The Haira Programming Language

Language Specification

Version 0.4.0

Haira Team

March 12, 2026

© 2026 Haira Team

This specification is licensed under CC BY 4.0.

Document Version: 0.4.0
Language Version: 0.4.0
Last Updated: March 12, 2026

Contents

Preface

HAIRA is a statically-typed, compiled programming language for building AI agents and workflows. It combines the simplicity of Go, the safety of Rust, and the expressiveness of modern functional languages, while making agents, tools, and workflows first-class language constructs—not library imports.

Design Philosophy

1. **Agents and workflows are primitives** – **`provider`**, **`tool`**, **`agent`**, and **`workflow`** are language keywords, not framework abstractions.
2. **Type-safe orchestration** – Agent inputs, outputs, and tool schemas are verified at compile time.
3. **Go-style simplicity** – Familiar error handling, explicit imports, and straightforward syntax.
4. **Zero boilerplate** – No framework setup, no dependency injection, no configuration files.
5. **Native performance** – Compiles to Go, then to standalone native binaries via `go build`.

Target Domains

HAIRA replaces the entire agentic stack:

- **Python + LangChain/LangGraph** – for code-first agent builders
- **n8n / Make / Zapier** – for workflow automation
- **CrewAI / AutoGen** – for multi-agent systems
- **Custom chatbot backends** – REST and WebSocket with streaming
- **General-purpose programming** – CLI tools, web services, data pipelines

Release History

Release	Date	Features
v0.1	Jan 2026	Lexer, parser, basic types, expressions, statements
v0.2	Jan 2026	Functions, error handling, providers, tools, agents, workflows
v0.3	Feb 2026	Modules, stdlib, MCP, streaming, generative UI, Workers target

Release	**Date**	**Features**
v0.4	Mar 2026	Eval framework, tool hooks, verify blocks, agent strategy, cross-harness export, stdlib agent templates, LSP, formatter

Part I

Core Language

Chapter 1

Overview

Haira is a statically-typed, compiled programming language for building AI agents and workflows. It combines the simplicity of Go, the safety of Rust, and purpose-built primitives for agentic orchestration — agents, tools, workflows, and providers are first-class language constructs, not library imports.

Haira compiles to native binaries via Cranelift, with no runtime dependencies beyond the standard library.

1.1 Hello, Agent

```
import "io"

provider local {
    backend: "ollama"
    host: "localhost:11434"
    model: "llama3:8b"
}

agent Greeter {
    provider: local
    system: "You are a friendly greeter. Keep it short."
}

fn main() {
    reply, err = Greeter.ask("Say hello to the world!")
    if err != nil {
        io.println("Error: " + err.message)
        return
    }
    io.println(reply)
}
```

To compile and run:

```
$ haira build hello.haira
$ ./hello
Hello, world! Great to meet you!
```

1.2 What Haira Replaces

HAIRA is designed to replace the fragmented toolchain currently used to build agentic systems:

Current Stack	Haira Equivalent
Python + LangChain/LangGraph	`agent` + `tool` + `workflow`
n8n / Make / Zapier	`workflow` with `@webhook`, `@cron` triggers
CrewAI / AutoGen	Multi-agent `workflow` composition
FastAPI / Flask	`http.Server()` with typed endpoints
`.env` files + API configs	`provider` blocks

Table 1.1: Haira replaces the fragmented agentic stack

1.3 Language Characteristics

1.3.1 Four Agentic Primitives

HAIRA adds four keywords to a general-purpose language:

- **`provider`** — Configure LLM backends (Anthropic, OpenAI, Ollama, local models)
- **`tool`** — Typed functions with LLM-visible descriptions; the compiler auto-generates JSON schemas
- **`agent`** — LLM-powered entities with a model, system prompt, tools, and optional memory
- **`workflow`** — Composable pipelines with triggers, typed I/O, and native concurrency

1.3.2 Explicit Imports

All dependencies are explicitly declared at the top of each file:

```
import "io"
import "json"
import "tools/http"
import "tools/postgres"
```

1.3.3 Go-Style Error Handling

Functions that can fail return a tuple of result and error:

```
import "io"
import "fs"

fn main() {
    content, err = fs.read_file("config.json")
    if err != nil {
        io.println("Error: " + err.message)
        return
    }
    io.println(content)
}
```

1.3.4 Agents and Tools

Agents are declared with a model, system prompt, and tools. Tools are typed functions with descriptions:

```
import "io"
import "tools/http"

provider anthropic {
    api_key: env("ANTHROPIC_API_KEY")
    model: "claude-sonnet-4-20250514"
}

tool search(query: string) -> [Result] {
    """Search the web for information"""

    resp, err = http.get("https://api.search.com?q={query}")
    if err != nil { return [], err }
    return json.decode(resp.body, [Result])
}

agent Researcher {
    provider: anthropic
    system: "You are a thorough researcher. Always cite
       sources."
    tools: [search]
    temperature: 0.2
}

fn main() {
    answer, err = Researcher.ask("What is quantum computing?")
    if err != nil {
        io.println("Error: " + err.message)
        return
    }
    io.println(answer)
}
```

1.3.5 Workflows with Triggers

Workflows are functions with triggers. They can be exposed as webhooks, scheduled with cron, or triggered by events:

```
@webhook("/summarize")
workflow Summarizer(url: string) -> Summary {
    content = fetch_page(url)
    summary = Researcher.ask("Summarize this: {content}")
    return Summary{ text: summary, word_count: summary.len() }
}

fn main() {
    server = http.Server([Summarizer])
    io.println("Running on :8080")
    server.listen(8080)
}
```

1.3.6 Streaming and Conversations

Agents support streaming responses and session-based memory for chatbot use cases:

```
agent Assistant {
    provider: anthropic
    system: "You are a helpful assistant."
    memory: conversation(max_turns: 50)
}

@websocket("/chat")
workflow Chat(message: string, session_id: string) ->
    stream<string> {
    return Assistant.stream(message, session: session_id)
}
```

1.4 Type System

Haira uses structural typing with type inference for local variables:

```
// Type inferred as int
count = 42

// Explicit type annotation
name: string = "Haira"

// Struct definition
struct Point {
    x: float
    y: float
}

// Option type (nullable)
maybe_value: int? = nil

// Arrays and maps
numbers: []int = [1, 2, 3]
scores: [string:int] = ["alice": 100, "bob": 95]
```

1.5 Control Flow

```
import "io"

fn main() {
    x = 10
    if x > 5 {
        io.println("Large")
    } else {
        io.println("Small")
    }

    for i in 0..5 {
        io.println(i)
    }

    status = 200
    match status {
        200 => io.println("OK")
        404 => io.println("Not Found")
        _ => io.println("Unknown")
    }
}
```

1.6 Functions

Functions are declared with **fn**:

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

fn divide(a: int, b: int) -> (int, Error?) {
    if b == 0 {
        return 0, Error{message: "division by zero"}
    }
    return a / b, nil
}
```

1.7 Concurrency

Haira provides Go-style concurrency with **spawn** and channels:

```
import "io"

fn main() {
    ch = chan<int>(10)

    spawn {
        for i in 0..5 {
            ch <- i
        }
        close(ch)
    }

    for value in ch {
        io.println(value)
    }
}
```

Inside workflows, **spawn** blocks run steps in parallel:

```
@manual
workflow ParallelReview(article: Article) -> [Review] {
    reviews = spawn {
        Reviewer.run(ReviewRequest{ article: article, focus:
            "accuracy" })
        Reviewer.run(ReviewRequest{ article: article, focus: "clarity" })
        Reviewer.run(ReviewRequest{ article: article, focus: "style" })
    }
    return reviews
}
```

1.8 Pipe Operator

The pipe operator `|>` enables functional composition:

```
import "io"
import "string"
import "array"

fn main() {
    result = "  hello, world  "
        |> string.trim
        |> string.to_upper
        |> string.split(", ")
        |> array.first

    io.println(result)  // "HELLO"
}
```

1.9 A Complete Example

A multi-agent workflow that researches a topic, writes an article, and reviews it:

```
import "io"

provider anthropic {
    api_key: env("ANTHROPIC_API_KEY")
    model: "claude-sonnet-4-20250514"
}

struct Article { title: string, body: string, sources: [string] }
struct Review { aspect: string, score: float, notes: string }

tool search(query: string) -> [string] {
    """Search the web and return relevant URLs"""

    resp, err = http.get("https://api.search.com?q={query}")
    if err != nil { return [], err }
    return json.decode(resp.body, [string])
}

agent Researcher {
    provider: anthropic
    system: "You are a thorough researcher."
    tools: [search]
    temperature: 0.2
}

agent Writer {
    provider: anthropic
    system: "You are a clear technical writer."
    temperature: 0.7
}

agent Reviewer {
    provider: anthropic
    system: "You are a critical but constructive editor."
    temperature: 0.1
}

@manual
workflow WriteArticle(topic: string) -> Article {
    research = Researcher.ask("Research this topic thoroughly:
       {topic}")

    draft: Article, err = Writer.run(
        { research: research, topic: topic }
    ) -> Article
    if err != nil { return Article{}, err }

    reviews = spawn {
        Reviewer.ask("Review for accuracy: {draft.body}")
        Reviewer.ask("Review for clarity: {draft.body}")
    }

    io.println("Reviews complete: {reviews.len()}")
    return draft
}

fn main() {
    article, err = WriteArticle("The future of agentic
       programming")
    if err != nil {
        io.println("Failed: " + err.message)
        return
    }
    io.println(article.title)
    io.println(article.body)
}
```

1.10 Implementation Status

This specification documents the complete HAIRA language. Implementation proceeds incrementally:

Feature	**Chapter**	**Target Release**
Lexer & Parser	2–3	v0.1
Expressions & Statements	4–5	v0.2
Functions & Error Handling	6–7	v0.3
Concurrency	8	v0.4
Modules & Stdlib	9–10	v0.5
Providers & Tools	11–12	v0.6
Agents	13	v0.7
Workflows & Chatbot Patterns	14–15	v0.8
Full Grammar & Compiler	16–17	v1.0

Table 1.2: Implementation roadmap by chapter

Chapter 2

Lexical Structure

This chapter defines the lexical grammar of HAIRA: the rules for converting source text into tokens.

2.1 Source Encoding

HAIRA source files are encoded in UTF-8. The file extension is `.haira`.

```
hello.haira
utils/string_helpers.haira
```

2.2 Line Structure

HAIRA is a line-oriented language. Statements are terminated by newlines, not semicolons.

```
x = 1
y = 2
z = x + y
```

2.2.1 Line Continuation

Lines can be continued with a trailing backslash or when the line ends with an operator or open bracket:

```
// Explicit continuation
long_result = very_long_function_name(arg1, arg2) \
    + another_function(arg3)

// Implicit continuation (ends with operator)
total = first_value +
    second_value +
    third_value

// Implicit continuation (open bracket)
users = [
    "alice",
    "bob",
    "charlie"
]
```

2.3 Comments

2.3.1 Single-Line Comments

```
// This is a single-line comment
x = 42  // Inline comment
```

2.3.2 Multi-Line Comments

```
/*
    This is a multi-line comment.
    It can span multiple lines.
*/
```

Multi-line comments can be nested:

```
/*
    Outer comment
    /* Nested comment */
    Still in outer comment
*/
```

2.3.3 Documentation Comments

```
/// This is a documentation comment for the following item.
/// It can span multiple lines.
fn calculate_area(width: float, height: float) -> float {
    return width * height
}
```

2.4 Tokens

The lexer produces the following token types:

1. Keywords
2. Identifiers
3. Literals (numbers, strings, booleans)
4. Operators
5. Delimiters

2.5 Keywords

The following identifiers are reserved as keywords:

General-purpose keywords				
`and`	`as`	`break`	`catch`	`chan`
`continue`	`defer`	`else`	`enum`	`export`
`false`	`fn`	`for`	`from`	`if`
`impl`	`import`	`in`	`match`	`nil`
`not`	`or`	`pub`	`return`	`select`
`spawn`	`struct`	`trait`	`true`	`try`
`type`	`while`			
Agentic keywords				
`agent`	`provider`	`tool`	`workflow`	

Table 2.1: Reserved keywords

Note

The four agentic keywords (**`agent`**, **`provider`**, **`tool`**, **`workflow`**) are first-class language constructs for building AI-powered systems. See Chapters 11–14.

2.6 Identifiers

Identifiers name variables, functions, types, and modules.

```
identifier      = letter { letter | digit | "_" } ;
letter          = "a".."z" | "A".."Z" | "_" ;
digit           = "0".."9" ;
```

Valid identifiers:

```
x
name
userName
user_name
_private
Point3D
MAX_SIZE
```

Invalid identifiers:

```
3d_point     // Cannot start with digit
my-name      // Hyphens not allowed
class        // Reserved keyword (if added)
```

2.6.1 Naming Conventions

2.7 Literals

2.7.1 Integer Literals

```
integer_literal = decimal_literal
                | hex_literal
```

Entity	Convention
Variables	snake_case
Functions	snake_case
Types/Structs	PascalCase
Constants	SCREAMING_SNAKE_CASE
Modules	snake_case

Table 2.2: Naming conventions

```
                | octal_literal
                | binary_literal ;

decimal_literal = digit { digit | "_" } ;
hex_literal     = "0" ("x"|"X") hex_digit { hex_digit | "_" } ;
octal_literal   = "0" ("o"|"O") octal_digit { octal_digit | "_"
   } ;
binary_literal  = "0" ("b"|"B") binary_digit { binary_digit |
   "_" } ;

hex_digit       = digit | "a".."f" | "A".."F" ;
octal_digit     = "0".."7" ;
binary_digit    = "0" | "1" ;
```

Examples:

```
decimal = 42
with_underscores = 1_000_000
hex = 0xFF
octal = 0o755
binary = 0b1010_1100
```

Note

All integer literals are of type `int` (64-bit signed).

2.7.2 Floating-Point Literals

```
float_literal = decimal_literal "." decimal_literal [ exponent ]
              | decimal_literal exponent ;

exponent = ("e"|"E") ["+"|"-"] decimal_literal ;
```

Examples:

```
pi = 3.14159
scientific = 6.022e23
negative_exp = 1.6e-19
```

2.7.3 Boolean Literals

```
is_valid = true
is_empty = false
```

2.7.4 String Literals

```
string_literal = '"' { string_char } '"' ;
string_char    = any_char - ('"' | '\' | newline)
               | escape_sequence ;
```

Escape sequences:

Sequence	Meaning
`\n`	Newline (LF)
`\r`	Carriage return (CR)
`\t`	Horizontal tab
`\\`	Backslash
`\"`	Double quote
`\0`	Null character
`\xNN`	Hex byte (00-FF)
`\u{NNNN}`	Unicode code point

Table 2.3: String escape sequences

Examples:

```
simple = "Hello, World!"
with_escapes = "Line 1\nLine 2"
unicode = "Hello \u{1F44B}"  // Wave emoji
```

2.7.5 String Interpolation

Strings can contain interpolated expressions using `${...}`:

```
name = "Alice"
age = 30
greeting = "Hello, ${name}! You are ${age} years old."
```

Implementation Note

String interpolation is desugared to concatenation during parsing:

```
greeting = "Hello, " + name + "! You are " +
    to_string(age) + " years old."
```

2.7.6 Raw Strings

Raw strings preserve backslashes literally:

```
regex_pattern = r"^\d{3}-\d{4}$"
```

```
windows_path = r"C:\Users\Alice\Documents"
```

2.7.7 Triple-Quoted Strings

Triple-quoted strings use `"""..."""` for multi-line content:

```
query = """
    SELECT *
    FROM users
    WHERE active = true
"""
```

The leading whitespace common to all lines is stripped.
Triple-quoted strings are also used for:

- Tool descriptions (mandatory, see Chapter 12)
- Agent system prompts (optional, for long prompts, see Chapter 13)

```
tool search(query: string) -> [Result] {
    """Search the web for information"""
    // ...
}

agent Assistant {
    provider: anthropic
    system: """
        You are a helpful assistant.
        Always be polite and concise.
    """
}
```

2.7.8 Nil Literal

```
empty: User? = nil
```

2.8 Operators

2.8.1 Arithmetic Operators

Operator	Description
`+`	Addition
`-`	Subtraction (or unary negation)
`*`	Multiplication
`/`	Division
`%`	Modulo (remainder)

Operator	Description
`==`	Equal
`!=`	Not equal
`<`	Less than
`>`	Greater than
`<=`	Less than or equal
`>=`	Greater than or equal

2.8.2 Comparison Operators

2.8.3 Logical Operators

Operator	Description
`and`, `&&`	Logical AND
`or`, `||`	Logical OR
`not`, `!`	Logical NOT

2.8.4 Assignment Operators

Operator	Description
`=`	Assignment
`+=`	Add and assign
`-=`	Subtract and assign
`*=`	Multiply and assign
`/=`	Divide and assign

2.8.5 Bitwise Operators

2.8.6 Bitwise Assignment Operators

2.8.7 Other Operators

2.9 Delimiters

2.10 Whitespace

Whitespace (spaces, tabs) is used to separate tokens but is otherwise insignificant, except for indentation in multi-line strings.

```
x=1+2          // Valid
x = 1 + 2      // Also valid, preferred for readability
```

2.11 Lexical Grammar Summary

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT (complement)
«	Left shift
»	Right shift (arithmetic for signed, logical for unsigned)
»>	Logical right shift (always fills with zeros)

Operator	Description
&=	Bitwise AND and assign
|=	Bitwise OR and assign
^=	Bitwise XOR and assign
«=	Left shift and assign
»=	Right shift and assign
»>=	Logical right shift and assign

```
token = keyword
      | identifier
      | integer_literal
      | float_literal
      | string_literal
      | operator
      | delimiter ;

keyword = "agent" | "and" | "assert" | "async" | "break"
        | "catch" | "const" | "continue" | "default" | "defer"
        | "else" | "enum" | "err" | "errdefer" | "export"
        | "false" | "fn" | "for" | "from" | "if" | "impl"
        | "import" | "in" | "let" | "match" | "nil" | "none"
        | "not" | "ok" | "oncancel" | "onerror" | "onsuccess"
        | "or" | "orelse" | "provider" | "pub" | "return"
        | "select" | "some" | "spawn" | "step" | "struct"
        | "test" | "tool" | "trait" | "true" | "try" | "type"
        | "while" | "workflow" ;

identifier = letter { letter | digit | "_" } ;

operator = "+" | "-" | "*" | "/" | "%"
         | "==" | "!=" | "<" | ">" | "<=" | ">="
         | "and" | "or" | "not" | "&&" | "||" | "!"
         | "&" | "|" | "^" | "~" | "<<" | ">>" | ">>>"
         | "=" | "+=" | "-=" | "*=" | "/="
         | "&=" | "|=" | "^=" | "<<=" | ">>=" | ">>>="
         | "|>" | ".." | "..=" | "<-" | "->" | "=>"
         | "@" ;

delimiter = "(" | ")" | "[" | "]" | "{" | "}"
          | "," | ":" | "." | "?" ;
```

Operator	Description
`|>`	Pipe (function composition)
`..`	Range (exclusive end)
`..=`	Range (inclusive end)
`<-`	Channel send
`->`	Function return type
`=>`	Match arm
`@`	Decorator (workflow triggers)

Delimiter	Usage
`( )`	Grouping, function calls, tuples
`[ ]`	Array literals, indexing
`{ }`	Blocks, struct literals, maps
`,`	Separator
`:`	Type annotations, map entries
`.`	Member access
`?`	Optional type marker

Chapter 3

Types

HAIRA is a statically-typed language with structural typing and type inference for local variables. All types are known at compile time.

3.1 Type System Overview

- **Static typing** – All types resolved at compile time
- **Structural typing** – Type compatibility based on structure, not name
- **Type inference** – Local variables infer types from initializers
- **Explicit annotations** – Required for function signatures
- **No null** – Option types (`T?`) replace null pointers

3.2 Primitive Types

3.2.1 Integer Types

Type	Size	Range
`int`	64 bits	-2^{63} to $2^{63}-1$
`i8`	8 bits	-128 to 127
`i16`	16 bits	-32768 to 32767
`i32`	32 bits	-2^{31} to $2^{31}-1$
`i64`	64 bits	-2^{63} to $2^{63}-1$
`u8`	8 bits	0 to 255
`u16`	16 bits	0 to 65535
`u32`	32 bits	0 to $2^{32}-1$
`u64`	64 bits	0 to $2^{64}-1$

Table 3.1: Integer types

```
count: int = 42
byte: u8 = 255
offset: i32 = -100
```

Note

The default `int` is an alias for `i64`. Integer literals without explicit type default to `int`.

3.2.2 Floating-Point Types

Type	Size	Precision
`float`	64 bits	IEEE 754 double
`f32`	32 bits	IEEE 754 single
`f64`	64 bits	IEEE 754 double

Table 3.2: Floating-point types

```
pi: float = 3.14159
temperature: f32 = 98.6
```

3.2.3 Boolean Type

```
is_valid: bool = true
is_empty: bool = false
```

Boolean values are used in conditionals and logical expressions. Only `true` and `false` are valid boolean values—there is no implicit conversion from other types.

3.2.4 String Type

Strings are immutable sequences of UTF-8 encoded bytes:

```
name: string = "Haira"
greeting: string = "Hello, " + name
```

String operations:

```
import "string"

s = "Hello, World!"
length = string.len(s)         // 13
upper = string.to_upper(s)     // "HELLO, WORLD!"
parts = string.split(s, ", ") // ["Hello", "World!"]
```

3.3 Composite Types

3.3.1 Arrays

Arrays are ordered, homogeneous, dynamically-sized collections:

```
// Array literal
numbers: []int = [1, 2, 3, 4, 5]
```

```

// Empty array
empty: []string = []

// Array operations
import "array"

first = numbers[0]              // 1
length = array.len(numbers)     // 5
numbers = array.push(numbers, 6)
```

```
array_type = "[" "]" type ;
```

3.3.2 Maps

Maps are unordered key-value collections:

```
// Map literal
scores: [string:int] = [
    "alice": 100,
    "bob": 95,
    "charlie": 87
]

// Empty map
empty: [string:User] = [:]

// Map operations
alice_score = scores["alice"]     // 100
scores["david"] = 92              // Insert
```

```
map_type = "[" type ":" type "]" ;
```

Note

Map keys must be comparable types: primitives, strings, or structs containing only comparable fields.

3.3.3 Tuples

Tuples are fixed-size, heterogeneous collections:

```
// Tuple type
point: (int, int) = (10, 20)

// Accessing elements
x = point.0   // 10
y = point.1   // 20

// Multiple return values use tuples
fn divide(a: int, b: int) -> (int, int) {
    return a / b, a % b
}

quotient, remainder = divide(17, 5)
```

```
tuple_type = "(" type { "," type } ")" ;
```

3.4 Option Types

Option types represent values that may or may not exist, replacing null pointers:

```
// Optional value
maybe_name: string? = nil
maybe_age: int? = 25

// Checking for nil
if maybe_name != nil {
    io.println(maybe_name)
}

// Default value with 'or'
name = maybe_name or "Unknown"
```

```
option_type = type "?" ;
```

3.4.1 Unwrapping Options

```
value: int? = get_value()

// Safe unwrap with default
result = value or 0

// Conditional unwrap
if value != nil {
    // value is automatically unwrapped here
    io.println(value)
}

// Match on option
match value {
    nil => io.println("No value")
    v => io.println("Got: " + to_string(v))
}
```

3.5 Struct Types

Structs are named product types with named fields:

```
struct User {
    id: int
    name: string
    email: string
    active: bool
}

// Creating a struct
user = User{
    id: 1,
    name: "Alice",
    email: "alice@example.com",
    active: true
}

// Accessing fields
io.println(user.name)

// Updating fields (structs are mutable by default)
user.active = false
```

3.5.1 Struct Methods

Methods can be defined on structs:

```
struct Rectangle {
    width: float
    height: float
}

Rectangle.area() -> float {
    return self.width * self.height
}

Rectangle.perimeter() -> float {
    return 2 * (self.width + self.height)
}

// Usage
rect = Rectangle{width = 10.0, height = 5.0}
io.println(rect.area())       // 50.0
io.println(rect.perimeter())  // 30.0
```

3.5.2 Nested Structs

```
struct Address {
    street: string
    city: string
    country: string
}

struct Person {
    name: string
    address: Address
}

```

```
person = Person{
    name: "Bob",
    address: Address{
        street: "123 Main St",
        city: "New York",
        country: "USA"
    }
}

io.println(person.address.city)  // "New York"
```

3.6 Enum Types

Enums define a type with a fixed set of named values:

```
enum Color {
    Red,
    Green,
    Blue
}

enum Status {
    Pending,
    Active,
    Completed,
    Failed
}

// Usage
color: Color = Color.Red
status: Status = Status.Active

match color {
    Color.Red => io.println("Stop")
    Color.Green => io.println("Go")
    Color.Blue => io.println("Info")
}
```

3.6.1 Enums with Associated Data

Enums can carry associated data (tagged unions):

```
enum Result<T> {
    Ok(T),
    Err(Error)
}

enum Message {
    Text(string),
    Number(int),
    Pair(int, int)
}

// Usage
msg: Message = Message.Text("Hello")
```

```

match msg {
    Message.Text(s) => io.println("Text: " + s)
    Message.Number(n) => io.println("Number: " + to_string(n))
    Message.Pair(a, b) => io.println("Pair: " + to_string(a) +
        ", " + to_string(b))
}
```

3.7 Type Aliases

Type aliases create alternative names for existing types:

```
type UserId = int
type UserMap = [string:User]
type Handler = fn(Request) -> Response

// Usage
id: UserId = 42
users: UserMap = [:]
```

3.8 Function Types

Functions are first-class values with types:

```
// Function type syntax
type Comparator = fn(int, int) -> bool
type Transformer = fn(string) -> string
type Callback = fn()

// Higher-order functions
fn apply(f: fn(int) -> int, x: int) -> int {
    return f(x)
}

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

result = apply(double, 21)  // 42
```

3.9 Generic Types

Generic types parameterize over other types:

```
// Generic struct
struct Pair<T, U> {
    first: T
    second: U
}

// Usage
pair: Pair<int, string> = Pair{first: 1, second: "one"}
```

```

// Generic function
fn swap<T, U>(p: Pair<T, U>) -> Pair<U, T> {
    return Pair{first: p.second, second: p.first}
}

swapped = swap(pair)  // Pair<string, int>
```

3.9.1 Type Constraints

Generics can be constrained to types implementing certain behaviors:

```
// Constraint definition
constraint Comparable {
    fn compare(other: Self) -> int
}

// Constrained generic
fn max<T: Comparable>(a: T, b: T) -> T {
    if a.compare(b) > 0 {
        return a
    }
    return b
}

// Multiple constraints
fn process<T: Comparable + Printable>(value: T) {
    io.println(value.to_string())
}
```

3.10 Stream Types

The `stream<T>` type represents a lazy, asynchronous sequence of values produced over time. Streams are the primary mechanism for handling token-by-token output from agents and for streaming responses over WebSocket connections.

```
// Stream of string tokens from an agent
for chunk in Assistant.stream("Tell me a story") {
    io.print(chunk)
}
```

3.10.1 Stream Type Syntax

```
stream_type = "stream" "<" type ">" ;
```

A stream is parameterized by the element type it yields:

```
// Common stream types
stream<string>    // Stream of text tokens (agent output)
stream<byte>      // Stream of raw bytes
stream<Event>     // Stream of typed events
```

3.10.2 Consuming Streams

Streams are consumed with `for` loops. Each iteration yields the next value as it becomes available:

```
// Iterate over a stream
tokens: stream<string> = Researcher.stream("Explain quantum
   computing")

for token in tokens {
    io.print(token)  // Print each token as it arrives
}
```

3.10.3 Streams as Return Types

Functions and workflows can return streams, enabling streaming pipelines:

```
// Workflow returning a stream (e.g., for WebSocket)
@websocket("/chat")
workflow StreamingChat(message: string, session_id: string) ->
   stream<string> {
     return Assistant.stream(message, session: session_id)
}
```

When a workflow returns `stream<T>`, the runtime streams each element to the client as it is produced—no buffering.

3.10.4 Stream Properties

- **Lazy** – Values are produced on demand, not computed upfront
- **Single-use** – A stream can only be iterated once
- **Asynchronous** – The `for` loop suspends until the next value is available
- **Finite** – Streams eventually terminate (the loop exits when the source is exhausted)

Note

Streams are distinct from channels (`chan<T>`). A channel is a bidirectional communication primitive between concurrent tasks. A stream is a unidirectional, read-only sequence—typically produced by an agent or an external data source.

3.11 The Error Type

The built-in `Error` type represents errors:

```
struct Error {
    message: string
    code: int?
    source: Error?
}

// Creating errors
err = Error{message: "file not found"}
```

```
err_with_code = Error{message: "permission denied", code: 403}

// Error chaining
wrapped = Error{
    message: "failed to read config",
    source: err
}
```

3.12 Type Inference

Haira infers types for local variables from their initializers:

```
// Types inferred
x = 42                // int
pi = 3.14             // float
name = "Haira"        // string
active = true         // bool
numbers = [1, 2, 3]   // []int

// Explicit annotation when needed
count: i32 = 100      // Specific integer type
```

Warning

Type annotations are **required** for:

- Function parameters
- Function return types
- Struct fields
- Uninitialized variables

3.13 Structural Typing

Types are compatible based on structure, not name:

```
struct Point2D {
    x: float
    y: float
}

struct Vector2D {
    x: float
    y: float
}

fn magnitude(p: {x: float, y: float}) -> float {
    return math.sqrt(p.x * p.x + p.y * p.y)
}

// Both work because they have the same structure
point = Point2D{x: 3.0, y: 4.0}
```

```
vector = Vector2D{x: 3.0, y: 4.0}

io.println(magnitude(point))   // 5.0
io.println(magnitude(vector))  // 5.0
```

3.14 Type Grammar

```
type = primitive_type
     | array_type
     | map_type
     | tuple_type
     | option_type
     | function_type
     | generic_type
     | stream_type
     | struct_type
     | identifier ;

primitive_type = "int" | "i8" | "i16" | "i32" | "i64"
               | "u8" | "u16" | "u32" | "u64"
               | "float" | "f32" | "f64"
               | "bool" | "string" ;

array_type = "[" "]" type ;

map_type = "[" type ":" type "]" ;

tuple_type = "(" type { "," type } ")" ;

option_type = type "?" ;

function_type = "fn" "(" [ type_list ] ")" [ "->" type ] ;

generic_type = identifier "<" type_list ">" ;

stream_type = "stream" "<" type ">" ;

type_list = type { "," type } ;
```

Chapter 4

Expressions

Expressions produce values. Every expression in HAIRA has a type determined at compile time.

4.1 Primary Expressions

4.1.1 Literals

```
42                  // int literal
3.14                // float literal
"hello"             // string literal
true                // bool literal
nil                 // nil literal
```

4.1.2 Identifiers

```
x                   // Variable reference
user.name           // Field access
array[0]            // Index access
```

4.1.3 Parenthesized Expressions

```
(1 + 2) * 3         // Grouping for precedence
```

4.2 Arithmetic Expressions

```
a + b               // Addition
a - b               // Subtraction
a * b               // Multiplication
a / b               // Division
a % b               // Modulo (remainder)
-a                  // Unary negation
```

4.2.1 Integer Division

Integer division truncates toward zero:

```
7 / 3               // 2
-7 / 3              // -2
7 / -3              // -2
```

4.2.2 Modulo

The modulo operator returns the remainder:

```
7 % 3               // 1
-7 % 3              // -1
7 % -3              // 1
```

4.3 Comparison Expressions

```
a == b              // Equal
a != b              // Not equal
a < b               // Less than
a > b               // Greater than
a <= b              // Less or equal
a >= b              // Greater or equal
```

All comparison expressions produce `bool` values.

4.3.1 Equality

Equality (==) compares by value for primitives and by structural equality for composite types:

```
1 == 1                         // true
"hello" == "hello"             // true
[1, 2] == [1, 2]              // true
User{id: 1} == User{id: 1}   // true (structural)
```

4.4 Logical Expressions

```
a and b             // Logical AND (also &&)
a or b              // Logical OR (also ||)
not a               // Logical NOT (also !)
```

4.4.1 Short-Circuit Evaluation

Logical operators use short-circuit evaluation:

```
// 'b' is not evaluated if 'a' is false
a and b
```

```

// 'b' is not evaluated if 'a' is true
a or b
```

4.5 Bitwise Expressions

Bitwise operators perform bit-level operations on integer types. All bitwise operators work on any integer type (`int`, `i8`–`i64`, `u8`–`u64`).

```
a & b           // Bitwise AND
a | b           // Bitwise OR
a ^ b           // Bitwise XOR
~a              // Bitwise NOT (one's complement)
a << n          // Left shift by n bits
a >> n          // Right shift by n bits
a >>> n         // Logical right shift by n bits
```

4.5.1 Bitwise AND

Sets each bit to 1 only if both corresponding bits are 1:

```
0b1100 & 0b1010         // 0b1000 (8)
flags & MASK            // Extract bits using mask
value & 0xFF            // Keep only lowest 8 bits
```

4.5.2 Bitwise OR

Sets each bit to 1 if at least one corresponding bit is 1:

```
0b1100 | 0b1010         // 0b1110 (14)
flags | FLAG_ENABLED    // Set a flag bit
```

4.5.3 Bitwise XOR

Sets each bit to 1 if exactly one corresponding bit is 1:

```
0b1100 ^ 0b1010         // 0b0110 (6)
value ^ mask            // Toggle bits
a ^ b ^ b               // Returns a (XOR is self-inverse)
```

4.5.4 Bitwise NOT

Inverts all bits (one's complement):

```
~0b00001111             // 0b11110000 (for 8-bit)
~0u8                    // 255 (all bits set)
~0                      // -1 (for signed int)
```

Warning

The result of ~ depends on the bit width of the operand type. For `u8`, `~0` is 255; for `int`, `~0` is −1.

4.5.5 Left Shift

Shifts bits to the left, filling vacated positions with zeros:

```
1 << 4                  // 16 (0b10000)
value << n              // Equivalent to value * 2^n
0b0001 << 3             // 0b1000 (8)
```

Warning

Shifting by a negative amount or by more than the bit width of the type is undefined behavior.

4.5.6 Right Shift

The behavior of » depends on the signedness of the operand:

- **Unsigned types**: Logical shift (fills with zeros)
- **Signed types**: Arithmetic shift (fills with sign bit)

```
// Unsigned: logical shift (fills with 0)
x: u8 = 0b11110000
x >> 2                  // 0b00111100 (60)

// Signed: arithmetic shift (fills with sign bit)
y: i8 = -16             // 0b11110000 in two's complement
y >> 2                  // -4 (0b11111100, sign preserved)
```

4.5.7 Logical Right Shift

The »> operator always performs a logical right shift, filling with zeros regardless of signedness:

```
x: i8 = -16             // 0b11110000
x >>> 2                 // 60 (0b00111100, zeros filled)

// Useful for treating signed integers as bit patterns
hash: int = -1
hash >>> 1              // Large positive number
```

4.5.8 Common Bit Manipulation Patterns

```
// Check if bit n is set
is_set = (value >> n) & 1 == 1

```

```
// Set bit n
value = value | (1 << n)

// Clear bit n
value = value & ~(1 << n)

// Toggle bit n
value = value ^ (1 << n)

// Check if power of two (positive integers)
is_power_of_two = value > 0 and (value & (value - 1)) == 0

// Extract n bits starting at position p
bits = (value >> p) & ((1 << n) - 1)

// Count trailing zeros (manual approach)
fn count_trailing_zeros(x: u32) -> int {
    if x == 0 { return 32 }
    count = 0
    while (x & 1) == 0 {
        count += 1
        x = x >> 1
    }
    return count
}
```

4.5.9 Compound Bitwise Assignment

```
flags &= MASK            // flags = flags & MASK
flags |= FLAG            // flags = flags | FLAG
flags ^= TOGGLE          // flags = flags ^ TOGGLE
value <<= 2              // value = value << 2
value >>= 1              // value = value >> 1
value >>>= 1             // value = value >>> 1
```

4.6 String Expressions

4.6.1 Concatenation

```
"Hello, " + "World!"        // "Hello, World!"
```

4.6.2 String Interpolation

```
name = "Alice"
age = 30
"Hello, ${name}! You are ${age} years old."
```

Expressions inside `${...}` are evaluated and converted to strings.

4.7 Array and Map Expressions

4.7.1 Array Literals

```
[]                             // Empty array (type from context)
[1, 2, 3]                      // []int
["a", "b", "c"]                // []string
```

4.7.2 Map Literals

```
[:]                            // Empty map (type from context)
["a": 1, "b": 2]               // [string:int]
[1: "one", 2: "two"]           // [int:string]
```

4.7.3 Index Access

```
array[0]                       // First element
array[array.len - 1]           // Last element
map["key"]                     // Map lookup (returns T?)
```

Note

Array indexing with an out-of-bounds index causes a runtime panic. Map indexing returns an option type.

4.8 Field Access

```
user.name                      // Struct field
point.x                        // Struct field
tuple.0                        // Tuple element
```

4.9 Function Calls

```
func()                         // No arguments
func(a, b, c)                  // With arguments
obj.method()                   // Method call
obj.method(a, b)               // Method with arguments
```

4.9.1 Chained Calls

```
text.trim().to_upper().split(" ")
```

4.10 Pipe Expressions

The pipe operator `|>` passes the left operand as the first argument to the right function:

```
// These are equivalent:
result = f(g(h(x)))
result = x |> h |> g |> f
```

4.10.1 Pipe with Arguments

When the right side has arguments, the left value becomes the first argument:

```
// These are equivalent:
result = string.split(text, ",")
result = text |> string.split(",")
```

4.10.2 Complex Pipelines

```
import "io"
import "string"
import "array"

fn main() {
    data = "  apple, banana, cherry  "

    result = data
        |> string.trim
        |> string.split(", ")
        |> array.map(string.to_upper)
        |> array.filter(fn(s) { string.len(s) > 5 })
        |> string.join(" - ")

    io.println(result)  // "BANANA - CHERRY"
}
```

4.11 Range Expressions

```
0..10            // 0 to 9 (exclusive end)
0..=10           // 0 to 10 (inclusive end)
```

Ranges are used primarily in `for` loops:

```
for i in 0..5 {
    io.println(i)  // 0, 1, 2, 3, 4
}

for i in 0..=5 {
    io.println(i)  // 0, 1, 2, 3, 4, 5
}
```

4.12 Conditional Expressions

4.12.1 If Expression

if can be used as an expression:

```
max = if a > b { a } else { b }

status = if score >= 90 {
    "A"
} else if score >= 80 {
    "B"
} else {
    "C"
}
```

Warning

When used as an expression, all branches must have the same type and **else** is required.

4.12.2 Match Expression

match expressions provide pattern matching:

```
result = match value {
    0 => "zero"
    1 => "one"
    n if n < 0 => "negative"
    _ => "other"
}
```

See Chapter 5 for complete pattern matching semantics.

4.13 Block Expressions

Blocks are expressions that evaluate to their last expression:

```
result = {
    x = compute_x()
    y = compute_y()
    x + y  // This is the value of the block
}
```

4.14 Struct Instantiation

```
user = User{
    id: 1,
    name: "Alice",
    email: "alice@example.com"
}

```

```
// With shorthand (field name matches variable)
name = "Bob"
email = "bob@example.com"
user = User{id: 2, name, email}
```

4.15 Closure Expressions

Anonymous functions (closures):

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

// Short syntax for single expression
double = fn(x: int) -> int { x * 2 }

// Type inference for closures
numbers = [1, 2, 3]
doubled = array.map(numbers, fn(n) { n * 2 })
```

4.16 Option Expressions

4.16.1 Nil Coalescing

The `or` operator provides a default for optional values:

```
name = maybe_name or "Unknown"
value = get_value() or default_value
```

4.16.2 Optional Chaining

```
// Returns nil if any part is nil
street = user?.address?.street
```

4.17 Type Assertions

```
// Assert that value is a specific type
x = value as int

// Safe assertion (returns option)
x = value as? int
```

4.18 Operator Precedence

From highest to lowest:

1. Postfix: `()` `[]` `.` `?.`
2. Unary: `!` `-` `~` `not`
3. Multiplicative: `*` `/` `%`
4. Additive: `+` `-`
5. Shift: `«` `»` `»>`
6. Bitwise AND: `&`
7. Bitwise XOR: `^`
8. Bitwise OR: `|`
9. Range: `..` `..=`
10. Pipe: `|>`
11. Comparison: `<` `>` `<=` `>=`
12. Equality: `==` `!=`
13. Logical AND: `and` `&&`
14. Logical OR: `or` `||`
15. Assignment: `=` `+=` `-=` `*=` `/=` `&=` `|=` `^=` `«=` `»=` `»>=`

4.19 Expression Grammar

```
expression = assignment ;

assignment = or_expr [ ("=" | "+=" | "-=" | "*=" | "/="
           | "&=" | "|=" | "^=" | "<<=" | ">>=" | ">>>=")
               assignment ] ;

or_expr = and_expr { ("or" | "||") and_expr } ;

and_expr = equality { ("and" | "&&") equality } ;

equality = comparison { ("==" | "!=") comparison } ;

comparison = pipe { ("<" | ">" | "<=" | ">=") pipe } ;

pipe = range { "|>" range } ;

range = bitwise_or [ (".." | "..=") bitwise_or ] ;

bitwise_or = bitwise_xor { "|" bitwise_xor } ;

bitwise_xor = bitwise_and { "^" bitwise_and } ;

bitwise_and = shift { "&" shift } ;

shift = additive { ("<<" | ">>" | ">>>") additive } ;

```

```
additive = multiplicative { ("+" | "-") multiplicative } ;

multiplicative = unary { ("*" | "/" | "%") unary } ;

unary = ("!" | "-" | "~" | "not") unary | postfix ;

postfix = primary { call | index | field } ;

call = "(" [ arguments ] ")" ;

index = "[" expression "]" ;

field = "." identifier ;

primary = identifier
        | literal
        | "(" expression ")"
        | array_literal
        | map_literal
        | struct_literal
        | closure
        | if_expr
        | match_expr
        | block ;

arguments = expression { "," expression } ;
```

Chapter 5

Statements

Statements perform actions. Unlike expressions, statements do not produce values.

5.1 Variable Declarations

5.1.1 With Inference

```
x = 42                    // Type inferred as int
name = "Haira"            // Type inferred as string
items = [1, 2, 3]         // Type inferred as []int
```

5.1.2 With Explicit Type

```
x: int = 42
name: string = "Haira"
count: i32 = 100
```

5.1.3 Multiple Declarations

```
a, b = 1, 2
x, y, z = compute()       // Destructuring tuple
```

5.1.4 Uninitialized Variables

```
x: int                    // Must be assigned before use
```

Warning

Using an uninitialized variable is a compile-time error.

5.2 Assignment Statements

```
x = 10                  // Simple assignment
x += 5                  // Add and assign (x = x + 5)
x -= 3                  // Subtract and assign
x *= 2                  // Multiply and assign
x /= 4                  // Divide and assign
```

5.2.1 Multiple Assignment

```
a, b = b, a             // Swap values
x, y = get_coordinates()
quotient, remainder = divide(17, 5)
```

5.2.2 Field Assignment

```
user.name = "Bob"
point.x = 10.0
```

5.2.3 Index Assignment

```
array[0] = 100
map["key"] = "value"
```

5.3 Expression Statements

Any expression can be used as a statement:

```
io.println("Hello")     // Function call
process(data)           // Function call, ignoring return
x + y                   // Valid but pointless
```

Note

The compiler may warn about expression statements that have no side effects.

5.4 If Statements

```
if condition {
    // executed if condition is true
}

if condition {
    // true branch
} else {
    // false branch
}
```

```

if condition1 {
    // first
} else if condition2 {
    // second
} else {
    // default
}
```

5.4.1 Condition Binding

```
// Bind and check in one statement
if value = get_optional_value(); value != nil {
    io.println(value)
}
```

5.5 For Statements

5.5.1 Range-Based For

```
// Exclusive range
for i in 0..10 {
    io.println(i)  // 0 through 9
}

// Inclusive range
for i in 0..=10 {
    io.println(i)  // 0 through 10
}
```

5.5.2 Collection Iteration

```
// Array iteration
for item in items {
    io.println(item)
}

// With index
for i, item in items {
    io.println("${i}: ${item}")
}

// Map iteration
for key, value in scores {
    io.println("${key}: ${value}")
}
```

5.5.3 String Iteration

```
for char in "hello" {
    io.println(char)  // Each UTF-8 character
}

for i, char in "hello" {
    io.println("${i}: ${char}")
}
```

5.6 While Statements

```
while condition {
    // body
}

// Infinite loop
while true {
    if should_stop() {
        break
    }
}
```

5.7 Match Statements

5.7.1 Basic Matching

```
match value {
    1 => io.println("one")
    2 => io.println("two")
    3 => io.println("three")
    _ => io.println("other")
}
```

5.7.2 Multiple Patterns

```
match value {
    1 | 2 | 3 => io.println("small")
    4 | 5 | 6 => io.println("medium")
    _ => io.println("large")
}
```

5.7.3 Range Patterns

```
match score {
    90..=100 => io.println("A")
    80..90 => io.println("B")
    70..80 => io.println("C")
    60..70 => io.println("D")
    _ => io.println("F")
}
```

5.7.4 Guard Clauses

```
match value {
    n if n < 0 => io.println("negative")
    n if n == 0 => io.println("zero")
    n if n > 0 => io.println("positive")
}
```

5.7.5 Destructuring Patterns

```
// Struct destructuring
match user {
    User{name: "admin", active: true} => io.println("Active
        admin")
    User{name: n, active: false} => io.println("Inactive: " + n)
    _ => io.println("Regular user")
}

// Tuple destructuring
match point {
    (0, 0) => io.println("origin")
    (x, 0) => io.println("on x-axis at " + to_string(x))
    (0, y) => io.println("on y-axis at " + to_string(y))
    (x, y) => io.println("at (" + to_string(x) + ", " +
        to_string(y) + ")")
}

// Enum destructuring
match result {
    Result.Ok(value) => io.println("Got: " + to_string(value))
    Result.Err(err) => io.println("Error: " + err.message)
}
```

5.7.6 Match Exhaustiveness

```
enum Direction { North, South, East, West }

match direction {
    Direction.North => go_north()
    Direction.South => go_south()
    Direction.East => go_east()
    Direction.West => go_west()
    // No _ needed: all cases covered
}
```

Note

The compiler verifies that match expressions are exhaustive. If not all cases are covered and there is no wildcard (_), the compiler produces an error.

5.8 Block Statements

```
{
    x = 10
    y = 20
    io.println(x + y)
}
```

Blocks create a new scope. Variables declared inside are not visible outside.

5.9 Return Statements

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

fn early_return(x: int) -> int {
    if x < 0 {
        return 0
    }
    return x * 2
}

// Multiple returns
fn divide(a: int, b: int) -> (int, Error?) {
    if b == 0 {
        return 0, Error{message: "division by zero"}
    }
    return a / b, nil
}
```

5.9.1 Implicit Return

The last expression in a function is implicitly returned:

```
fn add(a: int, b: int) -> int {
    a + b  // No return keyword needed
}
```

5.10 Break Statements

```
for i in 0..100 {
    if i > 10 {
        break
    }
    io.println(i)
}

while true {
    if done() {
        break
    }
}
```

5.10.1 Labeled Break

```
outer: for i in 0..10 {
    for j in 0..10 {
        if i * j > 25 {
            break outer
        }
    }
}
```

5.11 Continue Statements

```
for i in 0..10 {
    if i % 2 == 0 {
        continue  // Skip even numbers
    }
    io.println(i)
}
```

5.11.1 Labeled Continue

```
outer: for i in 0..5 {
    for j in 0..5 {
        if j == 2 {
            continue outer
        }
        io.println("${i}, ${j}")
    }
}
```

5.12 Import Statements

Import statements must appear at the top of the file:

```
import "io"
import "json"
import db from "haira/postgres"
import { User, Post } from "models"
```

See Chapter 9 for complete import semantics.

5.13 Defer Statements

`defer` schedules a statement to run when the current scope exits:

```
fn read_file(path: string) -> (string, Error?) {
    file, err = fs.open(path)
    if err != nil {
        return "", err
    }
    defer fs.close(file)

    content, err = fs.read_all(file)
    if err != nil {
        return "", err  // file is closed here
    }

    return content, nil  // file is closed here too
}
```

5.13.1 Multiple Defers

Multiple **`defer`** statements execute in reverse order (LIFO):

```
fn example() {
    defer io.println("first")   // Runs third
    defer io.println("second")  // Runs second
    defer io.println("third")   // Runs first
}
// Output: third, second, first
```

5.14 Statement Grammar

```
statement = variable_decl
          | assignment
          | expression_stmt
          | if_stmt
          | for_stmt
          | while_stmt
          | match_stmt
          | return_stmt
          | break_stmt
          | continue_stmt
          | defer_stmt
          | block ;

variable_decl = identifier [ ":" type ] "=" expression
              | identifier_list "=" expression_list ;

assignment = lvalue assign_op expression ;

assign_op = "=" | "+=" | "-=" | "*=" | "/=" ;
```

```

lvalue = identifier | field_access | index_access ;

expression_stmt = expression ;

if_stmt = "if" [ simple_stmt ";" ] expression block
          [ "else" ( if_stmt | block ) ] ;

for_stmt = "for" [ identifier "," ] identifier "in" expression
   block ;

while_stmt = "while" expression block ;

match_stmt = "match" expression "{" { match_arm } "}" ;

match_arm = pattern [ "if" expression ] "=>" ( expression |
   block ) ;

return_stmt = "return" [ expression_list ] ;

break_stmt = "break" [ identifier ] ;

continue_stmt = "continue" [ identifier ] ;

defer_stmt = "defer" statement ;

block = "{" { statement } "}" ;
```

Chapter 6

Functions

Functions are the primary unit of code organization in HAIRA. They are first-class values that can be passed as arguments, returned from other functions, and stored in variables.

6.1 Function Declarations

```
fn function_name(param1: Type1, param2: Type2) -> ReturnType {
    // body
}
```

6.1.1 Basic Examples

```
fn greet() {
    io.println("Hello!")
}

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

fn is_positive(n: int) -> bool {
    return n > 0
}
```

6.1.2 No Return Type

Functions without a return type implicitly return `()`:

```
fn log_message(msg: string) {
    io.println("[LOG] " + msg)
}
```

6.2 Parameters

6.2.1 Required Parameters

All parameters must have explicit type annotations:

```
fn calculate(x: int, y: int, z: float) -> float {
    return (x + y) as float * z
}
```

6.2.2 Default Parameters

Parameters can have default values:

```
fn greet(name: string, greeting: string = "Hello") {
    io.println(greeting + ", " + name + "!")
}

greet("Alice")              // "Hello, Alice!"
greet("Bob", "Hi")          // "Hi, Bob!"
```

Note

Parameters with defaults must come after parameters without defaults.

6.2.3 Named Arguments

Functions can be called with named arguments:

```
fn create_user(name: string, email: string, age: int = 0) ->
    User {
    return User{name: name, email: email, age: age}
}

// Positional
create_user("Alice", "alice@example.com", 30)

// Named
create_user(name: "Bob", email: "bob@example.com")

// Mixed (positional first, then named)
create_user("Charlie", email: "charlie@example.com", age: 25)
```

6.2.4 Variadic Parameters

Functions can accept variable numbers of arguments:

```
fn sum(numbers: ...int) -> int {
    total = 0
    for n in numbers {
        total += n
    }
    return total
}

sum(1, 2, 3)        // 6
sum(1, 2, 3, 4, 5)  // 15
```

6.3 Return Values

6.3.1 Single Return

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

6.3.2 Multiple Returns

Functions can return multiple values using tuples:

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

quotient, remainder = divide(17, 5)  // 3, 2
```

6.3.3 Implicit Return

The last expression in a function body is implicitly returned:

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

fn max(a: int, b: int) -> int {
    if a > b { a } else { b }  // Implicit return
}
```

6.3.4 Early Return

Use **`return`** for early exit:

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

6.4 Function Types

Functions have types that can be used in type annotations:

```
// Function type: fn(int, int) -> int
type BinaryOp = fn(int, int) -> int

fn apply(op: BinaryOp, a: int, b: int) -> int {
    return op(a, b)
}

fn add(a: int, b: int) -> int { a + b }
fn mul(a: int, b: int) -> int { a * b }

apply(add, 2, 3)  // 5
apply(mul, 2, 3)  // 6
```

6.5 Closures

Closures are anonymous functions that can capture values from their environment:

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

counter = make_counter()
counter()  // 1
counter()  // 2
counter()  // 3
```

6.5.1 Closure Syntax

```
// Full syntax
fn(a: int, b: int) -> int {
    return a + b
}

// Short syntax (single expression)
fn(a: int, b: int) -> int { a + b }

// With type inference (in context)
numbers = [1, 2, 3]
doubled = array.map(numbers, fn(n) { n * 2 })
```

6.5.2 Capture Semantics

Closures capture variables by reference:

```
fn example() {
    values = []

    for i in 0..3 {
        // Captures 'i' by reference
        values = array.push(values, fn() { i })
    }

    // All closures see final value of i
    values[0]()  // 3
    values[1]()  // 3
    values[2]()  // 3
}
```

To capture by value, use explicit binding:

```
fn example() {
    values = []

    for i in 0..3 {
        captured = i  // Capture current value
        values = array.push(values, fn() { captured })
    }

    values[0]()  // 0
    values[1]()  // 1
    values[2]()  // 2
}
```

6.6 Methods

Methods are functions associated with a type:

```
struct Rectangle {
    width: float
    height: float
}

Rectangle.area() -> float {
    return self.width * self.height
}

Rectangle.perimeter() -> float {
    return 2.0 * (self.width + self.height)
}

Rectangle.scale(factor: float) -> Rectangle {
    return Rectangle{
        width = self.width * factor,
        height = self.height * factor
    }
}

// Usage
rect = Rectangle{width = 10.0, height = 5.0}
rect.area()       // 50.0
rect.perimeter()  // 30.0
rect.scale(2.0)   // Rectangle{width: 20.0, height: 10.0}
```

6.6.1 Method Semantics

Methods use `self` to access the receiver. All methods can mutate the receiver (pointer semantics). Method visibility is inherited from the type: if the type is **`pub`**, all its methods are accessible to importers. Methods cannot be defined on imported types.

```
struct Point {
    x: float
    y: float
}

Point.distance_to(other: Point) -> float {
    dx = self.x - other.x
    dy = self.y - other.y
    return math.sqrt(dx*dx + dy*dy)
}

Point.translate(dx: float, dy: float) {
    self.x += dx
    self.y += dy
}
```

6.7 Generic Functions

Functions can be parameterized over types:

```
fn identity<T>(value: T) -> T {
    return value
}

fn swap<T, U>(pair: (T, U)) -> (U, T) {
    return (pair.1, pair.0)
}

fn first<T>(items: []T) -> T? {
    if array.len(items) == 0 {
        return nil
    }
    return items[0]
}
```

6.7.1 Type Constraints

Generic types can be constrained:

```
constraint Comparable {
    fn compare(other: Self) -> int
}

fn max<T: Comparable>(a: T, b: T) -> T {
    if a.compare(b) > 0 {
        return a
    }
    return b
}
```

```

fn sort<T: Comparable>(items: []T) -> []T {
    // Sorting implementation
}
```

6.8 Higher-Order Functions

Functions that take or return other functions:

```
// Function as parameter
fn apply_twice(f: fn(int) -> int, x: int) -> int {
    return f(f(x))
}

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

apply_twice(double, 5)  // 20

// Function as return value
fn make_multiplier(factor: int) -> fn(int) -> int {
    return fn(n: int) -> int {
        return n * factor
    }
}

triple = make_multiplier(3)
triple(7)  // 21
```

6.9 Recursion

Functions can call themselves:

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

fn fibonacci(n: int) -> int {
    match n {
        0 => 0
        1 => 1
        _ => fibonacci(n - 1) + fibonacci(n - 2)
    }
}
```

6.9.1 Tail Call Optimization

HAIRA optimizes tail-recursive functions:

```
fn factorial_tail(n: int, acc: int = 1) -> int {
    if n <= 1 {
        return acc
    }
    return factorial_tail(n - 1, n * acc)  // Tail call
}
```

6.10 Function Overloading

Haira does not support function overloading. Each function name must be unique within its scope. Use different names or generic functions instead:

```
// Instead of overloading, use descriptive names
fn add_ints(a: int, b: int) -> int { a + b }
fn add_floats(a: float, b: float) -> float { a + b }

// Or use generics
fn add<T: Addable>(a: T, b: T) -> T { a + b }
```

6.11 The main Function

Every executable Haira program must have a `main` function:

```
fn main() {
    io.println("Hello, World!")
}

// With exit code
fn main() -> int {
    io.println("Hello, World!")
    return 0
}
```

6.12 Function Grammar

```
function_decl = "fn" [ receiver ] identifier [ type_params ]
                "(" [ params ] ")" [ "->" type ] block ;

receiver = "(" identifier ":" [ "*" ] type ")" ;

type_params = "<" type_param { "," type_param } ">" ;

type_param = identifier [ ":" constraint_list ] ;

constraint_list = identifier { "+" identifier } ;

params = param { "," param } ;

param = identifier ":" [ "..." ] type [ "=" expression ] ;

closure = "fn" "(" [ closure_params ] ")" [ "->" type ]
          ( block | expression ) ;
```

```

closure_params = closure_param { "," closure_param } ;

closure_param = identifier [ ":" type ] ;
```

Chapter 7

Error Handling

HAIRA uses explicit error handling inspired by Go. Errors are values, not exceptions. Functions that can fail return both a result and an optional error.

For concise code, HAIRA also provides a `?` propagation operator and **`try`**/**`catch`** blocks built on Go's `panic`/`recover` mechanism.

7.1 Philosophy

- **Errors are values** – Not special control flow
- **Explicit handling** – Callers must handle or propagate errors
- **Two modes** – Verbose-but-clear tuple checking, or concise `?` propagation
- **Composable** – Errors can be wrapped and chained

7.2 The Error Type

The built-in `Error` type represents errors:

```
struct Error {
    message: string
    code: int?
    source: Error?
}
```

7.2.1 Creating Errors

```
// Simple error
err = Error{message: "something went wrong"}

// With error code
err = Error{message: "permission denied", code: 403}

// Wrapped error (error chaining)
err = Error{
    message: "failed to read config",
    source: original_error
}
```

7.3 Error Return Pattern

Functions that can fail return a tuple of `(result, Error?)`:

```
fn divide(a: int, b: int) -> (int, Error?) {
    if b == 0 {
        return 0, Error{message: "division by zero"}
    }
    return a / b, nil
}

fn read_file(path: string) -> (string, Error?) {
    // Implementation
}

fn parse_int(s: string) -> (int, Error?) {
    // Implementation
}
```

7.4 Handling Errors

7.4.1 Basic Error Checking

The standard pattern: call a function, check the error, handle or return:

```
result, err = divide(10, 0)
if err != nil {
    io.println("Error: " + err.message)
    return
}
io.println("Result: " + to_string(result))
```

7.4.2 Propagating Errors

```
fn process_file(path: string) -> (Data, Error?) {
    content, err = read_file(path)
    if err != nil {
        return Data{}, err  // Propagate error
    }

    data, err = parse_data(content)
    if err != nil {
        return Data{}, err  // Propagate error
    }

    return data, nil
}
```

7.4.3 Wrapping Errors

Add context when propagating:

```
fn load_config(path: string) -> (Config, Error?) {
    content, err = read_file(path)
    if err != nil {
        return Config{}, Error{
            message: "failed to load config from " + path,
            source: err
        }
    }

    config, err = parse_config(content)
    if err != nil {
        return Config{}, Error{
            message: "invalid config format",
            source: err
        }
    }

    return config, nil
}
```

7.5 Error Propagation Operator (?)

The `?` operator is syntactic sugar for "panic on error." It extracts the success value from a call that returns `(T, Error?)`. If the error is non-nil, it panics.

```
// These two are equivalent:

// Verbose
content, err = read_file(path)
if err != nil {
    panic(err)
}

// Concise
content = read_file(path)?
```

The `?` operator makes code much shorter when chaining multiple fallible calls:

```
fn setup() {
    config = load_config("app.toml")?
    db = connect_database(config.db_url)?
    cache = init_cache(config.cache_url)?
    // If any call fails, execution stops with a panic
}
```

Warning

The `?` operator panics on error. If used outside a **`try`** block, an unhandled error will crash the program. Use `?` inside **`try`**/**`catch`** blocks, or in initialization code where failure is fatal.

7.6 Try/Catch

The `try/catch` block recovers from panics caused by the `?` operator (or explicit `panic()` calls). It compiles to Go's `defer/recover` pattern.

```
try {
    config = load_config("app.toml")?
    db = connect_database(config.db_url)?
    start_server(db)
} catch err {
    io.println("Startup failed: " + err)
    os.exit(1)
}
```

The catch variable receives the error as a string. Variables assigned inside the `try` block are hoisted and remain accessible after the block.

7.6.1 Try/Catch in Functions

```
fn process_batch(items: [string]) -> int {
    count = 0
    for item in items {
        try {
            result = parse_item(item)?
            save_result(result)?
            count = count + 1
        } catch err {
            io.eprintln("Skipping item: " + err)
        }
    }
    return count
}
```

7.6.2 When to Use Each Pattern

Pattern	**Syntax**	**Use When**
Tuple check	`val, err = fn()`	You need to inspect or wrap the error
`?` + `try/catch`	`val = fn()?`	Multiple calls, same error handling
`?` without `try`	`val = fn()?`	Failure is fatal (init, main)

Table 7.1: Error handling patterns

7.7 Orelse Operator

The `orelse` operator provides a default value when a fallible call fails:

```
// If parse fails, use 0 as default
count = parse_int(input) orelse 0

// If config load fails, use defaults
config = load_config("app.toml") orelse Config{port: 8080}
```

This is useful for optional operations where a sensible default exists.

7.8 Defer and Errdefer

7.8.1 Defer

Use **`defer`** to ensure cleanup happens when the enclosing scope exits, regardless of how it exits:

```
fn with_file(path: string) -> (string, Error?) {
    file, err = fs.open(path)
    if err != nil {
        return "", err
    }
    defer fs.close(file)  // Always closes, even on error

    content, err = fs.read_all(file)
    if err != nil {
        return "", err  // file is closed by defer
    }

    return content, nil  // file is closed by defer
}
```

7.8.2 Errdefer

The **`errdefer`** statement runs cleanup only when the enclosing scope exits via a panic (from `?` or explicit `panic`). If the scope exits normally, the errdefer does not run.

```
fn setup_resources() {
    db = connect_database()?
    errdefer db.close()  // Only closes if a later ? panics

    cache = connect_cache()?
    errdefer cache.close()  // Only closes if a later ? panics

    // If migrate fails, both db and cache are closed
    migrate(db)?
    start_server(db, cache)
    // If we get here, errdefers do NOT run -- resources stay
        open
}
```

7.9 Error Checking Patterns

7.9.1 Guard Pattern

Check errors early and return:

```
fn process(input: string) -> (Output, Error?) {
    // Guard: validate input
    if string.len(input) == 0 {
        return Output{}, Error{message: "input cannot be empty"}
    }

    // Guard: check prerequisites
    ready, err = check_ready()
    if err != nil {
        return Output{}, err
    }
    if not ready {
        return Output{}, Error{message: "system not ready"}
    }

    // Main logic
    result = do_processing(input)
    return result, nil
}
```

7.10 Error Utilities

7.10.1 Checking Error Types

```
fn is_not_found(err: Error?) -> bool {
    if err == nil {
        return false
    }
    return err.code == 404 or string.contains(err.message, "not
       found")
}
```

7.10.2 Error Chain Traversal

```
fn root_cause(err: Error?) -> Error? {
    if err == nil {
        return nil
    }
    current = err
    while current.source != nil {
        current = current.source
    }
    return current
}
```

7.11 Option Types and Errors

For operations that may not find a value (not an error condition), use option types:

```
// Use option when "not found" is normal
fn find_user(id: int) -> User? {
    // Returns nil if not found
}

// Use error when "not found" is exceptional
```

```
fn get_user(id: int) -> (User, Error?) {
    // Returns error if not found
}
```

7.12 Panic

For truly unrecoverable situations, use `panic`:

```
fn assert(condition: bool, message: string) {
    if not condition {
        panic(message)
    }
}
```

Warning

`panic` crashes the program unless caught by **`try/catch`**. Use it only for programming errors and invariant violations. Prefer returning errors for all expected failure conditions.

7.13 Best Practices

- **Return errors, don't log and continue** – Propagate failures to the caller
- **Add context when wrapping** – Use `Error{message: "context", source: err}`
- **Handle at the right level** – Low-level returns specific errors, high-level decides what to do
- **Use `?` for chain-or-fail sequences** – Cleaner than repeated `if err != nil`
- **Use `orelse` for optional operations** – When a default is acceptable
- **Use `errdefer` for resource cleanup** – Pairs allocation with deallocation

7.14 Error Handling Grammar

```
error_return    = "(" type "," "Error?" ")" ;

error_check     = "if" identifier "!=" "nil" block ;

error_create    = "Error" "{"
                  "message" ":" expression
                  [ "," "code" ":" expression ]
                  [ "," "source" ":" expression ]
                  "}" ;

propagate_expr  = expression "?" ;

orelse_expr     = expression "orelse" expression ;

```

```
try_stmt         = "try" block "catch" identifier block ;

defer_stmt       = "defer" expression ;

errdefer_stmt    = "errdefer" expression ;
```

Part II

Concurrency and Modules

Chapter 8

Concurrency

HAIRA provides Go-style concurrency primitives: lightweight spawned tasks and channels for communication. The philosophy is "share memory by communicating" rather than "communicate by sharing memory."

8.1 Concurrency Model

- **Spawned tasks** – Lightweight concurrent execution units
- **Channels** – Typed conduits for communication
- **Select** – Multiplexing channel operations
- **No shared mutable state** – Communication via channels

8.2 Spawn

The **`spawn`** keyword starts a new concurrent task:

```
import "io"
import "time"

fn main() {
    spawn {
        time.sleep(1000)
        io.println("Hello from spawned task!")
    }

    io.println("Main continues immediately")
    time.sleep(2000)  // Wait for spawned task
}
```

8.2.1 Spawning Functions

```
fn worker(id: int) {
    io.println("Worker " + to_string(id) + " started")
    time.sleep(1000)
    io.println("Worker " + to_string(id) + " done")
}

```

```
fn main() {
    for i in 0..5 {
        spawn worker(i)
    }
    time.sleep(3000)  // Wait for all workers
}
```

8.3 Channels

Channels are typed conduits for passing values between tasks:

```
// Create a channel
ch = chan<int>()            // Unbuffered channel
ch = chan<int>(10)          // Buffered channel (capacity 10)
ch = chan<string>()         // String channel
```

8.3.1 Sending and Receiving

```
ch = chan<int>()

// Send (blocks until received for unbuffered)
ch <- 42

// Receive (blocks until value available)
value = <-ch
```

8.3.2 Basic Channel Example

```
import "io"

fn main() {
    ch = chan<string>()

    spawn {
        ch <- "Hello from spawned task!"
    }

    message = <-ch
    io.println(message)
}
```

8.3.3 Buffered Channels

```
ch = chan<int>(3)  // Buffer size 3

// These don't block (buffer has space)
ch <- 1
ch <- 2
ch <- 3
```

```

// This would block (buffer full)
// ch <- 4

// Receive
io.println(<-ch)  // 1
io.println(<-ch)  // 2
io.println(<-ch)  // 3
```

8.3.4 Closing Channels

```
ch = chan<int>(10)

spawn {
    for i in 0..10 {
        ch <- i
    }
    close(ch)  // Signal no more values
}

// Iterate until channel closed
for value in ch {
    io.println(value)
}

io.println("Channel closed, loop exited")
```

8.3.5 Checking Channel State

```
// Receive with closed check
value, ok = <-ch
if not ok {
    io.println("Channel closed")
}

// Non-blocking receive
value, ok = ch.try_receive()
if ok {
    io.println("Got: " + to_string(value))
} else {
    io.println("No value available")
}

// Non-blocking send
ok = ch.try_send(42)
if not ok {
    io.println("Channel full or closed")
}
```

8.4 Select

The `select` statement waits on multiple channel operations:

```
import "io"
import "time"

fn main() {
    ch1 = chan<string>()
    ch2 = chan<string>()

    spawn {
        time.sleep(1000)
        ch1 <- "from channel 1"
    }

    spawn {
        time.sleep(500)
        ch2 <- "from channel 2"
    }

    // Wait for first available
    select {
        msg = <-ch1 => io.println("Received: " + msg)
        msg = <-ch2 => io.println("Received: " + msg)
    }
}
```

8.4.1 Select with Default

Non-blocking select:

```
select {
    value = <-ch => io.println("Got: " + to_string(value))
    default => io.println("No value ready")
}
```

8.4.2 Select with Timeout

```
timeout = time.after(5000)  // 5 seconds

select {
    value = <-ch => io.println("Got: " + to_string(value))
    <-timeout => io.println("Timed out!")
}
```

8.4.3 Select in Loop

```
fn worker(jobs: chan<Job>, results: chan<Result>, done:
   chan<bool>) {
    while true {
        select {
            job = <-jobs => {
                result = process(job)
                results <- result
                }
                <-done => {
                    io.println("Worker shutting down")
                    return
                }
            }
        }
}
```

8.5 Common Patterns

8.5.1 Worker Pool

```
fn worker(id: int, jobs: chan<int>, results: chan<int>) {
    for job in jobs {
        io.println("Worker " + to_string(id) + " processing " +
            to_string(job))
        time.sleep(100)
        results <- job * 2
    }
}

fn main() {
    num_jobs = 10
    num_workers = 3

    jobs = chan<int>(num_jobs)
    results = chan<int>(num_jobs)

    // Start workers
    for i in 0..num_workers {
        spawn worker(i, jobs, results)
    }

    // Send jobs
    for j in 0..num_jobs {
        jobs <- j
    }
    close(jobs)

    // Collect results
    for _ in 0..num_jobs {
        result = <-results
        io.println("Result: " + to_string(result))
    }
}
```

8.5.2 Fan-Out, Fan-In

```
fn producer(out: chan<int>) {
    for i in 0..100 {
        out <- i
    }
    close(out)
}

fn worker(in: chan<int>, out: chan<int>) {
    for value in in {
        out <- value * value
    }
}

fn main() {
    input = chan<int>(100)
    output = chan<int>(100)

    // Producer
    spawn producer(input)

    // Fan-out to multiple workers
    for _ in 0..4 {
        spawn worker(input, output)
    }

    // Fan-in (collect results)
    spawn {
        time.sleep(1000)
        close(output)
    }

    for result in output {
        io.println(result)
    }
}
```

8.5.3 Pipeline

```
fn generate(out: chan<int>) {
    for i in 2..100 {
        out <- i
    }
    close(out)
}

fn filter(in: chan<int>, out: chan<int>, prime: int) {
    for n in in {
        if n % prime != 0 {
            out <- n
        }
    }
    close(out)
}

fn sieve() -> chan<int> {
    primes = chan<int>()

    spawn {
        ch = chan<int>(100)
        spawn generate(ch)

        while true {
            prime, ok = <-ch
            if not ok {
                break
            }
            primes <- prime

            next = chan<int>(100)
            spawn filter(ch, next, prime)
            ch = next
        }
        close(primes)
    }

    return primes
}

fn main() {
    for prime in sieve() {
        io.println(prime)
        if prime > 50 {
            break
        }
    }
}
```

8.5.4 Semaphore

```
struct Semaphore {
    ch: chan<bool>
}

fn new_semaphore(n: int) -> Semaphore {
    sem = Semaphore{ch: chan<bool>(n)}
    for _ in 0..n {
        sem.ch <- true
    }
    return sem
}

Semaphore.acquire() {
    <-self.ch
}

Semaphore.release() {
    self.ch <- true
}

// Usage: limit concurrent operations
fn main() {
    sem = new_semaphore(3)  // Max 3 concurrent

    for i in 0..10 {
        spawn {
            sem.acquire()
```

```
                defer sem.release()

                io.println("Task " + to_string(i) + " running")
                time.sleep(1000)
            }
        }

        time.sleep(5000)
}
```

8.6 Channel Types

8.7 Synchronization

8.7.1 WaitGroup

Wait for multiple tasks to complete:

```
import "sync"

fn main() {
    wg = sync.WaitGroup{}

    for i in 0..5 {
        wg.add(1)
        spawn {
            defer wg.done()
            io.println("Task " + to_string(i))
            time.sleep(1000)
        }
    }

    wg.wait()  // Block until all done
    io.println("All tasks completed")
}
```

8.7.2 Mutex

For rare cases requiring shared state:

```
import "sync"

struct Counter {
    mu: sync.Mutex
    value: int
}

Counter.increment() {
    self.mu.lock()
    defer self.mu.unlock()
    self.value += 1
}

Counter.get() -> int {
    self.mu.lock()
    defer self.mu.unlock()
    return self.value
}
```

Warning

Prefer channels over mutexes. Mutexes are provided for interoperability and performance-critical sections, but channel-based designs are generally safer and more idiomatic.

8.8 Concurrency Grammar

```
spawn_expr = "spawn" ( block | function_call ) ;

channel_type = "chan" "<" type ">"
             | "chan<-" type          (* send-only *)
             | "<-chan<" type ">"     (* receive-only *) ;

channel_create = "chan" "<" type ">" "(" [ expression ] ")" ;

send_stmt = expression "<-" expression ;

receive_expr = "<-" expression ;

select_stmt = "select" "{" { select_case } [ default_case ] "}"
    ;

select_case = pattern "=" receive_expr "=>" ( expression |
    block )
            | send_stmt "=>" ( expression | block ) ;

default_case = "default" "=>" ( expression | block ) ;
```

Chapter 9

Modules and Imports

Haira uses explicit imports for all dependencies. Every external symbol must be imported before use.

9.1 Module System Overview

- **Explicit imports** – All dependencies declared at file top
- **No implicit globals** – Everything must be imported
- **Simple resolution** – Standard library, project-local, external
- **Namespaced access** – `module.function()` syntax

9.2 Import Syntax

9.2.1 Basic Import

```
import "io"
import "json"
import "http"

fn main() {
    io.println("Hello")
}
```

9.2.2 Aliased Import

```
import fmt from "io"
import db from "haira/postgres"

fn main() {
    fmt.println("Hello")
    db.connect("localhost:5432")
}
```

9.2.3 Selective Import

Import specific symbols:

```
import { println, readln } from "io"
import { User, Post } from "models"

fn main() {
    println("Enter name:")
    name = readln()
    println("Hello, " + name)
}
```

9.2.4 Glob Import

Import all exports (use sparingly):

```
import * from "math"

fn main() {
    result = sqrt(pow(3.0, 2.0) + pow(4.0, 2.0))
    io.println(result)  // 5.0
}
```

Warning

Glob imports can cause naming conflicts and make code harder to read. Prefer explicit imports.

9.3 Module Resolution

Imports are resolved in this order:

1. **Standard library** – Built-in modules (`"io"`, `"json"`, etc.)
2. **Project-local** – Relative to project root (`"models/user"`)
3. **External packages** – From package registry (`"github.com/..."`)

9.3.1 Standard Library

```
import "io"          // Standard I/O
import "string"      // String manipulation
import "math"        // Mathematical functions
import "json"        // JSON encoding/decoding
import "http"        // HTTP client/server
import "fs"          // File system
import "os"          // Operating system
import "time"        // Time and duration
import "crypto"      // Cryptography
import "regex"       // Regular expressions
```

9.3.2 Project-Local Imports

```
// Project structure:
// myapp/
//   main.haira
//   models/
//     user.haira
//     post.haira
//   utils/
//     helpers.haira

// In main.haira:
import "models/user"
import "models/post"
import "utils/helpers"

// In models/post.haira:
import "models/user"    // Absolute from project root
```

9.3.3 External Packages

```
import "github.com/haira-libs/aws"
import "github.com/haira-libs/postgres"
import pg from "github.com/haira-libs/postgres"
```

External packages are declared in `haira.toml`:

```
[dependencies]
aws = "github.com/haira-libs/aws@1.0.0"
postgres = "github.com/haira-libs/postgres@2.1.0"
```

9.4 Module Definitions

9.4.1 File as Module

Each `.haira` file is a module. The file name determines the module name:

```
// File: utils/helpers.haira
// Module: utils/helpers

pub fn format_name(first: string, last: string) -> string {
    return first + " " + last
}

fn validate_email(email: string) -> bool {
    return string.contains(email, "@")
}
```

9.4.2 Visibility: The pub Keyword

Declarations are **private by default**. Use the `pub` keyword to make them available to importers:

```
// Public (exported) -- available to other modules
pub fn create_user(name: string) -> User { }
pub struct User { name: string, email: string }
pub enum Status { Active, Inactive }

// Private (not exported) -- file-local only
fn validate_email(email: string) -> bool { }
cache = [:]
```

Note

Agentic declarations (`provider`, `tool`, `agent`, `workflow`) are always public — they are inherently part of the module's external interface.

9.4.3 Package Initialization

Optional `init()` function runs when module is first imported:

```
// database.haira

pool: ConnectionPool?

fn init() {
    pool = create_pool()
    io.println("Database module initialized")
}

pub fn get_connection() -> (Connection, Error?) {
    if pool == nil {
        return Connection{}, Error{message: "not initialized"}
    }
    return pool.get(), nil
}
```

Note

`init()` functions are called in dependency order. Circular dependencies are a compile-time error.

9.5 Package Structure

9.5.1 Single-File Package

```
myapp/
  main.haira
  haira.toml
```

9.5.2 Multi-File Package

```
myapp/
  main.haira
  models/
    user.haira
    post.haira
    mod.haira       # Package entry point (optional)
  services/
    auth.haira
    api.haira
  utils/
    helpers.haira
  haira.toml
```

9.5.3 The mod.haira File

Re-exports from a directory:

```
// models/mod.haira
import { User } from "models/user"
import { Post } from "models/post"

// Re-export
export { User, Post }
```

Now consumers can import from the directory:

```
import { User, Post } from "models"
```

9.6 Import Order

Imports should be organized in groups:

```
// 1. Standard library
import "io"
import "json"
import "http"

// 2. External packages
import "github.com/haira-libs/aws"

// 3. Project-local
import "models/user"
import "services/auth"
```

9.7 Circular Dependencies

Circular imports are not allowed:

```
// a.haira
```

```
import "b"   // Error if b imports a

// b.haira
import "a"   // Creates circular dependency
```

Solution: Extract shared code to a third module:

```
// shared.haira
struct Common { }

// a.haira
import "shared"

// b.haira
import "shared"
```

9.8 The haira.toml File

Project configuration:

```
[package]
name = "myapp"
version = "1.0.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
aws = "github.com/haira-libs/aws@1.0.0"
postgres = "github.com/haira-libs/postgres@2.1.0"

[dev-dependencies]
testing = "github.com/haira-libs/testing@1.0.0"

[build]
target = "native"
optimization = "release"
```

9.9 Module Grammar

```
import_statement = "import" import_spec ;

import_spec = string_literal                                (* basic
   *)
            | identifier "from" string_literal              (*
              aliased *)
            | "{" identifier_list "}" "from" string_literal   (*
              selective *)
            | "*" "from" string_literal ;                   (* glob *)

export_statement = "export" "{" identifier_list "}" ;

identifier_list = identifier { "," identifier } ;

```

```
module = { import_statement } { export_statement } { top_level
    } ;
```

Chapter 10

Standard Library

The HAIRA standard library provides essential functionality for common programming tasks. All modules must be explicitly imported.

10.1 Core Modules

Module	Description
`io`	Input/output operations
`string`	String manipulation
`array`	Array operations
`map`	Map/dictionary operations
`math`	Mathematical functions
`conv`	Type conversions

Table 10.1: Core modules

10.2 io Module

Basic input/output operations.

```
import "io"

// Output
io.print("Hello")              // No newline
io.println("Hello")            // With newline
io.printf("Value: %d\n", 42)   // Formatted

// Input
line = io.readln()             // Read line from stdin

// Errors
io.eprintln("Error!")          // Print to stderr
```

10.2.1 io Functions

```
fn print(s: string)
fn println(s: string)
fn printf(format: string, args: ...any)
fn readln() -> string
fn eprintln(s: string)
fn eprintf(format: string, args: ...any)
```

10.3 string Module

String manipulation functions.

```
import "string"

s = "  Hello, World!  "

// Basic operations
string.len(s)                     // 18
string.is_empty("")               // true

// Trimming
string.trim(s)                    // "Hello, World!"
string.trim_left(s)               // "Hello, World!  "
string.trim_right(s)              // "  Hello, World!"

// Case
string.to_upper(s)                // "  HELLO, WORLD!  "
string.to_lower(s)                // "  hello, world!  "

// Search
string.contains(s, "World")       // true
string.starts_with(s, "  H")      // true
string.ends_with(s, "!  ")        // true
string.index_of(s, "o")           // 4
string.last_index_of(s, "o")      // 8

// Split/Join
string.split("a,b,c", ",")        // ["a", "b", "c"]
string.join(["a", "b"], "-")      // "a-b"

// Substring
string.substring(s, 2, 7)         // "Hello"
string.char_at(s, 2)              // "H"

// Replace
string.replace(s, "World", "Haira")   // "  Hello, Haira!  "
string.replace_all(s, " ", "_")       // "__Hello,_World!__"

// Repeat
string.repeat("ab", 3)            // "ababab"
```

10.3.1 string Functions

```
fn len(s: string) -> int
fn is_empty(s: string) -> bool
fn trim(s: string) -> string
fn trim_left(s: string) -> string
fn trim_right(s: string) -> string
fn to_upper(s: string) -> string
fn to_lower(s: string) -> string
fn contains(s: string, sub: string) -> bool
fn starts_with(s: string, prefix: string) -> bool
fn ends_with(s: string, suffix: string) -> bool
fn index_of(s: string, sub: string) -> int
fn last_index_of(s: string, sub: string) -> int
fn split(s: string, sep: string) -> []string
fn join(parts: []string, sep: string) -> string
fn substring(s: string, start: int, end: int) -> string
fn char_at(s: string, index: int) -> string
fn replace(s: string, old: string, new: string) -> string
fn replace_all(s: string, old: string, new: string) -> string
fn repeat(s: string, n: int) -> string
```

10.4 array Module

Array operations.

```
import "array"

arr = [1, 2, 3, 4, 5]

// Basic
array.len(arr)                    // 5
array.is_empty([])                // true

// Access
array.first(arr)                  // 1 (or nil if empty)
array.last(arr)                   // 5 (or nil if empty)
array.get(arr, 2)                 // 3 (or nil if out of bounds)

// Modification (returns new array)
array.push(arr, 6)                // [1, 2, 3, 4, 5, 6]
array.pop(arr)                    // ([1, 2, 3, 4], 5)
array.insert(arr, 0, 0)           // [0, 1, 2, 3, 4, 5]
array.remove(arr, 2)              // [1, 2, 4, 5]

// Slicing
array.slice(arr, 1, 4)            // [2, 3, 4]
array.take(arr, 3)                // [1, 2, 3]
array.drop(arr, 2)                // [3, 4, 5]

// Search
array.contains(arr, 3)            // true
array.index_of(arr, 3)            // 2
array.find(arr, fn(x) { x > 3 }) // 4

// Transform
array.map(arr, fn(x) { x * 2 })  // [2, 4, 6, 8, 10]
array.filter(arr, fn(x) { x > 2 })  // [3, 4, 5]
array.reduce(arr, 0, fn(acc, x) { acc + x })  // 15

// Sort
```

```
array.sort(arr)                     // [1, 2, 3, 4, 5]
array.sort_by(arr, fn(a, b) { b - a })  // [5, 4, 3, 2, 1]

// Other
array.reverse(arr)                  // [5, 4, 3, 2, 1]
array.concat(arr, [6, 7])           // [1, 2, 3, 4, 5, 6, 7]
array.flatten([[1, 2], [3, 4]])     // [1, 2, 3, 4]
array.unique([1, 2, 2, 3])          // [1, 2, 3]
```

10.5 map Module

Map/dictionary operations.

```
import "map"

m = ["a": 1, "b": 2, "c": 3]

// Basic
map.len(m)                          // 3
map.is_empty(m)                     // false

// Access
map.get(m, "a")                     // 1 (or nil if not found)
map.has(m, "a")                     // true

// Modification (returns new map)
map.set(m, "d", 4)                  // ["a": 1, "b": 2, "c": 3,
    "d": 4]
map.remove(m, "b")                  // ["a": 1, "c": 3]

// Iteration helpers
map.keys(m)                         // ["a", "b", "c"]
map.values(m)                       // [1, 2, 3]
map.entries(m)                      // [("a", 1), ("b", 2), ("c",
    3)]

// Transform
map.map_values(m, fn(v) { v * 2 })  // ["a": 2, "b": 4, "c": 6]
map.filter(m, fn(k, v) { v > 1 })   // ["b": 2, "c": 3]

// Merge
map.merge(m, ["d": 4])              // ["a": 1, "b": 2, "c": 3,
    "d": 4]
```

10.6 math Module

Mathematical functions.

```
import "math"

// Constants
math.PI                             // 3.14159265358979
math.E                              // 2.71828182845905

```

```
// Basic
math.abs(-5)                     // 5
math.min(3, 7)                   // 3
math.max(3, 7)                   // 7
math.clamp(5, 0, 10)             // 5
math.clamp(-5, 0, 10)            // 0

// Rounding
math.floor(3.7)                  // 3.0
math.ceil(3.2)                   // 4.0
math.round(3.5)                  // 4.0
math.trunc(3.7)                  // 3.0

// Power/Root
math.pow(2.0, 10.0)              // 1024.0
math.sqrt(16.0)                  // 4.0
math.cbrt(27.0)                  // 3.0

// Exponential/Logarithm
math.exp(1.0)                    // 2.718...
math.log(math.E)                 // 1.0
math.log10(100.0)                // 2.0
math.log2(8.0)                   // 3.0

// Trigonometry
math.sin(0.0)                    // 0.0
math.cos(0.0)                    // 1.0
math.tan(0.0)                    // 0.0
math.asin(0.0)                   // 0.0
math.acos(1.0)                   // 0.0
math.atan(0.0)                   // 0.0
math.atan2(1.0, 1.0)             // 0.785... (PI/4)

// Random
math.random()                    // Random float 0.0..1.0
math.random_int(1, 100)          // Random int 1..100
```

10.7 conv Module

Type conversion utilities.

```
import "conv"

// To string
conv.int_to_string(42)           // "42"
conv.float_to_string(3.14)       // "3.14"
conv.bool_to_string(true)        // "true"

// From string
conv.string_to_int("42")         // (42, nil)
conv.string_to_int("abc")        // (0, Error)
conv.string_to_float("3.14")     // (3.14, nil)
conv.string_to_bool("true")      // (true, nil)

// Number conversions
conv.int_to_float(42)            // 42.0
conv.float_to_int(3.7)              // 3 (truncates)

// Base conversions
conv.int_to_hex(255)                // "ff"
conv.int_to_binary(10)              // "1010"
conv.int_to_octal(8)                // "10"
conv.hex_to_int("ff")               // (255, nil)
```

10.8 Extended Modules

10.8.1 fs Module

File system operations.

```
import "fs"

// Read/Write
content, err = fs.read_file("file.txt")
err = fs.write_file("file.txt", "content")
err = fs.append_file("file.txt", "more")

// File operations
exists = fs.exists("file.txt")
err = fs.remove("file.txt")
err = fs.rename("old.txt", "new.txt")
err = fs.copy("src.txt", "dst.txt")

// Directory operations
err = fs.mkdir("dir")
err = fs.mkdir_all("path/to/dir")
entries, err = fs.read_dir("dir")
err = fs.remove_all("dir")

// File info
info, err = fs.stat("file.txt")
info.size                           // Size in bytes
info.is_dir                         // Is directory?
info.modified                       // Modification time
```

10.8.2 os Module

Operating system interface.

```
import "os"

// Environment
os.getenv("HOME")                   // Get env variable
os.setenv("KEY", "value")           // Set env variable
os.environ()                        // All env variables

// Process
os.args                             // Command line arguments
os.exit(0)                          // Exit with code
os.getpid()                         // Process ID

```

```
// Execution
output, err = os.exec("ls", ["-la"])
```

10.8.3 time Module

Time and duration.

```
import "time"

// Current time
now = time.now()
now.year                         // 2024
now.month                        // 1-12
now.day                          // 1-31
now.hour                         // 0-23
now.minute                       // 0-59
now.second                       // 0-59

// Formatting
time.format(now, "2006-01-02")   // "2024-03-15"

// Parsing
t, err = time.parse("2024-03-15", "2006-01-02")

// Duration
time.sleep(1000)                 // Sleep 1 second (ms)
duration = time.since(start)     // Time since start

// Timers
timer = time.after(5000)         // Channel that fires after 5s
ticker = time.tick(1000)         // Channel that fires every 1s
```

10.8.4 json Module

JSON encoding/decoding.

```
import "json"

struct User {
    name: string
    age: int
}

// Encoding
user = User{name: "Alice", age: 30}
data, err = json.encode(user)     // '{"name":"Alice","age":30}'

// Decoding
user, err = json.decode<User>(data)

// Pretty printing
data, err = json.encode_pretty(user)

// Raw JSON
value, err = json.parse('{"key": "value"}')
```

```
value["key"]                      // "value"
```

10.8.5 http Module

HTTP client and server.

```
import "http"

// GET request
response, err = http.get("https://api.example.com/users")
response.status                   // 200
response.body                     // Response body

// POST request
response, err = http.post(
    "https://api.example.com/users",
    ["Content-Type": "application/json"],
    '{"name": "Alice"}'
)

// Server
server = http.Server{port: 8080}

server.handle("/", fn(req: http.Request) -> http.Response {
    return http.Response{
        status: 200,
        body: "Hello, World!"
    }
})

server.listen()
```

10.8.6 regex Module

Regular expressions.

```
import "regex"

// Compile pattern
pattern, err = regex.compile(r"\d+")

// Match
regex.is_match(pattern, "abc123")  // true

// Find
match = regex.find(pattern, "abc123def")  // "123"
matches = regex.find_all(pattern, "a1b2c3")  // ["1", "2", "3"]

// Replace
regex.replace(pattern, "a1b2", "X")      // "aXb2"
regex.replace_all(pattern, "a1b2", "X") // "aXbX"

// Capture groups
pattern, _ = regex.compile(r"(\w+)@(\w+)")
captures = regex.captures(pattern, "alice@example")
```

```
// captures[0] = "alice@example"
// captures[1] = "alice"
// captures[2] = "example"
```

10.8.7 encoding Module

Encoding and decoding utilities.

```
import "encoding"

// Base64
encoding.base64_encode(data)
encoding.base64_decode(string)

// Hex
encoding.hex_encode(data)
encoding.hex_decode(string)
```

10.9 Agentic Modules

10.9.1 agents Module

Pre-built agent configuration templates for common use cases:

```
import "agents"

// Returns a map with name, system, temperature, max_steps
config = agents.code_reviewer()
config = agents.planner()
config = agents.security_reviewer()
config = agents.summarizer()
config = agents.data_analyst()
config = agents.customer_support()
config = agents.tdd_guide()
config = agents.doc_writer()

// Use with create_agent() for dynamic agent creation
reviewer = create_agent(config, my_provider, [my_tool])
```

10.9.2 observe Module

Observability and cost monitoring:

```
import "observe"
import "langfuse"

// Track costs
total = observe.cost()
agent_cost = observe.agent_cost("Writer")
usage = observe.usage()

// Export to Langfuse
observe.use(langfuse.exporter("", "", ""))
```

```

// Start observability dashboard
observe.start(server)  // Serves /_observe
```

10.10 Testing

Haira has built-in testing support. Tests are declared with the **test** keyword followed by a name and a block of assertions.

10.10.1 Test Blocks

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

fn is_even(n: int) -> bool {
    return n % 2 == 0
}

test "add returns correct sum" {
    assert add(1, 2) == 3, "1+2 should equal 3"
    assert add(0, 0) == 0
    assert add(-1, 1) == 0
}

test "is_even checks parity" {
    assert is_even(4)
    assert not is_even(3)
}
```

10.10.2 Assertions

The **assert** statement checks a condition. If it fails, the test reports the failure and continues.

```
// Boolean assertion
assert is_valid(input)

// Equality check (reports "got X, want Y" on failure)
assert result == 42

// Inequality check
assert a != b

// With custom message
assert count > 0, "count should be positive"
```

10.10.3 Running Tests

```
$ haira test file.haira
```

The compiler generates Go test functions and runs them with `go test`. Each **`test`** block becomes a subtest. Output follows standard test format:

```
--- PASS: TestHaira/add_returns_correct_sum (0.00s)
--- PASS: TestHaira/is_even_checks_parity (0.00s)
PASS
```

Part III

Agentic Orchestration

Chapter 11

Providers

Providers configure LLM backends. They are declared at the top level and referenced by agents.

11.1 Provider Declaration

```
provider provider_name {
    field: value
    field: value
}
```

11.1.1 Cloud Providers

```
provider anthropic {
    api_key: env("ANTHROPIC_API_KEY")
    model: "claude-sonnet-4-20250514"
}

provider openai {
    api_key: env("OPENAI_API_KEY")
    model: "gpt-4o"
}
```

11.1.2 Backend Resolution

The `backend` field selects a known provider and automatically resolves the API endpoint. This eliminates the need to manually construct endpoint URLs:

```
// Backend auto-resolves the endpoint
provider fast {
    backend: "groq"
    api_key: env("GROQ_API_KEY")
    model: "llama-3.1-70b"
}

// Cloudflare Workers AI (needs account_id)
provider cf {
    backend: "cloudflare"
    api_key: env("CF_API_TOKEN")
    account_id: env("CF_ACCOUNT_ID")
    model: "@cf/meta/llama-3.1-70b-instruct"
}

// Local Ollama (defaults to localhost:11434)
provider local {
    backend: "ollama"
    model: "llama3:8b"
}
```

Backend	Auto-Resolved Endpoint
`"openai"`	`https://api.openai.com/v1`
`"azure"`	User-provided `endpoint`
`"cloudflare"`	`https://api.cloudflare.com/.../{account_id}/ai/v1`
`"ollama"`	`http://{host}/v1` (default `localhost:11434`)
`"groq"`	`https://api.groq.com/openai/v1`
`"together"`	`https://api.together.xyz/v1`
`"mistral"`	`https://api.mistral.ai/v1`
`"deepseek"`	`https://api.deepseek.com`
`"fireworks"`	`https://api.fireworks.ai/inference/v1`
`"openrouter"`	`https://openrouter.ai/api/v1`
`"xai"`	`https://api.x.ai/v1`
`"cerebras"`	`https://api.cerebras.ai/v1`

Table 11.1: Known backends and auto-resolved endpoints

The `endpoint` field always overrides auto-resolution when explicitly set.

11.2 Provider Fields

Field	Type	Required	Description
`model`	`string`	Yes	Default model identifier
`api_key`	`string`	No	API authentication key
`backend`	`string`	No	Backend identifier (auto-resolves endpoint)
`endpoint`	`string`	No	Explicit API endpoint (overrides backend)
`host`	`string`	No	Backend host address (local providers)
`account_id`	`string`	No	Account identifier (Cloudflare, etc.)
`api_version`	`string`	No	API version (Azure OpenAI)
`temperature`	`float`	No	Default temperature (0.0–2.0)
`max_tokens`	`int`	No	Default max tokens per response
`input_token_cost`	`float`	No	USD per 1M input tokens
`output_token_cost`	`float`	No	USD per 1M output tokens

Table 11.2: Provider fields

11.3 Environment Variables

The `env()` function reads environment variables at runtime:

```
provider anthropic {
    api_key: env("ANTHROPIC_API_KEY")
    model: env("ANTHROPIC_MODEL") or "claude-sonnet-4-20250514"
}
```

Warning

Never hardcode API keys in source files. Always use `env()` to read them from the environment.

11.4 Referencing Providers

Providers are referenced by name in agent declarations:

```
agent Writer {
    provider: anthropic       // uses anthropic provider
    system: "You are a writer."
}

agent FastHelper {
    provider: local           // uses local Ollama provider
    system: "You are a fast helper."
}
```

11.5 Multiple Providers

A program can declare multiple providers. Agents choose which to use:

```
provider anthropic {
    api_key: env("ANTHROPIC_API_KEY")
    model: "claude-sonnet-4-20250514"
}

provider local {
    backend: "ollama"
    host: "localhost:11434"
    model: "llama3:8b"
}

// Use expensive cloud model for important tasks
agent Analyst {
    provider: anthropic
    system: "You analyze data carefully."
}

// Use cheap local model for simple tasks
agent Classifier {
    provider: local
    system: "Classify the input into categories."
}
```

11.6 MCP Providers

The Model Context Protocol (MCP) enables agents to use tools from external processes. MCP providers spawn a subprocess (or connect via HTTP) and discover its tools automatically.

11.6.1 Stdio Transport

The most common MCP transport. The runtime spawns a subprocess and communicates via JSON-RPC over stdin/stdout:

```
provider filesystem {
    transport: "mcp"
    command: "npx"
    args: ["-y", "@modelcontextprotocol/server-filesystem",
       "/tmp"]
}
```

11.6.2 SSE Transport

For remote MCP servers accessible over HTTP:

```
provider remote_tools {
    transport: "sse"
    endpoint: "http://localhost:9000/sse"
}
```

11.6.3 MCP Provider Fields

Field	**Type**	**Required**	**Description**
`transport`	`string`	Yes	`"mcp"` (stdio) or `"sse"` (HTTP)
`command`	`string`	Stdio only	Executable to spawn
`args`	`[string]`	No	Command arguments
`endpoint`	`string`	SSE only	Server URL
`env`	`map`	No	Environment variables for subprocess
`headers`	`map`	No	HTTP headers (SSE only)

Table 11.3: MCP provider fields

11.6.4 Using MCP in Agents

Agents reference MCP providers with the `mcp` field. The runtime discovers tools from all listed MCP providers and makes them available alongside regular tools:

```
agent Assistant {
    provider: azure_openai
    system: "You are a helpful assistant with file system
       access."
    tools: [greet]              // regular tools
    mcp: [filesystem]           // MCP tools (auto-discovered)
    memory: conversation(max_turns: 10)
}
```

11.6.5 MCP Server

Haira workflows can be exposed as MCP tools for other applications to consume:

```
import "mcp"

workflow Summarize(text: string) -> { summary: string } {
    """Summarize the given text into key points."""
    summary, err = Summarizer.ask(text)
    if err != nil { return { summary: "Error" } }
    return { summary: summary }
}

fn main() {
    mcp_server = mcp.Server([Summarize])
    mcp_server.listen(9000)  // SSE on port 9000
}
```

The `mcp.Server` exposes each workflow as an MCP tool. Tool names, descriptions, and input schemas are derived from the workflow declaration. The server supports both SSE (`.listen(port)`) and stdio (`.serve()`) transports.

11.7 Provider Grammar

```
provider_decl  = "provider" identifier "{" { provider_field }
   "}" ;
provider_field = identifier ":" expression ;
```

Chapter 12

Tools

Tools are typed functions with LLM-visible descriptions. The compiler auto-generates JSON schemas from the type signature, enabling agents to call tools with structured arguments.

12.1 Tool Declaration

```
tool tool_name(param: Type, param: Type = default) ->
    ReturnType {
    """Description visible to the LLM"""

    // implementation
}
```

A tool is like **fn** with two additions:

1. The **tool** keyword instead of **fn**
2. A mandatory triple-quoted description as the first element of the body

12.2 Basic Examples

12.2.1 HTTP Tool

```
import "tools/http"
import "json"

struct SearchResult {
    url: string
    title: string
    snippet: string
}

tool search(query: string, max_results: int = 5) ->
    [SearchResult] {
    """
    Search the web for information.
    Returns up to max_results results.
    Supports boolean operators in query.
    """

    resp, err =
       http.get("https://api.search.com?q={query}&n={max_results}")
    if err != nil { return [], err }
    return json.decode(resp.body, [SearchResult])
}
```

12.2.2 Database Tool

```
import "tools/postgres"

tool get_user(user_id: string) -> User {
    """Look up a user by their ID"""

    user, err = pg.query_one("SELECT * FROM users WHERE id =
       $1", [user_id])
    if err != nil { return User{}, err }
    return user
}
```

12.2.3 External Service Tool

```
import "tools/http"

tool send_slack(channel: string, message: string) -> bool {
    """Send a message to a Slack channel"""

    resp, err =
       http.post("https://slack.com/api/chat.postMessage", {
        channel: channel,
        text: message
    })
    return err == nil
}
```

12.2.4 Agent-Delegating Tool

Tools can call agents internally:

```
tool summarize(text: string) -> string {
    """Summarize text into key points"""

    result, _ = Writer.ask("Summarize concisely: {text}")
    return result
}
```

12.3 Triple-Quote Description

The description is mandatory and must be the first element of the tool body. It is a triple-quoted string (`"""..."""`) that can span multiple lines:

```
tool analyze(data: [DataPoint]) -> Analysis {
    """
    Analyze a dataset and produce a statistical summary.

    Includes:
    - Mean, median, and standard deviation
    - Outlier detection
    - Trend analysis over time

    Use this tool when the user asks about patterns or
       statistics.
    """

    // implementation
}
```

Note

The compiler will emit an error if a tool is missing its description. This is enforced because the description is sent to the LLM as part of the tool-calling protocol.

12.4 Auto-Generated JSON Schema

The compiler generates a JSON schema from the tool's type signature. For example:

```
tool search(query: string, max_results: int = 5) ->
    [SearchResult] {
    """Search the web for information"""
    // ...
}
```

Generates (conceptually):

```
{
  "name": "search",
  "description": "Search the web for information",
  "parameters": {
    "type": "object",
    "properties": {
      "query": { "type": "string" },
      "max_results": { "type": "integer", "default": 5 }
    },
    "required": ["query"]
  }
}
```

This happens at compile time. No hand-written JSON schemas are needed.

12.5 Tool Packs (Standard Library)

Haira ships with built-in tool packs for common integrations:

```
import "tools/http"       // http.get, http.post, http.put,
    http.delete
import "tools/slack"      // slack.send, slack.read_channel
import "tools/github"     // github.create_issue,
    github.list_prs, github.get_diff
import "tools/postgres"   // pg.query, pg.query_one, pg.insert,
    pg.update
import "tools/email"      // email.send
import "tools/fs"         // fs.read, fs.write, fs.list
```

Tool packs provide pre-built `tool` declarations. They can be used directly in agent `tools` lists or called from within other tools and workflows.

12.6 Error Handling in Tools

Tools follow Haira's standard error handling pattern:

```
tool fetch_page(url: string) -> string {
    """Fetch the content of a web page"""

    resp, err = http.get(url)
    if err != nil {
        return "", err
    }
    if resp.status != 200 {
        return "", Error{message: "HTTP {resp.status}"}
    }
    return resp.body
}
```

When a tool returns an error, the agent receives the error message and can decide how to proceed (retry, try a different tool, or report the error to the user).

12.7 Lifecycle Hooks

Tools support optional `@before` and `@after` hooks for pre/post-processing logic. Hooks are useful for validation, logging, rate limiting, or cleanup.

12.7.1 @before Hook

The `@before` block executes before the tool body:

```
tool create_user(name: string, email: string) -> User {
    """Create a new user account."""

    @before {
        assert name != "", "name is required"
        assert email.contains("@"), "invalid email"
    }

    user = db.insert("users", { name: name, email: email })
    return user
}
```

12.7.2 @after Hook

The `@after` block executes after the tool body completes:

```
tool query_db(sql: string) -> [map] {
    """Run a database query and return results."""

    @before {
        io.println("Query: " + sql)
    }

    @after {
        io.println("Query complete")
    }

    results, err = db.query(sql)
    if err != nil { return [], err }
    return results
}
```

Note

Hooks are optional. `@before` and `@after` must appear after the triple-quoted description. If a `@before` hook fails (e.g., via `assert`), the tool body does not execute.

12.8 Tool Grammar

```
tool_decl      = "tool" identifier "(" [ params ] ")" [ "->"
   type ]
                 "{" triple_string [ before_hook ] [ after_hook ]
                 { statement } "}" ;

triple_string = '"""' { any_char } '"""' ;
before_hook    = "@before" block ;
after_hook     = "@after" block ;
```

Chapter 13

Agents

Agents are LLM-powered entities. They have a provider, a system prompt, optional tools, and optional memory. Agents are the primary way to interact with language models in HAIRA.

13.1 Agent Declaration

```
agent AgentName {
    provider: provider_name
    system: "System prompt describing the agent's behavior."
    tools: [tool1, tool2]
    temperature: 0.5
    timeout: 60
    memory: conversation(max_turns: 50)
}
```

13.2 Agent Fields

Field	Type	Required	Description
`provider`	identifier	Yes	Provider to use
`system`	`string`	Yes	System prompt
`tools`	`[tool]`	No	List of tools the agent can call
`temperature`	`float`	No	Sampling temperature (0.0–2.0)
`max_tokens`	`int`	No	Max tokens per response
`max_steps`	`int`	No	Max tool-calling iterations
`memory`	memory type	No	Memory configuration (default: `none`)
`handoffs`	`[agent]`	No	Agents this agent can delegate to
`strategy`	`string`	No	Handoff strategy: `"parallel"` or `"sequential"`
`scope`	`string`	No	Text description of what agent is allowed to help with
`scope_deny`	`string`	No	Rejection message for out-of-scope requests
`timeout`	`int`	No	Timeout in seconds (default: 120)

Table 13.1: Agent fields

13.2.1 Multiline System Prompts

For longer system prompts, use triple-quoted strings:

```
agent SupportBot {
    provider: anthropic
    system: """
        You are a customer support agent for Acme Corp.
        Use the knowledge base to answer questions.
        Look up orders when customers ask about purchases.
        Create tickets for issues you cannot resolve.
        Always be polite and helpful.
    """
    tools: [search_kb, lookup_order, create_ticket]
    temperature: 0.3
}
```

13.3 Agent Methods

Agents have three methods for interacting with the LLM:

13.3.1 .ask() — Text In, Text Out

The simplest interaction. Send a text prompt, get a text response:

```
answer, err = Researcher.ask("What is quantum computing?")
if err != nil {
    io.println("Error: " + err.message)
    return
}
io.println(answer)
```

Signature: `.ask(prompt: string, session: string?) -> (string, Error?)`

13.3.2 .run() — Structured In, Structured Out

For typed interactions. Send a struct, get a typed response. The output type is inferred from the left-side type annotation:

```
struct ResearchRequest {
    topic: string
    depth: string = "detailed"
}

struct ResearchResult {
    summary: string
    sources: [string]
    confidence: float
}

research: ResearchResult, err = Researcher.run(ResearchRequest{
    topic: "quantum computing"
})
```

The compiler generates JSON schemas for both the input and output types. The agent is instructed to return structured data matching the output schema.

Signature: `.run(input: T, session: string?) -> (U, Error?)`

Where `U` is inferred from the left-side type annotation.

13.3.3 .stream() — Text In, Streaming Out

For real-time token-by-token responses. Returns an iterable stream:

```
for chunk in Assistant.stream("Tell me a story") {
    io.print(chunk)
}
io.println()  // newline after stream completes
```

Signature: `.stream(prompt: string, session: string?) -> stream<string>`

The `stream<string>` type is iterable with `for`. Each iteration yields the next token (or chunk) as the LLM generates it.

13.4 Memory

By default, agents are stateless — each call is independent. Adding memory enables agents to remember conversation history across calls.

13.4.1 No Memory (Default)

```
agent Classifier {
    provider: local
    system: "Classify inputs into categories."
}

// Each call is independent
cat1 = Classifier.ask("This is a bug report")      // "technical"
cat2 = Classifier.ask("What did I just send you?") // has no
    context
```

13.4.2 Conversation Memory

Keeps the last N turns of chat history:

```
agent Assistant {
    provider: anthropic
    system: "You are a helpful assistant."
    memory: conversation(max_turns: 50)
}
```

13.4.3 Summary Memory

Summarizes older messages to stay within a token budget. Useful for long-running conversations:

```
agent ProjectHelper {
    provider: anthropic
    system: "You help with long-running software projects."
    memory: conversation(max_turns: 50)
}
```

13.4.4 Memory Types

Type	Parameter	Behavior
`none`	—	Stateless (default)
`conversation(max_turns: N)`	N: int	Keeps last N message pairs

Table 13.2: Built-in memory types

13.5 Sessions

When an agent has memory, calls are scoped to a **session**. A session groups related interactions — for example, one user's conversation.

```
// With session --- memory persists within same session
answer1, _ = Assistant.ask("My name is Alice", session:
    "user-123")
answer2, _ = Assistant.ask("What is my name?", session:
    "user-123")
// answer2 = "Your name is Alice"

// Different session --- separate memory
answer3, _ = Assistant.ask("What is my name?", session:
    "user-456")
// answer3 = "I don't know your name yet"
```

The `session:` parameter is a named argument available on all agent methods:

```
// .ask() with session
reply, err = Agent.ask("Hello", session: session_id)

// .run() with session
result: T, err = Agent.run(input, session: session_id)

// .stream() with session
for chunk in Agent.stream("Hello", session: session_id) {
    io.print(chunk)
}
```

Without a `session:` parameter, each call starts a fresh conversation — even if the agent has memory configured.

13.6 Handoffs

Agents can transfer a conversation to another agent. This is useful when a generalist agent detects that a specialist should take over.

13.6.1 Declaring Handoffs

Use the `handoffs` field to list agents that this agent can delegate to:

```
agent FrontDesk {
    provider: anthropic
    system: """
        You are a front desk agent. Greet users and help them.
        If they have a billing question, hand off to
           BillingAgent.
        If they have a technical issue, hand off to TechAgent.
    """
    handoffs: [BillingAgent, TechAgent]
    memory: conversation(max_turns: 10)
}

agent BillingAgent {
    provider: anthropic
    system: "You handle billing questions. You can look up
       invoices and process refunds."
    tools: [lookup_invoice, process_refund]
    memory: conversation(max_turns: 30)
}

agent TechAgent {
    provider: anthropic
    system: "You handle technical support. You can search docs
       and check system status."
    tools: [search_docs, check_status]
    memory: conversation(max_turns: 30)
}
```

13.6.2 How Handoffs Work

When an agent has `handoffs`, the runtime injects synthetic tools for each target agent (e.g., `transfer_to_BillingAgent`). The LLM decides when to call these tools based on the system prompt and conversation context.

The handoff process:

1. The LLM calls a handoff tool (e.g., `transfer_to_BillingAgent`)
2. The runtime transfers the conversation history to the target agent's session
3. The target agent processes the original message with full context
4. The target agent's response is returned to the caller

13.6.3 .ask() — Automatic Handoffs

When using `.ask()`, handoffs are automatic. The caller sees a seamless response regardless of which agent ultimately handled it:

```
@webhook("/api/chat")
workflow Chat(message: string, session_id: string) -> { reply:
   string } {
    // FrontDesk may hand off to BillingAgent or TechAgent
       internally
    // The caller just gets the final reply
    reply, err = FrontDesk.ask(message, session: session_id)
    if err != nil { return { reply: "Something went wrong." } }
    return { reply: reply }
}
```

13.6.4 .run() — Manual Handoff Control

When using `.run()`, the workflow receives an `AgentResult` that includes handoff information, allowing manual inspection or override:

```
@webhook("/api/chat")
workflow Chat(message: string, session_id: string) -> { reply:
   string } {
    result: AgentResult, err = FrontDesk.run(message, session:
       session_id)
    if err != nil { return { reply: "Something went wrong." } }

    if result.handed_off_to != nil {
        io.println("Handed off to: " + result.handed_off_to)
    }

    return { reply: result.reply }
}
```

13.6.5 Handoff Chains

Target agents can themselves have handoffs, creating delegation chains. The runtime follows the chain until an agent produces a final response (up to a configurable depth limit):

```
agent L1Support {
    provider: anthropic
    system: "You handle basic support. Escalate complex issues
       to L2."
    handoffs: [L2Support]
}

agent L2Support {
    provider: anthropic
    system: "You handle complex technical issues. Escalate
       critical outages to L3."
    tools: [search_docs, check_status]
    handoffs: [L3Support]
}

agent L3Support {
    provider: anthropic
    system: "You handle critical outages. You have full system
       access."
    tools: [search_docs, check_status, restart_service,
       page_oncall]
}
```

13.7 Tool-Calling Loop

When an agent has tools, the runtime executes a loop:

1. Send the prompt (and conversation history) to the LLM
2. If the LLM requests a tool call, execute the tool
3. Send the tool result back to the LLM
4. Repeat until the LLM produces a final response (or `max_steps` is reached)

This is transparent to the caller:

```
// The agent may call search() multiple times internally
// before producing a final answer
answer, err = Researcher.ask("Find recent papers on quantum
    error correction")
```

The `max_steps` field limits how many tool-calling iterations are allowed:

```
agent Researcher {
    provider: anthropic
    system: "You are a thorough researcher."
    tools: [search, read_url]
    max_steps: 10      // at most 10 tool calls per request
    temperature: 0.2
}
```

13.8 Error Handling

Agent calls can fail. Always check for errors:

```
answer, err = Researcher.ask("Complex query here")
if err != nil {
    io.println("Agent error: " + err.message)
    // Handle: retry, fallback, or report
}
```

Common error cases:

- LLM provider unreachable or returns an error
- Max steps exceeded (tool-calling loop did not converge)
- Structured output did not match expected schema
- Timeout exceeded

13.9 Examples

13.9.1 Stateless Classifier

```
agent Classifier {
    provider: local
    system: "Classify the input as: positive, negative, or neutral. Reply with just the label."
    temperature: 0.0
}

fn main() {
    label, _ = Classifier.ask("I love this product!")
    io.println(label)  // "positive"
}
```

13.9.2 Research Agent with Tools

```
agent Researcher {
    provider: anthropic
    system: "You are a thorough researcher. Always cite your sources."
    tools: [search, read_url]
    temperature: 0.2
    max_steps: 15
}

fn main() {
    answer, err = Researcher.ask("What are the latest advances in fusion energy?")
    if err != nil {
        io.println("Error: " + err.message)
        return
    }
    io.println(answer)
}
```

13.9.3 Conversational Assistant

```
agent Tutor {
    provider: anthropic
    system: "You are a patient math tutor. Explain concepts step by step."
    memory: conversation(max_turns: 30)
    temperature: 0.5
}

fn main() {
    session = "student-001"

    reply1, _ = Tutor.ask("What is a derivative?", session: session)
    io.println(reply1)

    reply2, _ = Tutor.ask("Can you give me an example?", session: session)
    io.println(reply2)

    reply3, _ = Tutor.ask("Now explain the chain rule",
        session: session)
    io.println(reply3)
    // Tutor remembers the full conversation context
}
```

13.10 Delegation Strategy

When an agent has `handoffs`, the default behavior is LLM-driven: the agent decides when to delegate via synthetic tool calls. The `strategy` field overrides this with deterministic routing:

13.10.1 Parallel Strategy

Fan out to all handoff targets concurrently, aggregate their replies:

```
agent ResearchTeam {
    provider: anthropic
    system: "You coordinate research across multiple
        specialists."
    handoffs: [WebSearcher, PaperAnalyzer, NewsScanner]
    strategy: "parallel"
}

// All three agents run concurrently, results are combined
result, err = ResearchTeam.ask("Latest advances in quantum
    computing")
```

13.10.2 Sequential Strategy

Chain through handoff targets in order, each output feeds the next:

```
agent Pipeline {
    provider: anthropic
    system: "You orchestrate a content pipeline."
    handoffs: [Researcher, Writer, Editor]
    strategy: "sequential"
}

// Researcher -> Writer -> Editor, each builds on the previous
result, err = Pipeline.ask("Write an article about AI safety")
```

Strategy	**Behavior**	**Use Case**
(default)	LLM decides via tool calls	Dynamic routing
`"parallel"`	Fan-out to all targets concurrently	Research, voting
`"sequential"`	Chain targets in order	Pipelines, refinement

Table 13.3: Handoff strategies

13.11 Pre-built Agent Templates

The `agents` stdlib package provides reusable agent configurations:

```
import "agents"

// Get a pre-configured template
config = agents.code_reviewer()

// Use with create_agent() to instantiate
reviewer = create_agent(config, claude, [analyze_code])
result, err = reviewer.ask("Review this pull request")
```

Available templates: `code_reviewer()`, `planner()`, `security_reviewer()`, `summarizer()`, `data_analyst()`, `customer_support()`, `tdd_guide()`, `doc_writer()`.

13.12 Agent Grammar

```
agent_decl    = "agent" identifier "{" { agent_field } "}" ;

agent_field   = identifier ":" agent_value ;

agent_value   = expression
              | string_lit
              | triple_string
              | "[" ident_list "]"
              | memory_value ;

memory_value  = "conversation" "(" [ arguments ] ")"
              | "summary" "(" [ arguments ] ")"
              | "none" ;
```

Chapter 14

Workflows

Workflows are the central abstraction in HAIRA. A workflow is a composable function with a trigger, typed parameters, and a typed return value. Workflows replace both n8n-style visual automations and LangGraph-style agent graphs.

14.1 Workflow Declaration

A workflow is a function preceded by a trigger decorator:

```
@trigger_type("config")
workflow WorkflowName(param: Type, param: Type) -> ReturnType {
    // body
}
```

Workflows without a trigger decorator can be called as sub-workflows but cannot be triggered externally.

14.2 Triggers

Triggers define how a workflow is invoked. They use the `@` decorator syntax.

14.2.1 @webhook — HTTP Endpoint

```
@webhook("/summarize")
workflow Summarizer(url: string) -> Summary {
    content = fetch_page(url)
    summary = Researcher.ask("Summarize: {content}")
    return Summary{ text: summary, word_count: summary.len() }
}
```

The workflow parameters become the JSON request body. The return type becomes the JSON response body.

```
$ curl -X POST http://localhost:8080/summarize \
    -d '{"url": "https://example.com/article"}'

{"text": "...", "word_count": 150}
```

Options:

```
@webhook("/path")                          // POST (default)
@webhook("/path", method: "GET")    // GET
@webhook("/path", method: "PUT")    // PUT
```

14.2.2 @websocket — Bidirectional Streaming

```
@websocket("/chat")
workflow Chat(message: string, session_id: string) ->
    stream<string> {
    return Assistant.stream(message, session: session_id)
}
```

WebSocket workflows receive messages and can return `stream<string>` for real-time streaming responses.

14.2.3 @cron — Scheduled Execution

```
@cron("0 9 * * *")  // every day at 9 AM
workflow DailyReport() -> Report {
    data = fetch_metrics()
    report: Report, _ = Analyst.run(AnalyzeRequest{ data: data
        })
    send_email("team@company.com", "Daily Report",
        report.summary)
    return report
}
```

Cron expressions follow standard cron syntax (minute, hour, day-of-month, month, day-of-week).

14.2.4 @event — Event-Driven

```
@event("order.created")
workflow ProcessOrder(order: Order) -> OrderResult {
    validated = validate_order(order)
    if validated.has_errors {
        return OrderResult{ status: "rejected" }
    }
    charge_payment(order)
    return OrderResult{ status: "confirmed" }
}
```

Events are published using `haira.emit()`:

```
haira.emit("order.created", Order{ id: "123", total: 99.99 })
```

14.2.5 @manual — CLI Only

```
@manual
```

```
workflow Migrate(version: string) -> { status: string } {
    // only invoked from CLI or main()
    run_migrations(version)
    return { status: "done" }
}
```

```
$ haira run Migrate --input '{"version": "v2"}'
```

14.2.6 No Trigger (Sub-Workflow)

Workflows without a trigger can only be called from other workflows or from `main()`:

```
workflow EnrichLead(email: string) -> EnrichedLead {
    company = lookup_company(email)
    social = lookup_social(email)
    return EnrichedLead{ email: email, company: company,
       social: social }
}
```

14.3 Sequential Steps

The workflow body is sequential by default. Each line can use the results of previous lines:

```
@manual
workflow ArticlePipeline(topic: string) -> Article {
    // Step 1: Research
    research: ResearchResult, err =
       Researcher.run(ResearchRequest{ topic: topic })
    if err != nil { return Article{}, err }

    // Step 2: Write draft
    draft: Article, err = Writer.run(WriteRequest{
        content: research.summary,
        sources: research.sources
    })
    if err != nil { return Article{}, err }

    return draft
}
```

14.4 Named Steps

The **`step`** keyword creates named, observable regions within a workflow body. Each step has a string label and a block of statements. The runtime automatically emits structured telemetry for every step—start time, end time, duration, and success/failure status—giving production-grade observability with zero manual logging.

14.4.1 Syntax

```
step "Step Name" {
    // statements
}
```

A **step** is only valid inside a **workflow** body (including inside **spawn** blocks within workflows). The step name must be a string literal.

14.4.2 Transparent Scope

Steps are *not* functions—they are labeled regions of the enclosing workflow. Variables assigned inside a step are visible to all subsequent steps and to the rest of the workflow body. A **return** inside a step returns from the enclosing workflow.

```
@webhook("/api/process")
workflow ProcessData(input: string) -> { status: string, count:
   int } {
    step "Parse input" {
        records = parse_csv(input)?
        count = len(records)
    }

    step "Validate" {
        // `records` and `count` are visible here
        errors = validate_records(records)
        if len(errors) > 0 {
            // returns from ProcessData, not just the step
            return { status: "invalid", count: 0 }
        }
    }

    step "Save to database" {
        save_all(records)?
    }

    return { status: "ok", count: count }
}
```

This matches the n8n/Zapier mental model: every node's output flows to downstream nodes.

14.4.3 Automatic Telemetry

The runtime emits structured log events for each step without any explicit logging code:

```
[ProcessData] step:start  "Parse input"
[ProcessData] step:end    "Parse input"        duration=42ms
   status=success
[ProcessData] step:start  "Validate"
[ProcessData] step:end    "Validate"           duration=3ms
   status=success
[ProcessData] step:start  "Save to database"
[ProcessData] step:end    "Save to database"   duration=156ms
   status=success
```

If a step fails (via error propagation or early return), the telemetry reflects it:

```
[ProcessData] step:start  "Validate"
```

```
[ProcessData] step:end     "Validate"           duration=3ms
    status=failed    error="Validation failed"
```

This gives operations teams per-step visibility in production:

- Which step is slow (duration)
- Which step failed (status + error message)
- Dashboards and alerts per workflow per step

14.4.4 Parallel Steps

Use existing **spawn** blocks to run steps concurrently—no new syntax needed:

```
workflow ProcessOrder(order: Order) -> Result {
    step "Validate order" {
        validate(order)?
    }

    // These run in parallel
    spawn {
        step "Check inventory" {
            stock = check_stock(order.items)
        }
        step "Verify payment" {
            payment = verify_payment(order.card)
        }
    }

    // Back to sequential --- stock, payment available after
        spawn completes
    step "Fulfill" {
        fulfill(order, stock, payment)
    }

    return Result{ status: "fulfilled" }
}
```

Telemetry shows overlapping timestamps for parallel steps:

```
[ProcessOrder] step:start   "Check inventory"
[ProcessOrder] step:start   "Verify payment"
[ProcessOrder] step:end     "Check inventory"    duration=120ms
    status=success
[ProcessOrder] step:end     "Verify payment"     duration=200ms
    status=success
```

Variables from parallel steps flow to workflow scope after the **spawn** block completes (spawn is a synchronization point).

14.4.5 Real-World Example

A data pipeline with multiple stages, each automatically traced:

```
@webhook("/api/upload-config")
workflow UploadConfig(file_path: string) -> { status: string,
    message: string } {
```

```
    filename = file_path | string.split("/") | last()

    step "Open Excel" {
        wb = excel.open(file_path)?
        defer wb.close()
        index_rows = wb.read_sheet("index")?
    }

    step "Extract table data" {
        table_data = {}
        for row in index_rows {
            sheet = wb.read_sheet(row["sheet_name"])?
            table_data[row["table_name"]] = sheet
        }
    }

    step "Validate" {
        validation = validate_data(table_data)
        if !validation.is_valid {
            return { status: "error", message: "Validation
               failed: ${validation.errors}" }
        }
    }

    step "Push to GitLab" {
        branch = "feature/${filename}"
        create_branch(branch)?
        for name, data in table_data {
            sql = generate_sql(name, data)
            commit_file(branch, "${name}.sql", sql)?
        }
        create_merge_request(branch, "Update ${filename}")?
    }

    step "Wait for CI/CD" {
        poll_pipeline(branch)?
    }

    return { status: "success", message: "Deployed ${filename}"
       }
}
```

Without **step**, this workflow would need 10+ manual `io.println` calls to achieve the same observability. With **step**, every stage is automatically traced in production.

14.5 Parallel Execution

Use **spawn** blocks to run steps in parallel:

```
@manual
workflow ReviewArticle(article: Article) -> [Review] {
    reviews = spawn {
        Reviewer.run(ReviewRequest{ article: article, focus:
           "accuracy" })
        Reviewer.run(ReviewRequest{ article: article, focus:
           "clarity" })
        Reviewer.run(ReviewRequest{ article: article, focus:
            "engagement" })
    }
    return reviews
}
```

A **spawn** block runs all statements concurrently and returns a list of results when all complete. This is equivalent to `Promise.all()` in JavaScript.

14.6 Branching

Standard **if**/**else** and **match** work inside workflows:

```
@event("order.created")
workflow ProcessOrder(order: Order) -> OrderResult {
    validated = validate_order(order)

    if validated.has_errors {
        send_slack("#alerts", "Order {order.id} failed")
        return OrderResult{ status: "rejected" }
    }

    stock = check_stock(validated.items)

    if stock.available {
        charge_payment(order)
        send_email(order.customer_email, "Order confirmed!")
        return OrderResult{ status: "confirmed" }
    } else {
        send_email(order.customer_email, "Items out of stock")
        return OrderResult{ status: "backordered" }
    }
}
```

14.7 Loops

```
@manual
workflow ProcessBatch(urls: [string]) -> [string] {
    results = for url in urls {
        content = fetch_page(url)
        Researcher.ask("Summarize: {content}")
    }
    return results
}
```

Loops inside workflows are sequential by default. For parallel iteration, use **spawn**:

```
@manual
workflow ProcessBatchParallel(urls: [string]) -> [string] {
    results = spawn {
        for url in urls {
            fetch_page(url) | Researcher.ask("Summarize: {_}")
        }
    }
    return results
}
```

14.8 Sub-Workflows

Workflows can call other workflows like functions:

```
workflow ScoreLead(company: Company, social: SocialProfile) ->
   ScoreResult {
    score: ScoreResult, err = Analyst.run(ScoreRequest{
        company: company,
        social: social
    })
    return score
}

workflow EnrichLead(email: string) -> EnrichedLead {
    company = lookup_company(email)
    social = lookup_social(email)
    score = ScoreLead(company, social)
    return EnrichedLead{ email: email, company: company, score:
       score.value }
}

@webhook("/new-lead")
workflow SalesAutomation(email: string) -> { result: string } {
    lead = EnrichLead(email)

    if lead.score > 0.8 {
        send_slack("#sales", "Hot lead: {email}")
    }

    return { result: "processed" }
}
```

Sub-workflows compose naturally. The compiler validates type compatibility at each call site.

14.9 Error Handling

Workflows use standard HAIRA error handling:

```
@manual
workflow SafeProcess(data: string) -> { result: string } {
    result, err = risky_operation(data)
    if err != nil {
        send_slack("#errors", "Failed: {err.message}")
        result = fallback_operation(data)
    }
    return { result: result }
}
```

14.10 Running Workflows

14.10.1 From main()

```
fn main() {
    result, err = ArticlePipeline("AI agents")
    if err != nil {
        io.println("Failed: " + err.message)
        return
    }
    io.println(result.title)
}
```

14.10.2 Server Mode

Start a server that listens for all triggered workflows:

```
fn main() {
    server = http.Server([
        Summarizer,          // POST /summarize
        Chat,                // WebSocket /chat
        ProcessOrder,        // on order.created events
        DailyReport,         // cron 0 9 * * *
    ])

    io.println("Haira server running on :8080")
    server.listen(8080)
}
```

14.10.3 CLI

```
# Run a workflow directly
$ haira run ArticlePipeline --input '{"topic": "AI agents"}'

# Start server mode
$ haira serve main.haira --port 8080

# Run with environment
$ ANTHROPIC_API_KEY=sk-... haira serve main.haira
```

14.11 Orchestration Patterns

Haira workflows compose naturally into common orchestration patterns using existing language primitives. No special keywords or frameworks are needed.

14.11.1 Sequential Chain

Pipe the output of one agent into the next:

```
workflow Analyze(text: string) -> { result: string } {
    summary, err = Summarizer.ask(text)
    sentiment, err = SentimentAnalyzer.ask(summary)
    report, err = ReportWriter.ask(sentiment)
    return { result: report }
}
```

14.11.2 Parallel Fan-Out

Run multiple agents concurrently and collect results:

```
workflow Research(topic: string) -> { synthesis: string } {
    results = spawn {
        WebSearcher.ask(topic)
        PaperAnalyzer.ask(topic)
        NewsScanner.ask(topic)
    }
    synthesis, err = Synthesizer.ask(results)
    return { synthesis: synthesis }
}
```

14.11.3 Router / Dispatcher

One agent classifies intent, then `match` routes to a specialist:

```
workflow Support(message: string, session_id: string) -> {
   reply: string } {
    intent, err = Router.ask(message)
    reply, err = match intent {
        "billing"   => BillingAgent.ask(message, session:
           session_id)
        "technical" => TechAgent.ask(message, session:
           session_id)
        _           => GeneralAgent.ask(message, session:
           session_id)
    }
    return { reply: reply }
}
```

14.11.4 Handoff

Agents delegate to other agents automatically via the `handoffs` field. See Chapter 13, Handoffs section.

14.11.5 Iterative Refinement

An agent produces output, a critic evaluates it, and the loop repeats until approved:

```
workflow Refine(draft: string) -> { result: string } {
    current = draft
    for i in 0..3 {
        critique, err = Critic.ask(current)
        if critique == "approved" { break }
        current, err = Writer.ask("Improve based on:
           ${critique}\n\n${current}")
    }
    return { result: current }
}
```

14.11.6 Supervisor

A planner creates a task list, workers execute each step, and a supervisor evaluates results:

```
workflow ComplexTask(goal: string) -> { result: string } {
    plan, err = Planner.ask(goal)

    for step in plan.steps {
        worker = match step.type {
            "code"     => Coder
            "research" => Researcher
            "review"   => Reviewer
        }
        result, err = worker.ask(step.instruction)

        evaluation, err = Supervisor.ask("Evaluate: ${result}")
        if evaluation == "retry" {
            result, err = worker.ask("Try again:
               ${step.instruction}")
        }
    }

    return { result: result }
}
```

14.11.7 Voting / Consensus

Multiple agents answer the same question, then an aggregator picks the best:

```
workflow Decide(question: string) -> { answer: string } {
    votes = spawn {
        ExpertA.ask(question)
        ExpertB.ask(question)
        ExpertC.ask(question)
    }
    consensus, err = Aggregator.ask(votes)
    return { answer: consensus }
}
```

14.11.8 Human-in-the-Loop

For workflows that require human approval, split into two workflows connected by shared state. The first workflow does the AI work and sends a notification. The second workflow handles the human's response:

```
tool send_email(to: string, subject: string, body: string) ->
   string {
    """Send an email notification."""
    resp, err =
       http.post("https://api.sendgrid.com/v3/mail/send", {
        to: to, subject: subject, body: body
    })
    return resp.body
}

@webhook("/api/publish")
workflow SubmitForReview(content: string) -> { review_id:
   string } {
    edited, err = Editor.ask(content)
    review_id = save_review(edited)

    send_email(
        to: "admin@company.com",
        subject: "Review needed",
        body: "Approve:
           https://app.com/api/review?id=${review_id}&action=approve"
    )

    return { review_id: review_id }
}

@webhook("/api/review")
workflow HandleReview(id: string, action: string) -> { status:
   string } {
    if action == "approve" {
        content = get_review(id)
        publish(content)
        return { status: "published" }
    }
    return { status: "rejected" }
}
```

No new primitives are needed. The “event” is just another HTTP request hitting another workflow. Notifications go out via tools (email, Slack, etc.), and the callback URL points back to a `@webhook` workflow.

14.11.9 Pattern Summary

Pattern	**Primitives Used**	**New Syntax?**
Sequential chain	statements	No
Named steps	**`step`** blocks	Keyword
Parallel fan-out	**`spawn`**	No
Router / dispatcher	**`match`** + agents	No
Handoff	**`handoffs`** agent field	Field only
Iterative refinement	**`for`** + **`break`**	No
Supervisor	loop + **`match`** + agents	No
Voting / consensus	**`spawn`** + aggregation	No
Human-in-the-loop	two workflows + tools	No
Verification loop	**`verify`** + **`@retry`**	Keyword

Table 14.1: Orchestration patterns and their primitives

14.12 Verification Loops

The **`verify`** block groups assertions inside a workflow step. When combined with **`@retry`**, failed assertions automatically trigger a retry of the step.

```
@webhook("/api/generate")
workflow GenerateCode(spec: string) -> { code: string } {
    @retry(3)
    step "Generate and validate" {
        code, err = Coder.ask(spec)
        if err != nil { return { code: "" } }

        verify {
            assert code.contains("fn main"), "must have main
                function"
            assert !code.contains("unsafe"), "no unsafe code
                allowed"
        }
    }

    return { code: code }
}
```

When a **`verify`** assertion fails inside a **`@retry`** step, the step is retried up to the specified number of times. If all retries fail, the workflow continues with the last result.

Note

`verify` is only meaningful inside steps decorated with **`@retry`**. Without **`@retry`**, a failed assertion simply panics (as with standalone **`assert`**).

14.13 Lifecycle Hooks

Steps and workflows support lifecycle hooks for error handling, success callbacks, and cancellation:

```
@webhook("/api/process")
workflow ProcessData(input: string) -> { status: string } {
    step "Process" {
        result = heavy_computation(input)?
    }

    onerror(err) {
        send_slack("#alerts", "Process failed: ${err}")
    }

    onsuccess {
        send_slack("#ops", "Process completed successfully")
    }

    return { status: "ok" }
}
```

14.14 Eval Framework

The **eval** declaration defines automated agent evaluations with test cases and scoring thresholds:

```
eval "classifier accuracy" {
    agent: Classifier
    cases: [
        { input: "I love this!", expected: "positive" },
        { input: "This is terrible", expected: "negative" },
        { input: "It's okay I guess", expected: "neutral" }
    ]
    threshold: 0.8
}
```

Run evaluations with the CLI:

```
$ haira eval main.haira

Running eval: classifier accuracy
  [PASS] Case 1: I love this!
  [PASS] Case 2: This is terrible
  [FAIL] Case 3: It's okay I guess
         Expected: neutral
         Got:      positive

Score: 66.7% (2/3 passed)
Result: FAIL (threshold: 80%, got: 67%)
```

14.15 Cross-Harness Export

Haira programs can be exported as Claude Code agent configurations:

```
$ haira build main.haira --target claude-code
```

This generates:

```
.claude/
  agents/agent-name.md       # Agent system prompt + config
  commands/workflow-name.md  # Slash commands for workflows
  mcp-servers.json           # Points to MCP binary
bin/
  haira-tools                # MCP server with custom tools
```

Each agent becomes a Claude Code agent markdown file. Custom tools are compiled into an MCP server binary that Claude Code connects to via `mcp-servers.json`.

14.16 Workflow Grammar

```
workflow_decl = { decorator } "workflow" identifier
                "(" [ params ] ")" [ "->" type ] block
                { lifecycle_hook } ;

```

```
decorator      = "@" identifier [ "(" [ arguments ] ")" ] ;

spawn_block    = "spawn" "{" { statement } "}" ;

step_stmt      = "step" string_literal { decorator } block
                 { lifecycle_hook } ;

verify_stmt    = "verify" block ;

eval_decl      = "eval" string_literal "{" { eval_field } "}" ;

eval_field     = identifier ":" expression ;

lifecycle_hook = "onerror" "(" identifier ")" block
               | "onsuccess" [ "(" identifier ")" ] block
               | "oncancel" block ;
```

Part IV

Implementation

Chapter 15

Complete Grammar

This chapter provides the complete formal grammar for HAIRA in Extended Backus-Naur Form (EBNF), including all core language constructs and agentic extensions (providers, tools, agents, and workflows).

15.1 Notation

Notation	Meaning
`=`	Definition
`|`	Alternative
`[ ... ]`	Optional (0 or 1)
`{ ... }`	Repetition (0 or more)
`( ... )`	Grouping
`"..."`	Terminal string
`;`	End of production

Table 15.1: EBNF notation

15.2 Lexical Grammar

15.2.1 Characters

```
letter         = "a".."z" | "A".."Z" | "_" ;
digit          = "0".."9" ;
hex_digit      = digit | "a".."f" | "A".."F" ;
octal_digit    = "0".."7" ;
binary_digit   = "0" | "1" ;
any_char       = (* any Unicode character *) ;
```

15.2.2 Whitespace and Comments

```
whitespace    = " " | "\t" | "\r" ;
newline       = "\n" ;
line_comment  = "//" { any_char } newline ;
block_comment = "/*" { any_char | block_comment } "*/" ;
```

```
doc_comment    = "///" { any_char } newline ;
```

Note

Block comments nest: `/* outer /* inner */ still outer */` is valid. Doc comments (`///`) attach to the immediately following declaration and are preserved in compiled metadata.

15.2.3 Identifiers

```
identifier     = letter { letter | digit } ;
```

15.2.4 Keywords

```
keyword        = "agent" | "and" | "assert" | "async" | "break"
               | "catch" | "const" | "continue" | "default"
               | "defer" | "else" | "enum" | "err" | "errdefer"
               | "export" | "false" | "fn" | "for" | "from"
               | "if" | "impl" | "import" | "in" | "let"
               | "match" | "nil" | "none" | "not" | "ok"
               | "oncancel" | "onerror" | "onsuccess" | "or"
               | "orelse" | "provider" | "pub" | "return"
               | "select" | "some" | "spawn" | "step" | "struct"
               | "test" | "tool" | "trait" | "true" | "try"
               | "type" | "while" | "workflow" ;
```

Note

The keywords **`agent`**, **`provider`**, **`tool`**, and **`workflow`** are reserved for agentic declarations. The keyword **`unsafe`** is reserved for future use. All keywords are case-sensitive.

15.2.5 Literals

```
integer_lit    = decimal_lit | hex_lit | octal_lit | binary_lit ;
decimal_lit    = digit { digit | "_" } ;
hex_lit        = "0" ("x"|"X") hex_digit { hex_digit | "_" } ;
octal_lit      = "0" ("o"|"O") octal_digit { octal_digit | "_" }
    ;
binary_lit     = "0" ("b"|"B") binary_digit { binary_digit | "_"
    } ;

float_lit      = decimal_lit "." decimal_lit [ exponent ]
               | decimal_lit exponent ;
exponent       = ("e"|"E") ["+"|"-"] decimal_lit ;

string_lit     = ’"’ { string_char } ’"’
               | "r" ’"’ { raw_char } ’"’
               | triple_string ;
triple_string  = ’"""’ { any_char } ’"""’ ;
string_char    = any_char - (’"’ | "\" | newline)
```

```
                  | escape_seq ;
escape_seq        = "\" ( "n" | "r" | "t" | "\" | '"' | "0"
                  | "x" hex_digit hex_digit
                  | "u" "{" hex_digit { hex_digit } "}" ) ;
raw_char          = any_char - '"' ;

bool_lit          = "true" | "false" ;
nil_lit           = "nil" ;
```

Note

Triple-quoted strings (`"""..."""`) preserve newlines and leading indentation. They are used for tool descriptions, agent system prompts, and multi-line string literals. The common leading whitespace is stripped at compile time.

15.2.6 Operators

```
operator          = arith_op | comp_op | logic_op | bitwise_op
                  | shift_op | assign_op | other_op ;

arith_op          = "+" | "-" | "*" | "/" | "%" ;
comp_op           = "==" | "!=" | "<" | ">" | "<=" | ">=" ;
logic_op          = "&&" | "||" | "!" ;
bitwise_op        = "&" | "|" | "^" | "~" ;
shift_op          = "<<" | ">>" ;
assign_op         = "=" | "+=" | "-=" | "*=" | "/="
                  | "&=" | "|=" | "^=" | "<<=" | ">>=" | ">>>=" ;
other_op          = "|>" | ".." | "..=" | "<-" | "->" | "=>" | "?"
    | "@" ;
```

Note

The `|>` operator is the pipe operator for function composition. The `@` symbol is used for decorator syntax on workflow declarations. The `<-` operator is used for channel send and receive operations.

15.2.7 Delimiters

```
delimiter         = "(" | ")" | "[" | "]" | "{" | "}"
                  | "," | ":" | "." | ";" | "#" ;
```

15.3 Syntactic Grammar

15.3.1 Program Structure

```
program           = { import_stmt } { top_level } ;

top_level         = fn_decl
                  | struct_decl
                  | enum_decl
```

```
                | type_alias
                | const_decl
                | impl_block
                | constraint_decl
                | provider_decl
                | tool_decl
                | agent_decl
                | workflow_decl ;
```

15.3.2 Imports

```
import_stmt    = "import" import_spec ;

import_spec    = string_lit
               | identifier "from" string_lit
               | "{" ident_list "}" "from" string_lit
               | "*" "from" string_lit ;

ident_list     = identifier { "," identifier } ;
```

15.3.3 Core Declarations

```
fn_decl        = [ attributes ] "fn" [ receiver ] identifier
                 [ generic_params ] "(" [ params ] ")"
                 [ "->" type ] [ where_clause ] block ;

receiver       = "(" identifier ":" [ "*" ] type ")" ;

generic_params = "<" generic_param { "," generic_param } ">" ;

generic_param  = identifier [ ":" constraint_list ] ;

constraint_list = identifier { "+" identifier } ;

where_clause   = "where" where_bound { "," where_bound } ;

where_bound    = identifier ":" constraint_list ;

params         = param { "," param } ;

param          = identifier ":" [ "..." ] type [ "=" expression
   ] ;

struct_decl    = [ attributes ] "struct" identifier
                 [ generic_params ] "{" { field_decl } "}" ;

field_decl     = [ attributes ] identifier ":" type ;

enum_decl      = [ attributes ] "enum" identifier
                 [ generic_params ] "{" enum_variants "}" ;

enum_variants = enum_variant { "," enum_variant } [ "," ] ;

```

```
enum_variant   = identifier [ "(" type_list ")" ] ;

type_alias     = "type" identifier [ generic_params ] "=" type ;

const_decl     = "const" identifier [ ":" type ] "=" expression ;

impl_block     = "impl" [ generic_params ] identifier
                 "for" type [ where_clause ]
                 "{" { fn_decl } "}" ;

constraint_decl = "constraint" identifier [ generic_params ]
                  "{" { fn_signature } "}" ;

fn_signature   = "fn" identifier "(" [ type_params ] ")" [ "->"
    type ] ;

type_params    = type { "," type } ;
```

15.3.4 Attributes

Attributes use the `#[...]` syntax and can be applied to functions, structs, enums, and fields. They are distinct from the `@` decorator syntax used for workflow triggers.

```
attributes     = { attribute } ;

attribute      = "#[" identifier [ "(" attr_args ")" ] "]" ;

attr_args      = attr_arg { "," attr_arg } ;

attr_arg       = identifier [ "=" literal ] ;
```

15.3.5 Decorators

Decorators use the `@` syntax and are used exclusively for workflow trigger declarations.

```
decorator      = "@" identifier [ "(" [ arguments ] ")" ] ;
```

15.3.6 Provider Declarations

Providers configure LLM backends. They are declared at the top level and referenced by name in agent declarations.

```
provider_decl  = "provider" identifier "{" { provider_field }
    "}" ;

provider_field = identifier ":" expression ;
```

15.3.7 Tool Declarations

Tools are typed functions with a mandatory triple-quoted description. The compiler auto-generates JSON schemas from the type signature.

```
tool_decl      = "tool" identifier "(" [ params ] ")" [ "->"
    type ]
                 "{" triple_string { statement } "}" ;

triple_string = '"""' { any_char } '"""' ;
```

Note

The `triple_string` must be the first element of the tool body. The compiler emits an error if a tool is missing its description.

15.3.8 Agent Declarations

Agents are LLM-powered entities with a model, system prompt, optional tools, and optional memory.

```
agent_decl     = "agent" identifier "{" { agent_field } "}" ;

agent_field    = identifier ":" agent_value ;

agent_value    = expression
               | string_lit
               | triple_string
               | "[" ident_list "]"
               | memory_value ;

memory_value   = "conversation" "(" [ arguments ] ")"
               | "summary" "(" [ arguments ] ")"
               | "none" ;
```

Note

Agent fields include `model` (required), `system` (required), and optional fields such as `tools`, `temperature`, `max_tokens`, `max_steps`, and `memory`. The `model` field references a provider by name.

15.3.9 Workflow Declarations

Workflows are composable functions with optional trigger decorators. A workflow without a decorator can only be called as a sub-workflow.

```
workflow_decl = { decorator } "workflow" identifier
                "(" [ params ] ")" [ "->" type ] block ;
```

15.3.10 Types

```
type           = simple_type
               | array_type
               | map_type
               | tuple_type
               | option_type
```

```
                | fn_type
                | generic_type
                | stream_type
                | pointer_type
                | union_type
                | channel_type ;

simple_type    = "int" | "i8" | "i16" | "i32" | "i64"
                | "u8" | "u16" | "u32" | "u64"
                | "float" | "f32" | "f64"
                | "bool" | "string"
                | identifier ;

array_type     = "[" "]" type ;

map_type       = "[" type ":" type "]" ;

tuple_type     = "(" type { "," type } ")" ;

option_type    = type "?" ;

fn_type        = "fn" "(" [ type_list ] ")" [ "->" type ] ;

type_list      = type { "," type } ;

generic_type   = identifier "<" type_list ">" ;

stream_type    = "stream" "<" type ">" ;

pointer_type   = "*" type ;

union_type     = type { "|" type } ;

channel_type   = "chan" "<" type ">" ;
```

Note

The `stream<T>` type represents a lazy, asynchronous, single-use sequence of values. It is distinct from `chan<T>`, which is a bidirectional communication channel between concurrent tasks. Streams are typically produced by agent `.stream()` calls and consumed with `for` loops.

15.3.11 Statements

```
statement      = var_decl
                | assignment
                | expr_stmt
                | if_stmt
                | for_stmt
                | while_stmt
                | match_stmt
                | return_stmt
                | break_stmt
                | continue_stmt
                | defer_stmt
```

```
              | spawn_stmt
              | step_stmt
              | send_stmt
              | select_stmt
              | block ;

var_decl      = ident_list [ ":" type ] "=" expr_list ;

ident_list    = identifier { "," identifier } ;

expr_list     = expression { "," expression } ;

assignment    = lvalue_list assign_op expr_list ;

lvalue_list   = lvalue { "," lvalue } ;

lvalue        = identifier
              | lvalue "." identifier
              | lvalue "[" expression "]" ;

expr_stmt     = expression ;

if_stmt       = "if" [ simple_stmt ";" ] expression block
                [ "else" ( if_stmt | block ) ] ;

simple_stmt   = var_decl | assignment | expr_stmt ;

for_stmt      = "for" [ identifier "," ] identifier "in"
   expression block ;

while_stmt    = "while" expression block ;

match_stmt    = "match" expression "{" { match_arm } "}" ;

match_arm     = pattern [ "if" expression ] "=>" ( expression |
   block ) ;

return_stmt   = "return" [ expr_list ] ;

break_stmt    = "break" [ identifier ] ;

continue_stmt = "continue" [ identifier ] ;

defer_stmt    = "defer" statement ;

spawn_stmt    = "spawn" ( block | fn_call ) ;

step_stmt     = "step" string_lit block ;

send_stmt     = expression "<-" expression ;

select_stmt   = "select" "{" { select_case } [ default_case ]
   "}" ;

select_case   = pattern "=" "<-" expression "=>" ( expression |
   block )
              | expression "<-" expression "=>" ( expression |
                 block ) ;
```

```

default_case   = "default" "=>" ( expression | block ) ;

block          = "{" { statement } "}" ;

fn_call        = identifier "(" [ arguments ] ")"
               | identifier "." identifier "(" [ arguments ] ")"
                 ;
```

Note

Inside a workflow, a **spawn** block runs all its statements concurrently and returns a list of results when all complete. This is equivalent to `Promise.all()` in JavaScript. Outside a workflow, **spawn** launches a concurrent goroutine-style task. A **step** block is a named region with automatic telemetry; it is only valid inside **workflow** bodies. Variables assigned inside a step are visible in subsequent steps and the enclosing workflow scope.

15.3.12 Expressions

```
expression     = assign_expr ;

assign_expr    = or_expr [ ("=" | "+=" | "-=" | "*=" | "/="
               | "&=" | "|=" | "^=" | "<<=" | ">>=" | ">>>=")
                 assign_expr ] ;

or_expr        = and_expr { ("or" | "||") and_expr } ;

and_expr       = equality { ("and" | "&&") equality } ;

equality       = comparison { ("==" | "!=") comparison } ;

comparison     = pipe { ("<" | ">" | "<=" | ">=") pipe } ;

pipe           = range { "|>" range } ;

range          = bitwise_or [ (".." | "..=") bitwise_or ] ;

bitwise_or     = bitwise_xor { "|" bitwise_xor } ;

bitwise_xor    = bitwise_and { "^" bitwise_and } ;

bitwise_and    = shift { "&" shift } ;

shift          = additive { ("<<" | ">>") additive } ;

additive       = multiplicative { ("+" | "-") multiplicative } ;

multiplicative = unary { ("*" | "/" | "%") unary } ;

unary          = ("!" | "-" | "~" | "not") unary
               | postfix ;

postfix        = primary { postfix_op } ;

```

```
postfix_op     = call
               | index
               | field
               | optional_chain ;

call           = "(" [ arguments ] ")" ;

arguments      = argument { "," argument } ;

argument       = [ identifier ":" ] expression ;

index          = "[" expression "]" ;

field          = "." identifier ;

optional_chain = "?" "." identifier ;

primary        = identifier
               | literal
               | "(" expression ")"
               | array_lit
               | map_lit
               | struct_lit
               | closure
               | if_expr
               | match_expr
               | receive_expr
               | block ;

literal        = integer_lit | float_lit | string_lit
               | bool_lit | nil_lit ;

array_lit      = "[" [ expr_list ] "]" ;

map_lit        = "[" ":" "]"
               | "[" map_entry { "," map_entry } [ "," ] "]" ;

map_entry      = expression ":" expression ;

struct_lit     = identifier "{" [ field_inits ] "}" ;

field_inits    = field_init { "," field_init } [ "," ] ;

field_init     = identifier [ ":" expression ] ;

closure        = "fn" "(" [ closure_params ] ")" [ "->" type ]
                 ( block | "{" expression "}" ) ;

closure_params = closure_param { "," closure_param } ;

closure_param  = identifier [ ":" type ] ;

if_expr        = "if" expression block "else" ( if_expr | block
   ) ;

match_expr     = "match" expression "{" { match_arm } "}" ;

receive_expr   = "<-" expression ;
```

15.3.13 Patterns

```
pattern        = literal_pat
               | ident_pat
               | wildcard_pat
               | tuple_pat
               | struct_pat
               | enum_pat
               | array_pat
               | range_pat
               | or_pat ;

literal_pat    = integer_lit | float_lit | string_lit | bool_lit
    ;

ident_pat      = identifier ;

wildcard_pat   = "_" ;

tuple_pat      = "(" pattern { "," pattern } ")" ;

struct_pat     = identifier "{" [ field_pats ] "}" ;

field_pats     = field_pat { "," field_pat } ;

field_pat      = identifier [ ":" pattern ] ;

enum_pat       = identifier "." identifier [ "(" [ pattern_list
    ] ")" ] ;

pattern_list   = pattern { "," pattern } ;

array_pat      = "[" [ pattern_list ] [ "," "..." [ identifier ]
    ] "]" ;

range_pat      = literal ".." [ "=" ] literal ;

or_pat         = pattern { "|" pattern } ;
```

15.4 Grammar Summary

15.4.1 Entry Points

- `program` — Complete source file
- `expression` — Single expression (REPL)
- `statement` — Single statement
- `type` — Type annotation

15.4.2 Precedence (High to Low)

1. Postfix: `() [] . ?.`
2. Unary: `! - ~ not`

3. Multiplicative: `* / %`
4. Additive: `+ -`
5. Shift: `« » »>`
6. Bitwise AND: `&`
7. Bitwise XOR: `^`
8. Bitwise OR: `|`
9. Range: `.. ..=`
10. Pipe: `|>`
11. Comparison: `< > <= >=`
12. Equality: `== !=`
13. Logical AND: `and &&`
14. Logical OR: `or ||`
15. Assignment: `= += -= *= /= &= |= ^= «= »= »>=`

15.4.3 Associativity

- Left-associative: All binary operators except assignment
- Right-associative: Assignment operators, unary operators
- Non-associative: Comparison operators, range operators

Chapter 16

Compiler Architecture

This chapter describes the architecture and implementation of the Haira compiler.

16.1 Overview

The Haira compiler is written in Go and transforms `.haira` source files into native binaries by generating Go code and invoking the Go toolchain. Agentic constructs — **`provider`**, **`tool`**, **`agent`**, and **`workflow`** with `@` decorator triggers — are first-class elements of the compilation pipeline.

The compiler pipeline:

```
Source (.haira)
       |
       v
   +---------+
   | Lexer   |  --> Tokens
   +---------+
       |
       v
   +---------+
   | Parser |  --> AST
   +---------+
       |
       v
   +-----------+
   | Resolver |  --> Merged AST (multi-file imports resolved)
   +-----------+
       |
       v
   +-----------+
   | Checker   |  --> Type info + diagnostics
   +-----------+
       |
       v
   +-----------+
   | Codegen   |  --> Go source code
   +-----------+
       |
       v
   +-----------+
   | go build |  --> Native binary
   +-----------+
```

The key design decision: HAIRA compiles *to Go*, not to machine code directly. This means every HAIRA program benefits from Go's battle-tested runtime (goroutines, garbage collector, HTTP stack, TLS) without reimplementing any of it. The generated Go code is readable and can be inspected via `haira emit`.

Note

There is no "AI Phase" in the pipeline. All LLM interaction happens at *runtime* through agent method calls. The compiler validates, type-checks, and generates schemas for agentic constructs, then emits runtime library calls.

16.2 Lexer

The lexer is a hand-written scanner in `compiler/internal/lexer/` that converts UTF-8 source text into a stream of tokens.

16.2.1 Token Types

Tokens are defined in `compiler/internal/token/`:

```
// Literals
Int, Float, String, InterpolatedString, TripleQuoteString

// Keywords (agentic)
Provider, Tool, Agent, Workflow

// Keywords (core)
Fn, Struct, Enum, Type, Import, Export, From, Pub,
If, Else, For, While, Match, Return, Break, Continue,
True, False, None, Some, And, Or, Not, In,
Spawn, Select, Try, Catch, Defer, Errdefer,
Step, Test, Assert, Let, Const

// Keywords (reserved for future)
Trait, Impl, Async

// Operators
Plus, Minus, Star, Slash, Percent,
Eq, EqEq, Ne, Lt, Gt, Le, Ge,
PlusEq, MinusEq, StarEq, SlashEq, PercentEq,
Pipe, PipeArrow, Amp, AmpEq, Caret, CaretEq, Tilde,
Shl, ShlEq, Shr, ShrEq, PipeEq,
Arrow, FatArrow, DotDot, DotDotEq, Dot, Question, At, Ellipsis

// Delimiters
LParen, RParen, LBracket, RBracket, LBrace, RBrace, Comma, Colon
```

16.2.2 Lexer Features

- **Numeric literals** — Decimal, hex (`0xFF`), binary (`0b1010`), octal (`0o77`), with underscores (`1_000_000`)
- **String interpolation** — `${expr}` syntax with nested brace tracking
- **Triple-quoted strings** — `"""..."""` with automatic leading-whitespace dedenting

- **Nested block comments** — `/* ... /* nested */ ... */`
- **Escape sequences** —
  `n`,
  `t`,
  `r`,

  `,`
  `",`
  `{,`
  `}`

16.3 Parser

The parser in `compiler/internal/parser/` uses recursive descent for statements and declarations, combined with Pratt (precedence-climbing) parsing for expressions.

16.3.1 AST Nodes

The AST is defined in `compiler/internal/ast/` using Go interfaces as sum types. All nodes carry a `Span` for error reporting.

Top-level declarations:

- `TypeDef` — struct with typed fields and optional defaults
- `EnumDef` — enum with variants (optionally carrying fields)
- `TypeAlias` — `type Name = Type`
- `FunctionDef` — `fn name(params) -> Type { body }`
- `MethodDef` — `Type.method(params) -> Type { body }` with implicit `self`
- `ImportDecl` — four forms: basic, aliased, selective, glob
- `ExportDecl` — `export { Name1, Name2 }`
- `ProviderDecl` — `provider name { key: value }`
- `ToolDecl` — tool with params, return type, triple-quoted description, body
- `AgentDecl` — `agent name { key: value }`
- `WorkflowDecl` — workflow with optional trigger decorator, params, lifecycle hooks, body
- `TestDecl` — `test "name" { body }`

Expressions (27 types): Literals, identifiers, binary/unary ops, calls, method calls, field access, indexing, pipe, lambda, match, if-expressions, lists, maps, struct instances, ranges, error propagation (`?`), `orelse`, `some`/`none`, spawn, select, async blocks.

Statements (14 types): Assignment, let/const, if/else, for-in, while, return, try/catch, defer/errdefer, match, break/continue, step, assert, expression statements.

Patterns (6 types): Wildcard (`_`), literal, identifier, constructor, or-pattern (`A | B`), range (`1..5`).

Level	Operators
1	`orelse`
2	`|>` (pipe)
3	`or` (logical)
4	`and` (logical)
5	`|` (bitwise OR)
6	`^` (bitwise XOR)
7	`&` (bitwise AND)
8	`==`, `!=`
9	`<`, `>`, `<=`, `>=`, `..`, `..=`
10	`«`, `»`
11	`+`, `-`
12	`*`, `/`, `%`
13	`-`, `not`, `~` (unary)
14	`()`, `[]`, `.`, `?` (postfix)

Table 16.1: Operator precedence (lowest to highest)

16.3.2 Operator Precedence

The Pratt parser implements 14 precedence levels (lowest to highest):

16.4 Resolver

The resolver in `compiler/internal/resolver/` handles multi-file module resolution:

1. Reads the main source file and extracts import declarations.
2. Resolves each import: standard library (built-in), project-local (`path/file.haira` or `path/mod.haira`), or external.
3. Recursively resolves transitive imports.
4. Detects circular dependencies (compile-time error).
5. Filters exports: only `pub` items and agentic declarations are visible to importers.
6. Produces a `Program` with a merged item list in dependency order.

```
// Resolution order for merged items:
// 1. Transitive module items (deepest dependencies first)
// 2. Direct import items (in source order)
// 3. Main file items
```

16.5 Checker

The type checker in `compiler/internal/checker/` performs semantic analysis in two passes:

Pass 1 (Registration): Walk all top-level items and register:

- Struct types with field types

- Enum types with variants
- Function signatures
- Provider, tool, agent, workflow declarations
- Global variables (inferred from initializers)

Pass 2 (Validation): Check function/method bodies:

- Type inference for expressions (literals, binary ops, calls, field access, indexing)
- Variable definition tracking (warn on undefined variables)
- Const immutability enforcement
- Agent field validation (model references a provider, tools exist, known fields)
- Enum exhaustiveness warnings in match expressions
- Return type checking
- Method `self` binding

16.5.1 Type System

```
Primitive types: int, float, string, bool, any, void, error
Compound types: [T] (list), {K: V} (map)
Named types:    struct, enum
Function types: (T, U) -> V
```

Note

Generic types are parse-accepted but erased to `any` in v1. Full generics are planned for a future release.

16.5.2 Agentic Validation

The checker validates agentic declarations:

- **Providers:** Warn on unknown fields. Valid fields: `model`, `api_key`, `backend`, `host`, `temperature`, `max_tokens`, `transport`, `command`, `args`, `url`.
- **Agents:** Require `model` field referencing a declared provider. Validate tool references. Check handoff targets exist. Warn on unknown fields.
- **Tools:** Enforce mandatory triple-quoted description (parser-level). Register parameters for schema generation.
- **Workflows:** Validate decorator triggers. Type-check body.

16.6 Code Generation

The code generator in `compiler/internal/codegen/` transforms the typed AST into Go source code. This is the core of Haira's compilation strategy: generate idiomatic Go that calls into the Haira runtime library.

16.6.1 Emission Strategy

Code is emitted in 8 ordered passes to satisfy Go's declaration requirements:

1. **Providers** — `var Claude = &haira.Provider{...}`
2. **Tools** — Tool functions + registration
3. **Agents** — Agent variables + initialization functions
4. **Workflows** — HTTP handler functions + SSE streaming handlers
5. **Top-level variables** — Global `var` declarations
6. **Types and enums** — Struct definitions, enum iota constants
7. **Methods** — Receiver methods (dot-attach syntax)
8. **Functions** — User functions (non-main)
9. **Main function** — Entry point with agent initialization in topological order

16.6.2 Agent Initialization Order

Agents with handoff targets must be initialized after their targets. The codegen performs a topological sort on the handoff graph to determine initialization order:

```
// Haira source:
agent Triage { handoffs: [Billing, Support] }
agent Billing { provider: Claude }
agent Support { provider: Claude }

// Generated init order:
// 1. Billing (no dependencies)
// 2. Support (no dependencies)
// 3. Triage  (depends on Billing, Support)
```

16.6.3 Workflow Code Generation

Workflows with triggers generate HTTP handlers. Streaming workflows generate both a `StreamHandler` (SSE) and a `Handler` (JSON fallback):

```
// @webhook("/summarize")
// workflow Summarize(req: SumReq) -> Summary { ... }

// Generates:
// func SummarizeHandler(w, r)       -- JSON response
// func SummarizeStreamHandler(w, r) -- SSE streaming
// Route registration in main()
```

16.6.4 Tool Schema Generation

Each tool declaration generates a JSON schema from its parameter types:

Haira Type	JSON Schema Type
`int`	`"integer"`
`float`	`"number"`
`bool`	`"boolean"`
`string`	`"string"`
`[T]`	`"array"` with `items`
`struct`	`"object"` with `properties`

Table 16.2: Type-to-JSON-schema mapping

16.6.5 Import Detection

The codegen scans the AST to determine which Go imports are needed:

- `fmt` — string interpolation, print calls
- `encoding/json` — JSON marshal/unmarshal calls
- `sync` — spawn blocks (WaitGroup)
- `haira` — runtime library (agents, providers, tools, workflows, HTTP)

16.6.6 Go Compilation

After generating Go source code, the codegen:

1. Creates a temporary Go project directory with `go.mod`
2. Extracts the embedded runtime bundle (`bundle.tar.gz`) containing the haira Go package
3. Writes the generated `main.go`
4. Runs `go build` to produce a native binary
5. Moves the binary to the output path and cleans up

16.7 Runtime Library

The Haira runtime is a Go package (`package haira`) bundled into the compiler binary. It provides the execution environment for all agentic constructs.

16.7.1 Runtime Structure

The runtime is split into two layers:

- **Primitive** (`primitive/haira/`) — Core runtime: agent execution, provider management, tool registry, workflow orchestration, HTTP server, memory, I/O, strings, arrays, maps, math, JSON, regex, file system, time, environment variables.
- **Stdlib** (`stdlib/haira/`) — External integrations: PostgreSQL, SQLite, vector databases, Excel, Slack, GitHub, GitLab.

Both layers use `package haira` and are merged at bundle time into a single Go package, embedded in the compiler binary as `bundle.tar.gz`.

16.7.2 Provider Management

The runtime manages LLM provider connections:

- API key resolution from environment variables at startup
- Model routing based on provider configuration
- HTTP connection pooling to provider APIs
- Support for Anthropic, OpenAI, Ollama, and MCP (Model Context Protocol) transports

16.7.3 Agent Execution Loop

1. Assemble message payload (system prompt, conversation history, user message).
2. Send request to the LLM provider.
3. If response contains tool calls, execute each tool and collect results.
4. Send tool results back to the LLM.
5. Repeat steps 2–4 until final response or `max_steps` reached.
6. For `.run()` calls, parse JSON response into the annotated output type.
7. For `.stream()` calls, yield tokens incrementally via SSE.

16.7.4 Workflow Orchestration

- **HTTP server** — Built on Go's `net/http`. Handles `@webhook` (POST/GET/PUT/DELETE) and `@websocket` triggers.
- **Cron scheduler** — Executes `@cron`-triggered workflows on 5-field UNIX cron schedules.
- **Event bus** — In-process event dispatch for `@event`-triggered workflows via `haira.emit()`.
- **Step telemetry** — Named `step` blocks emit timing and status events for observability.
- **Run tracking** — Each workflow execution gets a unique run ID with step-level SSE streaming.

16.7.5 Session Management

- Each session maintains an isolated conversation history.
- `conversation(max_turns: N)` memory trims to the last N message pairs.
- Sessions are stored in-memory by default, with optional persistence via the Store interface.

16.8 Project Structure

The compiler is organized as a Go module:

```
compiler/
  main.go                          # CLI entry point
  internal/
    token/                         # Token type definitions
    lexer/                         # Hand-written scanner
    ast/                           # AST node definitions
    parser/                        # Recursive descent + Pratt
       parser
    resolver/                      # Multi-file import
       resolution
    checker/                       # Type checking + semantic
       analysis
    codegen/                       # Go code generation
      codegen.go                   #   8-pass emission
         orchestrator
      emitter.go                   #   Output buffer +
         indentation
      expressions.go               #   Expression-to-Go
         conversion
      statements.go                #   Statement emission
      agentic.go                   #
         Agent/tool/provider/workflow emission
      stdlib.go                    #   Standard library
         function mapping
      types.go                     #   Haira-type-to-Go-type
         conversion
      project.go                   #   Temp project setup + go
         build
      test_codegen.go              #   Test harness generation
    driver/                        # Pipeline orchestration
    errors/                        # Diagnostic system
       (pretty-printing)
    formatter/                     # Source code formatter
    lsp/                           # Language server protocol
    manifest/                      # package.haira handling
    runtime/                       # Embedded runtime bundle
       (bundle.tar.gz)

primitive/haira/                   # Core runtime (Go)
stdlib/haira/                      # External integrations (Go)
ui/sdk/                            # Web component SDK
   (TypeScript)
ui/application/                    # Default HTML templates
```

16.9 CLI Commands

```
haira build [file] [-o output]   # Compile to native binary
haira run [file]                  # Compile and execute
haira check [file]                # Type-check without codegen
haira emit [file]                 # Show generated Go code
haira test [file]                 # Run test blocks
```

Package	Responsibility
`token`	Token kind constants and Token struct
`lexer`	Tokenizes source text
`ast`	AST node types (Go interfaces as sum types)
`parser`	Builds AST from tokens
`resolver`	Resolves imports and produces merged program
`checker`	Type checking, inference, and semantic validation
`codegen`	Generates Go source code from typed AST
`driver`	Orchestrates the full compilation pipeline
`errors`	Diagnostic formatting with source context
`formatter`	Haira source code formatting
`lsp`	Language server (hover, completion, go-to-definition)
`manifest`	Package manifest (package.haira) support

Table 16.3: Compiler package responsibilities

```
haira fmt [file]                  # Format source in-place
haira parse [file]                # Show the AST
haira lex [file]                  # Show tokens
haira init                        # Create package.haira
   manifest
haira lsp                         # Start language server
   (stdio)
haira version                     # Show version
```

If no file is specified, `haira` looks for `package.haira` in the current directory and uses its entry point.

16.10 Error Messages

The compiler produces contextual error messages with source location, underlines, and help text.

16.10.1 Missing Tool Description

```
error: tool is missing a description
  --> tools.haira:5:1
    |
5   | tool search(query: string) -> [Result] {
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    |
    = help: add a triple-quoted description: """Search the web"""
    = note: tool descriptions are sent to the LLM and cannot be
        omitted
```

16.10.2 Agent Referencing Undefined Provider

```
error: undefined provider 'openai'
  --> agents.haira:8:12
    |
8  |     provider: openai
    |               ^^^^^^ not found in this scope
    |
    = help: declare a provider with: provider openai { model:
       "gpt-4o" ... }
```

16.10.3 Undefined Variable

```
warning: undefined variable 'foo'
  --> main.haira:10:5
   |
10 |     io.println(foo)
   |                ^^^
   |
   = hint: did you mean to declare it?
```

16.10.4 Workflow with Invalid Trigger

```
error: unknown trigger '@daily'
  --> workflows.haira:12:1
   |
12 | @daily
   | ^^^^^^ not a recognized trigger
   |
   = help: valid triggers: @webhook, @websocket, @cron, @event,
      @manual
   = help: for daily execution, use: @cron("0 0 * * *")
```

16.11 Runtime Bundling

The runtime is bundled into the compiler binary at build time:

1. `make bundle-runtime` copies `primitive/haira/*.go` and `stdlib/haira/*.go` into a staging directory.
2. The UI SDK is built (`bun build`) and its output included.
3. Everything is archived as `bundle.tar.gz`.
4. The archive is embedded in the compiler binary via Go's `go:embed` directive.
5. At compile time, the codegen extracts this bundle into a temporary Go project, writes the generated `main.go`, and runs `go build`.

This approach produces fully standalone binaries: the compiled HAIRA program contains everything it needs, including the full runtime library. No separate runtime installation is required.

Appendix A

Chatbot Patterns

One of the most common use cases for HAIRA is building chatbot backends. This chapter shows patterns from simple to advanced.

A.1 Simple REST Chatbot

A chatbot accessible via HTTP POST:

```
import "io"
import "http"

provider anthropic {
    api_key: env("ANTHROPIC_API_KEY")
    model: "claude-sonnet-4-20250514"
}

agent Assistant {
    provider: anthropic
    system: "You are a helpful assistant."
    memory: conversation(max_turns: 50)
    temperature: 0.7
}

@webhook("/chat")
workflow Chat(message: string, session_id: string) -> { reply:
    string } {
    reply, err = Assistant.ask(message, session: session_id)
    if err != nil {
        return { reply: "Sorry, something went wrong." }
    }
    return { reply: reply }
}

fn main() {
    server = http.Server([Chat])
    io.println("Chat server on :8080")
    server.listen(8080)
}
```

Usage:

```
$ curl -X POST http://localhost:8080/chat \
    -d '{"message": "Hello!", "session_id": "user-123"}'
```

```
{"reply": "Hi there! How can I help you today?"}

$ curl -X POST http://localhost:8080/chat \
    -d '{"message": "What did I just say?", "session_id":
       "user-123"}'

{"reply": "You said \"Hello!\""}
```

The agent remembers the conversation because both calls share the same `session_id`.

A.2 Streaming WebSocket Chatbot

For real-time token-by-token responses over WebSocket:

```
@websocket("/chat")
workflow StreamingChat(message: string, session_id: string) ->
   stream<string> {
    return Assistant.stream(message, session: session_id)
}

fn main() {
    server = http.Server([StreamingChat])
    server.listen(8080)
}
```

The `stream<string>` return type tells Haira to stream each token over the WebSocket as it arrives. No buffering — the client sees tokens in real time.

A.3 Dual-Mode Chatbot

Expose the same agent over both REST (for simple integrations) and WebSocket (for real-time UIs):

```
agent Assistant {
    provider: anthropic
    system: "You are a helpful assistant."
    memory: conversation(max_turns: 50)
}

@webhook("/chat")
workflow ChatREST(message: string, session_id: string) -> {
   reply: string } {
    reply, err = Assistant.ask(message, session: session_id)
    return { reply: reply or "Error occurred." }
}

@websocket("/chat/stream")
workflow ChatStream(message: string, session_id: string) ->
   stream<string> {
    return Assistant.stream(message, session: session_id)
}

fn main() {
    server = http.Server([ChatREST, ChatStream])
    server.listen(8080)
}
```

Both endpoints share the same agent and session memory.

A.4 RAG Chatbot

A chatbot that searches a knowledge base before answering:

```
import "tools/postgres"
import "tools/http"

tool search_kb(query: string) -> [KBArticle] {
    """Search the knowledge base for relevant articles"""

    results, err = pg.query(
        "SELECT * FROM articles WHERE content @@ to_tsquery($1)
           LIMIT 5",
        [query]
    )
    if err != nil { return [], err }
    return results
}

tool lookup_order(order_id: string) -> Order {
    """Look up a customer order by ID"""

    order, err = pg.query_one(
        "SELECT * FROM orders WHERE id = $1",
        [order_id]
    )
    if err != nil { return Order{}, err }
    return order
}

tool create_ticket(subject: string, body: string, priority:
    string = "normal") -> Ticket {
    """Create a support ticket"""

    resp, err = http.post("https://helpdesk.api/tickets", {
        subject: subject, body: body, priority: priority
    })
    if err != nil { return Ticket{}, err }
    return json.decode(resp.body, Ticket)
}

agent SupportBot {
    provider: anthropic
    system: """
        You are a customer support agent for Acme Corp.
        Use the knowledge base to answer questions.
        Look up orders when customers ask about purchases.
        Create tickets for issues you cannot resolve directly.
        Always be polite and helpful.
    """
    tools: [search_kb, lookup_order, create_ticket]
    memory: conversation(max_turns: 100)
    temperature: 0.3
}

@websocket("/support")
workflow CustomerSupport(message: string, session_id: string)
   -> stream<string> {
    return SupportBot.stream(message, session: session_id)
}
```

The agent automatically decides when to search the knowledge base, look up an order, or create a ticket based on the user's message.

A.5 Multi-Agent Routing

Route user messages to specialized agents based on intent:

```
agent Router {
    provider: anthropic
    system: """
        Classify the user message into one category: billing,
           technical, general.
        Reply with just the category name, nothing else.
    """
    temperature: 0.0
}

agent BillingAgent {
    provider: anthropic
    system: "You handle billing questions. You can look up
       invoices and process refunds."
    tools: [lookup_invoice, process_refund]
    memory: conversation(max_turns: 30)
}

agent TechAgent {
    provider: anthropic
    system: "You handle technical support. You can search docs
       and check system status."
    tools: [search_docs, check_status]
    memory: conversation(max_turns: 30)
}

agent GeneralAgent {
    provider: anthropic
    system: "You handle general inquiries."
    memory: conversation(max_turns: 30)
}

@websocket("/chat")
workflow SmartChat(message: string, session_id: string) ->
   stream<string> {
    category, _ = Router.ask(message)

    match category {
        "billing"   => return BillingAgent.stream(message,
           session: session_id)
        "technical" => return TechAgent.stream(message,
                session: session_id)
        _           => return GeneralAgent.stream(message, session: session_id)
    }
}
```

A.6 Agent Handoffs

For complex support bots, use handoffs instead of explicit routing. The front desk agent decides when to delegate, and the runtime handles the transfer automatically:

```
agent FrontDesk {
    provider: anthropic
    system: """
        You are a front desk agent. Greet users and help with general questions.
        If they have a billing question, hand off to BillingAgent.
        If they have a technical issue, hand off to TechAgent.
    """
    handoffs: [BillingAgent, TechAgent]
    memory: conversation(max_turns: 10)
}

agent BillingAgent {
    provider: anthropic
    system: "You handle billing questions. You can look up invoices and process refunds."
    tools: [lookup_invoice, process_refund]
    memory: conversation(max_turns: 30)
}

agent TechAgent {
    provider: anthropic
    system: "You handle technical support. You can search docs and check system status."
    tools: [search_docs, check_status]
    memory: conversation(max_turns: 30)
}

@webhook("/chat")
workflow SmartSupport(message: string, session_id: string) -> { reply: string } {
    // FrontDesk automatically hands off when appropriate
    reply, err = FrontDesk.ask(message, session: session_id)
    if err != nil { return { reply: "Sorry, something went wrong." } }
    return { reply: reply }
}

fn main() {
    server = http.Server([SmartSupport])
    io.println("Support server on :8080")
    server.listen(8080)
}
```

Compare this to the explicit routing in the previous section. With handoffs, the workflow code stays simple — the agent decides when to delegate, not the workflow. This is ideal for conversational bots where the routing logic is nuanced and best handled by the LLM itself.

A.7 Chatbot with Structured Actions

Sometimes you want the chatbot to return structured data alongside text:

```
struct ChatResponse {
    reply: string
    actions: [Action]
}

struct Action {
    type: string        // "navigate", "show_modal", "refresh"
    target: string
}

agent UIAssistant {
    provider: anthropic
    system: """
        You are a UI assistant. When answering, also suggest UI
            actions.
        For example, if the user asks about their order,
            suggest navigating
        to the orders page.
    """
    tools: [search_kb, lookup_order]
    memory: conversation(max_turns: 50)
}

@webhook("/assistant")
workflow UIChat(message: string, session_id: string) ->
    ChatResponse {
    response: ChatResponse, err = UIAssistant.run(
        { message: message },
        session: session_id
    )
    if err != nil {
        return ChatResponse{ reply: "Sorry, an error
            occurred.", actions: [] }
    }
    return response
}
```

A.8 Best Practices

1. **Always use sessions for conversation** — without a session, memory agents start fresh every call
2. **Set appropriate max_turns** — too high wastes tokens, too low loses context
3. **Use streaming for interactive UIs** — REST is fine for backend-to-backend, but users expect real-time responses

4. **Give tools clear descriptions** — the agent decides when to use tools based on descriptions
5. **Use a Router agent for complex bots** — instead of one agent doing everything, route to specialists
6. **Handle errors gracefully** — always provide a fallback response when agents fail

Appendix B

Generative UI

Generative UI allows agents to render rich, interactive components inline in a chat conversation instead of returning only text. When an agent calls a tool, the tool's result can be rendered as a structured UI component—a table, a card, a diff view, a form—rather than a plain text bubble.

HAIRA approaches generative UI at the language level. Tools declare what component renders their output. The runtime ships a built-in component catalog. No frontend framework, bundler, or client-side JavaScript is required.

B.1 Motivation

Consider a migration tool that returns a dry-run result:

```
tool dry_run(file: string) -> string {
    """Run SQL against the database in dry-run mode"""
    // ... returns "DRY-RUN FAILED on seeds.\n\nROOT CAUSE: ..."
}
```

The agent receives a string, streams it as markdown, and the user sees a wall of text. The information is there, but the presentation is poor—error details, affected tables, and fix SQL are all flattened into one block.

With generative UI, the same tool returns structured data and declares a renderer:

```
@render("status-card")
tool dry_run(file: string) -> DryRunResult {
    """Run SQL against the database in dry-run mode"""
    // ... returns structured result
}
```

The chat UI renders a status card with colored indicators, collapsible sections, and copy-paste SQL blocks—all from the same tool declaration, with zero frontend code.

B.2 The `@render` Decorator

Tools opt into generative UI with the `@render` decorator. It takes a component name from the built-in catalog:

```
@render("component-name")
tool tool_name(params) -> ReturnType {
    """Description"""
    // body
}
```

The decorator does three things:

1. Tells the compiler to include component metadata in the tool definition
2. Tells the runtime to emit a `tool_render` SSE event with the structured result
3. Tells the chat UI to render the named component instead of showing raw text

Tools without `@render` behave as before: the agent receives the result, interprets it, and streams a text response. Tools with `@render` still return their result to the agent for reasoning, but the UI additionally renders the component inline.

Note

The `@render` decorator is optional. Existing tools continue to work unchanged. Generative UI is additive.

B.3 Built-in Component Catalog

HAIRA ships a set of built-in components that cover common patterns. Each component expects a specific data shape and renders accordingly.

B.3.1 table — Tabular Data

Renders a table with headers and rows. Supports optional column alignment and row highlighting.

```
struct TableData {
    title: string
    headers: [string]
    rows: [[string]]
    highlight: [int]          // row indices to highlight
        (optional)
}

@render("table")
tool list_tables(file: string) -> TableData {
    """List all tables found in the Excel config file"""

    // ... extract tables ...
    return TableData{
        title: "Tables in " + filename,
        headers: ["Table", "Rows", "Type"],
        rows: [
            ["parameters", "42", "configuration"],
            ["batches", "8", "run"],
        ],
        highlight: [],
    }
}
```

B.3.2 status-card — Result Summary

Renders a card with a status indicator (success, error, warning), title, and detail sections.

```
struct StatusCard {
    status: string          // "success", "error", "warning",
        "info"
    title: string
    message: string
    sections: [CardSection]
}

struct CardSection {
    label: string
    content: string         // markdown supported
    style: string           // "default", "code", "error",
        "success"
}

@render("status-card")
tool validate(file: string) -> StatusCard {
    """Validate Excel data against the database schema"""

    // ... run validation ...
    return StatusCard{
        status: "error",
        title: "Validation Failed",
        message: "3 errors, 1 warning",
        sections: [
            CardSection{
                label: "Missing Columns",
                content: "`id_plant` in table `parameters`",
                style: "error",
            },
        ],
    }
}
```

B.3.3 code-block — Source Code or SQL

Renders a syntax-highlighted code block with a copy button and optional title.

```
struct CodeBlock {
    title: string           // optional header
    language: string        // "sql", "json", "haira", etc.
    code: string
}

@render("code-block")
tool generate_sql(file: string) -> CodeBlock {
    """Generate SQL INSERT statements from the config file"""

    // ... generate SQL ...
    return CodeBlock{
        title: "Generated Seeds SQL",
        language: "sql",
        code: sql_output,
    }
}
```

B.3.4 diff — Before/After Comparison

Renders a side-by-side or unified diff view.

```
struct DiffView {
    title: string
    before_label: string
    after_label: string
    before: string
    after: string
    language: string        // syntax highlighting language
}

@render("diff")
tool compare_schemas(table: string) -> DiffView {
    """Compare Excel schema with database schema for a table"""

    return DiffView{
        title: "Schema Diff: " + table,
        before_label: "Database",
        after_label: "Excel",
        before: db_schema,
        after: excel_schema,
        language: "sql",
    }
}
```

B.3.5 key-value — Property List

Renders a compact list of labeled values. Useful for metadata, summaries, and configuration display.

```
struct KeyValueList {
    title: string
    items: [KVItem]
}

struct KVItem {
    key: string
    value: string
    style: string           // "default", "success", "error",
        "muted"
}

@render("key-value")
tool file_info(file: string) -> KeyValueList {
    """Show metadata about the uploaded config file"""

    return KeyValueList{
        title: "File Info",
        items: [
            KVItem{ key: "Filename", value: filename, style: "default" },
            KVItem{ key: "Tables", value: "12", style: "default" },
            KVItem{ key: "Rows", value: "347", style: "default" },
            KVItem{ key: "Status", value: "Valid", style: "success" },
        ],
    }
}
```

B.3.6 progress — Multi-Step Progress

Renders a vertical progress tracker showing completed, active, and pending steps. Useful for tools that report pipeline state.

```
struct ProgressView {
    title: string
    steps: [ProgressStep]
}

struct ProgressStep {
    name: string
    status: string          // "done", "active", "pending", "failed"
    detail: string          // optional detail text
}

@render("progress")
tool pipeline_status(mr_id: string) -> ProgressView {
    """Check the deployment pipeline status"""

    return ProgressView{
        title: "Deployment Pipeline",
        steps: [
            ProgressStep{ name: "Branch Created", status: "done", detail: "" },
            ProgressStep{ name: "CI/CD Running", status: "active", detail: "2m 15s" },
            ProgressStep{ name: "Merge", status: "pending", detail: "" },
            ProgressStep{ name: "Deploy to QUA", status: "pending", detail: "" },
        ],
    }
}
```

B.3.7 form — Interactive Input

Renders a form that the user can fill and submit. The submission triggers a follow-up tool call or message. This is the foundation for human-in-the-loop approval flows.

```
struct FormView {
    title: string
    fields: [FormField]
    submit_label: string
    submit_action: string   // tool name or message prefix
}

struct FormField {
    name: string
    label: string
    field_type: string      // "text", "select", "checkbox",
        "textarea"
    value: string           // default value
    options: [string]       // for "select" type
    required: bool
}

@render("form")
tool request_approval(branch: string, mr_url: string) ->
    FormView {
    """Request user approval before pushing to GitLab"""

    return FormView{
        title: "Confirm Push to GitLab",
        fields: [
            FormField{
                name: "confirm",
                label: "Push branch '" + branch + "' and create
                    MR?",
                field_type: "select",
                value: "",
                options: ["Yes, push it", "No, cancel"],
                required: true,
            },
            FormField{
                name: "comment",
                label: "MR comment (optional)",
                field_type: "textarea",
                value: "",
                options: [],
                required: false,
            },
        ],
        submit_label: "Submit",
        submit_action: "handle_approval",
    }
}
```

When the user submits the form, the chat sends a message in the format: `"[Form: handle_approval] confirm=Yes, push it; comment=Looks good"`, which the agent can interpret and act upon.

B.3.8 Component Summary

B.4 Streaming Protocol

Generative UI extends the existing SSE protocol with a new event type: `tool_render`.

Component	Use Case	Data Shape
`table`	Tabular results, lists	Headers + rows
`status-card`	Success/error summaries	Status + sections
`code-block`	SQL, JSON, code output	Language + code string
`diff`	Before/after comparisons	Two text blocks
`key-value`	Metadata, config display	Key-value pairs
`progress`	Pipeline/step tracking	Steps with status
`form`	Approvals, user input	Fields + submit action

Table B.1: Built-in generative UI components

B.4.1 Current Protocol

The existing protocol emits three event types:

```
event: tool_start
data: {"tool": "validate", "args": "{\"file\":
   \"/tmp/config.xlsx\"}"}

event: tool_end
data: {"tool": "validate", "ok": true}

data: {"delta": "The validation passed with..."}
```

B.4.2 Extended Protocol

When a tool has an `@render` decorator, the runtime emits `tool_render` between `tool_start` and `tool_end`:

```
event: tool_start
data: {"tool": "validate", "args": "{\"file\":
   \"/tmp/config.xlsx\"}"}

event: tool_render
data: {
    "tool": "validate",
    "component": "status-card",
    "props": {
        "status": "error",
        "title": "Validation Failed",
        "message": "3 errors, 1 warning",
        "sections": [...]
    }
}

event: tool_end
data: {"tool": "validate", "ok": false}
```

The `tool_render` event carries:

- `tool`: The tool name (for matching with the active tool card)
- `component`: The component name from the catalog
- `props`: The tool's return value, JSON-serialized, used as component props

The chat UI receives `tool_render`, looks up the component in its registry, and renders it inline—replacing or augmenting the tool card.

B.4.3 Dual Consumption

The tool result is consumed twice:

1. **By the UI**: The `tool_render` event renders the component for the user
2. **By the agent**: The result is still appended to the message history, so the agent can reason about it and decide the next step

This means the agent sees the structured data (to make decisions) and the user sees the rich component (for readability). Neither path is sacrificed.

B.5 Agent Behavior with Generative UI

Agents do not need any changes to use generative UI. The `@render` decorator is on the tool, not the agent. An agent with a mix of rendered and non-rendered tools works naturally:

```
agent Maltimize {
    provider: azure_openai
    system: "You help deploy Excel configs to PostgreSQL."
    tools: [
        read_excel,            // no @render -- text result
        validate,              // @render("status-card")
        generate_sql,          // @render("code-block")
        dry_run,               // @render("status-card")
        push_to_gitlab,        // no @render -- text result
    ]
    memory: conversation(max_turns: 20)
    temperature: 0.3
}
```

When the agent calls `validate`, the user sees a status card. When it calls `push_to_gitlab`, the user sees the agent's text interpretation. Both are valid—generative UI is opt-in per tool.

B.6 Fallback Behavior

Generative UI degrades gracefully in two scenarios:

B.6.1 Non-Chat Contexts

When a tool with `@render` is called outside a chat workflow (e.g., from a form workflow or a plain function), the decorator is ignored. The tool returns its structured data normally. The `@render` decorator only takes effect when the tool is executed through an agent's streaming loop.

B.6.2 API Consumers

When the SSE stream is consumed by a programmatic client (not the built-in chat UI), the `tool_render` event provides the component name and props as JSON. The client can:

- Render its own implementation of the component

- Ignore `tool_render` events and use only `tool_end` for status
- Use the `props` data directly for custom processing

B.7 Complete Example

A full migration assistant with generative UI:

```
import "io"
import "http"
import "excel"
import "postgres"

provider azure_openai {
    type: "azure-openai"
    model: env("AZURE_OPENAI_DEPLOYMENT_NAME")
}

// --- Rendered tools ---

@render("table")
tool read_config(file_path: string) -> TableData {
    """Read an Excel config file and list the tables found"""

    wb = excel.open(file_path)?
    index = wb.read_sheet("index")?
    rows = []
    for entry in index {
        table = "" + entry["Table"]
        sheet = wb.read_sheet(get_sheet(table)) orelse []
        rows = rows + [[table, conv.int_to_string(len(sheet)),
            get_type(table)]]
    }
    wb.close()
    return TableData{
        title: "Config Tables",
        headers: ["Table", "Rows", "Type"],
        rows: rows,
        highlight: [],
    }
}

@render("status-card")
tool dry_run(file_path: string) -> StatusCard {
    """Dry-run the generated SQL against the local database"""

    // ... generate and test SQL ...
    if result["ok"] == true {
        return StatusCard{
            status: "success",
            title: "Dry Run Passed",
            message: "All SQL is valid",
            sections: [],
        }
    }
    return StatusCard{
        status: "error",
        title: "Dry Run Failed",
        message: result["error"],
        sections: [
            CardSection{
                label: "Root Cause",
                content: analysis["root_cause"],
                style: "error",
            },
            CardSection{
                label: "Fix SQL",
                content: "```sql\n" + analysis["fix"] + "\n```",
                style: "code",
            },
        ],
    }
}

// --- Agent ---

agent ConfigAssistant {
    provider: azure_openai
    system: "You deploy Excel configs to PostgreSQL via GitLab."
    tools: [read_config, dry_run, push_to_gitlab]
    memory: conversation(max_turns: 20)
    temperature: 0.3
}

// --- Workflow ---

@webui(title: "Config Assistant", description: "Chat-based config migration")
@webhook("/api/chat")
workflow Chat(message: string, session_id: string, file_path: file) -> stream {
    msg = message
    if file_path != "" { msg = "${message}\n\n[Attached file: ${file_path}]" }
    return ConfigAssistant.stream(msg, session: session_id)
}

fn main() {
    server = http.Server([Chat])
    server.listen(9001)
}
```

The user uploads an Excel file and says “deploy this”. The agent calls `read_config`—the chat renders a table showing all tables with row counts. Then calls `dry_run`—the chat renders a status card showing success or failure with detailed sections. If it passes, the agent asks for confirmation before pushing.

All of this happens in a standard HAIRA program. No React, no Next.js, no bundler, no client-side framework.

B.8 Custom Components

The built-in catalog covers common patterns, but domain-specific use cases need custom components. HAIRA supports two levels of customization: composite components built from built-in primitives, and fully custom components written as TypeScript Web Components.

B.8.1 Composite Components

The simplest way to extend the UI is to compose built-in components. A composite component is a Haira struct whose fields map to multiple built-in components, rendered together as a group:

```
@render("composite")
tool migration_report(file: string) -> MigrationReport {
    """Generate a full migration report with tables,
       validation, and SQL"""

    return MigrationReport{
        components: [
            Component{
                type: "key-value",
                props: KeyValueList{
                    title: "File Summary",
                    items: [
                        KVItem{ key: "File", value: filename,
                           style: "default" },
                        KVItem{ key: "Tables", value: "12",
                           style: "default" },
                    ],
                },
            },
            Component{
                type: "table",
                props: TableData{
                    title: "Tables",
                    headers: ["Name", "Rows", "Type"],
                    rows: table_rows,
                    highlight: [],
                },
            },
            Component{
                type: "status-card",
                props: StatusCard{
                    status: "success",
                    title: "Validation Passed",
                    message: "All checks passed",
                    sections: [],
                },
            },
        ],
    }
}
```

The `composite` renderer lays out multiple built-in components vertically in a single tool render. This requires no custom JavaScript—only data.

B.8.2 Custom Web Components

For fully custom rendering, users write a TypeScript Web Component in a `components/` directory next to their Haira source:

```
project/
  main.haira
  components/
    gantt-chart.ts
    schema-diagram.ts
```

A custom component file follows a simple contract:

```
// components/gantt-chart.ts

interface GanttProps {
    title: string;
    tasks: Array<{
        name: string;
        start: string;
        end: string;
        progress: number;
    }>;
}

export class HairaGanttChart extends HTMLElement {
    connectedCallback() {
        this.attachShadow({ mode: "open" });
    }

    // Called by the runtime with the tool's return value
    setProps(props: GanttProps) {
        const shadow = this.shadowRoot!;
        shadow.innerHTML = `
            <style>
                :host { display: block; }
                .chart {
                    padding: 1rem;
                    background: var(--haira-bg-card);
                    border: 1px solid var(--haira-border);
                    border-radius: var(--haira-radius);
                }
                .title {
                    font-weight: 600;
                    font-size: 0.85rem;
                    color: var(--haira-text);
                    margin-bottom: 0.75rem;
                }
                .bar-row {
                    display: flex;
                    align-items: center;
                    gap: 0.5rem;
                    margin: 0.35rem 0;
                }
                .bar-label {
                    font-size: 0.78rem;
                    color: var(--haira-text-dim);
                    min-width: 120px;
                }
                .bar-track {
                    flex: 1;
                    height: 20px;
                    background: var(--haira-bg-elevated);
                    border-radius: var(--haira-radius-sm);
                    overflow: hidden;
                }
                .bar-fill {
                    height: 100%;
                    background: var(--haira-gold);
                    border-radius: var(--haira-radius-sm);
                    transition: width 0.3s;
                }
                .bar-pct {
                    font-size: 0.7rem;
                    color: var(--haira-muted);
                    font-family: var(--haira-mono);
                    min-width: 35px;
                }
            </style>
            <div class="chart">
                <div
                   class="title">${this.esc(props.title)}</div>
                ${props.tasks.map(t => `
                    <div class="bar-row">
                        <span
                           class="bar-label">${this.esc(t.name)}</span>
                        <div class="bar-track">
                            <div class="bar-fill"
                                 style="width:
                                    ${t.progress}%"></div>
                        </div>
                        <span
                           class="bar-pct">${t.progress}%</span>
                    </div>
                `).join("")}
            </div>
        `;
    }

    private esc(s: string): string {
        return s.replace(/&/g, "&amp;")
                .replace(/</g, "&lt;")
                .replace(/>/g, "&gt;");
    }
}

// Export the component class and its custom element tag name
export default {
    tag: "haira-gantt-chart",
    component: HairaGanttChart,
};
```

The component must:

1. Extend `HTMLElement`
2. Use Shadow DOM for style isolation
3. Implement a `setProps(props)` method that receives the tool's return value as a typed object

4. Export a default object with `tag` (the custom element name) and `component` (the class)

The component is referenced in `@render` by its file name (without extension):

```
struct GanttData {
    title: string
    tasks: [Task]
}

struct Task {
    name: string
    start: string
    end: string
    progress: int
}

@render("gantt-chart")
tool project_timeline(project_id: string) -> GanttData {
    """Show the project timeline as a Gantt chart"""

    return GanttData{
        title: "Q1 Roadmap",
        tasks: [
            Task{ name: "Backend API", start: "2025-01-01",
                  end: "2025-02-15", progress: 80 },
            Task{ name: "Frontend", start: "2025-02-01",
                  end: "2025-03-15", progress: 40 },
        ],
    }
}
```

B.8.3 Component Resolution

When the compiler encounters `@render("name")`, it resolves the component in this order:

1. **Built-in catalog**: Check if `name` matches a built-in component (`table`, `status-card`, `composite`, etc.)
2. **Local components**: Check if `components/name.ts` exists relative to the `.haira` source file
3. **Compile error**: If neither matches, emit a diagnostic with the list of available component names

Example diagnostic:

```
error: unknown render component "gantt-chart"
  --> main.haira:42:9
   |
42 | @render("gantt-chart")
   |         ^^^^^^^^^^^^^
   |
   = available: table, status-card, code-block, diff,
                key-value, progress, form, composite
   = hint: create components/gantt-chart.ts to register
           a custom component
```

B.8.4 Build Process

The HAIRA compiler handles custom components as part of its normal build pipeline. The existing pipeline is:

```
.haira source --> Lexer --> Parser --> Checker --> Codegen -->
   main.go
                                                                |
go-runtime/haira/ (linked via go.mod replace) ---+
   |
                                                  |

                                                     v
                                          go build (produces
                                             binary)
```

The runtime's built-in UI is a pre-compiled JavaScript bundle (`ui/dist/haira-ui.js`) embedded in the Go binary via `//go:embed`. HTML templates contain a `<script>` tag that loads this bundle at `/_ui/assets/haira-ui.js`.

Custom components extend this pipeline with one additional step:

```
haira build main.haira

1. Parse, check, codegen as normal
2. Scan components/ directory for .ts files
3. Generate components/_register.ts (entry point):
      import c0 from "./gantt-chart.ts";
      import c1 from "./schema-diagram.ts";
      customElements.define(c0.tag, c0.component);
      customElements.define(c1.tag, c1.component);
4. Bundle components/_register.ts --> components.js
   (using bun build --minify --target browser)
5. Embed components.js as a Go string constant in
   the generated main.go:
      const customComponentsJS = `...minified JS...`
6. Register a second asset route:
      GET /_ui/assets/components.js
7. HTML templates include a second <script> tag
   after the built-in UI bundle
```

The generated `main.go` includes the custom bundle:

```
// Generated by haira build
package main

import "haira-go-runtime/haira"

// Custom components bundle (empty if no components/ dir)
const customComponentsJS = `(function(){class HairaGanttChart
extends HTMLElement{...}customElements.define("haira-gantt-
chart",HairaGanttChart);...})()`

func main() {
    server := haira.NewServer(workflows)
    server.SetCustomComponentsJS(customComponentsJS)
    server.Listen(9001)
}
```

The server serves the custom bundle at `/_ui/assets/components.js` and the HTML template includes both scripts:

```
<script src="/_ui/assets/haira-ui.js"></script>
<script src="/_ui/assets/components.js"></script>
```

If no `components/` directory exists, the custom bundle is empty and the second `<script>` tag is omitted. Existing projects are unaffected.

Note

The compiler requires `bun` to be installed for TypeScript compilation of custom components. If `bun` is not found and custom components exist, the compiler emits a clear error with installation instructions. Projects without custom components do not require `bun`.

B.8.5 Referencing Components from Tools

The connection between a tool and its component is the `@render` decorator. The decorator name must match either a built-in component or the filename (without `.ts`) of a file in `components/`:

```
// Built-in: name matches catalog entry
@render("table")
tool list_items(...) -> TableData { ... }

// Custom: name matches components/gantt-chart.ts
@render("gantt-chart")
tool project_timeline(...) -> GanttData { ... }

// Custom: name matches components/schema-diagram.ts
@render("schema-diagram")
tool show_schema(...) -> SchemaView { ... }
```

The compiler validates this mapping at compile time. The runtime connects them at render time: when a `tool_render` SSE event arrives with `"component": "gantt-chart"`, the chat UI looks up the custom element registered under that name, creates an instance, and calls `setProps()` with the tool's result data.

The custom element tag name is defined by the component's `export default { tag: "..." }` — the convention is `haira-{name}` (e.g., `haira-gantt-chart` for `gantt-chart.ts`), but the developer controls it. The runtime matches by the `@render` name, not the tag name.

Note

Custom components run in the browser's Shadow DOM sandbox. They cannot access the parent page's DOM or interfere with other components. The HAIRA runtime provides theme CSS custom properties (`-haira-gold`, `-haira-bg`, etc.) that custom components can use for consistent styling.

B.8.6 Theme Integration

Custom components inherit HAIRA's theme through CSS custom properties. The runtime injects these variables into the document root, making them available inside Shadow DOM:

```css
/* Inside a custom component's Shadow DOM styles: */
:host {
    display: block;
    font-family: var(--haira-font);
    color: var(--haira-text);
}
.chart-bar {
    background: var(--haira-gold);
    border-radius: var(--haira-radius-sm);
}
.error { color: var(--haira-error); }
.success { color: var(--haira-success); }
```

Available CSS custom properties:

Variable	**Purpose**
`-haira-bg`	Page background
`-haira-bg-card`	Card/panel background
`-haira-bg-elevated`	Elevated surface background
`-haira-border`	Border color
`-haira-gold`	Brand accent (gold)
`-haira-gold-light`	Lighter gold variant
`-haira-text`	Primary text color
`-haira-text-dim`	Secondary text color
`-haira-muted`	Muted/disabled text
`-haira-success`	Success state (green)
`-haira-error`	Error state (red)
`-haira-warn`	Warning state (yellow)
`-haira-info`	Info state (blue)
`-haira-font`	System font stack
`-haira-mono`	Monospace font stack
`-haira-radius`	Standard border radius
`-haira-radius-sm`	Small border radius

Table B.2: CSS custom properties available to custom components

B.8.7 Interactive Components

Custom components can be interactive. When a user interacts with a component (clicks a button, selects a row, submits a form), the component dispatches a custom DOM event that the chat UI captures and sends as a follow-up message to the agent:

```javascript
// Inside a custom component's setProps():
const btn = shadow.getElementById("approve-btn")!;
btn.addEventListener("click", () => {
    this.dispatchEvent(new CustomEvent("haira-action", {
        bubbles: true,
        composed: true,    // crosses Shadow DOM boundary
        detail: {
            action: "approve",
            data: { branch: "feature/config-123" },
        },
    }));
});
```

The chat UI listens for `haira-action` events and sends the action to the agent as a message: `"[Action: approve] {"branch": "feature/config-123"}"`. The agent can then interpret this and call the appropriate tool.

This pattern enables:

- Clickable table rows that trigger drill-down queries
- Approval/reject buttons on deployment cards
- Filter controls that re-run a tool with different parameters
- Form submissions within custom components

B.8.8 Example: Custom Schema Diagram

A complete custom component for visualizing database table relationships:

```
// components/schema-diagram.ts

interface SchemaProps {
    title: string;
    tables: Array<{
        name: string;
        columns: Array<{
            name: string;
            type: string;
            pk: boolean;
            fk: boolean;
        }>;
    }>;
}

export class HairaSchemaDiagram extends HTMLElement {
    connectedCallback() {
        this.attachShadow({ mode: "open" });
    }

    setProps(props: SchemaProps) {
        const shadow = this.shadowRoot!;
        shadow.innerHTML = ‘
            <style>
                :host { display: block; }
                .diagram {
                    padding: 1rem;
                    background: var(--haira-bg-card);
                    border: 1px solid var(--haira-border);
                    border-radius: var(--haira-radius);
                    display: flex;
                    flex-wrap: wrap;
                    gap: 0.75rem;
                }
                .table-box {
                    border: 1px solid var(--haira-border);
                    border-radius: var(--haira-radius-sm);
                    min-width: 150px;
                }
                .table-name {
                    padding: 0.4rem 0.6rem;
                    background: var(--haira-gold);
                    color: #1a0e04;
                    font-weight: 700;
                    font-size: 0.8rem;
                    border-radius:
                        var(--haira-radius-sm)
                        var(--haira-radius-sm) 0 0;
                }
                .col {
                    padding: 0.25rem 0.6rem;
                    font-size: 0.75rem;
                    color: var(--haira-text-dim);
                    font-family: var(--haira-mono);
                    border-top: 1px solid var(--haira-border);
                }
                .col.pk { color: var(--haira-gold); }
                .col.fk { color: var(--haira-info); }
                .title {
                    font-weight: 600;
                    font-size: 0.85rem;
                    color: var(--haira-text);
                    width: 100%;
                    margin-bottom: 0.25rem;
                }
            </style>
            <div class="diagram">
                <div
                   class="title">${this.esc(props.title)}</div>
                ${props.tables.map(t => `
                    <div class="table-box">
                        <div
                           class="table-name">${this.esc(t.name)}</div>
                        ${t.columns.map(c => `
                            <div class="col ${c.pk ? "pk" :
                               c.fk ? "fk" : ""}">
                                ${this.esc(c.name)}:
                                   ${this.esc(c.type)}
                            </div>
                        `).join("")}
                    </div>
                `).join("")}
            </div>
        `;
    }

    private esc(s: string): string {
        return s.replace(/&/g, "&amp;")
                .replace(/</g, "&lt;")
                .replace(/>/g, "&gt;");
    }
}

export default {
    tag: "haira-schema-diagram",
    component: HairaSchemaDiagram,
};
```

Used in HAIRA:

```
struct SchemaView {
    title: string
    tables: [TableView]
}

struct TableView {
    name: string
    columns: [ColumnView]
}

struct ColumnView {
    name: string
    type: string
    pk: bool
    fk: bool
}

@render("schema-diagram")
tool show_schema(table_names: string) -> SchemaView {
    """Visualize the database schema for the given tables"""

    // ... query schema, build view ...
    return SchemaView{
        title: "Database Schema",
        tables: tables,
    }
}
```

The result: the agent calls `show_schema`, and the chat renders an interactive diagram with color-coded primary and foreign keys—no React, no build system, just a `.ts` file next to the `.haira` source.

B.9 Design Principles

1. **Tools own their rendering.** The `@render` decorator lives on the tool, not the agent or workflow. This keeps rendering co-located with the data source.

2. **Structured data, not markup.** Tools return typed structs, not HTML or JSX. The runtime maps data to components. This keeps tools testable and platform-independent.

3. **Catalog first, custom when needed.** The built-in catalog covers common patterns with zero configuration. Custom components extend the catalog with a single `.ts` file—no framework, no bundler, no build pipeline.

4. **Additive, not breaking.** Every tool works without `@render`. Adding it improves the UI without changing the tool's logic or the agent's behavior.

5. **Dual consumption.** The UI and the agent both receive the tool result. The UI renders a component; the agent reasons about the data. Neither path is secondary.

6. **Progressive complexity.** Simple needs use built-in components (data only). Moderate needs use composites (multiple built-ins). Advanced needs use custom TypeScript Web Components (one `.ts` file). The learning curve matches the complexity of the use case.

7. **Interactivity through events.** Custom components communicate back to the agent through a single event contract (`haira-action`). No callback registration, no bidirectional binding—just events that become messages.

B.10 External Frontend API

The built-in chat UI is a convenience, not a requirement. HAIRA exposes a complete HTTP API that gives external frontends—React, Vue, Svelte, mobile apps, or any HTTP client—the same capabilities as the built-in UI. The built-in UI is itself just a consumer of this API.

B.10.1 Discovery: `GET /_api/`

Returns metadata about all registered workflows. This is the same data the built-in UI uses to render the index page and route to form/chat views.

```
GET /_api/

{
    "workflows": [
        {
            "name": "UploadConfig",
            "path": "/api/upload-config",
            "method": "POST",
            "ui_type": "form",
            "title": "Maltimize Config",
            "description": "Excel-to-PostgreSQL configuration
               migration",
            "params": [
                { "name": "file_path", "type": "file",
                   "required": true },
                { "name": "debug", "type": "string",
                   "required": false }
            ],
            "steps": [
                "Open Excel", "Extract table data", "Query
                   schema",
                "Validate", "Generate SQL", "Dry-run SQL",
                "Push to GitLab", "Wait for CI/CD",
                "Wait for merge", "Verify deployment"
            ]
        },
        {
            "name": "Chat",
            "path": "/api/chat",
            "method": "POST",
            "ui_type": "chat",
            "title": "Maltimize Chat",
            "description": "Chat-based config migration
               assistant",
            "params": [
                { "name": "message", "type": "string",
                   "required": true },
                { "name": "session_id", "type": "string",
                   "required": true },
                { "name": "file_path", "type": "file",
                    "required": false }
            ],
            "steps": [],
            "tools": [
                {
                    "name": "read_config_excel",
                    "description": "Read an Excel configuration
                        file...",
                    "render": null
                },
                {
                    "name": "validate_config",
                    "description": "Validate Excel config
                        data...",
                    "render": "status-card"
                },
                {
                    "name": "dry_run_migration",
                    "description": "Run the generated SQL...",
                    "render": "status-card"
                }
            ],
            "components": [
                {
                    "name": "status-card",
                    "type": "builtin"
                },
                {
                    "name": "gantt-chart",
                    "type": "custom",
                    "tag": "haira-gantt-chart"
                }
            ]
        }
    ]
}
```

The `tools` array lists each tool available to the agent, including its `render` component (or `null` if the tool has no UI). The `components` array lists all generative UI components used by the workflow, distinguishing built-in from custom.

B.10.2 Workflow Metadata: `GET /_api/{path}`

Returns detailed metadata for a single workflow:

```
GET /_api/api/chat

{
    "name": "Chat",
    "path": "/api/chat",
    "method": "POST",
    "ui_type": "chat",
    "title": "Maltimize Chat",
    "description": "Chat-based config migration assistant",
    "params": [...],
    "tools": [...],
    "components": [...]
}
```

B.10.3 Streaming: `POST /{path}` with SSE

The same endpoint used by the built-in UI. An external client sends a request with `Accept: text/event-stream` to receive SSE events:

```
$ curl -N -H "Accept: text/event-stream" \
    -d '{"message":"deploy this","session_id":"abc123"}' \
    http://localhost:9001/api/chat
```

The SSE stream emits these event types:

Event	When	Data
`tool_start`	Tool begins	`{"tool": "name", "args": "{...}"}`
`tool_render`	Tool has UI result	`{"tool": "name", "component": "status-card", "props": {...}}`
`tool_end`	Tool finishes	`{"tool": "name", "ok": true}`
(default)	Text delta	`{"delta": "text chunk"}`
`step`	Pipeline step event	`{"name": "...", "status": "start|end|failed"}`
`[DONE]`	Stream complete	—

Table B.3: SSE event types

An external frontend handles these events the same way the built-in UI does:

1. On `tool_start`: show a loading indicator for the tool
2. On `tool_render`: render the component using the `component` name and `props` data. For built-in components, the frontend implements its own version (e.g., a React `<StatusCard>`). For custom components, it uses the `props` JSON directly.
3. On `tool_end`: mark the tool as complete (success or failure)
4. On text deltas: append to the assistant message
5. On `[DONE]`: finalize the response

B.10.4 Form Submission: `POST /{path}` with JSON

For non-streaming workflows (form UI), the same endpoint accepts JSON and returns JSON:

```
$ curl -X POST http://localhost:9001/api/upload-config \
    -F "file_path=@config.xlsx" \
    -F "debug=true"

# Response with step-by-step SSE:
event: step
data: {"name": "Open Excel", "status": "start"}

event: step
```

```
data: {"name": "Open Excel", "status": "end", "duration_ms":
   120}

# ... more steps ...

event: result
data: {"status": "success", "message": "Configuration deployed"}
```

Without `Accept: text/event-stream`, the response is plain JSON:

```
$ curl -X POST http://localhost:9001/api/upload-config \
    -F "file_path=@config.xlsx"

{"status": "success", "message": "Configuration deployed to
   QUA"}
```

B.10.5 CORS Support

When an external frontend runs on a different origin (e.g., `localhost:3000` for a React dev server), the HAIRA server enables CORS automatically when the `HAIRA_CORS_ORIGIN` environment variable is set:

```
$ HAIRA_CORS_ORIGIN="http://localhost:3000" ./maltimize
```

The wildcard `*` is also supported for development:

```
$ HAIRA_CORS_ORIGIN="*" ./maltimize
```

B.10.6 External Frontend Example

A React frontend consuming the HAIRA API:

```
// React component consuming Haira's SSE API

function Chat() {
    const [messages, setMessages] = useState([]);
    const [tools, setTools] = useState(new Map());

    async function send(text: string, file?: File) {
        // Build request
        const body = new FormData();
        body.append("message", text);
        body.append("session_id", sessionId);
        if (file) body.append("file_path", file);

        // Open SSE connection
        const response = await fetch("/api/chat", {
            method: "POST",
            body,
            headers: { Accept: "text/event-stream" },
        });

        const reader = response.body.getReader();
        const decoder = new TextDecoder();

        while (true) {
            const { done, value } = await reader.read();
            if (done) break;

            const lines = decoder.decode(value).split("\n");
            for (const line of lines) {
                if (line.startsWith("event: tool_start")) {
                    // Show loading state for the tool
                }
                if (line.startsWith("event: tool_render")) {
                    const data = JSON.parse(/* next data line
                       */);
                    // data.component = "status-card"
                    // data.props = { status: "error", ... }
                    // Render <StatusCard {...data.props} />
                }
                if (line.startsWith("data: ")) {
                    const data = JSON.parse(line.slice(6));
                    if (data.delta) {
                        // Append text to assistant message
                    }
                }
            }
        }
    }

    return (
        <div>
            {messages.map(msg =>
                msg.component
                    ? <ComponentRenderer
                          name={msg.component}
                          props={msg.props} />
                    : <MessageBubble text={msg.text} />
            )}
        </div>
    );
}

// Map Haira component names to React components
function ComponentRenderer({ name, props }) {
    const components = {
        "status-card": StatusCard,
        "table": DataTable,
        "code-block": CodeBlock,
        "gantt-chart": GanttChart,  // custom component
    };
    const Component = components[name];
    return Component ? <Component {...props} /> : null;
}
```

The external frontend renders its own components for each `tool_render` event. The `component` name and `props` JSON provide all the information needed—the frontend maps component names to its own implementations (React, Vue, Svelte, or native mobile components).

B.10.7 API Parity Guarantee

The built-in UI has no privileged access. Every feature available in the built-in chat and form UIs is available through the HTTP API:

Feature	Built-in UI	External API
Workflow discovery	Index page	`GET /_api/`
Workflow metadata	`<script id="haira-meta">`	`GET /_api/{path}`
Form submission	`<haira-form>` submit	`POST /{path}` (JSON)
Chat streaming	`<haira-chat>` SSE	`POST /{path}` (SSE)
Tool execution status	`<haira-tool-card>`	`tool_start`/`tool_end` events
Generative UI render	`<haira-*>` components	`tool_render` event + props
File upload	Drag & drop / attach	`multipart/form-data`
Step pipeline	`<haira-pipeline>`	`step` events

Table B.4: API parity between built-in UI and external frontends

Note

The built-in UI can be disabled entirely with the `HAIRA_DISABLE_UI` environment variable. The API endpoints remain active. This is useful when deploying Haira as a pure backend behind a separate frontend application.

B.11 Grammar Extension

The `@render` decorator uses the existing decorator syntax. No new grammar is required:

```
tool_decl      = { decorator } "tool" identifier "(" [ params ]
   ")" [ "->" type ]
                 "{" triple_string { statement } "}" ;

decorator      = "@" identifier [ "(" decorator_args ")" ] ;
```

The compiler validates that the component name in `@render("name")` matches a known component from the catalog. Unknown component names produce a compile-time error.

Appendix C

Reserved Keywords

The following identifiers are reserved as keywords and cannot be used as identifiers:

```
agent      and        as         assert     break
catch      chan       const      continue   defer
else       enum       errdefer   eval       export
false      fn         for        from       if
impl       import     in         let        match
nil        not        or         provider   pub
return     select     some       spawn      step
struct     test       tool       trait      true
try        type       unsafe     verify     where
while      workflow
```

Appendix D

Operator Precedence

Operators are listed from highest to lowest precedence:

Precedence	**Operators**	**Associativity**
1 (highest)	`() [] .`	Left
2	`! - ~` (unary) `not`	Right
3	`* / %`	Left
4	`+ -`	Left
5	`« »`	Left
6	`&` (bitwise AND)	Left
7	`^` (bitwise XOR)	Left
8	`|` (bitwise OR)	Left
9	`.. ..=`	None
10	`|>` (pipe)	Left
11	`< > <= >=`	Left
12	`== !=`	Left
13	`and &&`	Left
14	`or ||`	Left
15 (lowest)	`= += -= *= /= &= |= ^= «= »=`	Right

Appendix E

Standard Library Reference

Quick reference for standard library modules. See Chapter 10 for full documentation.

Module	**Description**
Core Modules (built into the runtime)	
`io`	Input/output operations
`string/strings`	String manipulation
`math`	Mathematical functions
`array`	Array operations
`map/maps`	Map/dictionary operations
`json`	JSON encoding/decoding
`http`	HTTP client/server
`fs`	File system operations
`os`	Operating system interface
`time`	Time and duration
`regex`	Regular expressions
`conv`	Type conversions
`env`	Environment variable helpers
`upload`	File upload handling
`observe`	Observability and cost monitoring
`mcp`	MCP server/client
Stdlib Packages (tree-shaken, included only when imported)	
`postgres`	PostgreSQL client
`sqlite`	SQLite (auto-included for workflows)
`excel`	Excel file processing
`vector`	Vector embeddings and similarity search
`slack`	Slack integration
`github`	GitHub API
`gitlab`	GitLab API
`langfuse`	Langfuse observability exporter
`agents`	Pre-built agent templates
`auth`	Authentication and API key resolution
`websearch`	Web search integration
`healthcheck`	Health check endpoints