Compiler Design Project Proposal

Custom Native Compiler: "Trisynth"

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

REFINED PROBLEM STATEMENT

Standard industrial compilers often obscure the translation process from high-level source code to native machine code due to monolithic designs and complex legacy support. This project addresses the challenge of building a transparent, **strictly modular compiler system** that prioritizes syntactical simplicity and developer ease-of-use.

We propose **"Trisynth"**, a custom native compiler implemented in Python that demonstrates the complete execution pipeline from Lexical Analysis to Assembly Generation. The system will bridge the gap between abstract syntax and hardware execution by transforming source code into a machine-independent **Intermediate Representation (IR)**, performing critical optimizations such as constant folding and dead code elimination, and finally generating architecture-specific assembly for **x86-64 or RISC-V** targets.

SYSTEM ARCHITECTURE

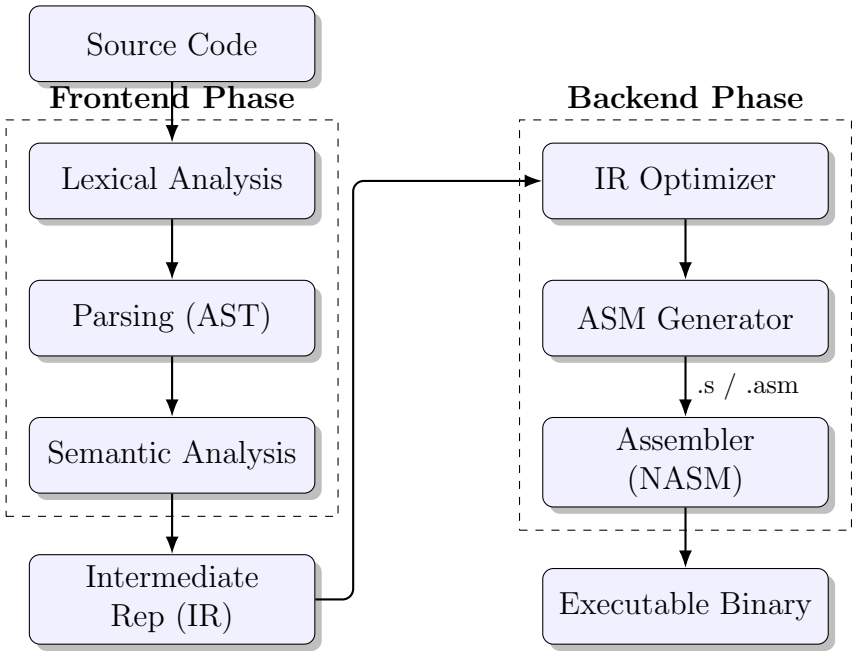

Figure 1: Compiler Workflow: A linear pipeline from Source to Native Executable.

MODULE DESCRIPTIONS

The compiler follows a linear pipeline architecture consisting of three primary phases.

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 (e.g., detecting type mismatches or undeclared variables).
- **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.

- **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.

Stretch Goal Optimizations:

- **Common Subexpression Elimination:** Identifying and calculating repeated expressions only once.
- **Loop Unrolling:** Reducing the overhead of loop control by executing multiple iterations per loop cycle.

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 $\rightarrow$ ./program).

LANGUAGE SPECIFICATION

The custom language is designed to be simple yet robust, featuring a strictly typed, procedural paradigm.

Language Paradigm

- Imperative, procedural programming model.
- Sequential execution with explicit control flow.
- No object-oriented or functional abstractions in the core language.

Type System

- Explicit, statically typed language.
- **Primitive Types:** `uint32`, `int`, `float`, `bool`, `char`, `void`.
- No implicit type conversions; type correctness enforced at compile time.

Variables and Constants

- Explicit declaration required before use.
- Support for mutable variables and immutable constants.
- Block-level lexical scoping with variable shadowing allowed in nested scopes.

Expressions & Operators

- **Arithmetic:** $+$, $-$, $*$, $/$, $\%$

- **Relational:** $<$, $>$, $<=$, $>=$, $==$, $!=$
- **Logical:** &&, ||, !
- **Increment:** Pre/Post $++$, $--$

Control Flow

- **Conditional:** `if`, `else`
- **Iterative:** `while`, `for`
- **Control:** `break`, `continue`

Functions

- Explicit definitions with named parameters and explicit return types.
- Pass-by-value parameter passing.
- Support for recursive calls and function hoisting.
- No function overloading.

Arrays

- Fixed-size arrays with homogeneous element types.
- Compile-time known array size.
- Index-based access.

Input and Output

- Output via built-in `print` function.
- Integer input via `readInt()` function.

TOOLS AND TECHNOLOGIES

- **Implementation Language:** Python
- **Parser Generator:** PLY (Python Lex-Yacc) or custom recursive-descent parser
- **Target Architectures:** x86-64 / RISC-V
- **Assemblers:** NASM (Netwide Assembler) or GNU Assembler (GAS)
- **Linker:** GNU ld or LLVM lld
- **Version Control:** Git/GitHub for collaborative development

WEEKLY MILESTONE PLAN

Week	Milestone & Deliverables	Outcome
1	**Language & Frontend Foundations** *Del:* NanoC specification, Token definitions, Functional Lexer.	Source successfully tokenized; Lexical correctness validated.
2	**Parsing & AST Construction** *Del:* Formal grammar (EBNF), Recursive-descent parser, AST node definitions.	Source code parsed into structured AST; Clear syntax error detection.
3	**Semantic Analysis** *Del:* Symbol table implementation, Scope and type checking.	Detection of undeclared variables and type mismatches.
4	**Intermediate Representation (IR)** *Del:* Machine-independent IR format, AST $\rightarrow$ IR translation.	High-level source code lowered to Linear IR.
5	**IR Validation** *Del:* Verification of IR correctness, Control-flow handling.	IR accurately represents program semantics; End-to-end frontend pipeline complete.
6	**Optimization Pass 1 (Constant Folding)** *Del:* Compile-time evaluation of constant expressions.	Reduced instruction count in IR.
7	**Optimization Pass 2 (Dead Code)** *Del:* Removal of unreachable and unused code.	Cleaner and more efficient IR.
8	**Optimization Pass 3 (Strength Reduction)** *Del:* Cheaper equivalent operations; Architecture-aware rules.	IR optimized for backend code generation.
9	**Backend: Assembly Generation** *Del:* IR $\rightarrow$ Assembly mapping, Register usage strategy.	Correct assembly (x86-64/RISC-V) generated from optimized IR.
10	**End-to-End Native Execution** *Del:* Integration with NASM/GAS, Full compilation pipeline.	Source code $\rightarrow$ Executable binary.
11	**Stabilization & Enhancement** *Del:* Bug fixes, Expanded test suite, Robustness.	Stable, reliable compiler pipeline.
12	**Documentation & Final Prep** *Del:* Final project report, Language manual.	Final-demo-ready project.

Note: Stretch goals (e.g., C++ transpiler backend, Loop Unrolling) will be attempted only after completing core deliverables in Week 12.