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 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)