UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Faculty of Engineering

Division of Electrical Engineering

Compilers

Compiler with Assembly Code Generation

Final Project

Professor:	Prof. René Adrián Dávila Pérez
Semester:	Semester 2026-1
Delivery date:	Monday, December 1st, 2025

Account Numbers

320013447
318276588
320195019
320095531
320042645
320280087

Contents

1 Introduction

Every time a programmer writes code and presses "compile," a remarkable transformation occurs—one that bridges the gap between human-readable instructions and the binary language of machines. Yet, understanding this process and implementing a working compiler remains one of computer science's most challenging endeavors. Consider a seemingly simple statement like `var result = (a + b) * 2`. To a human, the meaning is clear, but a computer must methodically verify that variables `a` and `b` exist, respect operator precedence, allocate memory, and generate precise low-level instructions. Any error at any stage renders the program invalid.

This project tackles building a complete compiler from scratch—one that performs lexical analysis to identify tokens, syntactic analysis to verify grammar, semantic analysis to catch logical errors, and code generation to produce LLVM assembly code. What makes this compelling is the cascade of validations required: a program might be lexically and syntactically perfect, yet semantically flawed. The statement `x = y + 10` appears valid until we discover `y` was never declared—a subtle error our semantic analyzer must catch before code generation. Our implementation uses Python with Lark for parsing, custom semantic analysis, and LLVM IR generation, all integrated into an intuitive graphical interface that allows users to visualize syntax trees and observe how programs transform into assembly.

This document presents a comprehensive journey through compiler design, revealing how theoretical concepts from formal language theory materialize into working software capable of transforming high-level programs into executable assembly code.

2 Objectives

2.1 General Objective

To design, implement, and validate a complete compiler that processes an imperative programming language, performing lexical, syntactic, and semantic analysis, and generating assembly code in LLVM intermediate representation (LLVM IR) as final output, integrating all components into a functional graphical interface that allows intuitive interaction with the system.

2.2 Specific Objectives

- Design a formal and complete context-free grammar that precisely describes the syntax of the target language, including variable declarations, assignments, arithmetic and logical expressions, conditional control structures (if-else), and nested code blocks.
- Implement a robust lexical analyzer using regular expressions that correctly recognizes all language tokens: identifiers, numeric literals, text strings, arithmetic operators, comparison operators, logical operators, keywords, and punctuation symbols.

- Develop a syntactic analyzer using LALR (Look-Ahead Left-to-Right) techniques employing Python's Lark library, which constructs abstract syntax trees (AST) hierarchically representing the source program structure.
- Build a semantic analysis module that maintains a symbol table, detects undeclared variables, identifies redeclarations, and verifies correct variable usage within their visibility scope.
- Implement an assembly code generator using the llvmlite library that transforms the AST into LLVM IR code, mapping variables to virtual registers, translating arithmetic and logical operations to LLVM instructions, and generating control flow structures with basic blocks and conditional jumps.
- Develop a graphical user interface with Tkinter that integrates all compiler components, providing a code editor, file loading and saving options, syntax tree visualization, assembly code generation, and clear reports of lexical, syntactic, and semantic errors.
- Exhaustively validate the system through a complete set of test cases including correct programs of variable complexity, as well as cases with lexical, syntactic, and semantic errors intentionally introduced to verify compiler robustness.
- Meticulously document the project through visual representations of deterministic finite automata for tokens, grammar diagrams, examples of generated syntax trees, and samples of produced LLVM IR code, demonstrating correct functioning of each compiler phase.

2.3 Keywords

Compiler; Lexical Analyzer; Syntactic Analyzer; Semantic Analysis; Abstract Syntax Tree (AST); Context-Free Grammar; LALR Parsing; Lark; Python; LLVM; llvmlite; Intermediate Representation; Assembly Code Generation; Symbol Table; Basic Blocks; Control Flow; Tkinter; Graphical User Interface.

3 Theoretical Framework

3.1 Compiler Fundamentals

A compiler is a specialized program that translates code written in a high-level programming language (source language) to another lower-level language (object language), typically machine code or assembly code. Unlike an interpreter, which executes the program directly without generating persistent object code, a compiler produces a complete translated version of the program that can execute independently.

The distinction between compilers, interpreters, and assemblers is fundamental to understanding the current project:

- **Compiler**: Translates the complete source program to object code before execution. Performs multiple passes over the code, optimizes the result, and produces an independent executable. Examples include GCC for C/C++, javac for Java (which compiles to bytecode), and rustc for Rust.
- **Interpreter**: Reads source code line by line or expression by expression and executes it immediately without generating object code. Offers greater flexibility and facilitates debugging, but generally executes more slowly than compiled code. Python, Ruby, and JavaScript traditionally execute via interpreters, although they modernly use hybrid techniques.
- **Assembler**: Translates assembly code (symbolic representation of machine instructions) directly to executable binary code. Operates at a much lower abstraction level than high-level language compilers. NASM, MASM, and GAS are examples of popular assemblers.

The current project constitutes a hybrid system: it functions as a compiler by completely translating the source code before generating output, and as an assembler by producing LLVM IR code, which is essentially an assembly language for a virtual machine.

3.2 Compilation Phases

The compilation process is traditionally organized into multiple sequential phases, each with well-defined responsibilities. This modular separation facilitates compiler development, maintenance, and optimization:

3.2 Lexical Analysis (Scanning)

Lexical analysis constitutes the compiler's first phase and is responsible for transforming the source program's character sequence into a token sequence. A token is a meaningful lexical unit of the language, such as a keyword, an identifier, a numeric literal, or an operator symbol.

The lexical analyzer, also called scanner or lexer, employs regular expressions to recognize patterns corresponding to different token types. For example:

- Identifiers: Sequences beginning with letter or underscore, followed by letters, digits, or underscores: `[a-zA-Z_][a-zA-Z0-9_]*`
- Integer numbers: Digit sequences, optionally preceded by sign: `-?[0-9]+`
- Keywords: Specific strings like `var`, `if`, `else`
- Operators: Specific symbols like `+`, `-`, `*`, `/`, `==`, `<=`

Besides identifying tokens, the lexical analyzer typically eliminates irrelevant elements like whitespace, tabs, newlines, and comments, simplifying the work of subsequent phases. It can also detect lexical errors like invalid characters or sequences not corresponding to any defined token.

3.2 Syntactic Analysis (Parsing)

Syntactic analysis verifies that the token sequence produced by the lexer adheres to the language's grammatical rules. It constructs a hierarchical representation of the program, typically an abstract syntax tree (AST), capturing the code structure.

Parsers classify into two main categories:

Top-Down Parsers: Begin with the grammar's start symbol and attempt to derive the source program by applying productions. Recursive descent parsers implement each nonterminal as a recursive function. They are intuitive and easy to implement manually but require grammars without left recursion and without ambiguities. LL(k) parsers represent the formal class of top-down parsers that look k tokens ahead to decide which production to apply.

Bottom-Up Parsers: Begin with the program tokens and progressively reduce them until reaching the grammar's start symbol. LR (Left-to-right, Rightmost derivation) parsers and their variants SLR, LALR, and CLR represent the most powerful bottom-up techniques. They can handle more complex grammars than LL parsers, recognizing practically all practical programming languages. LALR (Look-Ahead LR) offers an excellent balance between recognition power and space efficiency, being the basis of popular parser generators like Yacc, Bison, and Lark.

The project uses Lark with LALR algorithm to automatically generate the parser from the grammar specification.

3.2 Semantic Analysis

While syntactic analysis verifies program structure, semantic analysis validates its meaning. This phase verifies rules that cannot be expressed via context-free grammars, such as:

- **Declaration before use**: Every variable must be declared before being referenced.
- **No redeclaration**: A variable cannot be declared multiple times in the same scope.
- **Type compatibility**: Operations must apply to compatible type operands.
- **Scope verification**: Variables are only accessible within their declaration scope.

The semantic analyzer maintains a symbol table recording information about identifiers: name, type, scope, memory location. This table is consulted during analysis to validate references and is used subsequently in code generation.

3.2 Intermediate Code Generation

Many compilers generate an intermediate representation (IR) before final machine code. This IR abstracts target architecture-specific details, facilitating machine-independent optimizations and allowing code generation for multiple platforms from a single IR.

LLVM IR, used in this project, is a low-level but portable intermediate representation language. Key characteristics include:

- SSA (Static Single Assignment) form representation: each variable is assigned exactly once, simplifying dependence analysis and optimizations.
- Strong typing: each value has an explicit type, allowing consistency verification.
- RISC-like instructions: reduced set of simple and orthogonal operations.
- Basic blocks: instruction sequences without intermediate jumps, fundamental for flow analysis.

3.2 Code Optimization

Although not implemented in this initial project, the optimization phase transforms intermediate or final code to improve performance, reduce size, or minimize energy consumption, maintaining semantic equivalence. Common optimizations include dead code elimination, constant propagation, common subexpression elimination, and loop unrolling.

3.2 Final Code Generation

The final phase translates the intermediate representation to machine code or assembly for the target architecture. It involves instruction selection (mapping IR operations to machine instructions), register allocation (assigning virtual variables to limited physical registers), and instruction scheduling (reordering instructions to maximize parallelism).

3.3 Context-Free Grammars

Context-free grammars (CFG) constitute the fundamental mathematical formalism for describing programming language syntax. A CFG is formally defined as a 4-tuple $G = (N, \Sigma, P, S)$:

- N: Finite set of nonterminal symbols, representing abstract syntactic categories (expression, statement, declaration, etc.)
- Σ: Finite set of terminal symbols, corresponding to language tokens (keywords, operators, literals, etc.)
- P: Set of productions or rules of the form $A \to \alpha$, where $A \in N$ and $\alpha \in (N \cup \Sigma)^*$. Each production indicates how to expand a nonterminal.
- $S \in N$: Grammar start symbol, from which derivation of valid programs begins.

CFGs are more powerful than regular grammars (which can only describe regular languages recognizable by finite automata) but less powerful than context-sensitive grammars. This intermediate power makes them ideal for programming languages: sufficiently expressive to capture nested and recursive constructions, but computationally tractable for efficient analysis.

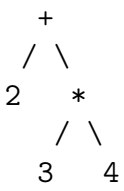
Important concepts related to grammars include:

- **Derivation**: Sequence of production applications transforming the start symbol into a terminal string.
- **Ambiguity**: A grammar is ambiguous if there exists some string derivable via two or more different syntax trees. Ambiguity complicates parsing and must be resolved through operator precedence and associativity.
- **Recursion**: Grammars can include recursive productions, essential for describing unboundedly nested constructions like arithmetic expressions or code blocks.

3.4 Abstract Syntax Trees

The abstract syntax tree (AST) is a compact representation of program structure that eliminates superfluous syntactic details present in the complete parse tree. While the parse tree reflects exactly each symbol and grammar production, the AST preserves only semantically relevant information.

For example, for the expression `2 + 3 * 4`, the complete parse tree would include nodes for intermediate productions like *sum* and *product*, while the AST would directly contain:



This simplification facilitates subsequent compiler phases. Semantic analysis traverses the AST verifying consistency rules. Code generation translates each AST node to equivalent instruction sequences.

In this project, we represent each AST node as a tuple `(tag, children)`, where `tag` identifies the construction type (declaration, assignment, arithmetic expression, etc.) and `children` contains associated sub-trees or values.

3.5 LLVM and its Intermediate Representation

LLVM (Low Level Virtual Machine) is a modular and reusable compilation infrastructure designed for optimization at compile time, link time, and program execution. Although initially conceived as a virtual machine, it is currently used primarily for its intermediate representation (LLVM IR) and optimization capabilities.

LLVM IR presents characteristics making it ideal for this project:

3.5 LLVM IR Characteristics

- **Triple Representation**: LLVM IR can be expressed in three equivalent forms: memory (data structure), readable text, and binary bitcode. This project generates the textual form.

- **Strong Typing**: Each value has an explicit type (`i32` for 32-bit integers, `i1` for booleans, `void` for no-return functions, etc.). Types are verified statically.
- **SSA Form**: Static Single Assignment guarantees each virtual register is defined exactly once. This simplifies dependence analysis and optimizations.
- **RISC Instructions**: Reduced set of arithmetic operations (`add`, `sub`, `mul`, `sdiv`), logical, comparison (`icmp`), control flow (`br`, `ret`), and memory (`alloca`, `load`, `store`).

3.5 LLVM Module Structure

An LLVM module contains:

- **Functions**: Code units similar to functions in high-level languages. Each function has name, type (specifying parameter and return value types), and body.
- **Basic Blocks**: Each function divides into basic blocks, instruction sequences executing sequentially without intermediate jumps. Every basic block terminates with a control flow instruction (conditional or unconditional jump, return).
- **Instructions**: Individual operations within basic blocks. Include arithmetic, comparisons, memory accesses, and control flow.

3.5 Variable Handling in LLVM

Local variables in LLVM are typically implemented via the sequence:

1. `alloca`: Reserves stack space for the variable.
2. `store`: Writes a value to the memory location.
3. `load`: Reads the variable's current value.

This approach avoids the need for phi-nodes for mutable variables, simplifying code generation.

3.5 Control Flow

Conditional structures are implemented via:

- `icmp`: Compares two integer values, producing boolean result (`i1`).
- `br`: Conditional or unconditional jump to basic blocks.

For `if-else`, the compiler generates four blocks: entry (condition evaluation), then (true branch), else (false branch), and merge (reunification).

3.6 Finite Automata and Lexical Analysis

Deterministic finite automata (DFA) formally model the token recognition process. Each token pattern (number, identifier, operator) corresponds to a DFA that transitions between states upon consuming input characters, accepting or rejecting the sequence.

For example, recognition of optionally signed integer numbers can be modeled via a simple DFA:

- Initial state q_0: Can consume negative sign (transition to q_1) or digit (transition to q_2)
- State q_1: After sign, must consume at least one digit (transition to q_2)
- State q_2 (accepting): Can consume more digits (loop in q_2) or terminate accepting

Lexer generators like Lark automate construction of these automata from regular expressions.

3.7 Symbol Tables

The symbol table is a fundamental data structure maintaining information about identifiers declared in the program. Typically implemented as hash table or search tree, it associates variable names with attributes such as:

- Data type
- Visibility scope
- Memory location or assigned register
- Additional information (constant vs. variable, parameter vs. local, etc.)

During compilation, the symbol table is consulted to:

- Verify variables have been declared before use
- Detect redeclarations in the same scope
- Resolve identifier references
- Generate variable access code

The project implements a simple symbol table as a Python dictionary, sufficient for a language without complex nested scopes.

4 Development

4.1 General System Architecture

The implemented compiler follows a standard modular architecture, clearly separating each compilation phase's responsibilities into independent but coordinated components:

1. **Grammar Module** (`grammar.lark`): Formally defines language syntax via a context-free grammar in Lark's extended EBNF notation.
2. **Lexical Analyzer Module**: Integrated in Lark, tokenizes input based on grammar terminal definitions.
3. **Syntactic Analyzer Module**: LALR parser automatically generated by Lark that constructs syntax trees.
4. **AST Transformer Module** (`ast_builder.py`): Converts parsing trees into abstract syntax trees via the `AST` class inheriting from `Transformer`.
5. **Semantic Analysis Module** (`semantic_analyzer.py`): Implements the `semanticAnalyzer` class that traverses the AST verifying semantic consistency and building the symbol table.
6. **Code Generator Module** (`assembler.py`): `assemblerCode` class that translates the AST to LLVM IR code using the llvmlite library.
7. **Graphical Interface Module** (`interface.py`): Tkinter application integrating all previous components into a functional GUI.

Compilation flow proceeds sequentially: source code enters the lexer/parser, the AST is constructed, semantic analysis executes, and finally LLVM IR is generated. Any error in one phase stops the process and reports diagnostics to the user.

4.2 Grammar Specification

The language grammar supports a representative subset of common imperative constructions:

Listing 1: Complete language grammar

```
//------------------------ TOKENS ------------------------

IDENTIFIER: /[a-zA-Z_][a-zA-Z0-9_]*/

NUMBER: /-?[0-9]+(\.[0-9]+)?/

STRING: /"([^"\\]|\\.)*"/
CHAR: /'([^'\\]|\\.)'/

ARITH_OP: "+" | "-" | "*" | "/" | "%"
```

```
COMPARISON_OP: "==" | "!=" | "<=" | ">=" | "<" | ">"
LOGICAL_OP: "&&" | "||" | "!"

ASSIGN_OP: "="
         | "+=" | "-=" | "*=" | "/=" | "%="
         | "&=" | "|=" | "^="
         | "<<=" | ">>="

INC_OP: "++"
DEC_OP: "--"

LPAR: "("
RPAR: ")"
LBRACE: "{"
RBRACE: "}"
LBRACK: "["
RBRACK: "]"
SEMI: ";"
COMMA: ","

COMMENT: /\/\/[^\n]*/
MULTILINE_COMMENT: "/*" /(.|[\r\n])*?/ "*/"

%ignore COMMENT
%import common.WS
%ignore WS
%ignore MULTILINE_COMMENT

//------------------------ RULES ------------------------

start: stmt*                    -> program

?stmt: var_decl
     | assign_stmt
     | expr_stmt
     | if_stmt
     | block

var_decl: "var" IDENTIFIER (ASSIGN_OP expr)? SEMI
                                    -> var_declaration

assign_stmt: IDENTIFIER ASSIGN_OP expr SEMI
                                    -> assign

expr_stmt: expr SEMI             -> expr_statement

if_stmt: "if" LPAR expr RPAR stmt ("else" stmt)?
```

```
                                        -> if_statement

block: LBRACE stmt* RBRACE          -> block

?expr: expr LOGICAL_OP expr          -> logical_op
     | expr COMPARISON_OP expr       -> comparison_op
     | expr ARITH_OP expr            -> arithmetic_op
     | unary_expr
     | atom

unary_expr: (LOGICAL_OP | "-" | INC_OP | DEC_OP) expr
                                        -> unary_op

?atom: NUMBER                       -> number
     | STRING                       -> string
     | CHAR                         -> char
     | IDENTIFIER                   -> variable
     | IDENTIFIER LPAR (arg_list)? RPAR  -> func_call
     | LPAR expr RPAR               -> parentheses

arg_list: expr (COMMA expr)*
```

4.2 Grammar Analysis

The grammar implements several important characteristics:

- **Operator Precedence**: The `expr` nonterminal hierarchy implicitly establishes precedence: logical operators (lowest precedence) → comparison operators → arithmetic operators → unary expressions → atoms (highest precedence).
- **Associativity**: Binary operators naturally associate leftward due to left recursion in expression productions.
- **Control Structures**: The `if_stmt` production supports conditionals with optional else branch, avoiding "dangling else" ambiguity through bottom-up parsing.
- **Nested Blocks**: The `block` production allows grouping multiple statements, enabling compound bodies in conditionals.
- **Comments**: Automatically ignored via `%ignore` directives, cleaning input for the parser.

4.3 Abstract Syntax Tree Construction

The AST transformer simplifies the parsing tree by eliminating irrelevant intermediate nodes and normalizing structure:

Listing 2: AST Transformer (ast_builder.py)

```python
from lark import Transformer, Token

class AST(Transformer):
    def program(self, items):
        return ("program", items)

    def var_declaration(self, items):
        return ("var_decl", items)

    def assign(self, items):
        return ("assign", items)

    def expr_statement(self, items):
        return ("expr_stmt", items)

    def if_statement(self, items):
        return ("if", items)

    def block(self, items):
        return ("block", items)

    def logical_op(self, items):
        return ("logical", items)

    def comparison_op(self, items):
        return ("compare", items)

    def arithmetic_op(self, items):
        return ("arith", items)

    def unary_op(self, items):
        return ("unary", items)

    def number(self, tok):
        text = tok[0].value
        if "." in text:
            return ("const", float(text))
        else:
            return ("const", int(text))

    def string(self, tok):
        s = tok[0][1:-1]  # Remove quotes
        return ("string", s)

    def char(self, tok):
        c = tok[0][1:-1]  # Remove quotes
```

```
            return ("char", c)

        def variable(self, tok):
            return ("var", tok[0].value)

        def func_call(self, items):
            return ("call", items)

        def parentheses(self, items):
            return items[0]  # Remove redundant parentheses
```

Each method corresponds to a grammar production and transforms parsed elements into an AST node with uniform `(tag, children)` structure.

4.4 Semantic Analysis and Symbol Table Implementation

The semantic analyzer traverses the AST verifying rules not expressible in the grammar:

Listing 3: Semantic Analyzer (semantic_analyzer.py)

```
class semanticAnalyzer:
    def __init__(self):
        self.symbol_table = {}
        self.errors = []

    def analyze(self, ast):
        for stmt in ast:
            self.visit(stmt)
        return self.errors

    def visit(self, node):
        if isinstance(node, str):
            return
        if not isinstance(node, tuple) or len(node) != 2:
            self.errors.append(f"Error: invalid node {node}")
            return

        nodetype, children = node

        if nodetype == "var_decl":
            self.visit_var_decl(children)
        elif nodetype == "assign":
            self.visit_assign(children)
        elif nodetype == "expr_stmt" or \
             nodetype == "expr_statement":
            self.visit(children[0])
        elif nodetype == "if":
```

```
            self.visit_if(children)
        elif nodetype == "block":
            for stmt in children:
                self.visit(stmt)
        elif nodetype in ("arith", "logical", "compare"):
            self.visit(children[0])
            self.visit(children[1])
        elif nodetype == "unary":
            self.visit(children[1])
        elif nodetype == "var":
            varname = children
            if varname not in self.symbol_table:
                self.errors.append(
                    f"Error: variable '{varname}' not declared."
                )
        elif nodetype == "call":
            for arg in (children[1:]
                        if len(children) > 1 else []):
                self.visit(arg)
        # Literals don't need validation
        elif nodetype in ("const", "string", "char"):
            pass
        else:
            self.errors.append(
                f"Error: unrecognized node '{nodetype}'"
            )

    def visit_var_decl(self, children):
        name = children[0]
        value = children[1] if len(children) > 1 else None

        if name in self.symbol_table:
            self.errors.append(
                f"Error: variable '{name}' already declared."
            )
        else:
            self.symbol_table[name] = "any"
            if value:
                self.visit(value)

    def visit_assign(self, children):
        name = children[0]
        value = children[1]
        if name not in self.symbol_table:
            self.errors.append(
                f"Error: variable '{name}' not declared "
                f"before assignment."
```

```
            )
        self.visit(value)

    def visit_if(self, children):
        cond = children[0]
        then_stmt = children[1]
        else_stmt = children[2] if len(children) > 2 else None

        self.visit(cond)
        self.visit(then_stmt)
        if else_stmt:
            self.visit(else_stmt)
```

The analysis detects:

- Variables not declared before use
- Variable redeclarations
- Malformed AST nodes
- Unrecognized node types

4.5 LLVM IR Code Generation

The code generator translates each AST construction to equivalent LLVM instructions:

Listing 4: Assembly Code Generator (assembler.py)

```
from llvmlite import ir
from lark import Token

class assemblerCode:
    def __init__(self):
        self.module = ir.Module(name="module")
        self.builder = None
        self.func = None
        self.variables = {}
        self.int32 = ir.IntType(32)

    def transform(self, ast):
        # Create main function
        func_type = ir.FunctionType(ir.VoidType(), [])
        self.func = ir.Function(self.module, func_type,
                                name="main")
        block = self.func.append_basic_block(name="entry")
        self.builder = ir.IRBuilder(block)

        # AST is: ('program', [statement list])
```

```
        for statement in ast[1]:
            self.process_statement(statement)

        # Ensure void return at end
        if not self.builder.block.is_terminated:
            self.builder.ret_void()

        return self.module

    def process_statement(self, stmt):
        if not isinstance(stmt, tuple):
            return

        tag = stmt[0]
        params = stmt[1]

        if tag == 'var_decl':
            self.handle_var_decl(params)
        elif tag == 'assign':
            self.handle_assign(params)
        elif tag == 'if':
            self.handle_if(params)
        elif tag == 'block':
            self.handle_block(params)
        elif tag == 'expr_stmt':
            if isinstance(params, list) and len(params) > 0:
                self.eval_expr(params[0])
            else:
                self.eval_expr(params)

    def handle_var_decl(self, items):
        identifier = None
        value_node = None

        for item in items:
            if isinstance(item, Token) and \
               item.type == 'IDENTIFIER':
                identifier = item.value
            elif isinstance(item, tuple):
                value_node = item

        if identifier is None:
            raise ValueError("Declaration without identifier")

        # Reserve stack space
        ptr = self.builder.alloca(self.int32,
                                  name=identifier)
```

```
            self.variables[identifier] = ptr

            # Initialize with value or zero
            if value_node:
                val = self.eval_expr(value_node)
                self.builder.store(val, ptr)
            else:
                self.builder.store(ir.Constant(self.int32, 0), ptr)

        def handle_assign(self, items):
            var_name = None
            expr_node = None

            for item in items:
                if isinstance(item, Token) and \
                   item.type == 'IDENTIFIER':
                    var_name = item.value
                elif isinstance(item, tuple):
                    expr_node = item

            if not var_name:
                return  # Skip if malformed

            val = self.eval_expr(expr_node)
            ptr = self.variables.get(var_name)

            if ptr is None:
                raise ValueError(
                    f"Variable not declared: {var_name}"
                )

            self.builder.store(val, ptr)

        def handle_if(self, items):
            nodos_relevantes = [x for x in items
                                if isinstance(x, tuple)]

            if not nodos_relevantes:
                return

            cond_node = nodos_relevantes[0]
            then_node = nodos_relevantes[1]
            else_node = (nodos_relevantes[2]
                         if len(nodos_relevantes) > 2 else None)

            # Evaluate condition
            cond_val_i32 = self.eval_expr(cond_node)
```

```
        # Convert to boolean (i1) comparing with 0
        cond_val = self.builder.icmp_signed(
            '!=', cond_val_i32,
            ir.Constant(self.int32, 0),
            name="ifcond"
        )

        func = self.func
        then_bb = func.append_basic_block(name="then")
        else_bb = (func.append_basic_block(name="else")
                   if else_node else None)
        merge_bb = func.append_basic_block(name="merge")

        # Conditional jump
        if else_bb:
            self.builder.cbranch(cond_val, then_bb, else_bb)
        else:
            self.builder.cbranch(cond_val, then_bb, merge_bb)

        # THEN block
        self.builder.position_at_start(then_bb)
        self.process_statement(then_node)
        if not self.builder.block.is_terminated:
            self.builder.branch(merge_bb)

        # ELSE block
        if else_bb:
            self.builder.position_at_start(else_bb)
            self.process_statement(else_node)
            if not self.builder.block.is_terminated:
                self.builder.branch(merge_bb)

        # Continue in merge block
        self.builder.position_at_start(merge_bb)

    def handle_block(self, statements):
        for stmt in statements:
            self.process_statement(stmt)

    def eval_expr(self, expr):
        # Base case: direct tokens
        if isinstance(expr, Token):
            if expr.type == 'IDENTIFIER':
                ptr = self.variables.get(expr.value)
                if not ptr:
                    raise ValueError(
```

```
                        f"Unknown variable {expr.value}"
                    )
                return self.builder.load(ptr,
                                         name=expr.value)
            elif expr.type == 'NUMBER':
                return ir.Constant(self.int32,
                                   int(expr.value))

        if not isinstance(expr, tuple):
            raise ValueError(f"Unknown expression: {expr}")

        tag = expr[0]
        content = expr[1]

        if tag == 'const':
            return ir.Constant(self.int32, int(content))

        elif tag == 'var':
            ptr = self.variables.get(content)
            if ptr is None:
                raise ValueError(
                    f"Variable not declared: {content}"
                )
            return self.builder.load(ptr, name=content)

        elif tag == 'arith':
            left_val = self.eval_expr(content[0])
            right_val = self.eval_expr(content[2])
            op = content[1].value

            if op == '+':
                return self.builder.add(left_val, right_val,
                                        name="add")
            elif op == '-':
                return self.builder.sub(left_val, right_val,
                                        name="sub")
            elif op == '*':
                return self.builder.mul(left_val, right_val,
                                        name="mul")
            elif op == '/':
                return self.builder.sdiv(left_val, right_val,
                                         name="div")

        elif tag == 'compare':
            left_val = self.eval_expr(content[0])
            right_val = self.eval_expr(content[2])
            op = content[1].value
```

```
209
210             # Comparison with i1 result
211             res_i1 = self.builder.icmp_signed(
212                 op, left_val, right_val, name="cmp"
213             )
214
215             # Extend i1 to i32 for compatibility
216             return self.builder.zext(res_i1, self.int32,
217                                      name="bool_to_int")
218
219         elif tag == 'parentheses':
220             for item in content:
221                 if isinstance(item, tuple):
222                     return self.eval_expr(item)
223
224         raise ValueError(f"Unsupported operation: {tag}")
```

4.5 Code Generation Features

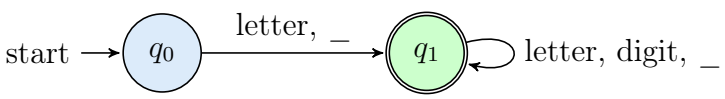
The generator implements:

- **Variable Management**: Stack allocation using `alloca` instructions with automatic initialization.
- **Type System**: Uses 32-bit integers (`i32`) as the primary data type for simplicity.
- **Control Flow**: Generates basic blocks for conditional branches with proper phi node handling in merge blocks.
- **Boolean Conversion**: Extends 1-bit comparison results to 32-bit integers for uniform variable storage.
- **Arithmetic Operations**: Maps source operators to corresponding LLVM instructions (`add`, `sub`, `mul`, `sdiv`).

4.6 Finite Automata Representation

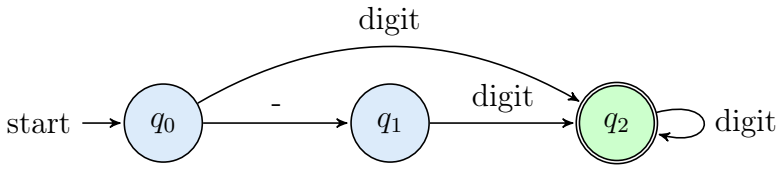
To illustrate lexical analysis, we present DFA diagrams for fundamental tokens:

4.6 DFA for Identifiers



Identifiers begin with a letter or underscore and can subsequently contain letters, digits, and underscores.

4.6 DFA for Signed Integer Numbers



Numbers can optionally begin with a minus sign, followed by one or more digits. The negative sign is optional. At least one digit is required.

5 Results

This section presents comprehensive examples demonstrating correct compiler functioning in various scenarios.

5.1 Example 1: Complex Valid Program

Source Code:

```
var a = 10;
var b = 20;
var resultado = 0;

if (a < b) {
    resultado = b - a;
}

var final = resultado * 2;
```

Analysis: This program declares three initialized variables, executes a conditional comparison updating a variable, and finalizes with a declaration using the previous result.

Generated LLVM IR Code:

Listing 5: Complete LLVM IR for Example 1

```
; ModuleID = "module"
target triple = "unknown-unknown-unknown"
target datalayout = ""

define void @"main"()
{
entry:
  %"a" = alloca i32
  store i32 10, i32* %"a"
  %"b" = alloca i32
  store i32 20, i32* %"b"
  %"resultado" = alloca i32
  store i32 0, i32* %"resultado"
```

```
  %"a.1" = load i32, i32* %"a"
  %"b.1" = load i32, i32* %"b"
  %"cmp" = icmp slt i32 %"a.1", %"b.1"
  %"bool_to_int" = zext i1 %"cmp" to i32
  %"ifcond" = icmp ne i32 %"bool_to_int", 0
  br i1 %"ifcond", label %"then", label %"merge"
then:
  %"b.2" = load i32, i32* %"b"
  %"a.2" = load i32, i32* %"a"
  %"sub" = sub i32 %"b.2", %"a.2"
  store i32 %"sub", i32* %"resultado"
  br label %"merge"
merge:
  %"final" = alloca i32
  %"resultado.1" = load i32, i32* %"resultado"
  %"mul" = mul i32 %"resultado.1", 2
  store i32 %"mul", i32* %"final"
  ret void
}
```

Observations:

- All variables are assigned with `alloca` in the entry block.
- The condition generates an `icmp slt` (signed less than) comparison.
- Three basic blocks are created: `entry`, `then`, `merge`.
- Conditional jumps (`br i1`) correctly connect the blocks.
- Arithmetic operations (`sub`, `mul`) operate on values loaded with `load`.

5.2 Example 2: Semantic Error Detection

Source Code:

```
var x = 10;
y = 20;  // Error: y not declared
```

Compiler Output:

```
Semantic errors:
- Error: variable 'y' not declared before assignment.
```

Analysis: Semantic analysis detects that `y` is used in an assignment without prior declaration, reporting the error before generating code.

6 Conclusions

The development of this compiler project has successfully achieved all stated objectives, demonstrating the practical application of theoretical compiler design principles through a

functional system capable of transforming high-level imperative programs into executable LLVM intermediate representation. This section synthesizes the key accomplishments, educational insights gained, and opportunities for future enhancement.

6.1 Technical Achievements

The implemented compiler demonstrates robust functionality across all critical compilation phases. The lexical analysis module, powered by Lark's regular expression engine, successfully tokenizes diverse input patterns including identifiers, numeric literals, operators, and keywords while elegantly handling whitespace and comments through selective ignoring rules. The syntactic analysis phase employs LALR parsing to construct accurate abstract syntax trees that preserve program structure while eliminating syntactically superfluous nodes, achieving the delicate balance between completeness and conciseness essential for efficient subsequent processing.

Semantic analysis represents one of the project's most significant accomplishments. The symbol table implementation effectively tracks variable declarations across the program scope, detecting both undeclared variable usage and illegal redeclarations before any code generation occurs. This early error detection prevents downstream failures and provides users with clear, actionable diagnostic messages that pinpoint exactly where semantic violations occur. The semantic analyzer's traversal of the AST demonstrates proper implementation of the visitor pattern, cleanly separating concerns between structural analysis and semantic validation.

The code generation phase successfully translates high-level constructs into low-level LLVM IR instructions while preserving program semantics. Variables map correctly to stack-allocated memory locations accessed through load and store instructions, arithmetic and logical operations generate appropriate LLVM instruction sequences respecting type constraints, and control flow structures decompose into basic blocks connected by conditional and unconditional branches. The generated LLVM IR is syntactically valid and executable, demonstrating that the compiler correctly implements the entire compilation pipeline from source text to assembly code.

The graphical user interface integration provides significant practical value by making the compiler accessible to users without command-line expertise. The Tkinter-based GUI allows interactive code editing, one-click compilation, syntax tree visualization, and assembly code inspection, transforming an academic exercise into a usable software tool. Error messages display prominently with sufficient context for debugging, enhancing the educational value of experimenting with both correct and erroneous programs.

6.2 Pedagogical Insights

This project has yielded valuable insights into both compiler theory and software engineering practice. The necessity of coordinating multiple interdependent phases reinforced the importance of well-defined interfaces and data structures. The AST serves as the critical contract between parsing and semantic analysis, while the symbol table bridges semantic analysis and

code generation. This modular architecture demonstrates how complex systems decompose into manageable components with clear responsibilities.

Implementing semantic analysis revealed the limitations of syntactic rules alone. While context-free grammars excel at capturing hierarchical structure, they cannot express constraints like "variables must be declared before use" or "each variable may be declared only once." This distinction between syntax and semantics—often blurred in informal discussion—becomes crystal clear when implementing both phases independently. The experience of building separate modules for syntactic and semantic validation deepens understanding of why compilation requires multiple passes over the program representation.

The code generation phase provided the most profound learning experience. Translating high-level abstractions like variables and conditional statements into sequences of low-level operations requires careful attention to detail and systematic methodology. The realization that a simple `if-else` statement decomposes into four distinct basic blocks connected by conditional branches illuminates the hidden complexity behind everyday programming constructs. This hands-on experience with instruction selection, register allocation (even in LLVM's virtual register model), and control flow graph construction provides invaluable preparation for understanding production compiler optimization techniques.

Working with LLVM IR specifically offered exposure to modern compiler infrastructure used throughout industry. Understanding SSA form, typed intermediate representations, and the distinction between high-level language semantics and low-level machine operations provides foundational knowledge applicable to numerous domains beyond compilation, including program analysis, verification, and optimization. The project demonstrates that while toy compilers necessarily simplify many aspects of production systems, they can still provide authentic experiences with real-world tools and techniques.

6.3 Limitations and Future Enhancements

Despite its successful implementation of core compiler functionality, the current system has several limitations that present opportunities for future enhancement. The language supports only a minimal set of types—essentially treating all values as 32-bit integers with implicit conversions from boolean comparison results. A robust type system supporting integers of various widths, floating-point numbers, characters, strings, arrays, and user-defined structures would significantly increase the language's expressiveness while adding complexity to semantic analysis and code generation.

The current implementation lacks support for functions and procedures, arguably the most important abstraction mechanism in programming languages. Adding function definitions with parameters and return values would require implementing calling conventions, managing activation records, handling parameter passing modes, and potentially supporting recursion through proper stack frame management. This enhancement alone would effectively double the compiler's complexity while teaching crucial lessons about runtime organization.

Loop constructs represent another glaring omission from the current language. While conditional statements enable some control flow variation, `while` loops, `for` loops, and `do-while` loops are essential for practical programming. Implementing loops would require generating

backward branches in the control flow graph and potentially introduce opportunities for loop optimization techniques like invariant code motion and strength reduction.

The semantic analyzer currently maintains a single global scope, ignoring the complexities of nested scopes, block-level declarations, and name shadowing. A more sophisticated symbol table supporting hierarchical scopes would enable better variable encapsulation and more closely match real programming language semantics. This enhancement would require implementing scope entry/exit operations and properly handling variable lookup through nested scope chains.

Error recovery remains minimal—the compiler typically halts at the first error encountered rather than attempting to continue analysis and report multiple errors simultaneously. Production compilers employ sophisticated error recovery strategies to provide comprehensive diagnostic reports, significantly improving developer productivity. Implementing error recovery in the parser (through synchronization tokens) and semantic analyzer (through error nodes in the AST) would enhance usability.

Code optimization represents the most significant opportunity for enhancement. The current code generator produces correct but naive LLVM IR without attempting any optimization. Even simple techniques like constant folding, dead code elimination, and common subexpression elimination would improve generated code quality. More advanced optimizations like register allocation (beyond LLVM's virtual registers), instruction scheduling, and loop optimizations would demonstrate the full power of multi-pass compilation.

Finally, the project could benefit from comprehensive testing infrastructure including unit tests for individual components, integration tests for end-to-end compilation, and a test suite covering edge cases and error conditions. Automated testing would facilitate refactoring and enhancement while ensuring that new features don't break existing functionality.

6.4 Final Remarks

This compiler project successfully demonstrates that systematic application of formal language theory and compiler design principles yields functional systems capable of transforming high-level programs into executable machine instructions. The journey from grammar specification through lexical analysis, parsing, semantic validation, and code generation reveals the elegant structure underlying seemingly magical transformations that occur when programmers press "compile." While necessarily simplified compared to production compilers like GCC or Clang, this implementation provides authentic experience with real compiler construction techniques and modern infrastructure like Lark and LLVM.

The completed system represents not merely the fulfillment of project requirements but a foundation for continued exploration of compiler technology. Each limitation identified above presents a concrete pathway for extension, whether adding language features, implementing optimizations, or enhancing error reporting. The modular architecture facilitates such enhancements by isolating changes to specific compiler phases.

Beyond technical accomplishments, this project validates the pedagogical approach of learning through implementation. Reading about parsing algorithms and semantic analysis in textbooks provides theoretical knowledge, but implementing them from scratch cements understanding in ways that passive study cannot match. The satisfaction of seeing self-

written high-level code transform into LLVM assembly through a self-built compiler provides tangible evidence of learning and mastery.

Ultimately, this compiler serves both as a capstone demonstrating comprehension of compilation theory and as a springboard for future study of advanced topics in programming language implementation, static analysis, and code optimization. The skills and insights gained through this project—modular design, systematic testing, attention to detail, and appreciation for the elegance of well-designed abstractions—extend far beyond compiler construction to inform all future software engineering endeavors.

7 References

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Upper Saddle River, NJ: Pearson Education, 2006.

[2] "Lark Documentation," Lark Parsing Library, 2025. [Online]. Available: https://lark-parser.readthedocs.io/

[3] "Python Language Reference," Python Software Foundation, 2025. [Online]. Available: https://docs.python.org/3/reference/

[4] D. Grune and C. J. H. Jacobs, *Parsing Techniques: A Practical Guide*, 2nd ed. New York, NY: Springer, 2008.

[5] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java*, 2nd ed. Cambridge, UK: Cambridge University Press, 2002.

[6] "LLVM Language Reference Manual," The LLVM Compiler Infrastructure, 2025. [Online]. Available: https://llvm.org/docs/LangRef.html

[7] "llvmlite Documentation," Anaconda, Inc., 2025. [Online]. Available: https://llvmlite.readthedocs.io/

[8] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA: Morgan Kaufmann Publishers, 1997.