Compiler Design Project: Custom Native Compiler "Trisynth"

V. Chitraksh CS23B054
P. Sathvik CS23B042
S. Danish Dada CS23B047

PROJECT OVERVIEW

The objective of this project is to design and implement a robust compiler for a custom-defined programming language. The language prioritises syntactical simplicity and developer ease-of-use.

The compiler follows a strictly modular architecture. Source programs are transformed into a machine-independent Intermediate Representation (IR), optimized for performance, and finally compiled into native assembly code using a pre-built assembler. While the primary goal is native execution, the modular IR design is extensible, allowing for potential future additions such as a C++ transpiler backend if time permits.

MODULE DESCRIPTIONS

1. Frontend Analysis

The frontend is responsible for validating the source code and converting it into a structured format.

- **Lexical & Syntax Analysis:** The source code is tokenized and parsed into an Abstract Syntax Tree (AST), ensuring adherence to the language grammar.
- **Semantic Analysis:** Performs type checking and scope resolution to ensure logical consistency.
- **IR Generation:** The verified AST is converted into a linear Intermediate Representation (IR), abstracting away high-level constructs.

2. Intermediate Code Optimization

The core of the compiler's efficiency lies in this phase. The optimizer performs multiple passes over the IR to reduce instruction count and execution time. Planned optimizations include:

- **Constant Folding:** Evaluating constant expressions (e.g., `3 + 5`) at compile-time rather than runtime.
- **Dead Code Elimination:** Removing code that is unreachable or does not affect the program output.
- **Strength Reduction:** Replacing expensive operations (like multiplication) with cheaper equivalents (like bit-shifts) where possible.

3. Native Backend

The backend synthesizes the final executable machine code:

- **Assembly Generation:** The optimized IR is mapped to architecture-specific instructions (x86-64 or RISC-V).
- **Assembly & Linking:** An external assembler (NASM/GAS) converts the generated assembly code into a binary executable (`.o` → `./program`).

TECHNOLOGY STACK

- **Implementation Language:** Python
- **Target Architectures:** x86-64 / RISC-V
- **External Tools:** NASM (Netwide Assembler) or GNU Assembler

SYSTEM ARCHITECTURE

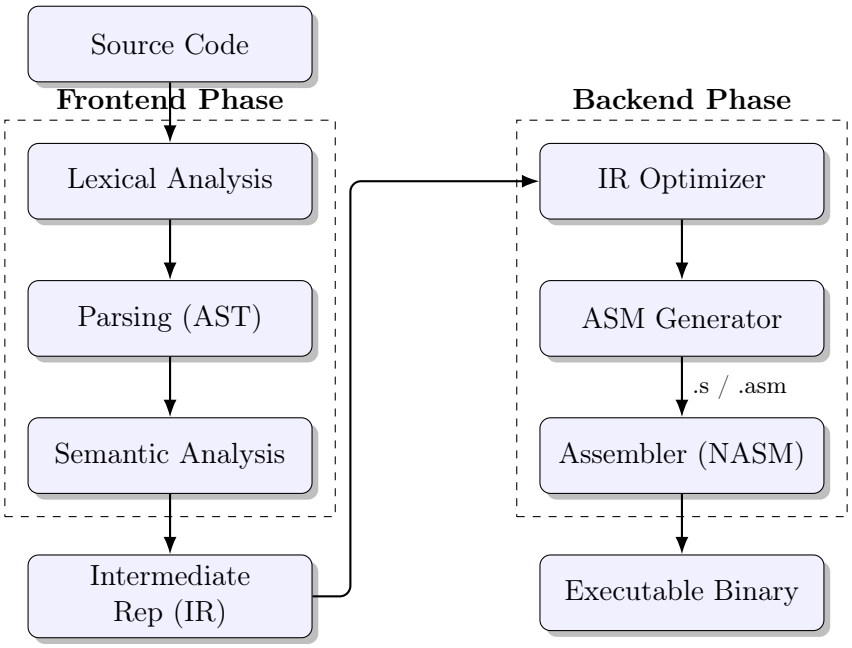

Figure 1: Compiler Workflow: A linear pipeline from Source to Native Executable.