अभा Programming Language - Official Guide (v0.1)

"Abha" (**अभा**) means *radiance*. Abha is a small, expressive, compiled language for **deterministic systems**, **probabilistic programs**, and **quantum workflows** with a clean surface syntax and a simple, auditable runtime.

Table of Contents

1. Introduction

Abha is designed for clarity and power:

- **Single syntax** for deterministic, probabilistic, and quantum tasks.
- **Total functions by default** with explicit `Result` for errorful paths.
- **Derivable protocols** (traits) for common behaviors (e.g. `Stringify`).
- **Minimal runtime** with explicit intrinsics.

A "hello world" with composite features:

```
@derive("ToString")
thing Pair:
  a: Int
  b: Int

task safe_div(a:Int, b:Int) -> Result[Int,String]:
  if b == 0:
    return err("div by zero")
  return ok(a / b)

task div_then_add(a:Int, b:Int, c:Int) -> Result[Int,String]:
  match safe_div(a,b):
    ok(v):  return ok(v + c)
    err(e): return err(e)

on start:
  print(div_then_add(10, 2, 3))  // ok(8)
  let t = Pair(3,4)
  print(t.to_string())           // Pair(a: 3, b: 4)
```

2. Quick Start

Requirements: Abha toolchain (CLI + runtime)

Run a program:

```
abha run sample.abha
```

Emit IR (for debugging):

```
abha run sample.abha --emit-ir
```

Seed probabilistic runtime:

```
abha run sample.abha --seed 123
```

3. Toolchain & CLI

- abha - command-line front end
    - `run <file>`: parse → typecheck → lower → execute on the VM
    - `--emit-ir`: print lowered IR for inspection
    - `--seed <int>`: set RNG seed for reproducibility
- Runtime backends
    - **Interpreter** (default)
    - (Optional) **native**/JIT paths for hot loops (implementation-defined)

4. Language Tour

4.1 Lexical Elements & Comments

- **Identifiers**: `snake_case` for functions/tasks; `UpperCamel` for `thing` types.
- **Literals**: `Int`, `Float`, `String` (double quotes), `Bool` (`true`/`false`).
- **Comments**: `//` to end of line.

```
// This is a comment
let x = 42
let s = "hello"
```

4.2 Types

Built-ins: `Int`, `Float`, `String`, `Bool`.

Generics (selected):

- `Result[T,E]`: success-or-error sum type.
- (Planned) `List[T]`, `Map[K,V]` may be provided via std.

4.3 Variables & Bindings

- `let` introduces an **immutable** binding; `var` introduces a **mutable** one.

```
let a = 10
var acc = 0
acc = acc + a
```

4.4 Expressions

Abha expressions are side-effect-aware. Expressions return values; `return` can appear in tail position (last expression) or explicitly.

```
task add(a:Int, b:Int) -> Int:
  a + b  // implicit return
```

4.5 Control Flow

if

```
task abs(x:Int) -> Int:
  if x < 0:
    return -x
  return x
```

match

match supports deconstructing Result and user-defined variants (when present).

```
task div_then_add(a:Int, b:Int, c:Int) -> Result[Int,String]:
  match safe_div(a,b):
    ok(v):  return ok(v + c)
    err(e): return err(e)
```

Loops

- for over ranges/iterables.
- Tail recursion via recur for self-tail-calls (lowered to loops by compiler).

```
// Tail recursion lowered to a loop

task fact_loop(i:Int, acc:Int) -> Int:
  if i <= 1:
    return acc
  recur(i-1, acc * i)

task fact(n:Int) -> Int:
  return fact_loop(n, 1)

// For-loop over a range

task sum_to(n:Int) -> Int:
  var acc = 0
  for i in range(n):
    acc = acc + i
  return acc
```

4.6 Functions (Tasks)

Abha uses the task keyword for functions. All parameters and returns are explicitly typed.

```
task greet(name:String) -> String:
  return "Hello, " + name
```

4.7 Data Types (Things)

`thing` declares a product type with named fields.

```
thing Point:
  x: Int
  y: Int

on start:
  let p = Point(1, 2)
  print(p.x + p.y) // 3
```

4.8 Protocols & Derives

Protocols (traits) define required methods; **derive** can autogenerate implementations.

```
protocol Stringify:
  task to_string(self: Self) -> String

@derive("ToString")
thing Pair:
  a: Int
  b: Int

on start:
  let t = Pair(3,4)
  print(t.to_string())   // Pair(a: 3, b: 4)
```

> The standard prelude provides `Stringify` impls for `Int`, `Float`, `Bool`, `String` so field `to_string()` calls always resolve.

4.9 Results & Errors

Abha uses `Result[T,E]` and helpers `ok`/`err`.

```
task safe_div(a:Int, b:Int) -> Result[Int,String]:
  if b == 0:
    return err("div by zero")
  return ok(a / b)

on start:
  print(safe_div(10, 2))  // ok(5)
  print(safe_div(3, 0))   // err("div by zero")
```

4.10 Iteration & Ranges

Use `range(n)` for `0..n-1`.

```
for i in range(5):
  print(i)
```

4.11 Streams & Handlers

on <event> declares a reactive handler. A standard event is stream_range(n).

```
on stream_range(3):
  print(it)   // prints 0, 1, 2
```

The special startup entry point is:

```
on start:
  // program entry
```

4.12 Probabilistic Primitives

Abha ships a small probabilistic core.

- prob_seed(Int) - set RNG seed.
- uniform(lo:Float, hi:Float) - distribution.
- bern(p:Float) - Bernoulli distribution.
- sample(dist) - draw a sample.
- trace(dist, n:Int) - draw n samples (or traces) for inspection.

```
on start:
  prob_seed(42)
  print(sample(uniform(0.0, 1.0)))
  print(trace(bern(0.5), 5))
```

4.13 Quantum Primitives

Abha includes a minimal, backend-agnostic quantum API (q.*).

- q.new(n:Int) - allocate n-qubit register
- q.h(reg, i) - Hadamard on qubit i
- q.cx(reg, c, t) - controlled-X from c to t
- measure(reg) - return a measurement distribution
- trace(dist, n) - sample strings from a distribution

```
on start:
  let qreg = q.new(2)
  q.h(qreg, 0)
  q.cx(qreg, 0, 1)
  let d = measure(qreg)
  print(d)            // distribution meta
  print("trace")
  print(trace(d, 8))  // sample bitstrings
```

4.14 Modules & Prelude

- The **prelude** is imported implicitly. It defines `print`, `range`, `ok`, `err`, `Stringify`, and primitive `to_string` implementations.
- Future versions may add `import`/`module` directives for multi-file projects.

5. Standard Library Overview

Core

- `print(x)` - print any value
- `stringify(x)` - convert to `String`
- `range(n:Int)` - iterator 0..n-1

Result

- `ok(value)`, `err(error)`

Probabilistic

- `prob_seed(seed:Int)`
- `uniform(lo:Float, hi:Float)`
- `bern(p:Float)`
- `sample(dist)`
- `trace(dist, n:Int)`

Quantum

- `q.new(n:Int)`, `q.h(reg,i)`, `q.cx(reg,c,t)`, `measure(reg)`

Protocols

- `protocol Stringify: task to_string(self:Self) -> String`

6. Build Targets & IR

The Abha compiler lowers surface syntax to a **linear, SSA-like IR**. Each task becomes a function with labeled blocks, explicit `mov`, `call`, and `rt_*` intrinsics. Debug with `--emit-ir`.

Why IR?

- **Deterministic lowering** for auditing and testing
- Enables **tail-call optimization** (`recur`) and loop transforms
- Clear interface to interpreter/JIT backends

7. Best Practices & Style Guide

- Prefer **pure tasks** returning `Result` on failures.
- Derive `ToString` for user types used in logs or prints.
- Seed RNG in `on start` for reproducible experiments.
- Keep quantum code **backend-agnostic**; defer to `q.*` bridge.
- Use tail-recursive style with `recur` for large loops.

Naming

- `thing` types: `UpperCamel` (`Point`, `Pair`)
- tasks: `snake_case` (`safe_div`, `sum_to`)

8. Cookbook

8.1 Safe arithmetic

```
task safe_add(a:Int, b:Int) -> Result[Int,String]:
  let s = a + b
  // Add domain checks if needed
  ok(s)
```

8.2 Pretty-print nested things

```
@derive("ToString")
thing Address:
  city: String
  zip: Int

@derive("ToString")
thing Person:
  name: String
  addr: Address

on start:
  let p = Person("Sai", Address("Hyd", 500081))
  print(p.to_string())
```

8.3 Monte Carlo π

```
task monte_carlo_pi(n:Int) -> Float:
  var inside = 0
  for _ in range(n):
    let x = sample(uniform(0.0,1.0))
    let y = sample(uniform(0.0,1.0))
    if (x*x + y*y) <= 1.0:
      inside = inside + 1
  return 4.0 * (inside as Float) / (n as Float)
```

8.4 Bell Pair

```
on start:
  let r = q.new(2)
  q.h(r,0)
  q.cx(r,0,1)
  print(trace(measure(r), 8))
```

8.5 Stream processing

```
on stream_range(5):
  // `it` is the current stream item
  print(it)
```

9. FAQ

Q: Why ``?
To emphasize runtime-awareness (deterministic/probabilistic/quantum) while keeping the same ergonomics as functions.

Q: Is the prelude mandatory?
Yes; it provides protocol impls and core functions. You can disable it in advanced builds, but then must provide equivalents.

Q: Can I add my own protocols?
Yes, declare with `protocol` and implement with `impl` (or use `@derive` hooks where available).

Q: How stable is the IR?
The IR is versioned; surface syntax remains stable as the IR evolves.

Appendix A - Reference Syntax (Grammar Sketch)

```
program      := decl*

decl         := thing_decl | task_decl | protocol_decl | impl_decl |
handler_decl | derive_ann thing_decl

thing_decl   := "thing" IDENT ":" NL INDENT field+ DEDENT
field        := IDENT ":" type NL

task_decl    := "task" IDENT "(" params? ")" "->" type ":" NL INDENT stmt*
DEDENT
params       := param ("," param)*
param        := IDENT ":" type
```

```
protocol_decl:= "protocol" IDENT ":" NL INDENT method+ DEDENT
method       := "task" IDENT "(" "self:" "Self" ("," params)? ")" "->" type
NL

impl_decl    := "impl" IDENT "for" IDENT ":" NL INDENT method_impl+ DEDENT

handler_decl := "on" ("start" | IDENT "(" args? ")") ":" NL INDENT stmt*
DEDENT

stmt         := let | var | assign | if | match | for | return | expr_stmt
let          := "let" IDENT "=" expr NL
var          := "var" IDENT "=" expr NL
assign       := IDENT "=" expr NL
if           := "if" expr ":" NL INDENT stmt* DEDENT ("else:" NL INDENT stmt*
DEDENT)?
match        := "match" expr ":" NL INDENT arm+ DEDENT
arm          := IDENT "(" IDENT ")" ":" stmt
for          := "for" IDENT "in" expr ":" NL INDENT stmt* DEDENT
return       := "return" expr NL
expr_stmt    := expr NL

expr         := literals | call | member | binary | unary | paren | construct
construct    := IDENT "(" args? ")"
args         := expr ("," expr)*
member       := expr "." IDENT
call         := expr "(" args? ")"
```