Building a parser

Project 2

Parser - Java

Submitted to: Professor: Llesh Miraj

March, 2024

Ashley Peleg, Ariana Contes, Andres Rodriguez, Aaron Amalraj, Melodie Cornelly, Stephanie Sicilian

Table of Contents

Introduction

Parser

The parser is a vital part of a compiler that interprets and arranges code into a structured format, known as the Abstract Syntax Tree (AST), following the lexical analysis phase. By reading tokens from the lexical analyzer in the Compiler.java file, the program demonstrates the process of lexical analysis in compiling C code. The lexical analyzer breaks down the input code into individual tokens, (such as keywords, identifiers, and operators). These tokens are then passed on to the Parser class, which constructs an Abstract Syntax Tree (AST) to represent the hierarchical structure of the code. This AST serves as a visual representation of the code's syntax, making it easier to analyze and understand. The Compiler class in Compiler.java processes the tokens and AST to generate an organized output that displays the AST in a tree format through pre-order traversal. This visualization helps developers better comprehend the code's structure and aids in the debugging and optimization processes. Through the compilation of C code, Compiler.java showcases a simplified workflow of how a compiler reads and analyzes tokens to effectively transform source code into executable programs.

Purpose/ Scope of Project

The project is focused on developing a compiler that can parse and organize code written in the Java programming language. The main purpose of this project is to create a tool that can efficiently analyze, compile, and optimize Java code, transforming it into executable machine code. This compiler aims to improve the overall performance and reliability of Java applications by optimizing the code during the compilation process. The project may also involve creating a user-friendly interface for developers to interact with the compiler and provide feedback on their code. Overall, the scope of this project includes developing a robust and efficient compiler for Java code that

can streamline the development process and enhance the performance of Java applications.

Project Overview

Description of the Java Program

Compiler.java

Function: create a tool that can efficiently analyze, compile, and optimize Java code, transforming it into executable machine code

Key Features:

1. Breaks down Java code into smaller parts (tokens) for analysis
2. Checks if the code follows the correct structure and syntax
3. Verifies the meaning of the code (semantics) to ensure it makes sense
4. Improves code performance through optimizations
5. Translates code into machine-readable instructions
6. Provides helpful error messages for debugging
7. Helps identify and fix issues in the code.

Compier.java Functionality:

The functions and key features of compiler.java may include the following:

1. Lexical analysis: This function involves breaking down the input Java code into a sequence of tokens, such as identifiers, keywords, operators, and literals.

2. Syntax analysis: This function checks the sequence of tokens for correct syntax based on the rules of the Java language, ensuring that the code is structured correctly.

3. Semantic analysis: This function involves checking the meaning of the code to ensure that it adheres to the semantic rules of the Java language, such as type compatibility and variable declarations.

4. Optimization: Compiler.java includes optimization techniques to improve the performance of the generated machine code, such as constant folding, dead code elimination, and loop optimizations.

5. Code generation: This function involves translating the parsed and optimized code into executable machine code that can be run on a specific hardware platform.

6. Error handling: The compiler may include robust error-handling mechanisms to provide informative error messages to developers in case of syntax or semantic errors in their code.

7. Debugging support: Compiler.java may also offer debugging features, such as line number mappings and debugging symbols, to help developers identify and fix issues in their code.

The compiler.java file is an important component of the Java development process, responsible for the translation of readable Java source code into machine-readable instructions that can be compiled and executed by a computer. The functionality of compiler.java involves the processing of Java source code provided as input. This process begins with parsing the code to check for syntax errors, ensuring that the code follows the correct structure specified by the Java programming language

Example scenario:

1. The developer has written the source code for the application in a file called "AreaCalculator.java".
2. When the developer wants to test the application, they compile the "AreaCalculator.java" file using the compiler.java component of their Java development environment. The compiler.java file processes the source code, checking for any syntax errors or issues that may prevent the code from running correctly.
3. If the compiler.java detects any errors in the source code, it will display error messages indicating the specific lines of code that need to be corrected. The developer can then review these messages, make the necessary adjustments to the code, and recompile the file using the compiler.java component.

4. Once the code has been successfully compiled without any errors, the compiler.java generates the corresponding executable code that can be run on a computer. The developer can then execute the compiled program to calculate the area of different shapes and verify that the application works as intended.

Java Packages Utilized:

1. import java.util.ArrayList;

This package allows you to use the ArrayList data structure in your Java code.

2. import java.util.List;

This package works with List implementations such as ArrayList, LinkedList, etc. This package allows you to use List-specific methods and functionalities in your code.

3. import java.util.regex.Matcher;

This package provides classes and interfaces for dealing with regular expressions in Java.

4. import java.util.regex.Pattern;

The Pattern class is another class from the java.util.regex package that represents a compiled version of a regular expression pattern.

5. import java.nio.file.Files;

This package provides classes and interfaces for file I/O operations in Java.

6. import java.nio.file.Paths;

The Paths class is part of the java.nio.file package and provides methods for converting a string or URI to a Path object.

7. `import java.io.IOException;`

This package provides classes for handling input/output operations in Java.

Token Types: enum TokenType is used to define the different token types.

- KeyWords: Keywords are reserved words in Java with predefined meanings that cannot be used as identifiers.They are identified via a regular expression pattern
  that matches specific keywords. Words Identified: int, float, char, if, else, while, for, and return
- Identifiers: Identifiers are names assigned to variables, methods, classes, and other entities in Java programs. They are recognized as sequences of characters that must begin with a letter or underscore, followed by zero or more letters, digits, or underscores. In the LexicalAnalyzer.java program, identifiers are identified using the following regular expression pattern:
  - \\b[a-zA-Z_]\\w*\\b

- Tokens conforming to this pattern, representing identifiers, are categorized as identifier tokens during the tokenization process.
- Elements above explained:
    - \\b: Represents a word boundary, ensuring that the identifier starts and ends with a complete word.
    - [a-zA-Z_]: Matches a single character that is either a lowercase letter (a-z), an uppercase letter (A-Z), or an underscore (_). This ensures that the identifier starts with a letter or underscore.
    - \\w*: Matches zero or more word characters, including letters, digits, and underscores. This allows identifiers to contain letters, digits, or underscores after the first character.
  - Constants: Constants represent fixed values in Java programs, such as numeric literals.
    - Identified as sequences of digits (\d+).
    - Tokens conforming to this pattern, representing constants, are categorized as constant tokens during the tokenization process.
  - Operators: Operators are symbols used to perform operations on operands in Java programs. Tokens conforming to patterns representing these symbols are categorized as operator tokens during the tokenization process. Symbols include:
    - Arithmetic operators: +, -, *, /
    - Assignment operator: =
    - Comparison operators: ==, !=, <, >, <=, >=
    - Logical operators: &&, ||
  - Punctuation: Punctuation symbols are special characters used to punctuate and structure Java code. Tokens conforming to patterns representing these symbols are categorized as punctuation tokens during the tokenization process. Symbols include:
    - Semicolons: ;
    - Commas: ,
    - Parentheses: (,)

- Braces: {, }
- Square brackets: [,]

Conclusion

The development of a parser and compiler for Java code is crucial in ensuring the efficient analysis, compilation, and optimization of Java programs. The Compiler.java program breaks down Java code into tokens, checks for syntactical correctness, and constructs an Abstract Syntax Tree (AST) to represent the hierarchical structure of the code. Through semantic analysis, optimization, and code generation, the compiler ensures that the Java code generates high-performance machine-readable instructions. The use of Java packages such as ArrayList, List, regex Matcher and Pattern, and NIO files provides functionalities for file I/O operations, regular expression handling, and data structure manipulation within the compiler. The development of a compiler like Compiler.java improves the development process, enhances code performance, and provides valuable error-handling and debugging support for Java developers. The project's focus is creating a user-friendly interface and optimizing the compilation process, ultimately aiming to improve the reliability and efficiency of Java applications.

Appendix

1. Source Code for Compiler.java:

```
import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.IOException;

/*to test it do:
javac Compiler.java
java Compiler sum.c */

// Token types
enum TokenType {
PREPROCESSOR_DIRECTIVE,
KEYWORD,
IDENTIFIER,
CONSTANT,
OPERATOR,
PUNCTUATION
}

// Token class
class Token {
TokenType type;
String value;

Token(TokenType type, String value) {
this.type = type;
this.value = value;
}
}

// Lexical Analyzer class
class LexicalAnalyzer {
public static ArrayList<Token> tokenize(String code) {
ArrayList<Token> tokens = new ArrayList<>();
// Define token patterns
String[] patterns = {
"#include <\\w+\\.h>",
"\\b(int|float|char|if|else|while|for|return)\\b", // keywords
"\\b[a-zA-Z_]\\w*\\b", // identifiers
"\\b\\d+\\b", // constants
"\\+|-|\\*|/|=|==|!=|<|>|<=|>=|\\|\\||&&", // operators
"[;,()\\[\\]{}]" // punctuation
};
```

```
Pattern pattern = Pattern.compile(String.join("|", patterns));

Matcher matcher = pattern.matcher(code);
while (matcher.find()) {
String matched = matcher.group();
TokenType type;
if (matched.matches("#include <\\w+\\.h>")) {
type = TokenType.PREPROCESSOR_DIRECTIVE;
} else if (matched.matches("\\b(int|float|char|if|else|while|for|return)\\b")) {
type = TokenType.KEYWORD;
} else if (matched.matches("\\b[a-zA-Z_]\\w*\\b")) {
type = TokenType.IDENTIFIER;
} else if (matched.matches("\\b\\d+\\b")) {
type = TokenType.CONSTANT;
} else if (matched.matches("\\+|-|\\*|/|=|==|!=|<|>|<=|>=|\\|\\||&&")) {
type = TokenType.OPERATOR;
} else if (matched.matches("[;,()\\[\\]{}]")) {
type = TokenType.PUNCTUATION;
} else {
throw new IllegalStateException("Unrecognized token: " + matched);
}
tokens.add(new Token(type, matched));
}

return tokens;
}
}

// Parser class
class Parser {
private ASTNode parseProgram(ArrayList<Token> tokens) {
ASTNode root = new ASTNode("Program", "");

ASTNode currentBlock = root; // Track the current block being filled

// Iterate through tokens to build the AST
for (int i = 0; i < tokens.size(); i++) {
Token token = tokens.get(i);
if (token.type == TokenType.PREPROCESSOR_DIRECTIVE) {
root.addChild(new ASTNode("Preprocessor Directive", token.value));
} else if (token.value.equals("int") && i + 1 < tokens.size() && tokens.get(i +
1).value.equals("main")) {
```

```
// Found the main function declaration, create a block for its body
ASTNode mainBlock = new ASTNode("Block", "");
currentBlock.addChild(mainBlock);
currentBlock = mainBlock;
} else if (token.value.equals("{")) {
// Found the start of a new block, create a nested block
ASTNode newBlock = new ASTNode("Block", "");
currentBlock.addChild(newBlock);
currentBlock = newBlock;
} else if (token.value.equals("}")) {
// Found the end of the current block, move back to the parent block
if (currentBlock != root) {
currentBlock = getParentBlock(root, currentBlock);
}
} else {
// Add the token as a child of the current block
currentBlock.addChild(new ASTNode(token.type.toString(), token.value));
}
}

return root;
}

private ASTNode getParentBlock(ASTNode root, ASTNode currentBlock) {
// Traverse the tree to find the parent block of the current block
if (root.children.contains(currentBlock)) {
return root;
}
for (ASTNode child : root.children) {
ASTNode parent = getParentBlock(child, currentBlock);
if (parent != null) {
return parent;
}
}
return null;
}

public ASTNode buildAST(ArrayList<Token> tokens) {
return parseProgram(tokens);
}
}
```

```
// Compiler class
public class Compiler {
public static void main(String[] args) {
if (args.length != 1) {
System.err.println("Usage: java Compiler <path_to_c_file>");
return;
}

String filePath = args[0];
try {
// Read the content of the C file
List<String> lines = Files.readAllLines(Paths.get(filePath));
StringBuilder codeBuilder = new StringBuilder();
for (String line : lines) {
codeBuilder.append(line).append("\n");
}
String code = codeBuilder.toString();

// Tokenize the C code
ArrayList<Token> tokens = LexicalAnalyzer.tokenize(code);

// Build the Abstract Syntax Tree (AST)
Parser parser = new Parser();
ASTNode ast = parser.buildAST(tokens);

// Print AST
System.out.println("Abstract Syntax Tree (AST):");
ast.printTree();
System.out.println("Parsing successful");
} catch (IOException e) {
System.err.println("Error reading file: " + e.getMessage());
}
}
}

// AST node classes
class ASTNode {
String type;
String value;
ArrayList<ASTNode> children;

ASTNode(String type, String value) {
```

```
this.type = type;
this.value = value;
this.children = new ArrayList<>();
}

void addChild(ASTNode child) {
children.add(child);
}

void print(String prefix, boolean isTail) {
System.out.println(prefix + (isTail ? "└── " : "├── ") + type + ": " + value);
for (int i = 0; i < children.size() - 1; i++) {
children.get(i).print(prefix + (isTail ? "    " : "│   "), false);
}
if (children.size() > 0) {
children.get(children.size() - 1).print(prefix + (isTail ? "    " : "│   "), true);
}
}

void printTree() {
print("", true);
}
}
```