Vadim Goryuk

Boston Karymshakov

JP McLaughlin

Jack Zhu

Conclusions - Post Mortem Documentation

The SwiftLang project successfully delivers a small, interpreter-based language that makes the classic “source code → execution” pipeline concrete and understandable. Rather than being just a collection of Python scripts, SwiftLang is organized into clear, cooperating phases including tokenization, parsing, semantic analysis, and interpretation that mirror the structure of real-world compilers and interpreters.

At the front of this pipeline, the SwiftLangAnalyzer in tokenizer_analyzer.py performs lexical analysis by converting raw source into a structured stream of tokens. It tracks variables, literals, reserved words, and declaration counts. This shows how even at the tokenization stage, a language implementation can begin to gather useful static information about a program before it is parsed or executed.

The recursive-descent Parser then builds a well-defined abstract syntax tree (AST) out of that token stream, using node types such as Program, DeclStmt, AssignStmt, IfStmt, WhileStmt, BlockStmt, PrintStmt, VarExpr, LiteralExpr, and BinaryExpr. The parser enforces the grammar of SwiftLang and handles operator precedence, parentheses, and block structure. This makes control flow (if/else, while) and expressions (+, -, *, /, comparisons, boolean logic) explicit and testable.

On top of this syntax layer, the SemanticAnalyzer walks the AST and builds a symbol table capturing the current state of each variable: its type, its value (when known), and whether it has been declared. This phase enforces important language rules, including prohibiting duplicate declarations, disallowing assignments to undeclared variables, and requiring that conditions in if and while evaluate to boolean values. At the same time, it demonstrates a simple form of dynamic typing: a variable's recorded type can change as it is reassigned from integers to floats, strings, or booleans.

Execution is handled by the tree-walking Interpreter, which evaluates AST nodes directly using the semantic symbol table as its runtime environment. It implements arithmetic expressions, comparisons, boolean operations, variable updates, print output, and structured control flow. Through this interpreter, constructs like if and while become concrete runtime behavior: branches are taken or skipped based on evaluated conditions, loops repeatedly check conditions and update state, and variables evolve over time. The separate symbol_table_generator.py demonstrates how symbol-table logic can be reused independently of full interpretation, by building and printing a symbol table using a custom hash-table-backed structure. This reinforces the idea that compiler-like analyses can power tools beyond just "run the program," such as static reports or IDE-style insights.

The main.py module ties all these pieces into a single command-line experience. Given a .sl file, it runs the tokenizer, parser, semantic analyzer, and interpreter in sequence, and reports errors at the appropriate stage with clear labels such as "Syntax Error," "Semantic Error," or "Runtime Error." This end-to-end driver turns the underlying architecture into a usable language tool, so that SwiftLang programs can be written, saved, and executed like programs in any other interpreted language.

Beyond the core implementation, SwiftLang has supporting materials. The pytest-based tests/ directory exercises all major components: tokenization, parsing, semantic rules, interpretation, symbol table generation, and CLI behavior. These unit tests not only help catch regressions when the language is extended, but they also act as executable documentation of expected behavior for both valid and invalid programs. The separate language tutorial provides a concrete introduction with a step-by-step example and an exercise, while the language reference manual summarizes keywords, operators, and syntax forms in a compact, technical format. The README includes install instructions, these documents transform SwiftLang from “just code” into a complete educational artifact that a new user—or a future student—could pick up and understand without needing the original author on hand.

At the same time, SwiftLang deliberately keeps its scope modest since it is meant to be a toy interpreter. Some reserved words (such as fun, for, or switch) are recognized by the tokenizer but are not yet fully implemented in the grammar or runtime. The current semantic analyzer performs only basic type checking and allows flexible type changes on assignment, leaving room for stricter and more advanced typing rules. Error reporting is clear but intentionally simple, without sophisticated recovery strategies or detailed source locations. These make the next steps obvious: adding first-class functions and procedures, enriching control structures, enforcing more rigorous type systems, improving error diagnostics, or targeting a bytecode or virtual machine instead of directly interpreting the AST.

Overall, SwiftLang demonstrates a language design and implementation at an educational scale. It shows how theoretical topics such as tokens, grammars, symbol tables, semantic checks, and interpreters can map directly onto concrete modules and observable behavior. With its interpreter architecture, symbol-table tooling, automated tests, and documentation, SwiftLang is a demonstration of how a small language can be designed, built, explained, and verified.