Lexical Analyzer

Project 1

Lexical Analyzer - Java

Submitted to: Professor: Llesh Miraj

February, 2024

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

Table of Contents

Introduction

Parser

A ***parser*** is pivotal in computer science, serving as a program or part of a program that analyzes the structure of text based on a programming language. Its fundamental role is to deconstruct a sequence of symbols into component parts, strictly following the rules of a programming language. This ability renders parsers essential in a variety of contexts: from analyzing and converting source code syntax for compiler processing to executing input programs directly in interpreters without needing compilation. Additionally, parsers are instrumental in extracting structured information from unstructured text and analyzing natural language sentences to understand their structure and meaning.

Purpose/ Scope of Project

This project centers on the practical application of parsing principles within the realm of C programming. It encompasses the development and analysis of two distinct programs: ***sum.c*** and ***LexicalAnalyzer.java***. sum.c is designed as a straightforward application to demonstrate basic C functionalities, including user input handling and simple arithmetic operations. In contrast, LexicalAnalyzer.java ventures into the more complex sphere of lexical analysis, a crucial preliminary stage in the parsing process of compilers. Its goal is to tokenize C source code, categorizing it into various token types like keywords, identifiers, constants, operators, and punctuation.

The objective of this report is to thoroughly detail the design, functionality, and code organization of these two programs. This approach not only illuminates the foundational and advanced aspects of parsing and lexical analysis but also underscores their significance in programming and compiler design. sum.c introduces basic programming constructs, while LexicalAnalyzer.java offers a deeper dive into the intricate process of preparing code for compilation through tokenization.

The sections that follow will provide an in-depth look at the design and functionality of each program, shedding light on their specific code organization and their integral role in the broader context of parsing and lexical analysis.

Project Overview

Description of the C Programs:

1. sum.c:
    a. Function: A user-interactive application for calculating the sum of two integers provided by the user.
    b. Key Features:
        i. Handles user input using the Scanner class.
        ii. Executes basic arithmetic operations.
        iii. Introduces exception handling mechanisms.

iv. Demonstrates core programming constructs such as variables, data types, and control structures (loops, conditionals).

1. LexicalAnalyzer.java:
    a. Function: A more complex program designed for tokenizing C source code.
    b. Key Features:
        i. Demonstrates lexical analysis within the context of programming language compilers.
        ii. Employs regular expressions to identify token types like keywords, identifiers, constants, operators, and punctuation, showcasing the importance of efficient pattern matching.
        iii. Provides a practical example of source code breakdown into tokens, aiding the understanding of compilation's initial stages.

Functionality

sum.c Functionality

- User Input Handling:

  The sum.c program efficiently handles user input through the utilization of the Scanner class, a standard C utility designed for parsing primitive types and strings from the input stream. Within the program, the Scanner object is instantiated to facilitate interaction with the standard input stream (System.in). This allows users to input integers directly from the console.

  Upon execution, the program prompts the user to enter the first integer, awaiting input. The Scanner class reads the input stream and parses the provided integer, storing it in the variable num1. Similarly, the program prompts the user to enter the second integer, and the Scanner class retrieves and parses this input, storing it in the variable num2.

  The interaction with the Scanner class ensures efficient and reliable user input handling, allowing for seamless integration of user-provided values into the arithmetic operations performed by the program. Additionally, the program may implement error handling mechanisms through the Scanner class to manage unexpected inputs or handle exceptions gracefully, thereby enhancing the robustness and user experience of the application.

- Arithmetic Operations:
    - Addition:
        - The program adds the two input integers using the + operator.
        - The operation is straightforward and doesn't involve complex algorithms.
        - The result is stored in the variable sum.

.

- Exception Handling:

    - Try-Catch Blocks:
        - The program implements exception handling mechanisms using try-catch blocks to manage potential errors during user input.
        - Specifically, the Scanner.nextInt() method, used to read integer input from the user, can throw InputMismatchException if the input provided is not an integer.
        - To handle this scenario, the program encloses the nextInt() method call within a try block and catches any InputMismatchException that may occur.

- Core Programming Constructs:
    - Variables:
        - Used to store input integers (num1 and num2) and the sum (sum).
        - Facilitate dynamic data storage during program execution.
    - Data Types:
        - int: Represents integer values for input and calculations.
        - Ensures consistent handling of numerical data.
    - Input Handling:
        - Relies on the Scanner class to manage user input.
        - Demonstrates how the program interacts with users.

LexicalAnalyzer.java Functionality

- Tokenization Process:

  In a lexical analyzer, "tokens" represent the smallest meaningful units of the source code, categorized into types such as keywords, identifiers, operators, punctuation symbols, literals (like numbers and strings), and other language-specific constructs. This process, known as tokenization, is vital in the initial stages of compilation, where the source code undergoes analysis and categorization into meaningful units for further processing.

  The program, LexicalAnalyzer.java, utilizes regular expressions to identify and extract tokens from the input Java source code. Regular expressions serve as patterns to match character combinations in strings, enabling efficient identification of tokens based on predefined rules. The tokenization algorithm iterates through the input source code, searching for patterns that match predefined token types using regular expressions. Upon finding a match, the program identifies the corresponding token type and adds the token to the list of extracted tokens

- **Java Packages Utilized:**

1. **java.util.regex.Matcher, and java.util.regex.Pattern** are used to work with regular expressions. Patterns are used for creating the regex pattern we are looking for, and it has a method called compile used for indicating the pattern. A matcher is a package that has a method called find to see if the given input matches the pattern, and other methods.

2. **java.io.IOException** is used for exception handling, in our program(LexicalAnalyzer.java) to catch an error and print out that error, using the getMessage() method, and throw an IllegalStateException in the case that we get an unrecognizable token.

3. **java.nio.file.Files and java.nio.file.Paths** are used to read the file sum.c in its respective path.

4. **java.util.ArrayList** is used to store all of the tokens found.

class Token defines a token, by having a TokenType (which is from the enum TokenType), and its associated value which is a String.

public class LexicalAnalyzer: *which contains*:

1. **private static final** (private-restricting visibility only within the class, static meaning these variables are for the whole class and not just an instance and final meaning that once initialized these values cannot be changed) defines a string array called **patterns** which define the **patterns to be recognized using regex** (once string/element pattern in array) for each tokentype.
2. **public static ArrayList<Token> tokenize(String program)** is a static method called tokenize, that takes in a String called ***"program"*** and returns an ArrayList of Token objects, which were defined in the class Token. It does this firstly, by creating an ArrayList of Token objects called tokens. Then, Pattern pattern = Pattern.compile(String.join("|", patterns)); creates a Pattern object by using the compile method to join the patterns string array using **| as a delimiter** for each element in that array. | is used as a delimiter because in Java regex it means to find "a match for any one of the patterns separated by | as in: cat|dog|fish". After this a matcher object is created by using our pattern string which is the one we just created and using the matcher object to create a matcher object by comparing pattern to String program which is the input that is received in this function. A while loop is then used to see if the matcher object which has information on the String ***program*** has any of the tokens from ***pattern*** the while condition uses the find() method. If find () is true it goes into the while loop it uses a method called group which returns the substring from ***program*** that matched with ***pattern***. It then checks this substring to find its corresponding tokentype, if it does not apply to any of the tokentypes regex or a matching error occurs an **IllegalStateException** is thrown. The substring if it has a tokentype is added to the arraylist and the process repeats iterating over each match with the program string by using the matcher object.
3. **Main method:** defines a file path. Uses a try catch statement first, Files.readAllBytes(Paths.get(filePath)) reads the path object as a byte array which is then converted to a string object (called program) because it is initialized as that. Then, an arraylist of tokens is created and assigned to a call to the method tokenize with our string program (which is the information from our file). Then a for loop is used to print each token with its type and value. If this does not work for whatever reason, there is a catch statement that catches exceptions/errors.

- **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

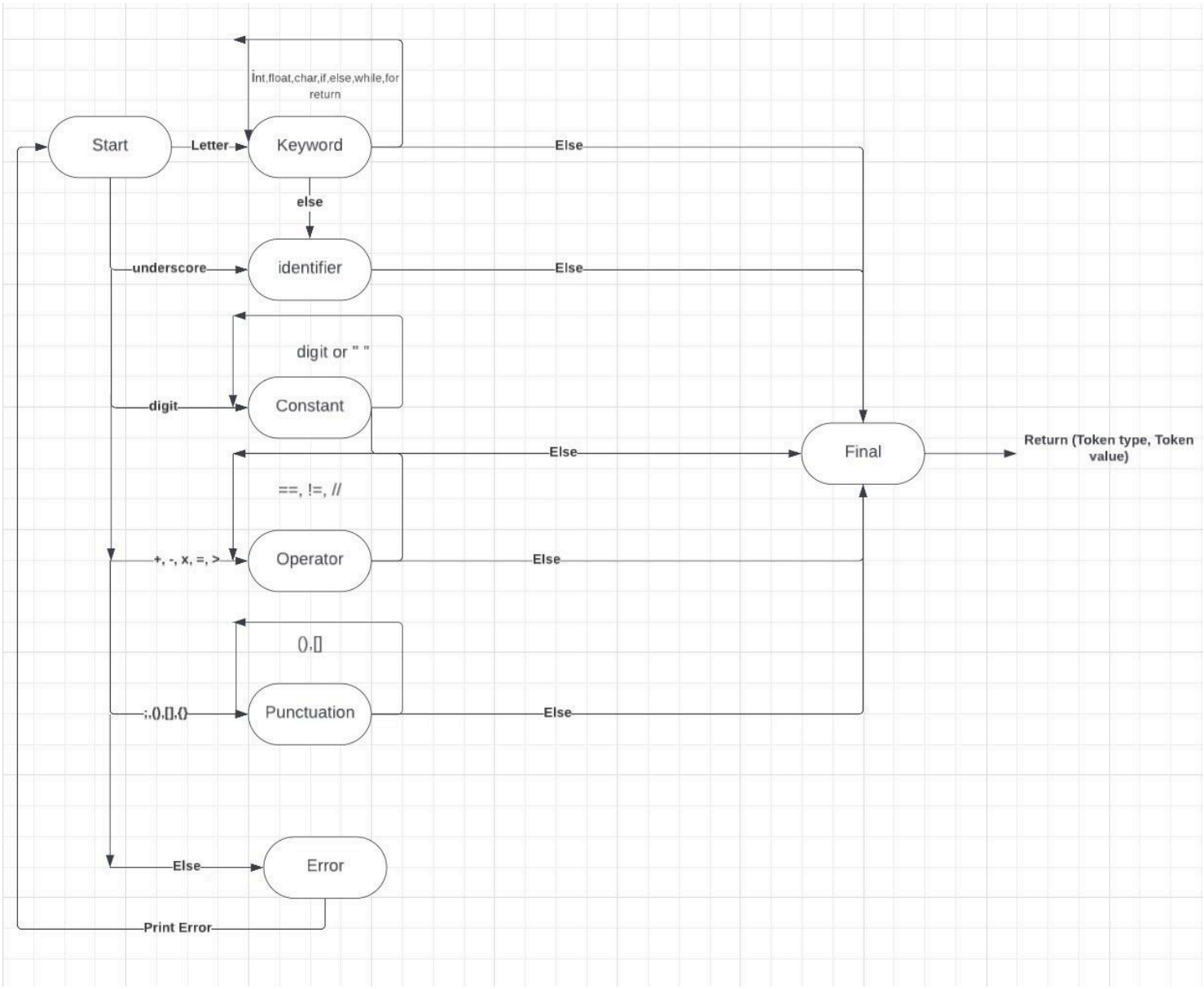
- Identifiers: Identifiers are names assigned to variables, methods, classes, and other entities in Java programs. They are recognized as sequences of characters that must begin with a letter or underscore, followed by zero or more letters, digits, or underscores. In the LexicalAnalyzer.java program, 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: [,]

State Diagram



Conclusion

This project provided an illustrative overview of lexical analysis and its role in parsing through the development of two Java programs - sum.c and LexicalAnalyzer.java. sum.c served as an introductory example to demonstrate core programming concepts like variable declaration, data types, arithmetic operations, user input handling, and exception handling in Java.

The more complex LexicalAnalyzer.java program provided practical insight into the lexical analysis phase of compilers. It highlighted the significance of breaking down source code into meaningful tokens through efficient pattern matching and regular expressions. The tokenization process categorized tokens into keywords, identifiers, constants, operators, and punctuation. This breakdown of source code is an essential first step in parsing and compiling programs.

Overall, this project underscored the importance of lexical analysis in the parsing process. The tokenization example in LexicalAnalyzer.java provided a useful reference for the techniques involved in this initial stage of compiling source code. The report detailed the key components of the program - the matching rules for different token types, the categorization process, and the role played by regular expressions in efficient pattern matching.

Through the development and analysis of these two programs, the project served as an effective learning tool to understand core programming concepts, as well as gain insight into advanced parsing mechanisms like lexical analysis. It illustrated the practical application of foundational and complex aspects of parsers within the domain of C programming.

Appendix

1. Source Code for sum.c:

```c
#include <stdio.h>

int main() {
  int num1, num2, sum;

  printf("Enter the first integer: ");
  scanf("%d", &num1);

  printf("Enter the second integer: ");
  scanf("%d", &num2);

  sum = num1 + num2;
```

```
  printf("The sum of %d and %d is %d\n", num1, num2, sum);

  return 0;
}
```

2. Source Code for Lexical Analyzer.java:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.IOException;
import java.util.ArrayList;



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

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

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

public class LexicalAnalyzer {

  // Define token patterns
  private static final String[] patterns = {
      "\\b(int|float|char|if|else|while|for|return)\\b", // keywords
      "\\b[a-zA-Z_]\\w*\\b", // identifiers
      "\\b\\d+\\b", // constants
```

```
        "\\+|-|\\*|/|=|==|!=|<|>|<=|>=|\\|\\||&&", // operators
        "[;,()\\[\\]{}]" // punctuation
    };

    public static ArrayList<Token> tokenize(String program) {
        ArrayList<Token> tokens = new ArrayList<>();
        Pattern pattern = Pattern.compile(String.join("|", patterns));

        Matcher matcher = pattern.matcher(program);
        while (matcher.find()) {
            String matched = matcher.group();
            TokenType type;
            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;
    }

    public static void main(String[] args) {
        String filePath = "/Users/arianacontes/CS361/sum.c";

        try {
            String program = new String(Files.readAllBytes(Paths.get(filePath)));

            ArrayList<Token> tokens = tokenize(program);

            for (Token token : tokens) {
                System.out.println("(" + token.type + ", " + token.value + ")");
            }
```

```
    } catch (IOException e) {
        System.err.println("Error reading file: " + e.getMessage());
    }
  }
}
```

Sample Output from Lexical Analyzer.java:

(IDENTIFIER, include)
(OPERATOR, <)
(IDENTIFIER, stdio)
(IDENTIFIER, h)
(OPERATOR, >)
(KEYWORD, int)
(IDENTIFIER, main)
(PUNCTUATION, ()
(PUNCTUATION,))
(PUNCTUATION, {)
(KEYWORD, int)
(IDENTIFIER, num1)
(PUNCTUATION, ,)
(IDENTIFIER, num2)
(PUNCTUATION, ,)
(IDENTIFIER, sum)
(PUNCTUATION, ;)
(IDENTIFIER, printf)
(PUNCTUATION, ()
(IDENTIFIER, Enter)
(IDENTIFIER, the)
(IDENTIFIER, first)
(IDENTIFIER, integer)
(PUNCTUATION,))
(PUNCTUATION, ;)
(IDENTIFIER, scanf)
(PUNCTUATION, ()
(IDENTIFIER, d)
(PUNCTUATION, ,)
(IDENTIFIER, num1)

(PUNCTUATION,))
(PUNCTUATION, ;)
(IDENTIFIER, printf)
(PUNCTUATION, ()
(IDENTIFIER, Enter)
(IDENTIFIER, the)
(IDENTIFIER, second)
(IDENTIFIER, integer)
(PUNCTUATION,))
(PUNCTUATION, ;)
(IDENTIFIER, scanf)
(PUNCTUATION, ()
(IDENTIFIER, d)
(PUNCTUATION, ,)
(IDENTIFIER, num2)
(PUNCTUATION,))
(PUNCTUATION, ;)
(IDENTIFIER, sum)
(OPERATOR, =)
(IDENTIFIER, num1)
(OPERATOR, +)
(IDENTIFIER, num2)
(PUNCTUATION, ;)
(IDENTIFIER, printf)
(PUNCTUATION, ()
(IDENTIFIER, The)
(IDENTIFIER, sum)
(IDENTIFIER, of)
(IDENTIFIER, d)
(IDENTIFIER, and)
(IDENTIFIER, d)
(IDENTIFIER, is)
(IDENTIFIER, d)
(IDENTIFIER, n)
(PUNCTUATION, ,)
(IDENTIFIER, num1)
(PUNCTUATION, ,)

(IDENTIFIER, num2)
(PUNCTUATION, ,)
(IDENTIFIER, sum)
(PUNCTUATION,))
(PUNCTUATION, ;)
(KEYWORD, return)
(CONSTANT, 0)
(PUNCTUATION, ;)
(PUNCTUATION, })