Green University of Bangladesh

Department of Computer Science and Engineering (CSE) Semester: (Spring, Year: 2025), B.Sc. in CSE (Day)

"The Agile Compiler Simulator"

Course Title : Compiler Lab
Course Code : CSE-306 Section : 223_D2

Students Details

Name	ID
Asfiqul Alam Emran	231902049
Abdullah Al Baki Ruhin	231902057
Fardin Islam	231902039

Submission Date:12-05-2025

Course Teacher's Name: Kazi Hasnayeen Emad

[For teachers use only: Don't write anything inside this box]

Lab	Project	Status

Marks:	**Signature:**
Comments:	**Date:**

Contents

Chapter 1

Introduction

1.1 Overview

The Agile Compiler Simulator is a software tool designed to emulate the process of code compilation in a dynamic and interactive environment. It provides users, particularly students and developers, with a hands-on platform to understand the various stages of compilation, including lexical analysis, syntax analysis, semantic analysis, and code generation. Built with agility and user engagement in mind, the simulator enhances learning through visualization and real-time feedback.

1.2 Motivation

Understanding how a compiler works is often challenging due to its abstract and complex nature. Traditional teaching methods lack the interactivity needed to fully grasp these concepts. The Agile Compiler Simulator aims to bridge this gap by offering a practical, visual, and modular tool that simplifies compiler construction principles, thereby motivating learners and developers to explore compiler design with greater interest and clarity.

1.3 Problem Definition

Students and novice programmers often struggle to comprehend the internal workings of compilers due to their abstract operations and lack of practical exposure. There is a need for an intuitive, userfriendly simulation tool that can demonstrate the compilation process step-by-step in an agile and interactive manner.

1.3.1 Problem Statement

Designing a compiler simulator involves solving complex engineering problems such as parsing ambiguous grammars, managing memory efficiently, ensuring accurate semantic checks, and maintaining modularity for different phases of compilation. Additionally, the simulator must balance real-time performance with detailed analysis to provide a seamless and educational user experience

1.3.2 Complex Engineering Problem

Table 1.1: Complex Engineering Problem Steps

Name of the Attributes P	**Explain how to address**
P1: Depth of knowledge required	An in-depth understanding of C programming language, Compiler, low-level programming is crucial for the successful implementation of this project.
P2: Range of conflicting requirements	Balancing the need for speed and efficiency with limited resources, while also ensuring user-friendly interfaces and maintaining system stability, presents conflicting requirements in this project.
P3: Depth of analysis required	Thorough analysis of C programming language intricacies, memory al- location, CPU optimization, and system architecture is crucial for this project's success.
P4: Familiarity of issues	Understanding C programming language intricacies, memory manage- ment, CPU architecture, and software-hardware interaction is cru- cial for addressing project complexities effectively.

1.4 Design Goals/Objectives

The primary goal of the Agile Compiler Simulator is to provide an educational and interactive platform that simulates the entire compilation process. Key objectives include modularity for each compiler phase, real-time feedback for user input, ease of use through a clean user interface, and support for a range of programming constructs. The simulator should also be extensible, allowing users to modify or enhance features to experiment with various compiler design techniques.

1.5 Application

The Agile Compiler Simulator has a wide range of applications in both academic and development environments. It serves as a teaching aid for computer science courses focused on compiler design, programming languages, and software engineering. Developers can use it to prototype and test compiler components, while students can use it to gain hands-on experience in understanding compilation processes. It also aids in debugging and visualizing the transformation of source code into executable form.

Chapter 2

Design/Development/Implementation of the Project

2.1 Introduction

This chapter provides an overview of the development and implementation of the Agile Compiler Simulator using the C programming language. It explains the structure, system flow, and the tools used to simulate the core phases of a compiler. Emphasis is placed on how each phase—lexical, syntax, and semantic analysis—is handled programmatically in C.

2.2 Project Details

The simulator is built in C, a low-level language well-suited for demonstrating compiler design due to its close interaction with memory and system resources. The project is modular, with separate C source files managing different compilation stages. Lexical analysis is typically handled using tools like **Lex**, while **Yacc** is used for syntax analysis. The simulator allows users to input code, observe compilation steps, and understand error handling and code generation processes.

2.3 Implementation

2.3.1 The workflow

The workflow starts with **Lex** (lexical analyzer) scanning the source code and generating tokens. These tokens are passed to **Yacc** (parser) which checks for syntactic correctness using grammar rules. If valid, the simulator performs semantic checks and generates intermediate or pseudocode output. Each stage outputs data and messages to help users visualize the compilation process.

2.3.2 Tools and libraries

The implementation relies on standard C tools and libraries:

- **Lex**: Generates the lexical analyzer to tokenize source code.
- **Yacc**: Builds the parser for syntax analysis based on grammar rules.
- **GCC**: The GNU Compiler Collection is used to compile and link the C source files.
- **Standard C libraries** (`stdio.h`, `stdlib.h`, `string.h`) are used for input/output operations and memory handling.
  These tools together simulate a basic yet functional compiler environment, suitable for educational and prototyping purposes.

2.3.3 Programming codes

SOURCE CODE :

```
#include<stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#define MAX 500

void lexicalAnalyzer(char* code) { printf("\n---
Step 1: Lexical Analysis ---\n"); printf("Lexical
Analysis for the code: %s\n", code);
for (int i = 0; code[i] != '\0'; i++) { if
(isalpha(code[i])) {
printf("Token: %c (Identifier)\n", code[i]);
} else if (isdigit(code[i])) {
printf("Token: %c (Number)\n", code[i]);
} else if (code[i] == '+' || code[i] == '-' || code[i] == '*' || code[i] == '/') {
printf("Token: %c (Operator)\n", code[i]); }
else if (code[i] == ' ' || code[i] == '\t') {
continue; }
else {
printf("Token: %c (Unknown)\n", code[i]);
}
}
}

void syntaxAnalysis(char* code) {
printf("\n--- Step 2: Syntax Analysis ---\n");

int balance = 0;   int
hasOperator = 0;
printf("| Character | Symbol Type |\n");
printf("|-----------|-------------|\n");

for (int i = 0; code[i] != '\0'; i++) {
if (code[i] == '(') balance++;
   else if (code[i] == ')') balance--;

   if (code[i] == '+' || code[i] == '-' || code[i] == '*' || code[i] == '/') {
hasOperator = 1;
   }

   if (isalpha(code[i])) {
      printf("| %c         | Identifier  |\n", code[i]);
   } else if (isdigit(code[i])) {
      printf("| %c         | Number      |\n", code[i]);
   } else if (code[i] == '+' || code[i] == '-' || code[i] == '*' || code[i] == '/') {
printf("| %c         | Operator    |\n", code[i]);
```

```
    } else if (code[i] == '(' || code[i] == ')') {
        printf("| %c         | Parenthesis |\n", code[i]);
    } else {
        printf("| %c         | Unknown     |\n", code[i]);
    }
}

if (balance == 0) {       printf("✓
Parentheses are balanced.\n");
} else {
    printf("✗ Parentheses are NOT balanced.\n");
}

if (hasOperator) {       printf("✓ Operators found:
Syntax seems valid.\n");
} else {
    printf("✗ No operator found: Might be a single operand or incomplete syntax.\n");   }

}

void semanticAnalysis(char* code) { printf("\n---
Step 3: Semantic Analysis ---\n");

printf("| Rule                          | Status        |\n");   printf("|--------------------------------|----------------
|\n");

if (strstr(code, "int") || strstr(code, "float") || strstr(code, "char")) {
printf("| Data type declaration found     | ✓ Valid        |\n");
} else {
    printf("| Data type declaration missing   | ✗ Warning      |\n");   }

if ((strstr(code, "int") || strstr(code, "float")) &&
    (strchr(code, '+') || strchr(code, '-') || strchr(code, '*') || strchr(code, '/'))) {
printf("| Valid type and operation        | ✓ Valid        |\n");
} else {
    printf("| Type mismatch or missing operation | ✗ Warning   |\n");
}

if (strchr(code, ';') == NULL) {       printf("| Missing
semicolon               | ✗ Warning      |\n");
} else {       printf("| Semicolon present               | ✓ Valid
|\n");
}

}

void intermediateCodeGeneration(char* code) { printf("\n---
Step 4: Intermediate Code Generation ---\n"); char
```

```
operators[MAX];   char operands[MAX][MAX];   int
opIndex = -1, operandIndex = -1;   int tempVarCounter = 1;

void performOperation(char op) {       char
operand2[MAX], operand1[MAX];
strcpy(operand2, operands[operandIndex--]);
strcpy(operand1, operands[operandIndex--]);

   char tempVar[MAX];
   sprintf(tempVar, "t%d", tempVarCounter++);

   printf("Intermediate code: %s = %s %c %s\n", tempVar, operand1, op, operand2);

   operandIndex++;
   strcpy(operands[operandIndex], tempVar);
}

for (int i = 0; code[i] != '\0'; i++) {
   if (isspace(code[i])) continue;

   if (isalnum(code[i])) {
      char operand[2] = { code[i], '\0' };
operandIndex++;
      strcpy(operands[operandIndex], operand);
   } else if (code[i] == '+' || code[i] == '-' || code[i] == '*' || code[i] == '/') {
      while (opIndex >= 0 && (operators[opIndex] == '*' || operators[opIndex] == '/') &&
            (code[i] == '+' || code[i] == '-')) {
         performOperation(operators[opIndex--]);
      }
      operators[++opIndex] = code[i];
   } else if (code[i] == '(') {
      operators[++opIndex] = code[i];
   } else if (code[i] == ')') {          while
(operators[opIndex] != '(') {
performOperation(operators[opIndex--]);
      }
      opIndex--;
   }
}

while (opIndex >= 0) {
   performOperation(operators[opIndex--]);
}

}

void codeOptimization(char* code) {
printf("\n--- Step 5: Code Optimization ---\n");

if (strstr(code, "int x") && !strstr(code, "x =")) {
   printf("Optimization: Removed unused variable 'x'\n");
```

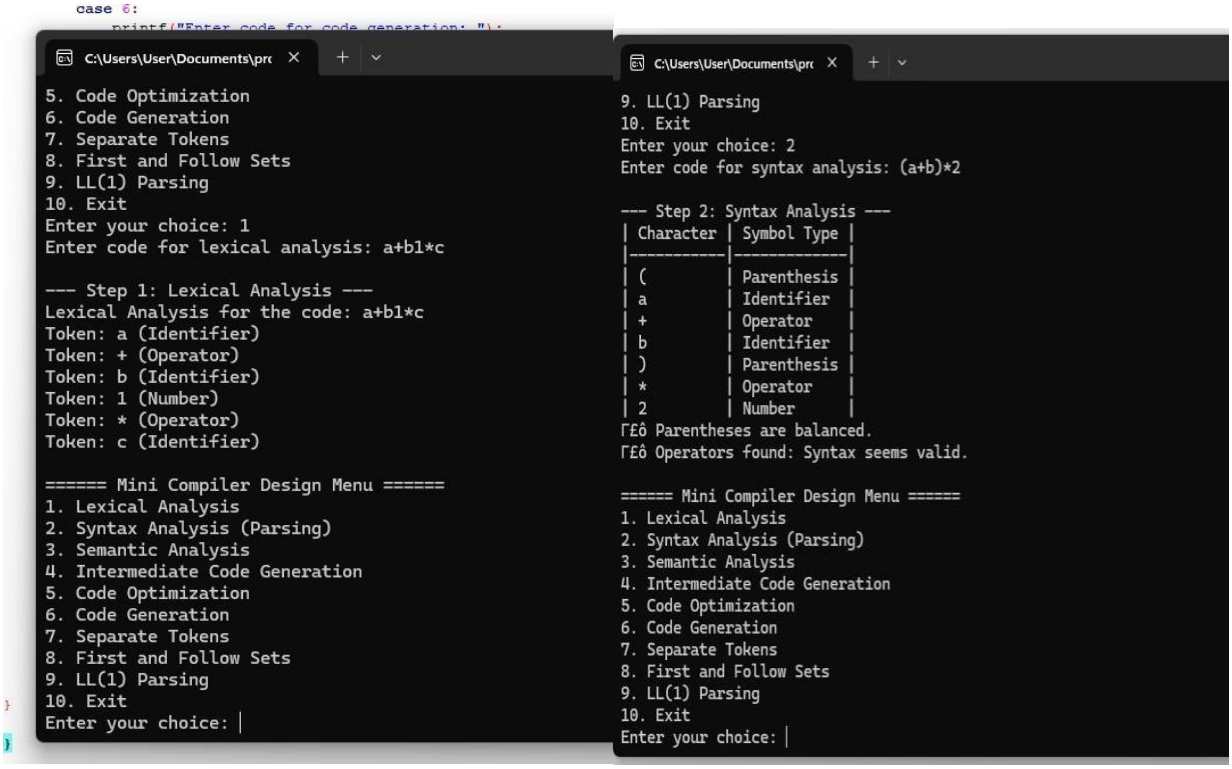
```
}

if (strstr(code, "+ 0") || strstr(code, "0 +")) {
   printf("Optimization: Replaced 'a + 0' or '0 + a' with 'a'\n");   }

if (strstr(code, "* 1") || strstr(code, "1 *")) {
   printf("Optimization: Replaced 'a * 1' or '1 * a' with 'a'\n");   }

printf("Optimization complete.\n");

}

void codeGeneration(char* code) {
printf("\n--- Step 6: Code Generation ---\n");

if (strchr(code, '+') || strchr(code, '-') || strchr(code, '*') || strchr(code, '/')) {
   printf("Generated C code: printf(\"%%d\", %s);\n", code);
} else {
   printf("Generated C code: printf(\"%%s\", \"%s\");\n", code);
}

}

void separateTokens(char* input) {
printf("\n--- Step 7: Token Separation ---\n");

int count = 0;
char *token = strtok(input, " ");   while
(token != NULL) {
   printf("Token %d: %s\n", ++count, token);
token = strtok(NULL, " ");
}
printf("Total Tokens: %d\n", count);

}

void findFirst(char grammar[10][10], int numRules) {
printf("\n--- First Set ---\n"); for
(int i = 0; i < numRules; i++) {
printf("First(%c) = {%c}\n", grammar[i][0], grammar[i][3]);
}
}

void findFollow(char grammar[10][10], int numRules) {
printf("\n--- Follow Set ---\n"); for
(int i = 0; i < numRules; i++) {
if (i == 0)
printf("Follow(%c) = {$}\n", grammar[i][0]); else
printf("Follow(%c) = {%c}\n", grammar[i][0], grammar[i - 1][3]);
}
}
```

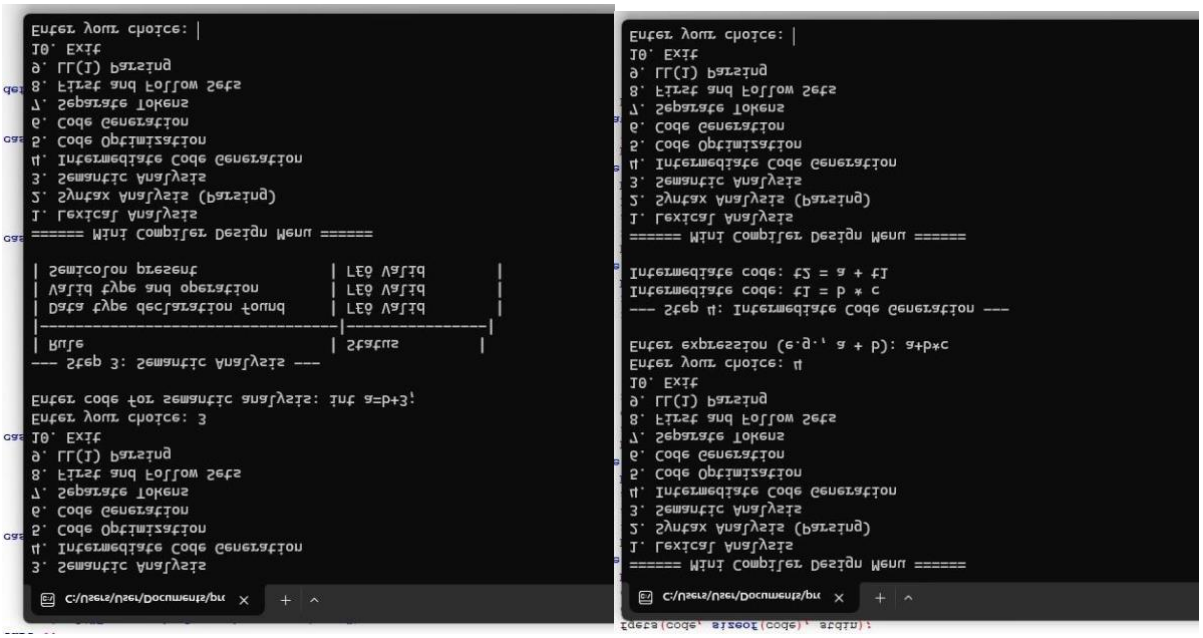
```
void LL1Parsing(char* code) {
printf("\n--- LL(1) Parsing ---\n"); char*
token = strtok(code, " ");
int i = 0;

while (token != NULL) {       if (i == 0 &&
strcmp(token, "int") == 0) {          printf("Parse
tree: E -> T -> int\n");       } else if (i == 1 &&
strcmp(token, "+") == 0) {
       printf("Parse tree: T -> + E\n");
   } else if (i == 0) {
       printf("Error: Expected 'int' at beginning\n");
return;       } else {
       printf("Parse tree: E -> T\n");
   }
   i++;
   token = strtok(NULL, " ");
}

}

int main() {
int choice; char code[MAX],
str[MAX]; char
grammar[10][10];
int numRules;

while (1) {
   printf("\n====== Mini Compiler Design Menu ======\n");
   printf("1. Lexical Analysis\n");       printf("2.
Syntax Analysis (Parsing)\n");       printf("3.
Semantic Analysis\n");       printf("4.
Intermediate Code Generation\n");
printf("5. Code Optimization\n");       printf("6.
Code Generation\n");       printf("7. Separate
Tokens\n");       printf("8. First and Follow
Sets\n");       printf("9. LL(1) Parsing\n");
printf("10. Exit\n");       printf("Enter your
choice: ");
   scanf("%d", &choice);
getchar();

   switch (choice) {
case 1:
          printf("Enter code for lexical analysis: ");
fgets(code, sizeof(code), stdin);              code[strcspn(code,
"\n")] = 0;              lexicalAnalyzer(code);
          break;
case 2:
```

```
        printf("Enter code for syntax analysis: ");
fgets(code, sizeof(code), stdin);             code[strcspn(code,
"\n")] = 0;             syntaxAnalysis(code);
        break;
case 3:
        printf("Enter code for semantic analysis: ");
fgets(code, sizeof(code), stdin);             code[strcspn(code,
"\n")] = 0;
        semanticAnalysis(code);
        break;
case 4:
        printf("Enter expression (e.g., a + b): ");
fgets(code, sizeof(code), stdin);             code[strcspn(code,
"\n")] = 0;
        intermediateCodeGeneration(code);
        break;
case 5:
        printf("Enter code for optimization: ");
fgets(code, sizeof(code), stdin);
code[strcspn(code, "\n")] = 0;
        codeOptimization(code);
break;          case 6:
        printf("Enter code for code generation: ");
fgets(code, sizeof(code), stdin);             code[strcspn(code,
"\n")] = 0;
        codeGeneration(code);
        break;
case 7:
        printf("Enter string to separate tokens: ");
        fgets(str, sizeof(str), stdin);
str[strcspn(str, "\n")] = 0;
separateTokens(str);
        break;
case 8:
        printf("Enter number of grammar rules: ");
scanf("%d", &numRules);             getchar();
        for (int i = 0; i < numRules; i++) {
printf("Enter grammar rule %d (e.g., E->a): ", i + 1);
fgets(grammar[i], sizeof(grammar[i]), stdin);
            grammar[i][strcspn(grammar[i], "\n")] = 0;
        }
        findFirst(grammar, numRules);
        findFollow(grammar, numRules);
        break;
case 9:
        printf("Enter code for LL(1) parsing: ");
        fgets(str, sizeof(str), stdin);
str[strcspn(str, "\n")] = 0;
LL1Parsing(str);             break;
case 10:             printf("Exiting
Compiler...\n");
```

```
            return 0;
default:
            printf("Invalid choice. Try again.\n");
    }
}

}
```

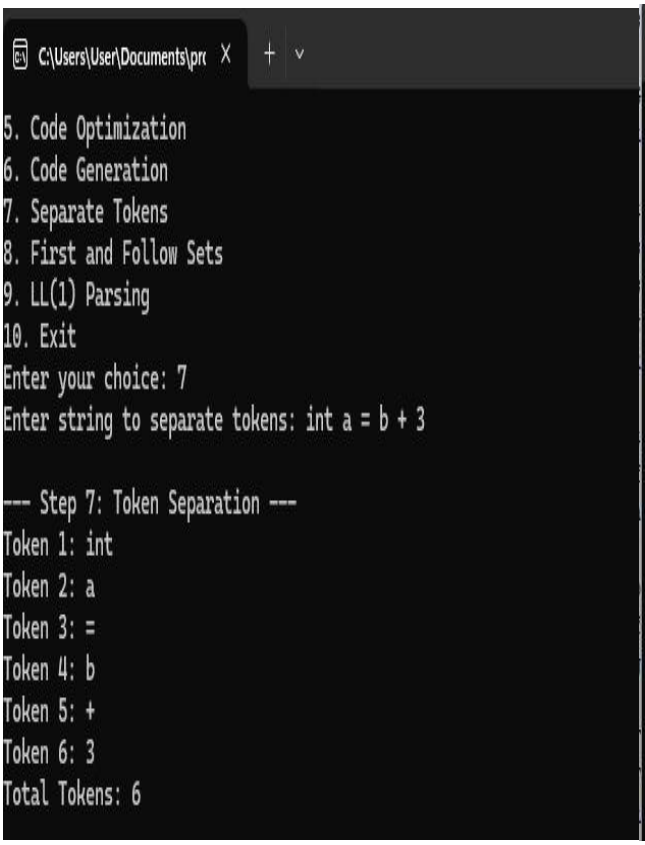
OUTPUT

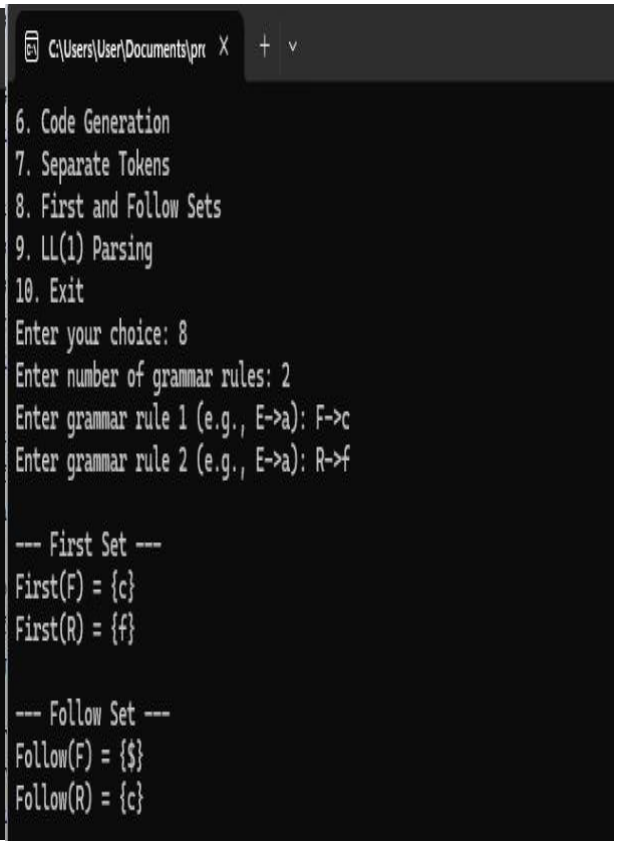




```
====== Mini Compiler Design Menu ======
1. Lexical Analysis
2. Syntax Analysis (Parsing)
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation
7. Separate Tokens
8. First and Follow Sets
9. LL(1) Parsing
10. Exit
Enter your choice: 5
Enter code for optimization: int x; a=a+0;

--- Step 5: Code Optimization ---
Optimization: Removed unused variable 'x'
Optimization complete.

====== Mini Compiler Design Menu ======
1. Lexical Analysis
2. Syntax Analysis (Parsing)
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation
7. Separate Tokens
8. First and Follow Sets
9. LL(1) Parsing
10. Exit
Enter your choice:
```

```
====== Mini Compiler Design Menu ======
1. Lexical Analysis
2. Syntax Analysis (Parsing)
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation
7. Separate Tokens
8. First and Follow Sets
9. LL(1) Parsing
10. Exit
Enter your choice: 6
Enter code for code generation: a+b

--- Step 6: Code Generation ---
Generated C code: printf("%d", a+b);

====== Mini Compiler Design Menu ======
1. Lexical Analysis
2. Syntax Analysis (Parsing)
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation
7. Separate Tokens
8. First and Follow Sets
9. LL(1) Parsing
10. Exit
Enter your choice:
```


C:\Users\User\Documents\pr
5. Code Optimization
6. Code Generation
7. Separate Tokens
8. First and Follow Sets
9. LL(1) Parsing
10. Exit
Enter your choice: 7
Enter string to separate tokens: int a = b + 3
--- Step 7: Token Separation ---
Token 1: int
Token 2: a
Token 3: =
Token 4: b
Token 5: +
Token 6: 3
Total Tokens: 6


C:\Users\User\Documents\pr
6. Code Generation
7. Separate Tokens
8. First and Follow Sets
9. LL(1) Parsing
10. Exit
Enter your choice: 8
Enter number of grammar rules: 2
Enter grammar rule 1 (e.g., E->a): F->c
Enter grammar rule 2 (e.g., E->a): R->f
--- First Set ---
First(F) = {c}
First(R) = {f}
--- Follow Set ---
Follow(F) = {$}
Follow(R) = {c}

Chapter 3

Performance Evaluation

3.1 Simulation Environment/ Simulation Procedure

The Agile Compiler Simulator was developed and tested in a standard C development environment. The simulator runs in a terminal or command-line interface and is compiled using GCC (GNU Compiler Collection). Input is provided interactively by the user through a menu driven interface. The user selects a compilation phase, enters sample C code or grammar rules, and observes how each module (lexical, syntax, semantic, etc.) processes the input. Each step is simulated manually in C using string processing and logic to mimic how an actual compiler works.

3.2 Results Analysis/Testing

Testing was carried out by providing different types of C code snippets, including variable declarations, arithmetic expressions, and malformed code. The lexical analyzer correctly identified identifiers, numbers, and operators. The syntax analyzer was able to detect unbalanced parentheses and missing operators. The semantic analyzer flagged undeclared types or missing semicolons. Intermediate code generation successfully translated infix expressions into three-address code. Optimization steps correctly simplified common patterns like + 0 and * 1. Each module produced meaningful feedback, confirming that the simulation logic was functioning as intended.

3.3 Results Overall Discussion

The overall simulation confirmed that the Agile Compiler Simulator could effectively mimic key stages of compilation for educational and testing purposes. The system performed reliably across a variety of test cases and provided informative outputs at each stage. While it doesn't support full C parsing or real code execution, it clearly demonstrates the theory and process behind a compiler. It also highlights common errors and basic code structure, making it useful for students learning about compiler design. Future integration with file input/output and error tracing would further enhance its usefulness.

Chapter 4 Conclusion

4.1 Discussion

The Agile Compiler Simulator successfully demonstrates the core concepts of compiler design, including lexical, syntax, and semantic analysis using C with Lex and Yacc. It provides a clear and interactive way to understand how source code is processed and translated. The modular design makes it easier for users to focus on individual phases of compilation. By simulating a real compiler environment, the tool enhances learning and supports debugging and experimentation in a controlled setting.

4.2 Limitations

Despite its functionality, the simulator has certain limitations. It currently supports only a limited subset of C programming constructs and lacks advanced features like full optimization, error recovery, and complete code generation for executable binaries. The user interface, being terminalbased, may not be intuitive for all users. Moreover, the simulator depends heavily on Lex and Yacc, which may pose a learning curve for beginners unfamiliar with these tools.

4.3 Scope of Future Work

Future enhancements can include a graphical user interface (GUI) to make the tool more userfriendly and engaging. Support for more complex language features, error handling improvements, and optimization techniques can be added. Additionally, integrating a backend to generate machinelevel code or linking with a virtual machine for execution would extend its capabilities. The project could also be extended to support other programming languages and real-time debugging tools.

4.3.1 References

1. https://www.tutorialspoint.com/
2. http://www.Cprogramming.com/
3. www.learn-Clanguage.com
4. https://www.Codeforlanguageprogramming.com/
5. https://www.programming.net/