COMPILER REPORT

DELORME Maximilien-Marie

Table des matières

1. Introduction

This project implements a mini-compiler capable of translating a simple imperative language into x86-64 assembly code (NASM), and then producing a functional ELF executable on Linux.
The compiler covers the entire compilation process: lexical analysis, syntax analysis, AST construction, intermediate representation (IR) generation, and final assembly code generation.

The project is designed for educational purposes, in order to clearly illustrate how a modern compiler works internally.

2. Target Environment

The compiler backend targets the Linux x86-64 architecture and directly uses Linux system calls for output and program termination.
All tests were performed on a Linux environment (Ubuntu), using the following tools:

- NASM for assembly
- ld for linking
- CMake for build automation
- C++ as the implementation language of the compiler

The generated binary follows the standard ELF format.

I developed on my mac M3, but I used my server to compile under a Debian 12 environment (it was easier for me since I already used nasm on linux X86, and I am not used to Apple Silicon ARM architecture).

3. Compiler Architecture

The compiler is designed in a modular way, with a clear separation of responsibilities:

Lexical Analysis

- Implemented using a scanner (scanner.lx, scanner.cpp)
- Converts source text into a sequence of tokens defined in tokens.hpp

Syntax Analysis

- Handled by the parser (parser.cpp, parser.hpp)

- Builds an Abstract Syntax Tree (AST)

AST (Abstract Syntax Tree)

- Describes the syntactic structure of the program
- Defined in ast.hpp
- Used as the central structure for the following phases

IR Generation

- Transforms the AST into an explicit intermediate representation
- Implemented in ir.cpp and ir.hpp

Code Generation

- Translates the IR into NASM assembly code
- Managed by codegen.cpp and codegen.hpp

Assembly and Linking

- The generated assembly code (output.asm) is assembled and linked to produce the final executable

4. Intermediate Representation (IR)

The intermediate representation is designed as a sequence of low-level instructions, similar to three-address code.
It includes:

- Assignments
- Arithmetic operations
- Conditional instructions
- Labels and jumps
- Output instructions
- Program termination instructions

Control structures (conditions, loops) are explicitly translated using labels and jumps, which simplifies the direct translation to x86-64 assembly.

5. Assembly Code Generation

The code generator translates each IR instruction into corresponding NASM instructions:

- Variables and temporaries are stored in the .bss section
- Constants and strings are stored in the .data section
- Executable code is generated in the .text section

The program entry point is _start, and Linux system calls are used to:

- Display values or messages
- Terminate the program with a return code

The output.asm file is a direct example of the compiler's output.

6. Runtime Support Functions

The compiler automatically generates some utility assembly functions when needed, including:

- String output
- Integer output

These routines are written directly in x86-64 assembly and follow Linux system call conventions and constraints.

7. Debugging and Issues Encountered

Several non-trivial issues were encountered during development:

Assembly code generated but not executed

This issue was related to the structure of the _start entry point and was fixed by ensuring that all IR instructions were properly translated.

Incorrect handling of some IR instructions

Some instructions were not correctly converted into assembly, which required adjustments in the code generator.

File organization and dependencies

Using CMake helped structure the project properly and simplified the build and testing process.

Debugging was mainly based on inspecting the generated assembly code and stepping through the produced binaries.

8. Final Result

The compiler is able to:

- Read a source file
- Generate a valid AST
- Produce a coherent IR
- Generate correct assembly code
- Produce a functional ELF executable

The output.asm and output.o files confirm that the full compilation pipeline works correctly.

9. Conclusion

This project demonstrates a complete implementation of a simple but functional compiler, from parsing to execution on a real architecture.
It highlights the importance of intermediate representations, control flow management, and low-level architecture-specific details.

The compiler provides a solid foundation for future extensions and optimizations. Personally, this project significantly improved my understanding of low-level programming concepts and provided a solid foundation for future projects in this field.