encode(map[string]string{
        "message": fmt.Sprintf("Hello, %s!", name),
    })
}
func main() {
    http.HandleFunc("/hello/", hello)
    http.ListenAndServe(":3000", nil)
}
```

Read File + Process Lines

Forge (4 lines):

```
let lines = split(fs.read("data.txt"), "\n")
let non_empty = filter(lines, (l) => len(l) > 0)
say "Found {len(non_empty)} non-empty lines"
```

Python (4 lines):

```
with open("data.txt") as f:
    lines = f.readlines()
non_empty = [l for l in lines if l.strip()]
print(f"Found {len(non_empty)} non-empty lines")
```

JavaScript (5 lines):

```
const fs = require("fs");
const lines = fs.readFileSync("data.txt", "utf8").split("\n");
const nonEmpty = lines.filter((l) => l.trim().length > 0);
console.log(`Found ${nonEmpty.length} non-empty lines`);
```

Database Query

Forge (4 lines):

```
let conn = db.open("app.db")
let users = db.query(conn, "SELECT name, age FROM users WHERE age > 21")
say users
db.close(conn)
```

Python (7 lines):

```
import sqlite3
conn = sqlite3.connect("app.db")
cursor = conn.cursor()
users = cursor.execute("SELECT name, age FROM users WHERE age > 21").fetchall()
print(users)
conn.close()
```

Go (15 lines):

```
db, _ := sql.Open("sqlite3", "app.db")
defer db.Close()
rows, _ := db.Query("SELECT name, age FROM users WHERE age > 21")
defer rows.Close()
for rows.Next() {
    var name string; var age int
    rows.Scan(&name, &age)
    fmt.Printf("%s: %d\n", name, age)
}
```

6.7 Appendix G: Project Statistics and Credits

6.7.1 Codebase Statistics

Metric	Value
Total Rust source lines	~26,000
Total source files	56
Rust tests	488
Forge integration tests	334
Unsafe blocks	0
Keywords recognized	80+
Built-in functions	230+
Standard library modules	16
CLI commands	13
Interactive tutorial lessons	30
Example programs	12

6.7.2 Largest Source Files

File	Lines	Component
`src/interpreter/mod.rs`	8,153	Tree-walk interpreter
`src/vm/machine.rs`	2,483	Bytecode VM engine
`src/parser/parser.rs`	1,851	Recursive descent parser
`src/vm/compiler.rs`	927	AST to bytecode compiler
`src/lexer/lexer.rs`	606	Lexer / tokenizer
`src/learn.rs`	520	Interactive tutorials
`src/runtime/server.rs`	352	HTTP server (axum)
`src/parser/ast.rs`	335	AST definitions
`src/repl/mod.rs`	299	Interactive REPL
`src/main.rs`	293	CLI entry point

6.7.3 Technology Stack

Component	Technology	Purpose
Language	Rust	Core implementation
CLI framework	clap	Argument parsing, subcommands
HTTP server	axum	Async HTTP server runtime
HTTP client	reqwest + rustls	HTTPS requests (pure Rust TLS)
Async runtime	tokio	Async I/O, task scheduling
SQLite	rusqlite	Embedded database support
PostgreSQL	tokio-postgres	PostgreSQL client
Error reporting	ariadne	Source-mapped error diagnostics
REPL	rustyline	Line editing, history, completion
Ordered maps	indexmap	Insertion-order-preserving maps
JSON	serde + serde_json	JSON parsing and serialization
TOML	toml	Manifest file parsing
CORS	tower-http	Cross-Origin Resource Sharing
Regex	regex	Regular expression engine
UUID	uuid	UUID v4 generation
Crypto	sha2, md5, base64	Cryptographic hash functions
CSV	csv	CSV parsing and writing

6.7.4 Design Principles

1. **Zero unsafe code** — The entire codebase uses safe Rust. No `unsafe` blocks, no raw pointer manipulation, no undefined behavior.
2. **Batteries included** — HTTP client and server, database access, cryptography, file I/O, regex, CSV, JSON, terminal UI, and AI integration ship with the language. No dependency hunting for common tasks.
3. **Dual syntax** — Every construct can be expressed in either classic programming syntax or natural English-like syntax. Both are first-class; neither is deprecated or secondary.

4. **Progressive complexity** — Simple programs are simple. `say "hello"` is a complete program. Advanced features (async, HTTP servers, database queries) are available but never required.
5. **Developer experience first** — Built-in formatter, test runner, REPL, tutorials, LSP, and project scaffolding. The toolchain is complete from day one.

6.7.5 Version History

Version	Milestone
0.1.0	Initial release: lexer, parser, interpreter, 8 stdlib modules
0.2.0	Bytecode VM, mark-sweep GC, 15 stdlib modules, LSP, tutorials, AI chat, formatter, test runner, package manager
0.3.0	73 new functions, 16 modules, GenZ debug kit, NPC module, structured errors, 30 tutorials, 822 tests

6.7.6 Acknowledgments

Forge is written by **Archith Rapaka**. The language draws inspiration from:

- **Python** for its readability and gentle learning curve
- **Go** for its simplicity and built-in tooling philosophy
- **Rust** for its safety guarantees and error handling patterns
- **Ruby** for its expressive, human-friendly syntax
- **JavaScript** for its object literal syntax and async patterns
- **Lua** for its register-based VM design
- **Swift** for its optional handling and guard statements

Special thanks to the Rust ecosystem for the excellent crates that power Forge's runtime: `tokio`, `axum`, `reqwest`, `rusqlite`, `ariadne`, `clap`, `rustyline`, `serde`, and `indexmap`.

This concludes Part IV and the Appendices of Programming Forge. *The complete source code is available at the project repository. Contributions, bug reports, and feature requests are welcome.*