A BASIC Compiler: GW-BASIC, that is…

Samuel Caldwell

Champlain College

CSCI-430: Compilers

Dr. Kenneth Revett

29 October 2025

A BASIC Compiler: GW-BASIC, that is…

This report documents a GW-BASIC→LLVM compiler and developer environment targeting AMD64 and ARM64. True to the Texas maxim that nothing worth doing is not worth overdoing, the system—built in 56 hours (Sat 00:15–Wed Oct 29, 2025, ~15:04 CDT)—produces LLVM IR and native binaries and emits rich logs for compiler diagnostics.

Source code: <https://github.com/sam-caldwell/csci-430>.

Figure : The Compiler's First Output

A screenshot of a computer program

AI-generated content may be incorrect.

In the above illustration, we see the project factorial in ‘build/demos/factorial’ which was compiled using ‘./build/basic\_compiler/basic\_compiler’ using the ‘make demo’ automation. We see alongside the factorial binary executable .asm files with the generated assembly language code, .bc and .ll files containing LLVM byte code and LLVM text as well as logs from each stage of the compiler process.

In the following pages we will examine how this compiler is constructed. We will look at this student’s belief in test-driven development as a means of rapidly developing software and the architecture of the GW-BASIC compiler.

*(Note: The assignment said we should “implement a basic compiler,” and this student being an Asperger’s case took the assignment literally and wrote a BASIC compiler, GW-BASIC having been his first language.)*

# The Source Code

Project development started with the following GW-BASIC program to calculate the factorial of some input:

Figure : factorial.bas

A screenshot of a computer

AI-generated content may be incorrect.

This program can be found at <https://github.com/sam-caldwell/csci-430/tree/main/demos>.

# Development Approach

With this starting point, and with a simple clang/LLVM environment setup in JetBrains Clion, we began implementing main.cpp to handle command line arguments which would evolve as the program evolved, starting with the lexical analyzer. As source code was defined for features, unit tests were written to ensure expected functionality and guard against regressions and defects as the project evolved. This is consistent with modern software engineering practice, slicing across the problem space instead of attacking the problem in layers (as illustrated, below):

Figure : Iterative "Slicing" the Problem Space

A diagram of a pyramid

AI-generated content may be incorrect.

By working along diagonal slices of the problem space, the developer is able to iterate and improve features across the feature boundary, revising across the stack in an adaptive manner. The alternative requires a well-planned end-to-end project to avoid rewrites, consuming additional time. By building tests and source code in parallel as the project started from main.cpp to the Lexer and beyond, this student was able to develop a working compiler in a short timeframe. As new features were added, new flags were added to the command line quickly, using a takeOptValue() helper function:

Figure : main() and the CLI flag handler

A computer screen shot of a program

AI-generated content may be incorrect.

As this project evolved, the following rules were applied:

1. Every unit/integration/end-to-end (e2e) test would be in a separate file.
2. Source code files would remain small, specific and well documented

This is demonstrated in the following file:

Figure : Example of Modular Coding Style

A screen shot of a computer program

AI-generated content may be incorrect.

This modular code style has kept the project simple, testable and easily readable with less change of an accidental change—as can happen in larger files.

# Compiler Structure

The modular coding style was preserved through the compiler structure. The project’s C++ code organized the CLI, token, AST, Lexer, Parser (Syntax/Semantics) and Optimizer into clearly delineated files and directories. Stages of the compiler such as the Lexer were constructed as C++ classes to further enhance the organization and testability of the code.

Figure : Modular File Hierarchy

A screenshot of a computer

AI-generated content may be incorrect.

Our Cmake + Ninja build system further extended this through file globbing to auto discover source files and through organizing the build output into a nicely organized hierarchy.

Figure : CMake File Modularity

A screenshot of a computer

AI-generated content may be incorrect.

This student must confess that much of this pattern was built over the past several years, but with this confession, this student must also assert that this well-defined habit of code organization and build automation has increased development velocity for this and other projects.

# The Compiler Internals

The compiler in this project is rather simple, partly because the GW-BASIC language is rather simple. In this section we discuss the different stages of the compiler.

## Lexer

The lexer (include/basic\_compiler/Lexer.h, src/basic\_compiler/lexer/\*.cpp) converts raw GW-BASIC source into a flat token stream with source positions. It maintains a cursor (pos\_, line\_, col\_) over the input and provides single-character lookahead (peek, peekNext). Whitespace handling (skip\_whitespace.cpp) treats spaces, tabs, and \r as ignorable and recognizes two comment forms: apostrophe (') to end-of-line (skip\_to\_eol.cpp) and the REM keyword, which is recognized in identifier\_or\_keyword.cpp and causes the rest of the line to be skipped, emitting a synthetic NewLine. Numeric literals are scanned by number() with a single optional dot; tokens are classified as Integer or Float based on whether a dot was seen. String literals (string\_literal.cpp) support basic escapes (\n, \t, \", \\), throw on unterminated quotes, and return a String token whose lexeme is the unescaped payload. Identifiers/keywords are recognized by identifierOrKeyword(), which uppercases the lexeme and maps it to TokenType for the supported vocabulary: LET, PRINT, IF, THEN, GOTO, END, FOR/TO/STEP/NEXT, GOSUB/RETURN, and INPUT. Punctuation and operators (+ - \* /, = <> < <= > >=, ( ) : ,) are emitted as their own token kinds; the single = token (Assign) serves both as assignment in statements and equality in expressions—the parser disambiguates by context. Newlines are surfaced as NewLine tokens (not discarded), which allows the parser to preserve BASIC’s line-oriented structure. Throughout tokenization, the lexer can optionally log each token (log\_token.cpp) and can be configured with a log path; lexical errors raise LexError with accurate line:col positions to enable clean diagnostics in later phases.

We can see evidence of the lexer operations in the .lex.log files:

Figure : Lexer Logs

A screenshot of a computer program

AI-generated content may be incorrect.

## Syntax

The “parser” class consumes the token vector and builds a typed AST. The root “Program” holds an ordered list of Line nodes; each Line contains its numeric label and a sequence of statement nodes. parseProgram() skips leading blank lines and repeatedly calls parseLine() until EndOfFile. parseLine() requires an Integer line number, then parses one or more statements separated by : until a newline; it also emits optional “syntax” log entries that include node kind and source position. Statement coverage matches the language subset: assignments (with optional LET, via parse\_assign\_or\_let.cpp), PRINT (strings or numeric expressions), GOTO, GOSUB/RETURN, END, single-line FOR … TO … [STEP …] … : NEXT [var] (body collected until NEXT, in parse\_for.cpp), INPUT name, and IF <comparison> THEN <line>. Expressions are parsed with standard precedence: parsePrimary() handles numbers, strings, identifiers, and parenthesized expressions; parseUnary() recognizes unary +/-; parseFactor() and parseTerm() implement \*// and +/- respectively; parseComparison() folds =, <>, <, <=, >, >= onto the arithmetic layer, producing BinaryExpr nodes tagged with BinaryOp. The parser enforces required tokens with consume() and throws ParseError on mismatches, producing precise messages (e.g., “Expected ')'”, “Expected line number after THEN”). Every AST node carries a SourcePos captured at parse time for downstream logs and error reporting. The resulting tree is a minimal, well-typed representation designed for a simple, single-pass code generator.

## Semantics

This phase is intentionally lightweight, due to the simplicity of GW-BASIC, and is implemented inside the code generator as explicit “declaration collection” passes (collect\_decls.cpp, collect\_stmt\_vars.cpp, collect\_expr\_vars.cpp) plus optional semantic logging. Rather than a separate type checker, the compiler assumes a single numeric type (double) for variables and treats strings as literal constants used only for printing. collectDecls() clears state, gathers and sorts all line numbers, and builds a lineMap\_ (line number → Line\*) used to resolve GOTO, GOSUB, and IF … THEN targets. It also walks every statement to discover the referenced variables and string literals. Variables encountered in either statements or expressions are inserted into variables\_; these drive stack allocation in the entry block and on-demand allocation via ensureVarAllocated() for cases like INPUT. String literals are interned in strLiteralId\_ so they can be emitted once as IR globals and referenced by id. The traversal is precise about control-flow requirements: IfStmt conditions are required to be comparisons, and the generator will later throw CodeGenError if non-comparison expressions reach conditional branching. Semantic logs (when enabled via setSemanticLogPath()) record discoveries such as VarRef x @ line:col, Assign x, and loop metadata For var=x. Because the language does not have declarations, scoping, or a static type system, there is no symbol table in the traditional sense and no type inference—semantic analysis is focused on building the minimal facts needed for correct IR layout (variables, strings, line targets) and validating simple invariants that codegen relies on.

## AST Optimization (constant folding & algebraic simplification)

Nothing says this student lives a riotous weekend life like the words “Algebraic Simplification.” In AstOptimizer.h and ast\_optimizer.cpp the compiler implements an optional, semantics-preserving optimization pass that rewrites expressions (and select statements) before IR emission. The optimizer does not depend on target details and operates purely on the AST. It performs constant folding across arithmetic and comparisons: binary nodes with constant operands are evaluated and replaced with a NumberExpr; unary plus is eliminated; unary minus of a constant is folded. A handful of algebraic identities simplify common cases without changing value: x + 0 → x, 0 + x → x, x - 0 → x, x \* 1 → x, 1 \* x → x, x / 1 → x, and any multiply by zero collapses to 0. For comparisons, constant operands are evaluated so IF 1 < 2 THEN … becomes an unconditional branch candidate. The pass also recognizes IF statements with constant conditions and rewrites them: if true, it replaces the statement with a direct GOTO to the target line; if false, it removes the statement entirely. Loop constructs are preserved, but trivial STEP 1.0 cases are normalized such that codegen can take the default path. The optimizer is deliberately conservative—no reordering, no CSE, and no cross-line control-flow edits—so it remains transparent and easy to reason about. It’s integrated via Compiler::compileStringOptimized(), which runs Lexer → Parser → AstOptimizer → CodeGenerator, letting tests exercise both optimized and non-optimized flows. This yields simpler IR, fewer runtime operations, and cleaner control flow with zero change in observable behavior.

## Code Generator (LLVM IR Emission)

The CodeGenerator (include/basic\_compiler/codegen/CodeGenerator.h, src/basic\_compiler/codegenerator/\*.cpp) produces textual LLVM IR for a single main function using a straightforward block-structured scheme keyed by BASIC line numbers. generate() orchestrates: it first calls collectDecls(), then emits a header with external declarations for printf/scanf (emit\_header.cpp), global constants for string literals and fixed format strings (emit\_globals.cpp), the function prologue allocating each discovered variable as alloca double initialized to 0.0 and branching to the first line (emit\_main\_prologue.cpp), one labeled block per source line (emit\_line\_block.cpp), and an exit epilogue returning i32 0 (emit\_main\_epilogue.cpp). Expressions are lowered recursively to SSA temporaries as double; arithmetic uses fadd/fsub/fmul/fdiv, unary minus uses fsub 0.0, %x (emit\_expr.cpp). Comparisons compile to fcmp oeq/one/olt/ole/ogt/oge on double, yielding i1 that drives br i1 in IF … THEN (emit\_comparison.cpp). PRINT selects @.fmt\_str for strings and @.fmt\_num for numbers, calling @printf with either a pointer to the string global or a double value; string payloads are escaped for IR via escape\_for\_ir.cpp. INPUT materializes @.fmt\_in and calls @scanf, passing the pointer to the variable’s alloca. GOTO is an unconditional branch to the target line label; GOSUB is implemented by inlining the callee’s lines into a synthesized entry/continuation label pair (emit\_subroutine\_inline.cpp) until a RETURN or fall-through occurs. FOR lowers to a small control-flow diamond with init, compare, body, and increment blocks (emit\_for.cpp), supporting optional STEP. The generator includes rich optional logs (both “semantic” discovery and per-IR emission) and raises CodeGenError for unsupported constructs or violated invariants, producing deterministic, readable IR suitable for llc/clang backends.

# The Final Result

The code for this project speaks for itself. While the assignment was to produce a compiler that produces LLVM IR, this student does not know LLVM as well as ARM64 or AMD64. Even when writing unit and integration tests, the ultimate test became “Would the <redacted> thing run?” (a question muttered many times Monday morning in the pre-dawn hours. The only way to ensure the LLVM output was correct was to produce an AMD64 binary (and later a ARM64 binary when the AMD64 laptop suffered a hardware failure Monday afternoon[[1]](#footnote-1)). Extending the project to produce a working binary appeared to be the shortest path to being certain that the LLVM would be correct.

Figure : make demo (proof the compiler compiles)

A screenshot of a computer program

AI-generated content may be incorrect.

We added ‘make demo’ to the project to run the compiler and compile the demos/factorial.bas program mentioned earlier. From that we were able to run the resulting file to produce the following:

Figure : Factorial.bas compiled binary execution results

A screenshot of a computer

AI-generated content may be incorrect.

In this screenshot, we see that the factorial program runs, that the user enters ‘5’ and presses <enter> and that the program returns the result 120. We can then work backward from this working ARM64 executable to validate any LLVM artifacts. But the project also includes multiple logs for various stages of the compiler process which aided in the rapid development of the solution:

Figure : Compiler logs

A black background with orange text

AI-generated content may be incorrect.

# Conclusions

This student hopes that the project as built thus far will allow rapid iteration over the next few week’s assignments. Using ‘make clean lint test build demo’, this project is able to test, compile and demonstrate the full project in approx. 3 minutes, 37 seconds to quickly identify and address problems. There are still bugs in this compiler, as in almost all software. And there are a few nits which should be fixed (e.g., ensuring that all which can be made const is made const). Nonetheless, this solution appears to be correct.

1. *“Caldwell’s Law of Unlimited Failure”* (coined in the 1990s) states that when you think you have reached rock bottom and everything possible has gone wrong, count to three because there is another level of hell beneath you…and you are standing on the trap door. This was coined during a particular evening when a lot of things were going wrong. This week’s issue was less severe but while debugging LLVM code, the original dev laptop crashed and a few hours of code was lost. [↑](#footnote-ref-1)