Universidad Nacional Autónoma de México

Computer Engineering

Compilers

Compiler documentation

TEAM 7:
320030772
320323719
320181520
423007262
424147295

Group:
5
Semester:
2026-I

Mexico City, Mexico. December 2025

Contents

1 Introduction

Compiler development is a central topic in computer science, as it facilitates the translation of high-level programming languages into machine-readable code as it is machine code. The main problem addressed in this report is the design and implementation of a basic compiler that can take a set of instructions written in a specific programming language and convert them into executable code. This process involves several phases that are grouped in two main phases: The analysis and synthesis phases. This report will focus primarily on the synthesis phase since the analysis phase has already been covered in previous reports. The synthesis phase is divided in 3 phases, which are the intermediate code generation, code optimization and target code generation, this phases are responsible for transforming the program into an executable format.

The motivation for focusing on the synthesis phase is to deepen our understanding of how compilers generate efficient executable code from a high-level programming language. After covering the analysis phases in previous reports, it is essential to explore how the compiler synthesizes intermediate representations, optimizes them, and finally generates target machine code. By developing these stages, students will gain a practical understanding of the process of turning high-level code into optimized machine code, which is critical for ensuring the performance and correctness of compiled programs.

For this being the last report, the main objective will be to accomplish the class objective, which is to implement compiler design tools and techniques to elaborate a base software that will optimize both memory and efficiency. The alumn as well will be able to discern to differentiate existing traductors to elaborate efficient software, fit to the problem to be solved. This main objective conglomerates a set of specific objectives, some that have already been covered on past reports, as well as some synthesis phase-directed, which may include:

- Apply and consolidate knowledge acquired in class by building a functional compiler that demonstrates understanding of the key concepts.
- Implement Intermediate Code Generation to translate the syntax tree into an intermediate representation of the program.
- Apply Code Optimization techniques to improve the intermediate code by reducing execution time and resource consumption.
- Implement Target Code Generation to convert the optimized intermediate code into machine-specific code that can be executed on the target platform.
- Consolidate and apply knowledge from previous phases to the synthesis phase, ensuring that the compiler produces correct and efficient executable code.

2 Theoretical Framework

2.1 Intermediate Representation (IR)

An Intermediate Representation is a program representation used internally by a compiler between the front-end and the back-end. It abstracts away details of the source language while remaining independent of the target machine. IRs are designed to be easy for the compiler to analyze, transform, and optimize. [1]

2.2 Three Address Code (TAC)

Linear intermediate representation in which each instruction contains at most one operator and three operands, usually two sources and one destination. TAC simplifies program analysis and optimization by breaking complex expressions into small, uniform operations. [2]

2.3 Static Single Assignment

Property of an IR in which each variable is assigned exactly once. New values created by the program are represented using distinct names.[3]

2.4 Basic Block

A basic block is a straight-line segment of code with no internal branches except to the entry at the beginning and at the end, respectively. Basic blocks serve as the fundamental unit for control-flow analysis and optimization.[4]

2.5 Code generator

A code generator is the compiler component responsible for converting intermediate representations into the target machine code. It transforms abstract operations into concrete instructions that conform to the constraints of the target architecture. [5]

2.6 Memory layout

The memory layout specifies how data types are represented in memory during execution. It defines useful data to be used by compiler to correctly allocate memory and access variables. [6]

2.7 Code optimization

Code optimization is the stage of compilation where the IR is transformed to produce a more efficient version of the program. Optimizations may target execution time, memory consumption, or power usage. Although compilers cannot change program semantics, they attempt to improve performance while preserving correctness.

This optimization is done by using a set of machine-independent optimizations that operate on the IR level and do not depend on the CPU architecture. Some of these said optimizations used in this project are listed below: [7]

- **Loop Optimizations:** Process of increasing execution speed and reducing the overheads associated with loops.
- **Constant Propagation:** Replaces variables with known constant values.
- **Constant Folding:** Evaluates constant expressions at compile time instead of runtime.
- **Dead Code Elimination:** Removes instructions or blocks that compute values never used.
- **Common Subexpression Elimination:** Detects and removes repeated expressions, using the first computation.
- **Algebraic Simplification:** Simplifies expressions using algebraic identities.

2.8 Target Code

Target code generation is the final stage of the synthesis phase. It converts the optimized intermediate representation into machine-specific instructions. This step must respect the constraints of the target architecture, such as instruction formats, register usage, and calling conventions. [8]

2.9 LLVM

LLVM is a collection of modular and reusable compiler and toolchain technologies. The LLVM project consists of a number of subprojects.

2.9.1 LLVM Core subproject

The **LLVM Core** libraries provide a modern source- and target-independent **optimizer** along with **code generation support** for many popular CPUs. These libraries are built around a well specified code representation known as the LLVM intermediate representation ("LLVM IR"). [9]

The LLVM representation aims to be light-weight and low-level while being expressive, typed, and extensible at the same time. It aims to be a “universal IR” of sorts, by being at a low enough level that high-level ideas may be cleanly mapped to it. By providing type information, LLVM can be used as the target of optimizations: for example, through pointer analysis, it can be proven that a C automatic variable is never accessed outside of the current function, allowing it to be promoted to a simple SSA value instead of a memory location. [10]

3 Development

3.1 Compiler Backend Implementation

The entry point of the compiler ('main') acts as the **Compiler Driver**, orchestrating the transition from the frontend's Abstract Syntax Tree (AST) to a fully functional executable. This process involves four distinct technical stages: LLVM infrastructure initialization, object code generation, runtime environment resolution, and dynamic linking.

3.1.1 LLVM Infrastructure Initialization

Before any code generation can occur, the compiler initializes the underlying LLVM core libraries. The calls to `LLVMInitializeAllTargetInfos`, `LLVMInitializeAllTargets`, and `LLVMInitializeAllAsmPrinters` are essential to configure the global state of the backend.

Purpose: These functions register the target architecture (e.g., x86-64) and the corresponding assembly printers. This ensures the compiler can emit machine code compatible with the host CPU where the compiler is running.

3.1.2 Object Code Generation

Following the semantic validation of the AST (via `validate_sdt`), the driver invokes the backend entry point:

```
codegen_generate_module(ast_root, "out.o");
```

This function translates the verified AST into LLVM Intermediate Representation (IR) in memory. Subsequently, the LLVM engine compiles this IR into native machine code (binary) and writes it to disk as a relocatable object file (`out.o`). At this stage, `out.o` contains the user's compiled logic but lacks the operating system interfaces required to execute (i.e., it has no entry point or system libraries).

3.1.3 Runtime Environment Resolution

To produce a standard Linux executable (ELF format), the object file must be linked with the C Runtime (CRT) startup files. Since the location of these files varies by Linux distribution, the compiler dynamically resolves their paths using `gcc` as a query tool via `popen`. The required files are:

`ld-linux-x86-64.so.2` The dynamic linker/loader required for program execution.

`crt1.o` Contains the `_start` symbol, which is the true entry point of the process. It initializes the stack and passes control to the user's `main` function.

`crti.o` **(CRT Init)** Contains the prologue for the `.init` section (global constructors).

`crtn.o` **(CRT End)** Contains the epilogue for the `.fini` section (global destructors).

3.1.4 Dynamic Linking

The final phase is the construction and execution of the linker command via `ld`. The compiler constructs a command string to combine the resolved paths with the generated object file. The linking order is strictly enforced to maintain binary compatibility:

1. **Dynamic Linker:** Specified via the `-dynamic-linker` flag.
2. **Prologue Files:** `crt1.o` and `crti.o` are placed *before* the user code.
3. **User Object:** The generated `out.o` file.
4. **System Libraries:** The `-lc` flag links the Standard C Library (libc), allowing the user's code to use system calls like `printf` or `malloc`.
5. **Epilogue File:** `crtn.o` is placed *last* to properly close the initialization sections.

Finally, the `system()` call executes this command, invoking the system linker to produce the final binary named `program`.

3.2 Code Generation Architecture

The `codegen.c` module serves as the semantic translator of the compiler. It traverses the verified Abstract Syntax Tree (AST) and emits LLVM Intermediate Representation (IR). This process is managed through the use of the LLVM C API, transforming high-level language constructs into low-level, target-independent instructions.

The code generation strategy relies on four key architectural components:

3.2.1 SSA Form and Memory Model

LLVM IR uses Static Single Assignment (SSA) form, where virtual registers are immutable. However, the source language (Mini-C) supports mutable variables. To bridge this gap, the backend employs a stack-based memory model:

- **Allocation:** Variables are allocated on the stack using the `alloca` instruction via the helper function `create_entry_alloca`.
- **Symbol Table:** The `sym_table` maps variable names to their memory addresses (`LLVMValueRef`), not their values.
- **Access:** Variable assignments generate `store` instructions, while variable reads generate `load` instructions. This delegates the complexity of register promotion (Mem2Reg) to LLVM's optimization passes later.

3.2.2 Module Lifecycle (`codegen_generate_module`)

This function acts as the backend entry point and orchestrates the compilation pipeline:

1. **Initialization:** Creates the global `LLVMModule` and the `LLVMBuilder`.
2. **Global Definitions:** Manually declares external functions (e.g., `printf`) to ensure ABI compatibility.
3. **AST Traversal:** Iterates through the root node's children to process functions.
4. **Object Emission:** Configures the `LLVMTargetMachine` based on the host architecture (e.g., x86-64) and emits the binary object file (`.o`) to disk using `LLVMTargetMachineEmitToFile`.

3.2.3 Function Generation (`codegen_function`)

Each function is translated into an LLVM Function object. The process involves:

- Defining the function prototype (return type and argument types).
- Creating the initial `Basic Block` (named "entry").
- Allocating stack space for parameters and storing the incoming argument values into these memory slots, ensuring parameters act as local variables.

3.2.4 Control Flow and Basic Blocks (`codegen_statement`)

Since LLVM IR lacks high-level flow control structures (like `if`, `while`, or `for`), these are constructed manually using a Control Flow Graph (CFG) of Basic Blocks and Branch instructions.

For example, the translation of a `while` loop involves creating three distinct blocks:

`cond_block`**:** Evaluates the loop condition. Uses `LLVMBuildCondBr` to jump to the body or the exit.

`body_block`**:** Contains the loop's statements. Ends with an unconditional jump back to `cond_block`.

`after_block`**:** The destination when the condition evaluates to false.

3.2.5 Expression Evaluation (`codegen_expr`)

This recursive function generates values (`LLVMValueRef`). It handles:

- **Arithmetic Logic:** Generating instructions like `Add`, `Sub`, `Mul`.
- **Ternary Operators:** Implemented using **Phi Nodes** (`LLVMBuildPhi`) to merge values coming from different predecessor blocks.
- **Function Calls:** Includes specific logic for variadic functions like `printf`, promoting arguments (e.g., `float` to `double`) to match C calling conventions.

3.3 Backend Architecture Visualization

To provide a comprehensive overview of the compiler's backend operations, this section visualizes two critical stages: the translation of high-level control flow into LLVM Intermediate Representation, and the subsequent construction of the final executable binary through system linking.

3.3.1 Control Flow Translation (Code Generation)

The translation of control structures, such as loops, requires manual construction of the Control Flow Graph (CFG). Since LLVM IR operates using basic blocks and branch instructions, high-level logic must be decomposed. Figure 1 illustrates how a `while` loop is semantically translated into three distinct basic blocks: a condition block, a body block, and an exit block.

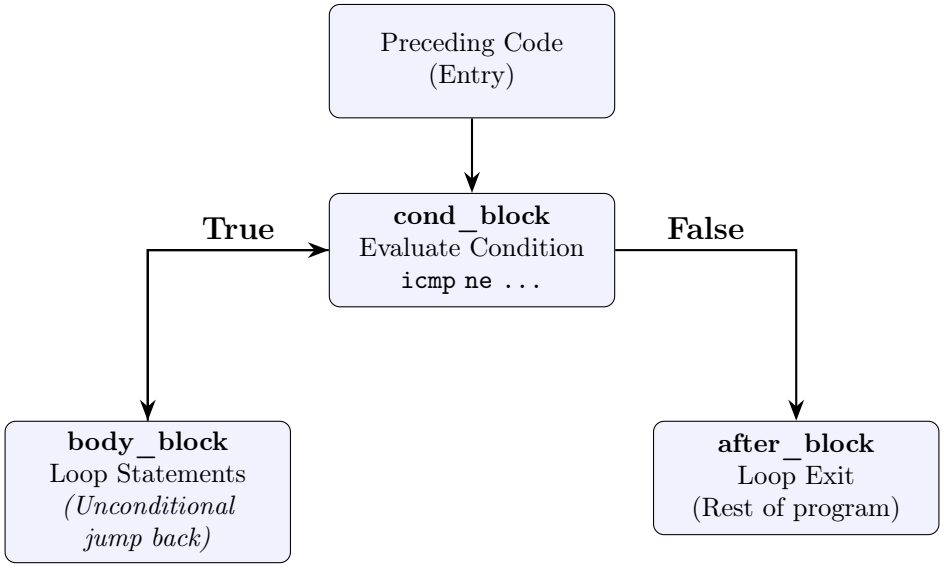

Figure 1: Translation of a `while` loop into LLVM IR Basic Blocks.

3.3.2 Executable Construction (Linking Process)

Once the semantic analysis and IR generation are complete, the compiler driver (‘main.c‘) generates a relocatable object file (‘out.o‘). To produce a standalone executable compatible with the Linux operating system, this object file must be linked with the C Runtime (CRT) startup files and standard libraries. Figure 2 details the data flow managed by the compiler driver to invoke the system linker (‘ld‘).

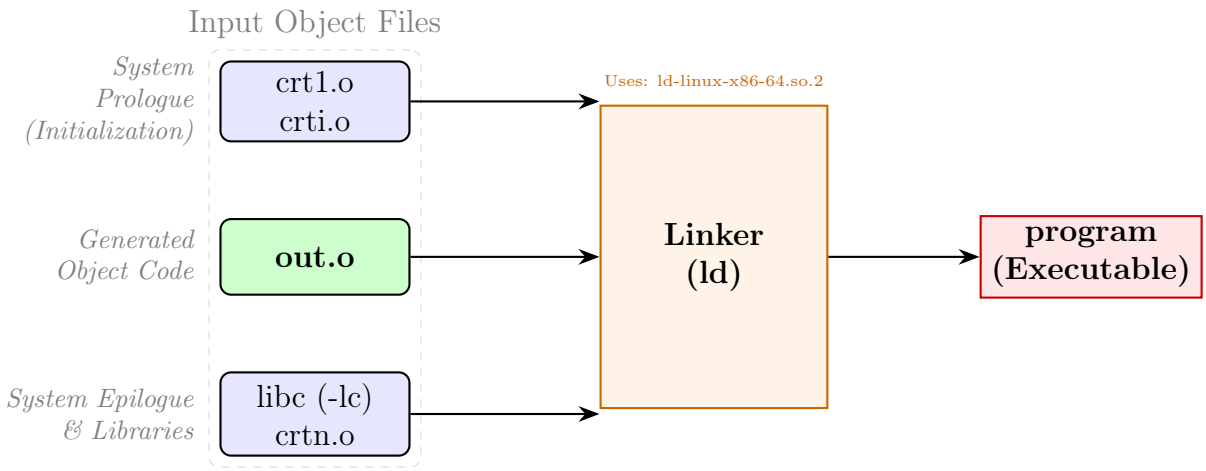

Figure 2: Diagram of the Linking Process managed by the Compiler Backend.

4 Results

The results obtained in the execution of the main program are as follows:

- Execution of the main program

```
[krunker_prp@vdlnx] 07 $ ./bin/main test/testCompiler1.c
[krunker_prp@vdlnx] 07 $
```

Figure 3: Execution of the main program providing a source file path.

```
[krunker_prp@vdlnx] 07 $ ./program
"Test 1: 105"[krunker_prp@vdlnx] 07 $
```

Figure 4: Execution of the result of compilation.

- Verbose option

```
[krunker_prp@vdlnx] 07 $ ./bin/main -s 'int main() {int a=10; printf("Hello World! %d", a); return 0;}' -v
PROGRAMA
  FUNCION
    TIPO: (Type Token 258)
    ID: main
    BLOQUE
      DECLARACION
        TIPO: (Type Token 258)
        OP_BINARIO: (Operator Token 307)
          VAR: a
          ENTERO: 10
      EXPR_SENTENCIA
        LLAMADA_FUNCION
          ID: printf
          CADENA: "Hello World! %d"
          ID: a
      RETURN
        ENTERO: 0
Total number of tokens: 21
OK: out.o succesfully generated (Object Code).
INFO: Linking executable (program) with ld...
OK: Compilation finished. Program 'program' generated.
Executing program ...
"Hello World! 10"[krunker_prp@vdlnx] 07 $
```

Figure 5: Execution of the main program with verbose option on.

4.1 Testing

A test suite was also developed to test a set of programs in order to verify the correct operation of the compiler:

Figure 6: Running testing suite.

5 Conclusions

With this being the final report, the complete development cycle of the compiler has allowed us to learn, apply and integrate all the concepts necessary to achieve the proposed objectives. Each report has addressed a different component of the compilation process and together they formed the foundation required to reach the main goal of the project, which is to design and implement a basic, functioning compiler capable of translating a specific programming language into executable code. Evaluating whether the specific objectives were met provides a clear measure of whether the main objective was accomplished as well.

The results presented in this report demonstrate that the compiler successfully completes the synthesis phase by generating a functional executable from the provided source code. This confirms that the concepts studied in class and those researched independently were understood and applied effectively within the process of the project.

Moreover, this project illustrates how the integration of the lexer, parser, semantic validator, intermediate representation, optimizer and code generator forms a coherent and interdependent compilation pipeline. By transforming the Abstract Syntax Tree into an intermediate representation, applying optimizations such as constant folding, constant propagation, dead-code elimination, algebraic simplification, loop transformations and ultimately producing object code that is linked into an executable binary, the compiler demonstrates a practical understanding of how high-level programs are systematically translated into efficient low-level instructions.

The implementation also highlights the importance of concepts such as memory layout, calling conventions, basic block construction and the role of SSA form in enabling optimization. These components reinforce how the synthesis phase abstracts the source program into architecture-aware machine-level operations while preserving correctness and adhering to target constraints. Furthermore, the successful use of the LLVM infrastructure and the correct handling of object emission and runtime linking validate the completeness and correctness of the backend.

In conclusion, the project confirms that all specific objectives were successfully achieved, thereby accomplishing the main objective established at the beginning of the course. This work demonstrates that the theoretical principles studied throughout the semester can indeed be translated into a functional system, reinforcing both conceptual understanding and practical proficiency in compiler construction.

References

[1] *Introduction to intermediate representation(ir)*, Website, Último acceso: 2025-12-01, Geeks For Geeks, 2022. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/introduction-to-intermediate-representationir/.

[2] *Three address code in compilers*, Website, Último acceso: 2025-12-01, Geeks For Geeks, 2025. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/three-address-code-compiler/.

[3] *Static single assignment (with relevant examples)*, Website, Último acceso: 2025-12-01, Geeks For Geeks, 2024. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/static-single-assignment-with-relevant-examples/.

[4] *Basic blocks in compiler design*, Website, Último acceso: 2025-12-01, Geeks For Geeks, 2025. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/basic-blocks-in-compiler-design/.

[5] *Simple code generator*, Website, Último acceso: 2025-12-01, Geeks For Geeks, 2025. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/simple-code-generator/.

[6] *Memory layout of c programs*, Website, Último acceso: 2025-12-01, Geeks For Geeks, 2024. [Online]. Available: https://www.geeksforgeeks.org/c/memory-layout-of-c-program/.

[7] *Code optimization in compiler design*, Website, Último acceso: 2025-12-01, Geeks For Geeks, 2025. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/code-optimization-in-compiler-design/.

[8] *Target code generation in compiler design*, Website, Último acceso: 2025-12-01, Geeks For Geeks, 2020. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/target-code-generation-in-compiler-design/.

[9] *Llvm — the llvm compiler infrastructure*, Website, Último acceso: 2025-12-01, LLVM Project, 2025. [Online]. Available: https://llvm.org/.

[10] *Llvm language reference manual*, Último acceso: 2025-12-01, LLVM Project, 2025. [Online]. Available: https://llvm.org/docs/LangRef.html.

[11] N. ACADEMY, *Compiler design*, https://www.youtube.com/playlist?list=PLBlnK6fEyqRjT3oJxFXRgjPNzeS-LFY-q, Accessed: 2025-11-03, Nov. 2023.

[12] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, 2006.

[13] A. W. Appel, *Modern compiler implementation in C*, 1st ed. THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE, 2004.

[14] Universidad Nacional Autónoma de México, *Introducción a la Programación*, Agosto 2017. Ciudad Universitaria, Coyoacán, Ciudad de México: Facultad de Contaduría y Administración, UNAM, 2017, Apunte electrónico. Plan de estudios 2012, actualizado 2016. ISBN en trámite. [Online]. Available: https://cedigec.fca.unam.mx/materiales/informatica/2/LI_1167_140120_A_Introduccion_Programacion_Plan2016.pdf.