SimpleLang Compiler

1. Introduction

1.1 Overview

The **SimpleLang Compiler** is a basic high-level language compiler designed to translate SimpleLang code into assembly language for an 8-bit CPU. The project demonstrates fundamental compiler construction techniques, including lexical analysis, parsing, and code generation.

1.2 Objectives

- Develop a lexer to tokenize SimpleLang source code.
- Implement a parser to analyse syntax and build an abstract representation.
- Generate assembly code for an 8-bit CPU.
- Handle variable declarations, assignments, arithmetic operations, and conditional statements

1.3 Scope

- Supports variable declarations (int a;).
- Handles arithmetic expressions (a = b + c;).
- Implements conditional statements (if (a == b) { ... }).
- Outputs assembly code compatible with an 8-bit CPU simulator.

2. System Design

2.1 Architecture

The compiler follows a **three-phase design**:

1. **Lexical Analysis (Lexer)** – Converts source code into tokens.
2. **Syntax Analysis (Parser)** – Validates structure and builds an intermediate representation.
3. **Code Generation** – Translates parsed code into assembly instructions.

2.2 Data Structures

- **Token Structure** – Stores token type (TOKEN_INT, TOKEN_IDENTIFIER, etc.) and text.
- **Symbol Table** – Tracks variable names and memory addresses.
- **Assembly Buffer** – Stores generated assembly code.

2.3 Workflow

1. **Input File** (input.sl) is read.
2. **Lexer** processes characters into tokens.
3. **Parser** validates syntax and generates an intermediate representation.
4. **Code Generator** emits assembly instructions (output.asm).

2. System Design

2.1 Architecture

The compiler follows a **three-phase design**:

4. **Lexical Analysis (Lexer)** – Converts source code into tokens.
5. **Syntax Analysis (Parser)** – Validates structure and builds an intermediate representation.
6. **Code Generation** – Translates parsed code into assembly instructions.

2.2 Data Structures

- **Token Structure** – Stores token type (TOKEN_INT, TOKEN_IDENTIFIER, etc.) and text.
- **Symbol Table** – Tracks variable names and memory addresses.
- **Assembly Buffer** – Stores generated assembly code.

2.3 Workflow

5. **Input File** (input.sl) is read.
6. **Lexer** processes characters into tokens.
7. **Parser** validates syntax and generates an intermediate representation.
8. **Code Generator** emits assembly instructions (output.asm).

3. Implementation Details

3.1 Lexical Analysis (Lexer)

Functions:

- getNextToken() – Reads characters and categorizes them into tokens.
- ungetToken() – Allows token lookahead by pushing back a token.

Token Types:

Token Type	Example
TOKEN_INT	int
TOKEN_IDENTIFIER	a, b
TOKEN_NUMBER	5, 10
TOKEN_ASSIGN	=
TOKEN_PLUS	+
TOKEN_IF	if
TOKEN_EQUAL	==

Example Lexer Output:

Token: TOKEN_INT ('int')

Token: TOKEN_IDENTIFIER ('a')

Token: TOKEN_SEMICOLON (';')

3.2 Syntax Analysis (Parser)

Functions:

- compileStatement() – Processes declarations, assignments, and conditionals.
- compileExpression() – Handles arithmetic operations (+, -).

Grammar Rules:

program → statement*

statement → declaration | assignment | if_statement

declaration → 'int' identifier ';'

assignment → identifier '=' expression ';'

if_statement → 'if' '(' condition ')' '{' statement* '}'

condition → identifier '==' (identifier | number)

expression → term (('+' | '-') term)*

term → identifier | number

Example Parsing:

```
int a;
a = 5 + 3;
if (a == 8) {
    a = a + 1;
}
```

- Recognizes int a; as a declaration.
- Parses a = 5 + 3; as an assignment with an arithmetic expression.
- Processes if (a == 8) as a conditional statement.

3.3 Code Generation

Functions:

- emit() – Writes assembly instructions to a buffer.
- getVarAddress() – Assigns memory addresses to variables.

Assembly Instructions Used:

Instruction	Description
LDA addr	Load value from memory
STA addr	Store value to memory
LDI val	Load immediate value
ADD addr	Add memory value to accumulator
SUB addr	Subtract memory value from accumulator
JZ label	Jump if zero
JMP label	Unconditional jump

SimpleLang Code:

```
int a;
a = 5 + 3;
if (a == 8) { a = a + 1; }
```

Generated Assembly Code:

```
; int a;
LDI 5
STA 16    ; a stored at address 16
LDI 3
ADD 16
STA 16
; if (a == 8)
LDA 16
SUBI 8
JZ L1
JMP L2
L1:
LDA 16
ADDI 1
STA 16
L2:
```

4. Testing and Results

4.1 Test Cases

Input (input.sl**)**	**Expected Output (**output.asm**)**
int a; a = 5;	LDI 5 STA 16
a = b + 3;	LDA 17 ADDI 3 STA 16
if (a == b) { a = a + 1; }	Generates jumps and conditional logic

4.2 Observations

- Successfully compiles variable declarations, assignments, and if statements.
- Handles arithmetic expressions with variables and constants.
- Generates correct assembly for branching logic.

---

5. Challenges and Solutions

5.1 Challenges Faced

1. **Token Lookahead** – Needed for parsing expressions.
2. **Handling Nested Statements** – Required recursive parsing.
3. **Memory Management** – Tracking variable addresses.

5.2 Solutions Implemented

- **Token Buffering** – Used ungetToken() for lookahead.
- **Recursive Parsing** – compileStatement() calls itself for nested blocks.
- **Symbol Table** – Managed variable addresses efficiently.

6. Conclusion

The **SimpleLang Compiler** successfully demonstrates:
Lexical analysis (tokenization)
Syntax parsing (grammar validation)
Assembly code generation