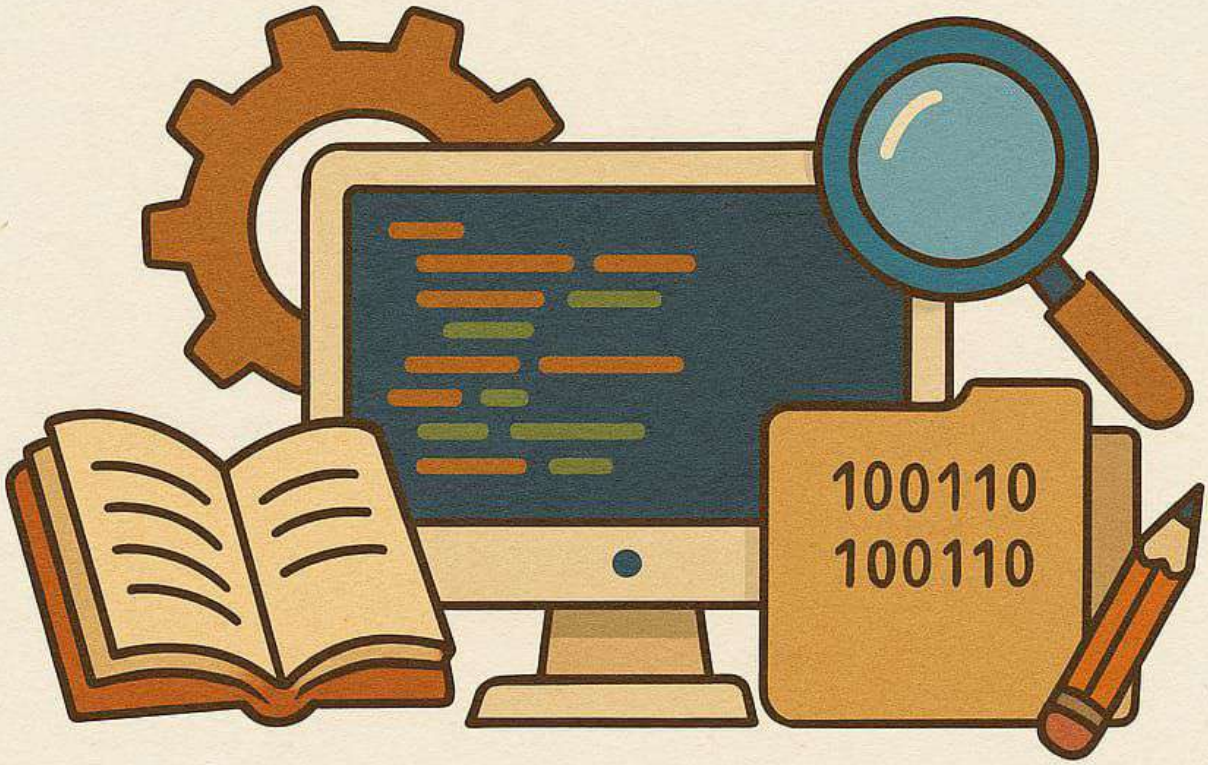
COMPILER DESIGN



Members:
Abhinandan Barua
Shahir Habib
Masud Alam
Sanket Chakraborty

Abstract

This report details the development of a compiler for a subset of the C programming language. The project encompasses the implementation of key compiler components: lexical analysis, syntax analysis using LR(1) parsing, symbol table management, and the generation of intermediate representations. The report outlines the methodologies employed, the structure of the implementation, challenges encountered, and potential areas for future enhancement.

Table of Contents

Introduction

Compilers are essential tools that translate high-level programming languages into machine code, enabling program execution on hardware. This project focuses on constructing a compiler for a subset of the C language, implementing core components to understand the intricacies of compiler design and construction.

Project Objectives

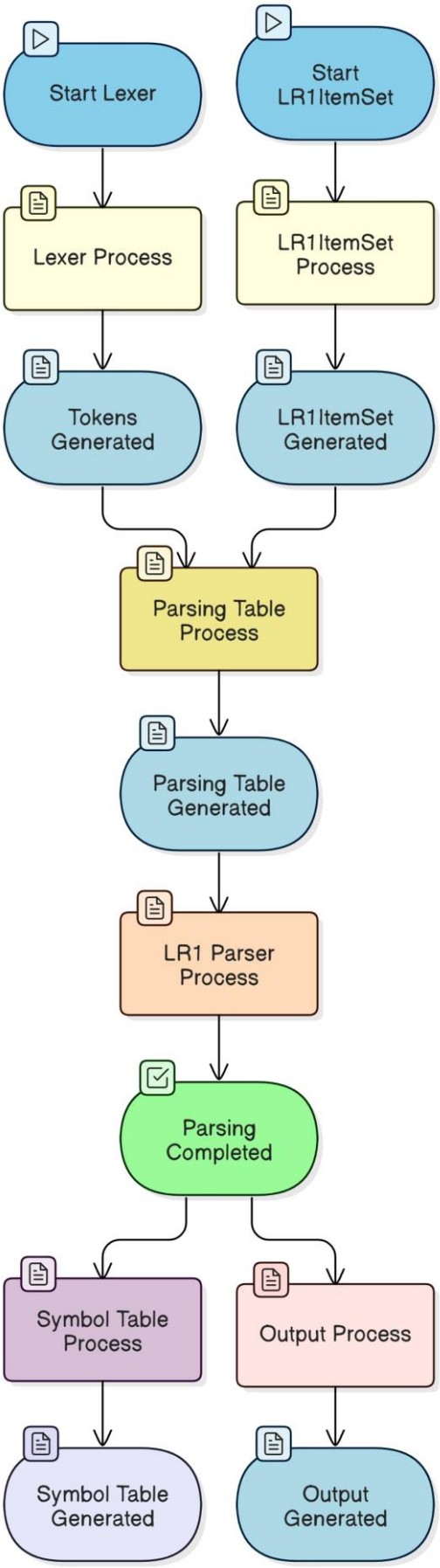
The primary objectives of this project are:

- To develop a functional compiler capable of processing a subset of the C language.
- To implement and integrate fundamental compiler components: lexical analysis, syntax analysis using LR(1) parsing, symbol table management, and intermediate representation generation.
- To gain practical experience with compiler construction techniques and tools.

Language Specification

- **Data Types: int, float, void**
- **Input/Output: read, print**
- **Operators: +, -, *, /, %, ++**
- **Conditionals: if, else (no nesting)**
- **Relational Operators: <,>,==**
- **Multiple functions allowed**
- **Declarations can appear anywhere with proper scope handling**

Flow Diagram:



Implementation Details

The compiler development is structured into four main components: lexical analysis, syntax analysis, symbol table management, and Parsing.

File Name	Description
`lexer.cpp`	Performs lexical analysis and generates tokens
`program.txt`	Sample input C-like code
`tokens.txt`	Output of lexer containing tokens
`symbolTable.cpp`	Manages variables, types, scopes
`symbol_table.txt`	Stores final symbol table output
`cfg.txt`	Context-Free Grammar definition
`lr1itemset.cpp`	Constructs LR(1) item sets from grammar
`lr1itemsets.txt`	Output of LR(1) item sets
`parsingTable.cpp`	Generates LR(1) parsing table
`parsingtable.txt`	Stores action/goto parsing table
`lr1parser.cpp`	Uses parsing table to parse token stream
`parsingresult.txt`	Output showing successful parsing or error

Lexical Analysis

Objective: To scan the source code and convert it into a stream of tokens.

Implementation:

- Lexer Development (lexer.cpp): The lexer reads the source code from program.txt, identifies lexemes, and classifies them into tokens such as keywords, identifiers, literals, operators, and punctuation. The resulting tokens are output to tokens.txt.

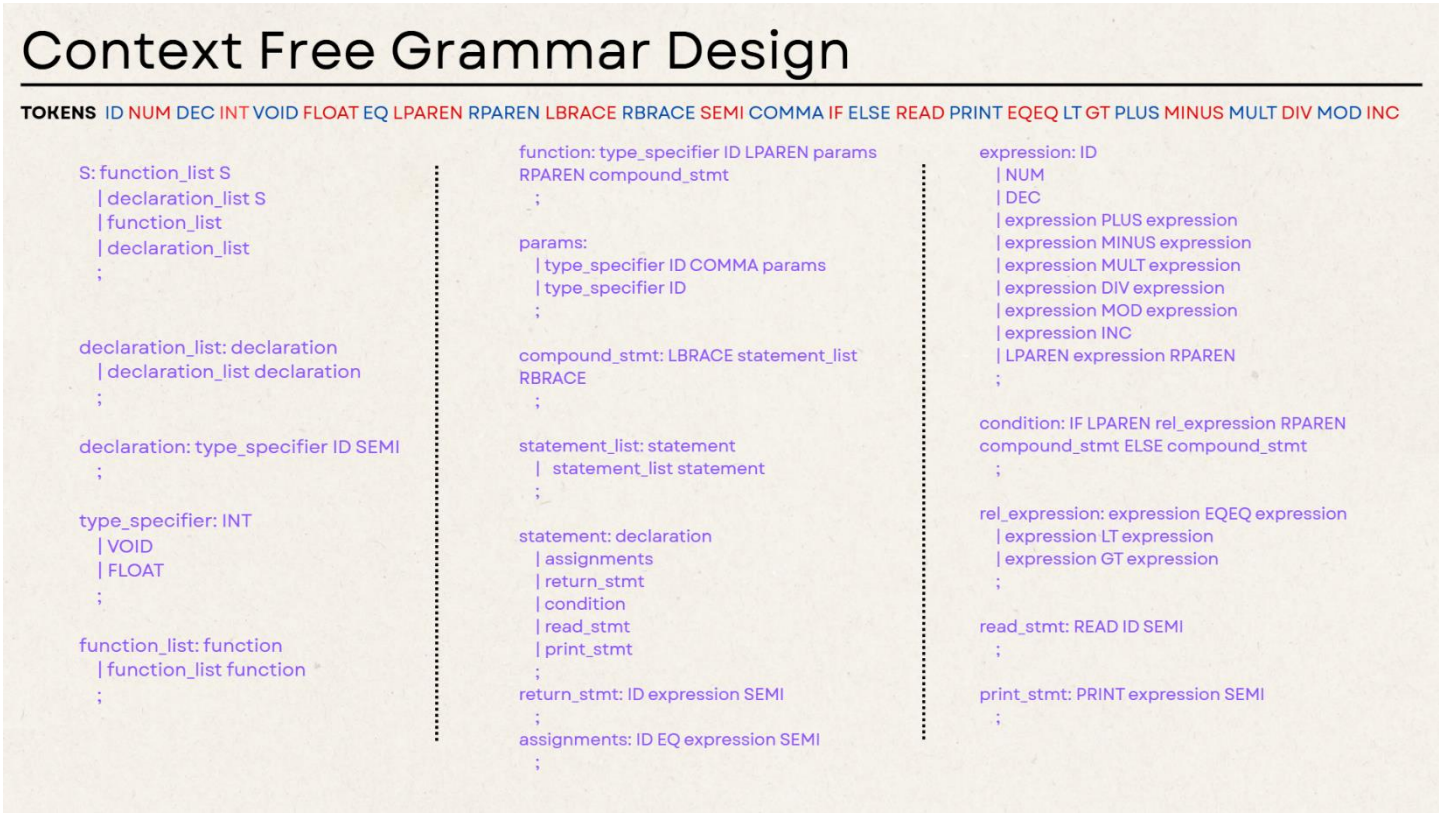
Challenges:

- Accurately distinguishing between similar lexemes (e.g., distinguishing between the keyword if and an identifier ifelse).
- Efficiently handling white spaces and comments.

Solutions:

- Implemented a longest match rule to resolve ambiguities between similar lexemes.
- Designed the lexer to ignore white spaces and comments, ensuring they do not interfere with tokenization.

Context-Free Grammar (CFG):



- Start Symbol: S defines a program that can consist of multiple declarations and functions.
- Declaration and Function Blocks:
  - *declaration_list* represents multiple variable declarations.
  - *function_list* allows defining one or more functions.
  - function has structure: return type + function name + parameters + compound statement.
- Parameter Parsing:

    - *params* allows functions to have typed parameters or be empty.
- Compound Statement:
    - Block enclosed in {} that holds a list of valid statements.
- **Statements include**:
    - declaration, assignments, return_stmt, condition, read_stmt, print_stmt
- **Expressions**:
    - Support arithmetic: +, -, *, /, %, and ++
    - Can include variables, numbers, or nested expressions
- **Conditionals**:
    - Format: if (rel_expression) { ... } else { ... }
    - No nested if-else support
- **Relational Expressions**:
    - Comparisons using ==, <, and >
- **I/O Statements**:
    - read_stmt and print_stmt use READ and PRINT tokens respectively

LR(1) Item Set Generation:

1.Reads the CFG.

2.Constructs LR(1) item sets using closure and goto functions.

3.Implements FIRST and FOLLOW computation.

4.Generates lr1itemsets.txt.

```
INC
_____P R O D U C T I O N S _____
0 S' -> S
1 S -> function_list S
2 S -> declaration_list S
3 S -> function_list
4 S -> declaration_list
5 declaration_list -> declaration
6 declaration_list -> declaration_list declaration
7 declaration -> type_specifier ID SEMI
8 type_specifier -> INT
9 type_specifier -> VOID
10 type_specifier -> FLOAT
11 function_list -> function
12 function_list -> function_list function
13 function -> type_specifier ID LPAREN params RPAREN compound_stmt
14 params ->
15 params -> type_specifier ID COMMA params
16 params -> type_specifier ID
17 compound_stmt -> LBRACE statement_list RBRACE
18 statement_list -> statement
19 statement_list -> statement_list statement
20 statement -> declaration
21 statement -> assignments
22 statement -> return_stmt
```

Follow:

```
138     S : $
139     S' :
140     assignments : FLOAT ID IF INT PRINT RBRACE READ VOID
141     compound_stmt : $ ELSE FLOAT ID IF INT PRINT RBRACE READ VOID
142     condition : FLOAT ID IF INT PRINT RBRACE READ VOID
143     declaration : $ FLOAT ID IF INT PRINT RBRACE READ VOID
144     declaration_list : $ FLOAT INT VOID
145     expression : DIV EQEQ GT INC LT MINUS MOD MULT PLUS RPAREN SEMI
146     function : $ FLOAT INT VOID
147     function_list : $ FLOAT INT VOID
148     params : RPAREN
149     print_stmt : FLOAT ID IF INT PRINT RBRACE READ VOID
150     read_stmt : FLOAT ID IF INT PRINT RBRACE READ VOID
151     rel_expression : RPAREN
152     return_stmt : FLOAT ID IF INT PRINT RBRACE READ VOID
153     statement : FLOAT ID IF INT PRINT RBRACE READ VOID
154     statement_list : FLOAT ID IF INT PRINT RBRACE READ VOID
155     type_specifier : ID
```

First:

```
117     S : FLOAT INT VOID
118     S' : FLOAT INT VOID
119     SEMI : SEMI
120     VOID : VOID
121     assignments : ID
122     compound_stmt : LBRACE
123     condition : IF
124     declaration : FLOAT INT VOID
125     declaration_list : FLOAT INT VOID
126     expression : DEC ID LPAREN NUM
127     function : FLOAT INT VOID
128     function_list : FLOAT INT VOID
129     params : FLOAT INT VOID epsilon
130     print_stmt : PRINT
131     read_stmt : READ
132     rel_expression : DEC ID LPAREN NUM
133     return_stmt : ID
134     statement : FLOAT ID IF INT PRINT READ VOID
135     statement_list : FLOAT ID IF INT PRINT READ VOID
136     type_specifier : FLOAT INT VOID
```

Parsing Table Generation:

1. Uses the LR(1) item sets to compute the parsing table.

2.Determines shift/reduce actions and goto transitions.

3.Outputs parsingtable.txt.

LR(1) Parsing Table

MINUS	MULT	DIV	MOD	INC	$	S	declaration_list	dec
						3	6	
					Accept			
					r5			
					r4	10	6	
					r11			
					r3	13	6	

LR(1) Parsing:

1.Reads tokens.txt and parsingtable.txt.

2.Simulates the parsing stack.

3.Logs reductions, matches, and errors.

```
    Lookahead : FLOAT
                        | Action : s1
    Shift  : FLOAT
    Lookahead : ID
                        | Action : r10
    Reduce by type_specifier -> FLOAT
    Lookahead : ID
                        | Action : s16
    Shift  : ID
    Lookahead : LPAREN
                        | Action : s19
    Shift  : LPAREN
    Lookahead : RPAREN
                        | Action : r14
    Reduce by params ->
    Lookahead : RPAREN
                        | Action : s23
    Shift  : RPAREN
    Lookahead : LBRACE
                        | Action : s25
    Shift  : LBRACE
    Lookahead : ID
                        | Action : s28
    Shift  : ID
    Lookahead : DEC
```

```
                        | Action : r19
    Reduce by statement_list -> statement_list statement
    Lookahead : RBRACE
                        | Action : s51
    Shift  : RBRACE
    Lookahead : $
                        | Action : r17
    Reduce by compound_stmt -> LBRACE statement_list RBRACE
    Lookahead : $
                        | Action : r13
    Reduce by function -> type_specifier ID LPAREN params RPAREN compound_stmt
    Lookahead : $
                        | Action : r11
    Reduce by function_list -> function
    Lookahead : $
                        | Action : r3
    Reduce by S -> function_list
    Lookahead : $
                        | Action : r2
    Reduce by S -> declaration_list S
    Lookahead : $
                        | Action : r1
    Reduce by S -> function_list S
    Lookahead : $
                        | Accepted 👌 👌 👌 👌
```

Symbol Table:

symbolTable.cpp

- Creates a symbol table during parsing.
- Stores variable names, types, and scope.

```
+-------------+--------+---------+------+-----+-------+------------------+
| Token Type  | Name   |  Value  | Line | Pos | Scope |   Memory Addr    |
+-------------+--------+---------+------+-----+-------+------------------+
| FLOAT       | fun    |         | 1    | 0   | 0     | 0x1000           |
+-------------+--------+---------+------+-----+-------+------------------+
| Referenced at:                                                          |
|                                                                         |
|    → Line 1   , Pos 6   , Scope 0                                       |
+-------------+--------+---------+------+-----+-------+------------------+
| INT         | x      |         | 3    | 4   | 1     | 0x1008           |
+-------------+--------+---------+------+-----+-------+------------------+
| Referenced at:                                                          |
|                                                                         |
|    → Line 3   , Pos 8   , Scope 1                                       |
+-------------+--------+---------+------+-----+-------+------------------+
| INT         | x      | 55      | 7    | 5   | 0     | 0x100c           |
+-------------+--------+---------+------+-----+-------+------------------+
| Referenced at:                                                          |
|                                                                         |
|    → Line 5   , Pos 4   , Scope 0                                       |
|    → Line 7   , Pos 4   , Scope 2                                       |
+-------------+--------+---------+------+-----+-------+------------------+
| INT         | main   |         | 6    | 0   | 0     | 0x1010           |
```

Conclusion:

The project demonstrates the end-to-end design of a simplified compiler. It uses LR(1) parsing, proper tokenization, and symbol table management. The modular design enables future extensions like semantic analysis and code generation.