

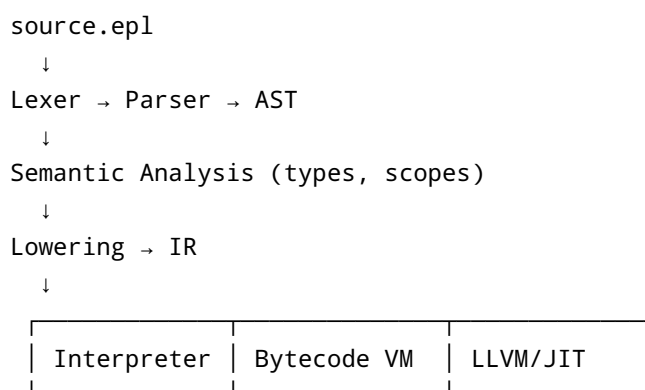
# EPL Language Development Plan

This document outlines a **complete roadmap** for developing EPL: a new programming language with three execution modes (Interpreter, Bytecode VM, LLVM/JIT). The plan is structured from 0% (design) to 100% (release).

---

## 1. Architecture Overview

**Pipeline:**



- **Interpreter:** Immediate execution (AST walker). - **Bytecode + VM:** Portable `.epc` files, platform-independent. - **LLVM/JIT:** Native high-performance compilation.

---

## 2. Development Phases

### Phase A — Project Setup & Language Spec

- Repo skeleton (`src/`, `tests/`, `docs/`).
- Language design doc: syntax, core types, control flow, functions.
- Toolchain setup (C + LLVM).

### Phase B — Frontend (Lexer, Parser, AST)

- Lexer for keywords, identifiers, numbers, strings, operators.
- Recursive-descent parser → AST with source locations.
- Parser unit tests.

### Phase C — Semantic Analysis & Type System

- Symbol table with scope management.
- Type checker (primitives: int, float, char, string, bool, void).
- Coercion rules and static error detection.

## Phase D — Interpreter & REPL

- AST-walking interpreter with runtime values.
- Symbol table for execution.
- REPL + script runner.
- Tests: factorial, loops, string ops.

## Phase E — Intermediate Representation (IR)

- Define TAC/SSA-style IR.
- AST → IR lowering with basic block support.

## Phase F — Bytecode Format & Compiler

- Define `.epc` binary format: header, constant pool, functions, code.
- Opcode set (stack-based VM for MVP).
- Compiler from IR → bytecode.

## Phase G — Virtual Machine (EPL VM)

- VM loop: fetch-decode-execute.
- Operand stack + call stack.
- GC/refcount for strings/objects.
- Native runtime functions: `print`, `read`, `malloc_wrapper`.

## Phase H — Runtime & Standard Library

- Core runtime (`print`, math, strings, I/O).
- Shared between VM and LLVM backend.

## Phase I — LLVM Backend

- IR → LLVM IR lowering.
- JIT compilation with ORC.
- AOT compilation to object/executable.
- CLI: `eplc --jit`, `eplc --aot`.

## Phase J — Tooling & Ecosystem

- CLI tool (`eplc`) with subcommands: run, compile, bytecode, disasm, repl.
- Package manager skeleton (`ep1pkg`).
- Language Server Protocol (LSP) server.
- CI pipeline with unit + integration tests.

## Phase K — Optimizations & Advanced Features

- IR-level optimizations (constant folding, DCE).
- Tiered execution (interpret → profile → JIT).
- Optional tracing GC.
- Concurrency features.

## Phase Z — Release & Maintenance

- v1.0 release with docs, tutorials, and installers.
  - Backwards compatibility policy.
  - Contributor onboarding.
- 

## 3. Repository Layout

```
/epl
  /src
    /front    (lexer.c/h, parser.c/h, ast.c/h)
    /ir       (ir.c/h, lower.c/h)
    /vm       (vm.c/h, bytecode.c/h, gc.c/h)
    /backend  (llvm_gen.c/h)
    /runtime  (runtime.c/h)
    /tools    (eplc.c)
  /tests
    /unit
    /integration
  /examples
  /docs
  Makefile or CMakeLists.txt
```

---

## 4. Bytecode Format (MVP)

- **Header:** Magic `EPLC`, version, constant count, function count.
  - **Constant Pool:** ints, floats, strings.
  - **Function Table:** name, arg count, local count, code offset, length.
  - **Opcodes:** Stack-based, minimal set:
    - `OP_CONST_INT`, `OP_LOAD_VAR`, `OP_STORE_VAR`
    - `OP_ADD`, `OP_SUB`, `OP_MUL`, `OP_DIV`
    - `OP_JMP`, `OP_JMP_IF`, `OP_CALL`, `OP_RET`
    - `OP_PRINT`, `OP_HALT`
- 

## 5. Testing & CI

- Unit tests: lexer, parser, semantic checks.
  - Parity tests: interpreter vs VM vs LLVM backend.
  - Fuzzing: parser and VM.
  - CI: cross-platform builds, sanitizers (ASAN, UBSAN).
-

## 6. Risk Management

- **Feature creep** → Start with MVP features only.
  - **Backend inconsistency** → Single frontend + IR.
  - **Memory safety** → Start with refcounting, add GC later.
  - **LLVM complexity** → Begin with `.ll` text generation, expand later.
- 

## 7. Next Immediate Steps

1. Create repo skeleton & write 1-page EPL spec.
  2. Implement lexer, parser, AST with tests.
  3. Build AST-walking interpreter and REPL.
  4. Design IR and lowering.
  5. Prototype minimal VM executing hard-coded bytecode.
- 

## 8. Completion Checklist (v1.0)

- Interpreter, VM, LLVM backends pass all tests.
  - Stable runtime & standard library.
  - Tooling (CLI, LSP, package manager).
  - Docs & tutorials.
  - CI pipelines green across platforms.
  - Distribution packages released.
-