A SIMPLE COMPILER IMPLEMENTATION

PS/2020/169 - NANDASIRI A.D.N.D.

COSC 44283 Theory Of Compilers

1. Project Objective

The objective of this project is to design and implement a simple compiler for a custom mini-language called MiniLang, which supports fundamental programming constructs such as:

- Variable declarations
- Arithmetic expressions
- Conditional statements
- Loops
- Print statements

This project provides hands-on experience with the core compilation phases: Lexical Analysis, Syntax Analysis, Semantic Analysis, and Intermediate Code Generation.

2. MiniLang Grammar Description

The following is a simplified context-free grammar (CFG) used in the MiniLang compiler:

```
program        → { statement }

statement      → declaration
                 | assignment
                 | if_statement
                 | while_statement
                 | print_statement

declaration    → 'int' IDENTIFIER ';'
assignment     → IDENTIFIER '=' expression ';'
if_statement   → 'if' '(' expression ')' block [ 'else' block ]
while_statement → 'while' '(' expression ')' block
print_statement → 'print' '(' expression ')' ';'

block          → '{' { statement } '}'

expression     → arithmetic_expression [ COMPARATOR arithmetic_expression ]
arithmetic_expression → term { ('+' | '-') term }
term           → factor { ('*' | '/') factor }
factor         → IDENTIFIER | NUMBER | '(' expression ')'
```

3. Compiler Phases Overview

3.1 Lexical Analyzer

- Implemented using Java regex patterns
- Recognizes keywords, identifiers, numbers, operators, punctuation
- Outputs a list of tokens

3.2 Syntax Analyzer

- Recursive-descent parser
- Implements CFG rules to validate the structure of statements
- Reports syntax errors with clear messages

3.3 Semantic Analyzer

- Tracks declared variables using a symbol table
- Detects:
    - Undeclared variable usage
    - Redeclaration of the same variable
- Terminates with a message if semantic errors are found

3.4 Intermediate Code Generation

- Generates 3-address code
- Supports expressions like: a = b + c * 2;
- Handles nested operations with temporary variables

4. Complete Source Code

SimpleLexer.java

```
package compiler;

import java.io.*;
import java.util.*;
import java.util.regex.*;

public class SimpleLexer {

    // 1. Token Types
    public enum LexType {
        KEYWORD,       // Reserved words: int, if, else, while, print
        IDENTIFIER,    // Variable names
        NUMBER,        // Integer literals
        ASSIGN_OP,     // =
        SEMICOLON,     // ;
        OPERATOR,      // +, -, *, /
        COMPARATOR,    // <, >
        LEFT_PAREN,    // (
        RIGHT_PAREN,   // )
        LEFT_BRACE,    // {
        RIGHT_BRACE    // }
    }

    // 2. Token Representation
    public static class LexToken {
        LexType lexType;
        String lexValue;

        public LexToken(LexType lexType, String lexValue) {
            this.lexType = lexType;
            this.lexValue = lexValue;
        }

        @Override
        public String toString() {
            return "(" + lexType + ", " + lexValue + ")";
        }
    }

    // 3. Regular Expression Patterns
```

```
    private static final Map<LexType, String> PATTERNS = new LinkedHashMap<>();
    private static final Set<String> RESERVED_WORDS = Set.of("int", "if", "else", "while",
"print");

    static {
        PATTERNS.put(LexType.KEYWORD,     "\\b(int|if|else|while|print)\\b");
        PATTERNS.put(LexType.IDENTIFIER,  "\\b[a-zA-Z_][a-zA-Z0-9_]*\\b");
        PATTERNS.put(LexType.NUMBER,      "\\b\\d+\\b");
        PATTERNS.put(LexType.ASSIGN_OP,   "=");
        PATTERNS.put(LexType.SEMICOLON,   ";");
        PATTERNS.put(LexType.OPERATOR,    "[+\\-*/]");
        PATTERNS.put(LexType.COMPARATOR,  "[<>]");
        PATTERNS.put(LexType.LEFT_PAREN,  "\\(");
        PATTERNS.put(LexType.RIGHT_PAREN, "\\)");
        PATTERNS.put(LexType.LEFT_BRACE,  "\\{");
        PATTERNS.put(LexType.RIGHT_BRACE, "\\}");
    }

    // 4. Lexical Analyzer Function
    public static List<LexToken> lexAnalyze(String sourceCode) {
        List<LexToken> lexTokens = new ArrayList<>();
        String combinedPattern = PATTERNS.values().stream()
                .reduce((p1, p2) -> p1 + "|" + p2)
                .orElseThrow(() -> new RuntimeException("No patterns defined"));

        Pattern pattern = Pattern.compile(combinedPattern);
        Matcher matcher = pattern.matcher(sourceCode);

        while (matcher.find()) {
            String matchedText = matcher.group();
            LexType matchedType = null;

            for (Map.Entry<LexType, String> entry : PATTERNS.entrySet()) {
                if (matchedText.matches(entry.getValue())) {
                    matchedType = entry.getKey();
                    break;
                }
            }

            // Convert identifiers to keywords if matched
            if (matchedType == LexType.IDENTIFIER &&
RESERVED_WORDS.contains(matchedText)) {
                matchedType = LexType.KEYWORD;
```

```
            }

            if (matchedType != null) {
                lexTokens.add(new LexToken(matchedType, matchedText));
            }
        }

        return lexTokens;
    }

    // 5. Main Method (Reads File and Prints Tokens)
    public static void main(String[] args) {
        String fileName = "src/input.minilang";
        StringBuilder sourceBuilder = new StringBuilder();

        try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
            String line;
            while ((line = br.readLine()) != null) {
                sourceBuilder.append(line).append("\n");
            }

            // Lexical Analysis
            List<LexToken> tokens = lexAnalyze(sourceBuilder.toString());
            System.out.println("✅ Lexical Tokens:");
            for (LexToken token : tokens) {
                System.out.println(token);
            }

            // Syntax Analysis
            SimpleParser parser = new SimpleParser(tokens);
            parser.parse();

        } catch (FileNotFoundException e) {
            System.err.println("❌ Error: File not found - " + fileName);
        } catch (IOException e) {
            System.err.println("❌ Error reading file: " + e.getMessage());
        } catch (Exception e) {
            System.err.println("❌ Unexpected error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

SimpleParser.java

```
package compiler;

import java.util.List;
import java.util.Set;
import java.util.HashSet;
import java.util.ArrayList;
import static compiler.SimpleLexer.*;

public class SimpleParser {

    private final List<LexToken> tokens;
    private int currentIndex = 0;
    private final Set<String> declaredVariables = new HashSet<>();
    private int tempCounter = 0;
    private List<String> threeAddressCode = new ArrayList<>();

    private String newTemp() {
        return "t" + (tempCounter++);
    }

    public SimpleParser(List<LexToken> tokens) {
        this.tokens = tokens;
    }

    // Entry Point
    public void parse() {
        while (!isAtEnd()) {
            parseStatement();
        }

        System.out.println("✅ Syntax Analysis: Passed.");

        // Print generated 3-address code (if you're using code generation)
        if (!threeAddressCode.isEmpty()) {
            System.out.println("Generated 3-Address Code:");
            for (String line : threeAddressCode) {
                System.out.println(line);
            }
        }

        // ✅ Print the Symbol Table (Declared Variables)
```

```
    System.out.println("Declared Variables (Symbol Table):");
    for (String var : declaredVariables) {
        System.out.println("- " + var);
    }
}



// ========== STATEMENTS ==========

private void parseStatement() {
    if (match(LexType.KEYWORD, "int")) {
        parseDeclaration();
    } else if (check(LexType.IDENTIFIER)) {
        parseAssignment();
    } else if (match(LexType.KEYWORD, "if")) {
        parseIfStatement();
    } else if (match(LexType.KEYWORD, "while")) {
        parseWhileStatement();
    } else if (match(LexType.KEYWORD, "print")) {
        parsePrintStatement();
    } else {
        error("Expected a valid statement.");
    }
}

private String consumeIdentifier(String errorMessage) {
    if (check(LexType.IDENTIFIER)) {
        String name = peek().lexValue;
        advance();
        return name;
    } else {
        error(errorMessage);
        return null; // unreachable
    }
}

private String parseExpressionWithCode() {
    String left = parseTermWithCode();
    while (match(LexType.OPERATOR, "+", "-")) {
        String op = previous().lexValue;
        String right = parseTermWithCode();
        String temp = newTemp();
```

```
            threeAddressCode.add(temp + " = " + left + " " + op + " " + right);
            left = temp;
        }
        return left;
    }

    private String parseTermWithCode() {
        String left = parseFactorWithCode();
        while (match(LexType.OPERATOR, "*", "/")) {
            String op = previous().lexValue;
            String right = parseFactorWithCode();
            String temp = newTemp();
            threeAddressCode.add(temp + " = " + left + " " + op + " " + right);
            left = temp;
        }
        return left;
    }

    private String parseFactorWithCode() {
        if (match(LexType.NUMBER) || match(LexType.IDENTIFIER)) {
            return previous().lexValue;
        } else if (match(LexType.LEFT_PAREN)) {
            String expr = parseExpressionWithCode();
            consume(LexType.RIGHT_PAREN, "Expected ')' after expression.");
            return expr;
        } else {
            error("Expected number, variable, or expression.");
            return null;
        }
    }

    private void parseDeclaration() {
        String varName = consumeIdentifier("Expected variable name after 'int'.");
        if (declaredVariables.contains(varName)) {
            error("Variable '" + varName + "' already declared.");
        }
        declaredVariables.add(varName);
        consume(LexType.SEMICOLON, "Expected ';' after declaration.");
    }

    private void parseAssignment() {
        String varName = consumeIdentifier("Expected variable name.");
        if (!declaredVariables.contains(varName)) {
```

```
        error("Variable '" + varName + "' not declared.");
    }
    consume(LexType.ASSIGN_OP, "Expected '=' in assignment.");
    String result = parseExpressionWithCode();
    threeAddressCode.add(varName + " = " + result);
    consume(LexType.SEMICOLON, "Expected ';' after assignment.");
}

private void parseIfStatement() {
    consume(LexType.LEFT_PAREN, "Expected '(' after 'if'.");
    parseExpression();
    consume(LexType.RIGHT_PAREN, "Expected ')' after condition.");
    parseBlock();
    if (match(LexType.KEYWORD, "else")) {
        parseBlock();
    }
}

private void parseWhileStatement() {
    consume(LexType.LEFT_PAREN, "Expected '(' after 'while'.");
    parseExpression();
    consume(LexType.RIGHT_PAREN, "Expected ')' after condition.");
    parseBlock();
}

private void parsePrintStatement() {
    consume(LexType.LEFT_PAREN, "Expected '(' after 'print'.");
    if (check(LexType.IDENTIFIER)) {
        String varName = peek().lexValue;
        if (!declaredVariables.contains(varName)) {
            error("Semantic Error: Variable '" + varName + "' not declared.");
        }
    }
    parseExpression();
    consume(LexType.RIGHT_PAREN, "Expected ')' after expression.");
    consume(LexType.SEMICOLON, "Expected ';' after print statement.");
}


private void parseBlock() {
    consume(LexType.LEFT_BRACE, "Expected '{' to start block.");
    while (!check(LexType.RIGHT_BRACE) && !isAtEnd()) {
        parseStatement();
```

```
    }
    consume(LexType.RIGHT_BRACE, "Expected '}' to close block.");
}

// ========== EXPRESSIONS ==========

private void parseExpression() {
    parseArithmetic();
    if (match(LexType.COMPARATOR)) {
        parseArithmetic();
    }
}

private void parseArithmetic() {
    parseTerm();
    while (match(LexType.OPERATOR, "+", "-")) {
        parseTerm();
    }
}

private void parseTerm() {
    parseFactor();
    while (match(LexType.OPERATOR, "*", "/")) {
        parseFactor();
    }
}

private void parseFactor() {
    if (match(LexType.IDENTIFIER) || match(LexType.NUMBER)) {
        return;
    } else if (match(LexType.LEFT_PAREN)) {
        parseExpression();
        consume(LexType.RIGHT_PAREN, "Expected ')' after expression.");
    } else {
        error("Expected number, variable, or expression.");
    }
}

// ========== UTILITY FUNCTIONS ==========

private boolean match(LexType type, String... values) {
    if (check(type)) {
        String lexeme = peek().lexValue;
```

```
            for (String v : values) {
                if (lexeme.equals(v)) {
                    advance();
                    return true;
                }
            }
        }
        return false;
    }

    private boolean match(LexType type) {
        if (check(type)) {
            advance();
            return true;
        }
        return false;
    }

    private void consume(LexType type, String errorMessage) {
        if (check(type)) {
            advance();
        } else {
            error(errorMessage);
        }
    }

    private boolean check(LexType type) {
        return !isAtEnd() && peek().lexType == type;
    }

    private LexToken peek() {
        return tokens.get(currentIndex);
    }

    private LexToken advance() {
        if (!isAtEnd()) currentIndex++;
        return previous();
    }

    private LexToken previous() {
        return tokens.get(currentIndex - 1);
    }
```

```
    private boolean isAtEnd() {
        return currentIndex >= tokens.size();
    }

    private void error(String message) {
        System.err.println("❌ Syntax Error: " + message + " at token: " + (isAtEnd() ? "EOF" :
peek()));
        System.exit(1);
    }
}
```

<u>input.minilang file</u>

The input.minilang file contains source code written in the MiniLang programming language, a simplified language often used in compiler design assignments.

Purpose:

- To serve as input to a MiniLang compiler or interpreter
- Used to test different compiler phases: lexical analysis, parsing, semantic analysis, and code generation

5. Input Code Examples and Outputs

Example 1: Arithmetic with Precedence

MiniLang Code:

```
int a;
int b;
a = 2 + 3 * 4;
b = (2 + 3) * 4;
print(a);
print(b);
```

Output:

```
✅ Lexical Tokens:
(KEYWORD, int)
(IDENTIFIER, a)
(SEMICOLON, ;)
(KEYWORD, int)
(IDENTIFIER, b)
(SEMICOLON, ;)
(IDENTIFIER, a)
(ASSIGN_OP, =)
(NUMBER, 2)
(OPERATOR, +)
(NUMBER, 3)
(OPERATOR, *)
(NUMBER, 4)
(SEMICOLON, ;)
(IDENTIFIER, b)
(ASSIGN_OP, =)
(LEFT_PAREN, ()
(NUMBER, 2)
(OPERATOR, +)
(NUMBER, 3)
(RIGHT_PAREN, ))
(OPERATOR, *)
(NUMBER, 4)
(SEMICOLON, ;)
(KEYWORD, print)
(LEFT_PAREN, ()
(IDENTIFIER, a)
(RIGHT_PAREN, ))
(SEMICOLON, ;)
```

```
(KEYWORD, print)
(LEFT_PAREN, ()
(IDENTIFIER, b)
(RIGHT_PAREN, ))
(SEMICOLON, ;)
✅ Syntax Analysis: Passed.
Generated 3-Address Code:
t0 = 3 * 4
t1 = 2 + t0
a = t1
t2 = 2 + 3
t3 = t2 * 4
b = t3
Declared Variables (Symbol Table):
- a
- b
```

Example 2: If-Else Statement

MiniLang Code:

```
int a;
a = 5;
if (a > 3) {
   print(a);
} else {
   a = a + 1;
}
```

Output:

```
✅ Lexical Tokens:
(KEYWORD, int)
(IDENTIFIER, a)
(SEMICOLON, ;)
(IDENTIFIER, a)
(ASSIGN_OP, =)
(NUMBER, 5)
(SEMICOLON, ;)
(KEYWORD, if)
(LEFT_PAREN, ()
(IDENTIFIER, a)
(COMPARATOR, >)
(NUMBER, 3)
```

```
(RIGHT_PAREN, ))
(LEFT_BRACE, {)
(KEYWORD, print)
(LEFT_PAREN, ()
(IDENTIFIER, a)
(RIGHT_PAREN, ))
(SEMICOLON, ;)
(RIGHT_BRACE, })
(KEYWORD, else)
(LEFT_BRACE, {)
(IDENTIFIER, a)
(ASSIGN_OP, =)
(IDENTIFIER, a)
(OPERATOR, +)
(NUMBER, 1)
(SEMICOLON, ;)
(RIGHT_BRACE, })
✅ Syntax Analysis: Passed.
Generated 3-Address Code:
a = 5
t0 = a + 1
a = t0
Declared Variables (Symbol Table):
- a
```

Example 3: While Loop

MiniLang Code:

```
int count;
count = 3;
while (count > 0) {
   print(count);
   count = count - 1;
}
```

Output:

```
✅ Lexical Tokens:
(KEYWORD, int)
(IDENTIFIER, count)
(SEMICOLON, ;)
(IDENTIFIER, count)
(ASSIGN_OP, =)
```

```
(NUMBER, 3)
(SEMICOLON, ;)
(KEYWORD, while)
(LEFT_PAREN, ()
(IDENTIFIER, count)
(COMPARATOR, >)
(NUMBER, 0)
(RIGHT_PAREN, ))
(LEFT_BRACE, {)
(KEYWORD, print)
(LEFT_PAREN, ()
(IDENTIFIER, count)
(RIGHT_PAREN, ))
(SEMICOLON, ;)
(IDENTIFIER, count)
(ASSIGN_OP, =)
(IDENTIFIER, count)
(OPERATOR, -)
(NUMBER, 1)
(SEMICOLON, ;)
(RIGHT_BRACE, })
✅ Syntax Analysis: Passed.
Generated 3-Address Code:
count = 3
t0 = count - 1
count = t0
Declared Variables (Symbol Table):
- count
```

6. Error Handling Demonstration

Example 1: Undeclared Variable

```
x = 10;
```

Output:

```
✅ Lexical Tokens:
(IDENTIFIER, x)
(ASSIGN_OP, =)
(NUMBER, 10)
(SEMICOLON, ;)
❌ Syntax Error: Variable 'x' not declared. at token: (ASSIGN_OP, =)
```

Example 2: Redeclaration

```
int x;
int x;
```

Output:

```
✅ Lexical Tokens:
(KEYWORD, int)
(IDENTIFIER, x)
(SEMICOLON, ;)
(KEYWORD, int)
(IDENTIFIER, x)
(SEMICOLON, ;)
❌ Syntax Error: Variable 'x' already declared. at token: (SEMICOLON, ;)
```

Example 3: Missing Semicolon

MiniLang Code:

```
int a
a = 5;
```

Output:

```
✅ Lexical Tokens:
(KEYWORD, int)
(IDENTIFIER, a)
(IDENTIFIER, a)
(ASSIGN_OP, =)
(NUMBER, 5)
```

```
(SEMICOLON, ;)
❌ Syntax Error: Expected ';' after declaration. at token: (IDENTIFIER, a)
```

Example 4: Invalid Print Statement (missing closing parenthesis)

MiniLang Code:

```
int x;
x = 10;
print(x;
```

Output:

```
✅ Lexical Tokens:
(KEYWORD, int)
(IDENTIFIER, x)
(SEMICOLON, ;)
(IDENTIFIER, x)
(ASSIGN_OP, =)
(NUMBER, 10)
(SEMICOLON, ;)
(KEYWORD, print)
(LEFT_PAREN, ()
(IDENTIFIER, x)
(SEMICOLON, ;)
❌ Syntax Error: Expected ')' after expression. at token: (SEMICOLON, ;)
```

7. Symbol Table

The symbol table is printed at the end of successful parsing and includes all variables declared in the source file.

Example:

```
Declared Variables (Symbol Table):
- a
- b
- total
```

8. Conclusion

This project successfully demonstrates the fundamental stages of compiler construction for a simple language. Through implementing the Lexical, Syntax, Semantic, and Code Generation phases, I have gained practical knowledge in:

- Grammar design
- Recursive-descent parsing
- Symbol table management
- Error handling
- Intermediate code generation