Ben Pallotti

Concepts of Programming Languages-4308-W02­ Summer 2023

**P3-Scanner-Parser-Interpreter**

7/16/23  
Professor Sharon Perry  
100% Complete

**SCANNER:**

1. **Introduction**

This project is to develop a scanner for a minimal form of the Julia language. The scanner is responsible for tokenizing the input Julia file, extracting individual tokens, and providing them to the parser for further processing. The scanner is implemented in Java.

1. **Scanner Implementation**

The scanner implementation consists of the JuliaScanner class, which uses the java.util.Scanner class to read the Julia input file and extract tokens. The getNextToken() method retrieves the next token from the input file using the next() method of Scanner. If no more tokens are available, it returns null to indicate the end of the file.

1. **Execution and Output**

The program reads tokens from a file and retrieves a token, then the next token, get the corresponding lexeme symbol, and then its . The program’s main method demonstrates the scanner by printing the input file contents and the lexemes with their symbols. The lexeme-symbol mappings are defined in a hashmap. Here is the execution output:

A screenshot of a computer program

Description automatically generated

1. **Conclusion**

The getNextToken() method retrieves the next token from the input file using the next() method of Scanner. If no more tokens are available, it returns null to indicate the end of the file. The program reads tokens from a file and retrieves a token, then the next token, get the corresponding lexeme symbol, and then its line number. The program's main method demonstrates the scanner by printing the input file contents and the lexemes with their symbols. The lexeme-symbol mappings are defined in a hashmap.

Source Code:  
package projects;

import java.io.File;

import java.io.FileNotFoundException;

import java.util.HashMap;

import java.util.Scanner;

public class JuliaScanner {

private Scanner fileScanner;

private int lineNumber;

private String currentToken;

private HashMap<String, String> lexemeSymbolTable;

public String getNextToken() {

// continuously loop while there are more tokens to read from the file

while (fileScanner.hasNext()) {

// Read the next token from the file

String token = fileScanner.next();

// check if the token is not blank (contains non-whitespace characters)

if (!token.isBlank()) {

// assign the current token to the field (no longer used in the code)

currentToken = token;

// return the token as the next token in the sequence

return token;

}

}

// return null if there are no more tokens in the file

return null;

}

public String getLexemeSymbol(String lexeme) {

return lexemeSymbolTable.getOrDefault(lexeme, "IDENTIFIER");

}

public int getLineNumber() {

// return the current line number being processed

return lineNumber;

}

public static void main(String[] args) {

try {

Scanner userInput = new Scanner(System.in);

System.out.println("Enter File Location:");

String filePath = userInput.nextLine(); // Read user input

JuliaScanner scanner = new JuliaScanner(filePath);

String token;

System.out.println("\nInput File Contents:");

// print the input file contents

while ((token = scanner.getNextToken()) != null) {

System.out.println(token);

}

scanner.close();

System.out.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

System.out.println("Lexeme Symbol");

System.out.println("-------------------------");

// reinitialize the scanner for lexeme and symbol printing

scanner = new JuliaScanner(filePath);

// print the lexemes and their symbols

while ((token = scanner.getNextToken()) != null) {

int lineNumber = scanner.getLineNumber();

String symbol = scanner.getLexemeSymbol(token);

System.out.printf("%-15s %s\n", token, symbol);

}

scanner.close();

} catch (FileNotFoundException e) {

System.err.println("File not available: " + e.getMessage());

}

}

// lexeme table as seen in the sample output

public JuliaScanner(String filename) throws FileNotFoundException {

fileScanner = new Scanner(new File(filename));

lineNumber = 1;

currentToken = null;

lexemeSymbolTable = new HashMap<>();

lexemeSymbolTable.put("function", "FUNCTION");

lexemeSymbolTable.put("(", "OPEN\_PARENTHESIS");

lexemeSymbolTable.put(")", "CLOSE\_PARENTHESIS");

lexemeSymbolTable.put("=", "ASSIGNMENT\_OPERATOR");

lexemeSymbolTable.put("!=", "NOT\_EQUAL\_OPERATOR");

lexemeSymbolTable.put("if", "IF");

lexemeSymbolTable.put("then", "THEN");

lexemeSymbolTable.put("print", "FUNCTION");

lexemeSymbolTable.put("else", "ELSE");

lexemeSymbolTable.put("end", "END");

}

public void close() {

fileScanner.close();

}

}

**PARSER:**

**Introduction**

The parser works in conjunction with the scanner from P1 to process the tokens produced by the scanner and identify the structure and syntax of the Julia program. The parser is implemented in Java.

**Parser Implementation**

Takes a file as input and parses the file line by line. For each line, the parser checks if the current token is a keyword, an identifier, or an error. If the current token is a keyword, the parser prints the keyword and its symbol. If the current token is an identifier, the parser prints the identifier and its symbol. If the current token is an error, the parser prints an error message.

**Execution and Output**

To demonstrate the execution of the parser the input Julia files named Test1.jl were used. The input files were given by the assignment.

Here is the execution output:

A screen shot of a computer screen

Description automatically generated

**Conclusion**

The parser implementation complements the scanner by analyzing the syntax of the Julia program. It successfully recognizes the statements and constructs. The recursive descent parsing technique handles the different language constructs. The parser has been tested with a sample text and produces the expected output.

Source code:  
  
package projects;

package projects;

/\*

\* Class: CS 4308 Section W02

\* Term: Summer 2023

\* Name: Ben Pallotti

\* Instructor: Sharon Perry

\* Project: P2-Parser

\*/

import java.io.File;

import java.io.FileNotFoundException;

import java.util.HashMap;

import java.util.Scanner;

public class JuliaParser {

private Scanner fileScanner;

private int lineNumber;

private String currentToken;

private HashMap<String, String> lexemeSymbolTable;

public void parse() throws FileNotFoundException {

while (fileScanner.hasNext()) {

currentToken = fileScanner.next();

lineNumber++;

// Check if the current token is a keyword

if (lexemeSymbolTable.containsKey(currentToken)) {

System.out.println(currentToken + " - " + lexemeSymbolTable.get(currentToken));

} else {

// Check if the current token is an identifier

if (Character.isLetter(currentToken.charAt(0))) {

System.out.println(currentToken + " - IDENTIFIER");

} else {

// The current token is not a keyword or an identifier, so it is an error

System.out.println("Error: Invalid token " + currentToken + " at line " + lineNumber);

}

}

}

}

public static void main(String[] args) throws FileNotFoundException {

Scanner userInput = new Scanner(System.in);

System.out.println("Enter File Location:");

String filePath = userInput.nextLine(); // Read user input

JuliaParser parser = new JuliaParser(filePath);

parser.parse();

userInput.close();

}

public JuliaParser(String filename) throws FileNotFoundException {

fileScanner = new Scanner(new File(filename));

lineNumber = 1;

currentToken = null;

lexemeSymbolTable = new HashMap<>();

lexemeSymbolTable.put("function", "FUNCTION");

lexemeSymbolTable.put("(", "OPEN\_PARENTHESIS");

lexemeSymbolTable.put(")", "CLOSE\_PARENTHESIS");

lexemeSymbolTable.put("=", "ASSIGNMENT\_OPERATOR");

lexemeSymbolTable.put("!=", "NOT\_EQUAL\_OPERATOR");

lexemeSymbolTable.put("+", "ADDITION\_OPERATOR");

lexemeSymbolTable.put("+=", "ADDITION\_ASSIGNMENT\_OPERATOR");

lexemeSymbolTable.put("<", "LESS\_THAN\_OPERATOR");

lexemeSymbolTable.put("==", "EQUAL\_TO\_OPERATOR");

lexemeSymbolTable.put("if", "IF");

lexemeSymbolTable.put("then", "THEN");

lexemeSymbolTable.put("print", "FUNCTION");

lexemeSymbolTable.put("else", "ELSE");

lexemeSymbolTable.put("end", "END");

lexemeSymbolTable.put("//", "COMMENT");

//Loop to parse numbers

for (int i = 0; i < 20; i++) {

lexemeSymbolTable.put(Integer.toString(i), "NUMBER");

}

}

}

**INTERPRETER:**

**Introduction:**

This is a Java program designed to execute Julia code. It uses a symbol table to store identifier values and can handle basic programs.

**Interpreter Implementation:**

Using a recursive descent parser, the interpreter breaks down Julia code into tokens and interprets them according to the language's grammar. It also employs a symbol table for efficient identifier value storage.

**Execution and Output:**

The interpreter executes the code by calling appropriate functions for each token, with results printed to the console. It can also be used for debugging, providing error messages for invalid Julia code.

**Here is the execution output:**

A screenshot of a computer program

Description automatically generated

**Conclusion:**

The program that enables the execution of Julia code. Implemented in Java, it effectively utilizes a symbol table to store identifier values and employs a recursive descent parser to interpret code according to the grammar of the language.

Source Code:

package projects;

import java.io.File;

import java.io.FileNotFoundException;

import java.util.HashMap;

import java.util.Scanner;

/\*

\* Class: CS 4308 Section W02

\* Term: Summer 2023

\* Name: Ben Pallotti

\* Instructor: Sharon Perry

\* Project: P3-Interpreter

\*/

public class JuliaInterpreter {

private Scanner fileScanner;

private int lineNumber;

private String currentToken;

private HashMap<String, String> lexemeSymbolTable;

private HashMap<String, Integer> identifierValues;

public void parse() throws FileNotFoundException {

while (fileScanner.hasNext()) {

currentToken = fileScanner.next();

lineNumber++;

// Check if the current token is a keyword

if (lexemeSymbolTable.containsKey(currentToken)) {

System.out.println(currentToken + " - " + lexemeSymbolTable.get(currentToken));

} else {

// Check if the current token is an identifier

if (Character.isLetter(currentToken.charAt(0))) {

if (identifierValues.containsKey(currentToken)) {

System.out.println(currentToken + " - " + identifierValues.get(currentToken));

} else {

// The identifier is not defined yet, so add it to the symbol table

identifierValues.put(currentToken, 0);

System.out.println(currentToken + " - IDENTIFIER");

}

} else {

// The current token is not a keyword or an identifier, so it is an error

System.out.println("Error: Invalid token " + currentToken + " at line " + lineNumber);

}

}

}

}

public static void main(String[] args) throws FileNotFoundException {

Scanner userInput = new Scanner(System.in);

System.out.println("Enter File Location:");

String filePath = userInput.nextLine(); // Read user input

JuliaInterpreter interpreter = new JuliaInterpreter(filePath);

interpreter.parse();

userInput.close();

}

public JuliaInterpreter(String filename) throws FileNotFoundException {

fileScanner = new Scanner(new File(filename));

lineNumber = 1;

currentToken = null;

lexemeSymbolTable = new HashMap<>();

lexemeSymbolTable.put("function", "FUNCTION");

lexemeSymbolTable.put("(", "OPEN\_PARENTHESIS");

lexemeSymbolTable.put(")", "CLOSE\_PARENTHESIS");

lexemeSymbolTable.put("=", "ASSIGNMENT\_OPERATOR");

lexemeSymbolTable.put("!=", "NOT\_EQUAL\_OPERATOR");

lexemeSymbolTable.put("+", "ADDITION\_OPERATOR");

lexemeSymbolTable.put("+=", "ADDITION\_ASSIGNMENT\_OPERATOR");

lexemeSymbolTable.put("<", "LESS\_THAN\_OPERATOR");

lexemeSymbolTable.put("==", "EQUAL\_TO\_OPERATOR");

lexemeSymbolTable.put("if", "IF");

lexemeSymbolTable.put("then", "THEN");

lexemeSymbolTable.put("print", "FUNCTION");

lexemeSymbolTable.put("else", "ELSE");

lexemeSymbolTable.put("end", "END");

lexemeSymbolTable.put("//", "COMMENT");

//Loop to parse numbers

for (int i = 0; i < 20; i++) {

lexemeSymbolTable.put(Integer.toString(i), "NUMBER");

}

identifierValues = new HashMap<>();

}

}