**Design and Implementation of the Compiler**

**Overview**

The compiler processes input code written in a SimpleLang programming language, translating it into assembly code. The design is divided into the following components:

1. **Lexer**: Tokenizes the input string.
2. **Parser**: Builds an Abstract Syntax Tree (AST) from tokens.
3. **Code Generator**: Translates the AST into assembly code.
4. **AST Nodes**: Represent the structure of the input program.

**1. Lexer**

**Design**

The **Lexer** processes the raw input string and converts it into a sequence of tokens. Each token has a type and associated text. The types include keywords (e.g., int, if), operators (+, -, =), and other symbols (e.g., parentheses, braces).

**Implementation**

**Key Functions**

* **currentChar()**: Returns the character at the current position.
* **advance()**: Moves to the next character.
* **skipWhitespace()**: Skips over spaces and tabs.
* **getNextToken()**: Extracts the next token based on the current character.

**Token Types**

* TOKEN\_INT, TOKEN\_IDENTIFIER, TOKEN\_NUMBER
* TOKEN\_ASSIGN, TOKEN\_PLUS, TOKEN\_MINUS
* TOKEN\_IF, TOKEN\_EQUAL, TOKEN\_NOT\_EQUAL
* TOKEN\_LPAREN, TOKEN\_RPAREN, TOKEN\_LBRACE, TOKEN\_RBRACE
* TOKEN\_SEMICOLON, TOKEN\_EOF

**2. Parser**

**Design**

The **Parser** takes tokens from the lexer and produces an Abstract Syntax Tree (AST) representing the program's structure. The parsing is recursive, starting with high-level constructs (e.g., statements) and drilling down into expressions.

**Implementation**

**Key Functions**

* **parseStatement()**: Parses high-level statements like variable declarations, assignments, and if conditions.
* **parseExpression()**: Parses arithmetic and logical expressions.
* **parseEquality()**: Handles equality comparisons.
* **parseAdditive()**: Parses addition and subtraction operations.
* **parsePrimary()**: Parses primary expressions like numbers and identifiers.

**Supported Constructs**

* Variable declarations: int x = 5;
* Assignments: x = 10;
* Conditional statements: if (x == 5) { ... }

**3. AST Nodes**

**Design**

The **AST Nodes** represent the program's structure. Each node corresponds to a construct in the language, such as expressions or statements.

**Types of Nodes**

**Expressions**

* **Number**: Represents a numeric literal.
* **Identifier**: Represents a variable.
* **BinaryOp**: Represents binary operations (e.g., +, -, ==).

**Statements**

* **VarDeclaration**: Represents variable declarations.
* **Assignment**: Represents assignments.
* **If**: Represents conditional statements.
* **Block**: Represents a block of statements.

**Key Methods**

Each node implements:

* **generateAssembly(CodeGenerator&)**: Generates assembly code for the node.

**4. Code Generator**

**Design**

The **Code Generator** translates the AST into assembly code. It uses virtual registers and manages variable declarations in memory.

**Implementation**

**Key Functions**

* **getNewRegister()**: Allocates a new virtual register.
* **getNewLabel()**: Allocates a new label for control flow.
* **emit()**: Emits assembly code.
* **declareVariable()**: Declares a variable in memory.
* **generatePrelude()**: Emits data section setup.
* **generatePostlude()**: Emits entry point setup.
* **generateEpilogue()**: Emits program termination instructions.

**Assembly Example**

.section .data

x: .word 0

.section .text

.global \_start

\_start:

MOV R0, #5

STR R0, [x]

MOV R7, #1

MOV R0, #0

SWI 0

**Example Walkthrough**

**Input Code**

int x = 5;

x = x + 10;

if (x == 15) {

int y = 20;

}

**Lexer Output**

Tokens:

[TOKEN\_INT, "int"]

[TOKEN\_IDENTIFIER, "x"]

[TOKEN\_ASSIGN, "="]

[TOKEN\_NUMBER, "5"]

[TOKEN\_SEMICOLON, ";"]

...

**Parser Output**

AST:

VarDeclaration: int x = 5

Assignment: x = (x + 10)

If:

Condition: (x == 15)

Block:

VarDeclaration: int y = 20

**Generated Assembly**

.section .data

x: .word 0

y: .word 0

.section .text

.global \_start

\_start:

MOV R0, #5

STR R0, [x]

LDR R1, [x]

MOV R2, #10

ADD R1, R1, R2

STR R1, [x]

CMP R1, #15

BNE L0

MOV R3, #20

STR R3, [y]

L0:

MOV R7, #1

MOV R0, #0

SWI 0

**Future Enhancements**

* Support for additional data types and operations.
* Loop constructs (e.g., for, while).
* Function declarations and calls.
* Enhanced error reporting in the parser.