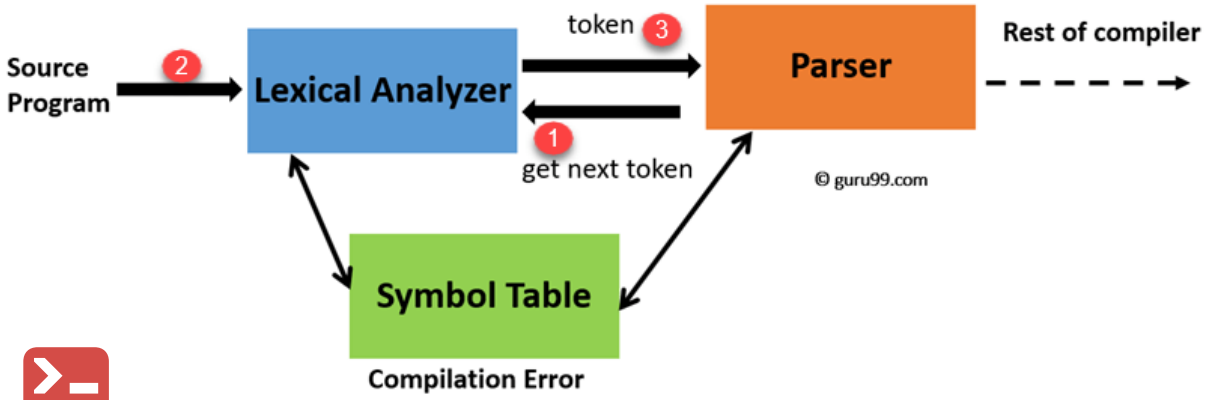


Assignment 02

CSCS 290: Introduction to Java Programming

Deadline: 13th August 2023

➔ Plagiarism is absolutely forbidden.
➔ Do not use any programming constructs that are not covered in class.
➔ Use only what has been covered in class (OOP, File handling)
➔ Submit the repository link of your assignment (pushed on Github) along with the actual assignment on classroom.
➔ **Your program must not crash**. If the user enters wrong input, your program should handle it and asks for the input again.

Implementing a Scanner (Lexical Analyzer) in Java

Introduction

In the process of compiling a program, the first step is lexical analysis. A lexical analyzer scans the source code and converts a sequence of characters into a sequence of tokens, or lexemes. A lexeme is a minimal unit of the source code that

conveys meaning, such as a keyword, an identifier, or an operator. This assignment focuses on creating a lexical analyzer to process and analyze source code files using object-oriented programming concepts in Java.

For example, in the following Java code snippet:

```
int x = 5;
```

The lexemes would be `int`, `x`, `=`, and `5`. The lexical analyzer identifies these lexemes and passes them to the next phase of the compiler, the parsing phase.

During lexical analysis, the lexer/scanner also discards any whitespace, comments, and other characters that are not relevant to the syntax of the program.

The role of the lexical analyzer is critical to the overall success of the compiler, as it is responsible for identifying and grouping the lexemes into meaningful units that can be parsed and compiled into machine code.

Objective

In this assignment, you will create an object-oriented Java program that reads a source code file, processes it according to the given specifications, and analyzes the lexemes in the code. The program will consist of three tasks.

Instructions

1. Use an appropriate IDE or text editor to write your Java code.
2. Organize your code using classes and separate files for each class.
3. Make sure to include appropriate packages, classes, and functions in your program.
4. Use appropriate object-oriented programming concepts.
5. Write proper documentation/commenting with step-by-step instructions.
6. Test your program with the given sample codes and ensure that it produces the expected output.

Classes

You must create separate classes for each task, such as a `Preprocessor` class for Task 1, a `Processor` class for Task 2, and a `LexicalAnalyzer` class for Task 3.

File Handling

To handle file input and output in Java, you should use the classes from the `java.io` package, such as:

```
import java.io.FileReader;
import java.io.FileWriter;
```

Command Line Arguments

Command line arguments are a useful way to provide input to a program when it is run from the command line, allowing for greater flexibility and customization. In this assignment, you will be using command line arguments to provide file names for your Java program to process.

To handle command line arguments in your program, modify the `main` method's signature as shown below:

```
public static void main(String[] args) {
    if (args.length < 1) {
        System.err.println("Usage: java ClassName <input_file>");
        System.exit(1);
    }

    String input_file = args[0];
}
```

Compilation and Execution of Your Programs

To compile and run your Java program, follow these instructions:

1. Open a terminal or powershell on your system.
2. Navigate to the directory where your source files are located using the `cd` command:

```
cd /path/to/your/source/files
```

3. Compile your source files using the Java compiler:

```
javac ClassName.java
```

4. After the compilation is successful, you will have a class file named "ClassName.class" in your current directory. To run the program with the input file as a command line argument, use the following command:

```
java ClassName code.java
```

Replace "ClassName" with the actual name of your class and "code.java" with the actual name of your input file. This input file is a `command line argument`.

Task 1: Preprocessing the Source Code

Create a Java program that accepts a file name as a command-line argument. The program should check the validity of the file and open it in read mode. It should then perform the following actions in the given sequence:

- Eliminate any blank lines in the program.
- Identify and eliminate all double slash and slash star comments.
- Eliminate unnecessary tabs and spaces in the program.
- Eliminate import statements and annotations.
- Write the updated program to a separate file named "out1.txt".
- Display the contents of the output file on the console.

You can implement this task using a `Preprocessor` class. The class should contain member functions for each of the preprocessing steps mentioned above.

An input sample and corresponding output is shown below for your convenience:

in1.txt

```
/***** in1.java *****/

// defining imports

import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

// defining class
public class in1
{


    public static void main(String [] args)
    {


        /*declaring variables*/
          int j = 2  ;
          int      motor   = 0, sensorValue =        0      ;
          if (motor    == 1)
          {
              sensorValue++;
          }
          else if (motor ==        0)
          {
              sensorValue--;
          }
    }

}
```

The output file for `in1.java` is:

out1.txt

```
public class in1
{
public static void main(String [] args)
{
int j = 2;
int motor = 0, sensorValue = 0;
if (motor == 1)
{
sensorValue++;
}
else if (motor == 0)
{
sensorValue--;
}
}
}
```

Task 2: Processing the Output File

In this task, you will take the output file from Task 1 and further process it as follows:

- File name will be provided as a command-line argument. Perform appropriate checks on the file and open it in read mode.
- Read the file character by character and write the contents of the file in a buffer as a linear array of characters. Ignore any newline characters and include all necessary spaces in the buffer.
- Place a sentinel value (`$`) at the end of the buffer carrying the linear representation of the program.
- Write the buffer to an output file `out2.txt`.
- Display the contents of the new output file on the console.

You can implement this task using a `Processor` class. This class should contain member functions for reading the file, processing it, and writing the output to a new file.

Output is shown below:

out2.txt

```
public class in1{public static void main(String [] args){int j = 2;int motor = 0, sensorValue = 0;if (motor == 1){sensorValue++;}else if (motor == 0){sensorValue--;}}}$

```

Task 3: Lexical Analysis

In this task, you will take the output file from Task 2 and further process it to perform lexical analysis. You will implement a `LexicalAnalyzer` class to identify lexemes in the input program.

A lexeme is a sequence of characters in the source program that matches the pattern for a token.

Here is a table of possible lexemes that can be found in a Java program:

Category	Lexemes
Keywords	`if`, `else`, `while`, `for`, `do`, `int`, `float`, `double`, `char`, `void`, `boolean`, `true`, `false`, `return`, `class`, `public`, `private`, `protected`, `static`, `final`, `try`, `catch`, `throw`, `interface`, ...
Identifiers	`x`, `y`, `z`, `sum`, `count`, `temp`, `myVar`, `myFunction`, ...
Operators	`+`, `-`, `*`, `/`, `%`, `=`, `+=`, `-=` , `*=` , `/=`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `++`, `--`, `&&`, `
Punctuators	`{`, `}`, `[`, `]`, `(`, `)`, `,`, `;`, `:`, `.`, `...`
Literals	`42`, `3.14`, `"Hello, world!"`, `'A'`, `'9'`, `true`, `false`, `null`
Comments	`// This is a single line comment`, `/* This is a multiline comment */`
Annotations	`@Override`, `@Deprecated`, `@SuppressWarnings`, ...
Import	`import java.util.List;`, `import static java.lang.Math.*;`, ...

Sample Run 1:

public static void main(){int j = 2;}$

The output of the program should be as follows:

Lexeme: public
Lexeme: static
Lexeme: void
Lexeme: main
Lexeme: (
Lexeme:)
Lexeme: {
Lexeme: int
Lexeme: j
Lexeme: =

Lexeme: 2
Lexeme: ;
Lexeme: j
Lexeme: }

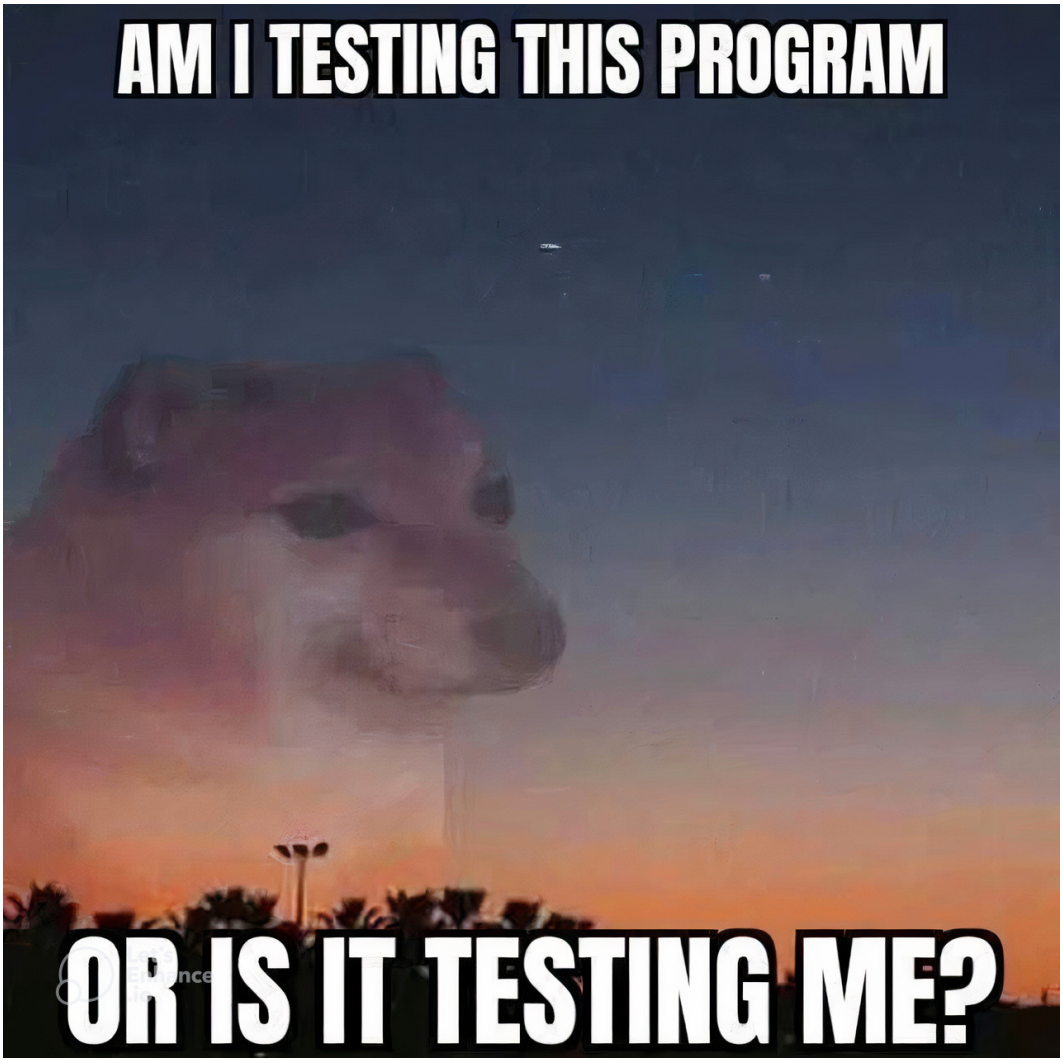
Sample Run 2:

public static void main(){System.out.println(”Hello World”);}$

The output of the program should be as follows:

Lexeme: public
Lexeme: static
Lexeme: void
Lexeme: main
Lexeme: (
Lexeme:)
Lexeme: {
Lexeme: System
Lexeme: .
Lexeme: out
Lexeme: .
Lexeme: println
Lexeme: (
Lexeme: "Hello World"
Lexeme:)
Lexeme: ;
Lexeme: }

End.


AM I TESTING THIS PROGRAM
OR IS IT TESTING ME?