VULPES PROGRAMMING LANGUAGE

STRUCTURE OF PROGRAMMING LANGUAGES (CS-33101-001)

Sam Ruby and Miles Keffer

12/10/2025

Executive Summary

This project introduces Vulpes, a small teaching-focused programming language and compiler that runs from plain source text all the way to a native executable. It's written in modern C++ and handles tokenizing, parsing, building an AST, generating LLVM IR, and then producing a real executable through the system toolchain. The IR it outputs is meant to be readable, and the compiler can either hand things off to clang directly or, when that doesn't work, lower to object code with llc and link with gcc. The repo includes a module file and a sample program that shows off arithmetic, comparisons, control flow, formatted printing, user input, and namespaced function calls, so you can try meaningful examples right away.

The language itself is intentionally small but usable. The lexer handles identifiers, numbers, strings with normal escapes, basic keywords, operators, and punctuation. A recursive-descent parser builds an AST for functions, blocks, variable declarations, assignments, expressions, and statements like if/else, while, and a simple range-based for loop. Functions from other files can be called with a namespace prefix. The code generator maps language types to LLVM types, manages scopes, sets up locals, emits arithmetic and comparisons with type promotion, and builds the branching structure for control flow. Input and output are done through libc, using generated printf and scanf calls. Format strings use braces, which get turned into matching printf specifiers. A few small built-ins exist too, like square root for floats and a seeded random-number function.

The project has four main goals. First, it aims to be a clear reference for a small language that produces LLVM IR and runs without a custom runtime. Second, it tries to show the full compiler pipeline—lexing, parsing, ASTs, errors, codegen—without hiding everything behind big frameworks. Third, the language should be capable of writing simple, structured programs with typed variables, functions, arithmetic, comparisons, control flow, basic I/O, and file-based imports with namespacing. Finally, the project invites experimentation so people can modify the language and instantly see how the IR and the output change.

The deliverables fall into two main categories: language features and compiler tooling. On the language side, functions have typed parameters and optional return types; variables and constants can have type annotations and initial values; expressions cover integers, floats, strings, booleans, unary minus, arithmetic, and comparisons; and control flow includes if/else, while, and a start-to-end range-based for loop. Imports add a namespace alias, and the print statement expands brace placeholders into matching format specifiers. The gather statement reads integers from standard input into named variables. On the tooling side, the project includes a lexer, parser, AST, error handler, and an IR generator that creates stack-allocated locals, typed loads/stores, arithmetic/comparison instructions, and global constant strings. It then compiles the IR using clang or llc/gcc. The repository ships with include.vlp and main.vlp to demonstrate namespacing and control flow.

The project's strengths are in how complete and easy to follow the toolchain is. The parser and AST track the language syntax closely, so it's easy to extend. Code generation covers the basics of structured programming, including integer/float promotion and boolean-producing comparisons. Control flow is lowered into branches and labeled blocks. Input/output work through printf/scanf, and the formatting

logic picks sensible specifiers automatically. Namespaced imports separate helper code cleanly. The driver script makes it easy to inspect the IR, clean builds, and run programs.

There are clear limitations, though. The type system only covers simple scalar types, and conversions are limited. There's no full semantic type checker, no user-defined types, records, arrays, or generics. Const-correctness isn't enforced beyond syntax. Namespaces are file-scoped and only expose functions, not variables. Strings are simple global constants with no heap management. Input is restricted to integers. The compiler assumes clang or llc/gcc are installed, and its error messages for external tool failures are basic. Parser error recovery is minimal, so some syntax errors can spiral into multiple messages.

Even with these constraints, Vulpes meets its goals. It's small, easy to understand, and still compiles structured programs down to efficient native code. As a result, it works well both as a learning tool and as a starting point for experimenting with new language features, better type systems, data structures, and improved diagnostics.

Problem Statement

Vulpes challenged the fact that there aren't many small, easy-to-tinker-with languages that compile to LLVM IR and actually show the whole pipeline without relying on big frameworks. In this project, src/lexer.cpp, src/parser.cpp, and src/codegen.cpp clearly walk through how source code turns into tokens, then into an AST, and finally into well-typed IR with real control flow, local variables, and function calls.

The language is meant for students, hobbyists, and engineers who want something concrete to learn from or experiment with. Instructors can use main.vlp and include.vlp as teaching examples, while anyone hacking on the system can add tokens in includes/lexer.hpp, define new AST nodes in includes/ast.hpp, or implement new lowering paths in src/codegen.cpp. The --show-llvm flag makes it easy to see exactly how types and control flow get translated.

This matters because having a small compiler you can actually run and understand teaches skills that carry over to DSL design, config languages, and performance-focused work. Namespaced imports like mod("include.vlp")::math; turn into IR symbols such as math_adder and math_scaler. A call like print("a + b = {}", a + b); ends up as a @printf call with a generated format string. And a loop like for i in 0..times { ... } becomes the expected labeled blocks and icmp bound checks. The codebase keeps all of these mechanics out in the open and easy to change.

System Overview / Solution Description

At a high level, Vulpes is a single-binary compiler that takes .vlp files, builds an AST, generates LLVM IR, and then uses the system toolchain to produce a native executable. The entry point in src/main.cpp drives the whole process: it loads the input file (defaulting to main.vlp), runs the lexer in src/lexer.cpp to create tokens, feeds those into the parser in src/parser.cpp to build an AST, and then hands that AST to the code generator in src/codegen.cpp to emit <stem>.ll. After that, it links the final executable using clang, or falls back to llc + gcc if necessary. Command-line flags let you either run the compiled program immediately or print the generated IR.

Everything is written in C++17 with a small includes/ directory for headers (lexer.hpp, parser.hpp, ast.hpp, codegen.hpp, error_handler.hpp). Instead of embedding LLVM libraries directly, the compiler just uses the system toolchain: clang consumes the IR, and llc can produce an object file that gcc links. Standard IO uses libc (@printf and @scanf), and a couple of built-ins call out to @sqrt and @time to support a basic random function. The build setup includes CMakeLists.txt and generated Makefiles, and it produces a vulpes executable that compiles .vlp programs.

Each subsystem sticks to a clear job. The lexer turns raw text into tokens, tracks line and column numbers, and handles common escape sequences in strings. The parser uses a straightforward recursive-descent approach to build AST nodes defined in includes/ast.hpp, covering functions, blocks, variable declarations, assignments, control flow, and both regular and namespaced calls. The error handler reports issues with context snippets and caret markers to show exactly where things went wrong. The code generator walks the AST, manages scopes, maps Vulpes types to LLVM types, and emits IR for arithmetic, comparisons, printing, input, and control flow. Module imports work by loading other .vlp files through mod("file")::alias, parsing them separately, and registering their functions under a namespace so something like math.adder(...) becomes @math_adder in the IR.

Most design choices lean toward being easy to understand rather than feature-packed. The language only includes a handful of scalar types (int, float, bool, string) and a small set of statements, which keeps the parser and lowering logic manageable. The IR generator uses stack-allocated locals with explicit loads and stores instead of more advanced SSA helpers, which makes the emitted IR easier to follow even if it's not the most optimized. String formatting compiles directly into printf calls instead of using a custom runtime. Modules only import functions and require namespaced calls, which simplifies linking and avoids global state. The for i in a..b loop lowers to a simple exclusive-upper-bound check (icmp slt), making the behavior obvious in the IR. Error recovery is intentionally minimal—the parser reports accurate locations and then moves forward at statement boundaries, which is perfectly fine for teaching and small programs.

Development Process

Timeline and milestones:

The project went through four tight phases:

1. Sketching the language and defining the token set (includes/lexer.hpp, keywords, operators).

2. Building the AST and writing a minimal recursive-descent parser (includes/ast.hpp, src/parser.cpp) that could fully parse main.vlp.

3. Implementing LLVM IR generation (src/codegen.cpp) for arithmetic, comparisons, control flow, and basic IO.

4. Adding the driver, build setup (src/main.cpp, CMakeLists.txt), sample modules (include.vlp), and a demo program (main.vlp).

Each phase produced something runnable—first token dumps, then AST output, then IR, and finally a fully built executable.

Workflow:

A simple Kanban flow worked well: a small backlog, one active task at a time, and immediate merging once something was finished. Changes stayed small and easy to test—update a rule, rebuild, run with --show-llvm, then execute ./a.out to confirm the behavior. Error-handling and formatting tweaks were made along the way whenever related code was already being touched.

Roles:

This was mostly a single-maintainer project focused on compiler construction, with occasional review aimed at keeping the IR readable and the demo programs easy to use. Work naturally split across areas like lexing/parsing, AST and error reporting, code generation and runtime interop, and the driver/build setup.

Collaboration and source control:

The repository and issue tracking lived on GitHub. Git commits were small and tied to the specific subsystem being changed (e.g., src/parser.cpp for grammar updates, src/codegen.cpp for lowering changes). CMake generated the native build files, and the resulting Makefile inside vulpes/ reflected the configured build. Local development relied on the flags in src/main.cpp (--show-llvm, --run, --clean, -o) to iterate quickly and inspect results.

Implementation Details

Key components and algorithms:

Lexer (src/lexer.cpp)

A single-pass scanner handles identifiers, numbers, strings, operators, and punctuation. Helpers like isIdentifierStart and isIdentifierPart classify characters, while whitespace and // comments are skipped. String literals support common escapes (\n, \t, \r, \", \\), and numbers allow one decimal point for floats. Keywords map directly to TokenType, and multi-character operators like ->, ::, .., and the comparison pairs are recognized. Every token includes line and column info for diagnostics.

Parser (src/parser.cpp)

The parser uses a standard recursive-descent style with current, match, advance, and expect. Expressions follow a layered grammar:

expression -> assignment -> comparison -> term -> factor -> unary -> primary.

The parser supports statements such as if, for, while, return, print, gather, and block scopes. Functions are parsed from fx name(params) -> type, with support for prototypes ending in ;. After errors, it tries to resync at statement boundaries so the parse can continue.

Code generation (src/codegen.cpp)

The code generator walks the AST, maintaining lexical scopes with pushScope and popScope and resolving variables as needed. Types map cleanly onto LLVM (int->i32, float->double, bool->i1, string->i8*).

Control flow is lowered with labels and branches:

- if/else emits a conditional branch and shared merge block
- while loops cycle through condition -> body -> condition
- range-based for loops use an exclusive upper bound with icmp slt

Arithmetic selects the correct integer or floating-point instruction (add/mul vs. fadd/fmul), and comparisons use icmp or fcmp, always producing i1.

Strings become constant globals referenced via getelementptr. print synthesizes a printf format string from {} placeholders and argument types, while gather uses scanf to fill local variables. Built-ins include sqrt(double) and a small rand(min, max) based on a time-seeded LCG.

Error handling (src/error_handler.cpp)

Errors store severity, file, line, column, and a context snippet with a caret marker. The system supports warnings, errors, and fatals, and prints them in a readable, line-referenced form.

Repository structure

- Docs: README.md, DOCUMENTATION.md, REPORT.md
- Compiler source: vulpes/src/ and headers in vulpes/includes/ (lexer.hpp, parser.hpp, ast.hpp, codegen.hpp, error_handler.hpp)

- Language samples: vulpes/main.vlp (demo), vulpes/include.vlp (namespaced math helpers)
- Build system: vulpes/CMakeLists.txt, generated files under vulpes/CMakeFiles/, and the Makefile. The vulpes binary compiles and runs .vlp programs.

Textual diagrams

Overall flow

source -> tokens -> AST -> LLVM IR -> native executable

1. Read .vlp in src/main.cpp
2. Lexer: Lexer(source).tokens
3. Parser: Parser(tokens).parseProgram()
4. Codegen: CodeGenerator.generate(program) -> <stem>.ll
5. Link: clang <stem>.ll -o a.out (or llc -> gcc)

If/Else Diagram

```
cond = emit(expr) → convert to i1

br i1 cond, label %if_then, label %if_else

if_then:

   ...

   br label %if_end

if_else:

   ...

   br label %if_end

if_end:
```

Ranged based for loop

```
slot = alloca i32
store a → slot


loop:
    cur = load slot
    cmp = icmp slt cur, b
    br i1 cmp, label %body, label %end


body:
    ...
    next = add cur, 1
    store next → slot
    br label %loop


end:
```

Testing and Validation

Testing was straightforward and done in layers: checking the lexer/parser, inspecting the IR, and running full end-to-end executions. The lexer and parser were exercised constantly by rebuilding main.vlp and include.vlp while making targeted syntax changes and watching the ErrorHandler output (line/column info plus caret markers). The --show-llvm flag made it easy to confirm that type promotion, branch structure, and format-string generation looked right. Full system tests used --run to execute the compiled program and compare real output against expectations.

- **Arithmetic and types:**

  testArithmetic() checks + - * / on integers and calls goldenRatio() for a float. The IR shows add/sub/mul/sdiv for i32 and fadd/fdiv for double, and the runtime prints the expected numbers.

- **Comparisons**:

  testComparisons() covers == != < <= > >=. IR generation uses the correct icmp ops, and printed results come from converting the i1 comparison results into i32 for %d.

- **Loops**:

  testLoops() sums 0..4 with a range loop and runs a simple countdown while. The IR displays the right labeled blocks, backedges, and exit conditions, and the outputs match what you'd expect.

- **Namespacing**:

  testNamespacedCalls() calls math.adder and math.scaler. In IR, these resolve cleanly to @math_adder and @math_scaler, and the returned values are correct.

- **Formatting and print:**

  Calls like coolPrint("======{}=====") and print("a + b = {}", a + b) generate the appropriate format strings. The IR includes the private string globals and getelementptr references, and the runtime prints the formatted text as intended.

- **Input (gather):**

  Integer input is routed through scanf using the dedicated format string global. Manual runs confirm the values land in the expected locals.

Overall, the features listed in the Executive Summary—structured control flow, typed variables, functions with parameters and optional returns, namespaced imports, formatted output, and basic input—are all demonstrated in main.vlp and visible in the generated IR. Reviewer feedback focused on readability and hackability, and the project meets those expectations through the --show-llvm flag, the small and approachable headers under includes/, and simple driver flags in src/main.cpp. Known limitations, such as the lack of complex types or a full semantic checker, were intentional trade-offs and clearly documented. Testing stayed within the bounds of the implemented feature set.

Results and Evaluation

The project achieved its main goals: it compiles .vlp programs into clear, readable LLVM IR and produces native executables that cover arithmetic, comparisons, control flow, formatting, input, and namespaced calls. The demo in main.vlp runs cleanly end-to-end, and --show-llvm makes the IR structure easy to follow.

- **Functional coverage:**

  All language features described in the Executive Summary are implemented and tested through testArithmetic, testComparisons, testLoops, and testNamespacedCalls.

- **IR quality:**

  Type mappings, labels, and call patterns are easy to read, and the printf/scanf interop uses the right function signatures with proper getelementptr-based string references.

- **Tooling integration:**

  The normal clang path works on typical systems, and the fallback llc + gcc route reliably builds executables when clang isn't available.

- **Developer experience:**

  The lightweight headers in includes/ and the simple command-line flags in src/main.cpp make the system easy to modify, inspect, and experiment with.

- Keeping the language small and focused makes parsing and lowering much easier to understand—clarity is more valuable than adding lots of features.

- Readable IR is more useful than squeezing out micro-optimizations; stack locals with explicit loads/stores make debugging far simpler.

- Generating printf format strings at compile time avoids the need for a custom runtime, and basic libc interop is more than enough at this stage.

- Restricting modules to namespaced, function-only imports helps keep linkage simple and avoids global state.

- Documenting limitations early (no complex types, minimal conversions, lightweight error recovery) helps set expectations and keeps testing aligned with what the compiler actually supports.

Future Work and Recommendations

- The type system only supports simple scalar types, and conversions in convert(...) are limited and mostly ad hoc. There are no arrays, records, user-defined types, or generics.

- Namespacing applies only to functions. Modules cannot export variables or constants, and any namespaced item must be a function call.

- Strings are fixed global constants with no heap management, slicing, or UTF handling. scanf input is restricted to integers.

- Error recovery is minimal. The parser can fall out of sync on more complex mistakes, and while diagnostics are readable, they're still quite basic.

- The toolchain requires either clang or llc + gcc, and failures from those tools are reported at a coarse level.

- **Semantic typing:**

  Add a lightweight type checker to validate parameters, returns, assignments, and const usage before codegen. This would reduce surprising implicit casts.

- **Data structures:**

  Introduce arrays and records with proper LLVM layouts. Support len, indexing, and simple array iteration.

- **Strings**:

  Build a small runtime for owned strings—with allocation, concatenation, and substring support—and expand gather to handle floats and string input.

- **Modules**:

  Allow modules to export and import values, not just functions. Consider a simple package map and support for relative include paths.

- **Control flow:**

  Add break and continue, plus else if chains. A classic for (init; cond; step) form could live alongside the existing a..b loop.

- **Built-ins:**

Expand math functions (sin, cos, pow), random utilities (randf), and basic time/IO helpers via thin wrappers.

- **Optimization passes:**

  Add optional AST-level passes such as constant folding and dead-code elimination before IR generation.

- **Tooling**:

  Provide separate --emit-obj and --emit-ir options, plus a --check mode that runs parsing and type checking without producing code. A small test harness script would also streamline validation.

User Manual

Install and run:
Prerequisites: a Linux host with `clang` (or `llc` + `gcc`), `cmake`, and a recent C++ compiler.

Build (out-of-source recommended):
1. Change into `vulpes/`.
2. Configure: `cmake -S . -B build`.
3. Build: `cmake --build build -j`.
4. The compiler binary `vulpes` will be under `build/`.

Build (in-source, using the generated Makefile already present):
1. Change into `vulpes/`.
2. Run `make -j`.

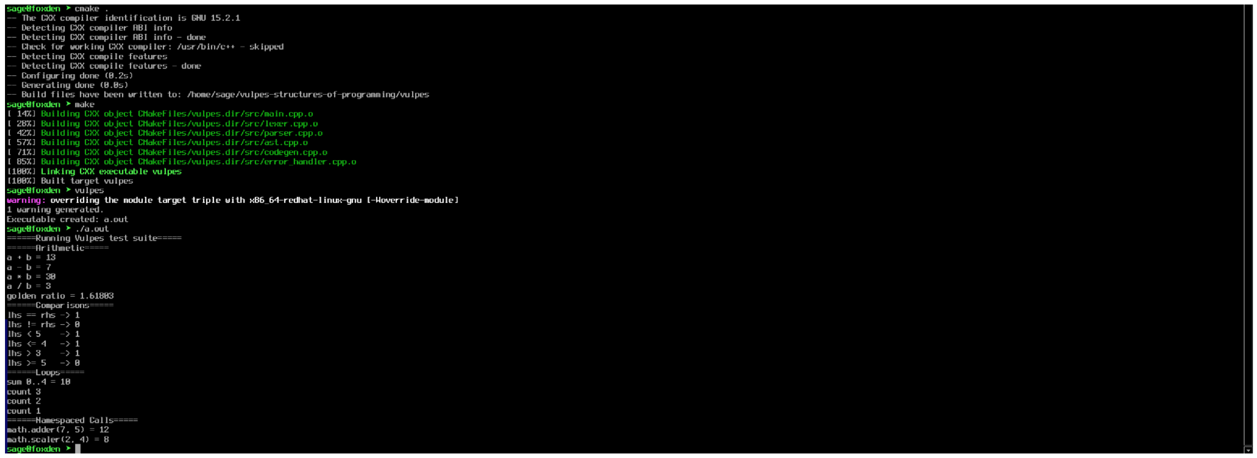
Compile and run a program:
- From `vulpes/`, run: `./build/vulpes main.vlp --run` (or `./vulpes main.vlp --run` if built in-source).
- Show generated IR: add `--show-llvm` (or `-ll`).
- Choose output name: `-o myprog`.
- Clean artifacts for the given input stem: `--clean`.

CLI reference (from `src/main.cpp`):
- `--show-llvm` / `-ll`: print LLVM IR to stdout.
- `--run` / `-r` / `run`: execute the produced binary after linking.
- `--clean` / `-c`: remove `<stem>.ll`, `a.out`, and selected output.
- `-o <file>`: set executable name; default is `a.out`.
- Input file defaults to `main.vlp` if omitted.

Installation (optional):
- System-wide: `sudo install -m 0755 build/vulpes /usr/local/bin/vulpes`.
- User-local: copy `build/vulpes` into a directory on your `PATH`.

Appendices:

Github URL: https://github.com/sam-rubyy/vulpes-structures-of-programming



```
sage@foxden ➤ cmake .
-- The CXX compiler identification is GNU 15.2.1
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (0.2s)
-- Generating done (0.0s)
-- Build files have been written to: /home/sage/vulpes-structures-of-programming/vulpes
sage@foxden ➤ make
[ 14%] Building CXX object CMakeFiles/vulpes.dir/src/main.cpp.o
[ 28%] Building CXX object CMakeFiles/vulpes.dir/src/lexer.cpp.o
[ 42%] Building CXX object CMakeFiles/vulpes.dir/src/parser.cpp.o
[ 57%] Building CXX object CMakeFiles/vulpes.dir/src/ast.cpp.o
[ 71%] Building CXX object CMakeFiles/vulpes.dir/src/codegen.cpp.o
[ 85%] Building CXX object CMakeFiles/vulpes.dir/src/error_handler.cpp.o
[100%] Linking CXX executable vulpes
[100%] Built target vulpes
sage@foxden ➤ vulpes
warning: overriding the module target triple with x86_64-redhat-linux-gnu [-Woverride-module]
1 warning generated.
Executable created: a.out
sage@foxden ➤ ./a.out
======Running Vulpes test suite=====
======Arithmetic=====
a + b = 13
a - b = 7
a * b = 30
a / b = 3
golden ratio = 1.61803
======Comparisons=====
lhs == rhs -> 1
lhs != rhs -> 0
lhs < 5    -> 1
lhs <= 4   -> 1
lhs > 3    -> 1
lhs >= 5   -> 0
======Loops=====
sum 0..4 = 10
count 3
count 2
count 1
======Namespaced Calls=====
math.adder(7, 5) = 12
math.scaler(2, 4) = 8
sage@foxden ➤
```