Course: Compiler Construction **(Fall 2023)**

Resource Person: Nazifa Fatima **Assignment – 1(Lexical Analysis)**

Total Points: 100	Assigned: Tuesday, 19th December, 2023	Due: Wednesday, 27th December, 2023

Lexical Analysis

During the lecture, you were introduced to the initial step in a compiler's workflow: lexical analysis. This process involves a lexical analyzer, or tokenizer, transforming the unprocessed character sequence of code into a series of tokens. To accomplish this, the lexical analyzer systematically examines the code's character sequence, clusters them into lexemes, and assigns a corresponding token class to each lexeme. Your programming assignment provides a practical opportunity to apply this knowledge and implement your own lexical analyzer.

PART-I (20 Points)

A. Your Task

Your task is to write a function *tokenizeCode* in a programming language of your preference either python or java. Your code should accept the source code as a string and produce a list of tokens.

B. Sample Input (Source Code)

```
Source_code = "if(x >= 0){
                 print("Hello World");
               }
               else{
                  int sum = 0;
                  for(int i = 0; i < 10; i=i+1){
                     sum = i + 12.34 + 21E-2 + .21;
                }
                }";
```

C. Expected Output

The output of the program should be similar to the following:

```
<KEYWORD, if>
<LPAR, (>
<ID, x>
<RELOP, GEQ>
<NUMBER, 0>
<RPAR, )>
<L-CURLY-BRACE, {>
<ID, print>
<LPAR, (>
<STRING, "Hello World">
<RPAR, )>
<SEMICOLON, ;>
<R-CURLY-BRACE, }>
<KEYWORD, else>
<L-CURLY-BRACE, {>
<KEYWORD, int>
<ID, sum>
<RELOP, EQ>
<NUMBER, 0>
<SEMICOLON, ;>
<KEYWORD, for>
<LPAR, (>
<KEYWORD, int>
<ID, i>
<RELOP, EQ>
<NUMBER, 0>
<SEMICOLON, ;>
<ID, i>
<RELOP, GT>
<NUMBER, 10>
<SEMICOLON, ;>
<ID, i>
<RELOP, EQ>
<ID, i>
<OP, +>
```

```
<NUMBER, 1>
<RPAR, )>
<L-CURLY-BRACE, {>
<ID, sum>
<RELOP, EQ>
<ID, i>
<OP, +>
<NUMBER, 12.34>
<OP, +>
<NUMBER, 21E-2>
<OP, +>
<Number, .21>
<SEMICOLON, ;>
<R-CURLY-BRACE, }>
<R-CURLY-BRACE, }>
```

Valid Tokens:

We have provided below a list of token classes you must support:

- **Keywords (10 Points)**: any tokens from the list [***if*/*else***, ***for***, ***while***]
  - Token class: *KEYWORD*
- **Identifiers (10 Points)**: any tokens that begin with an ***alphabetic*** (including both capital and lowercase) character or an ***underscore*** (_), followed by alphanumeric characters and/or underscore (EXCEPT for the keyword tokens)
  - Examples of valid identifiers: **test**, **test1**, **_id1**, and **test_1_id_2**
  - Token class: *ID*
- **Numbers (10 Points)**: any numerical tokens optionally containing a ***decimal point/period*** (.), ***exponential number*** *i.e.*, both integers, floating-point and exponential numbers
  - Examples of valid numbers: **1**, **1.0**, **1.01**, **1E-2** and **.01**
  - Token class: *NUMBER*
- **Strings (10 Points)**: any tokens represented by a sequence of characters (including the empty sequence) that begins and ends with double quotes ("). You are **not** required to handle escape characters like \".

- Examples of strings: **"Hello"**, **""**, and **"1.01"**
- Token class: *STRING*

- **Comments (10 Points)**: any tokens represented by a sequence of characters beginning with a double slash (**//**) and that ends with a newline (**\n**)
  - Examples of comments: **//Hello\n**, **//""\n**, and **//"1.01"\n**
  - Token class: *COMMENT*
- **Relational Operators (10 Points)**: any tokens represented by a relational operator, specifically from the list [***>***, ***<***, ***>=***, ***<=***, ***==***, ***=***]
  - Token classes: *RELOP*
- **Operators (10 Points)**: any tokens represented by a arthimetic operator, specifically from the list [***+***, ***-***, *******, ***/***]
  - Token classes: *OP*
- **Parentheses, Braces, and Semicolons (10 Points)**: any tokens from the list [***(***, ***)***, ***{***, ***}***, ***;***]
  - Token classes: *LPAR*, *RPAR*, *LBRACE*, *RBRACE*, *SEMICOLON*

You are not required to handle standalone whitespaces (*e.g.*, **\t**, **\n**, **\r**, etc.); if you encounter them in the character stream, please make sure to properly ignore them (*e.g.*, do not characterize them as identifiers).

Extra Credit (20 Points)

You learned in lecture that lexical analysis can handle errors. You now have the opportunity to figure out how to handle these errors inside your lexical analyzer by detecting and localizing them.

- **Detection (10 Points)**: determine if an error exists at all and output the result.
- **Localization (10 Points)**: determine the cause(s) of any error that exists. There are 2 different causes of lexical errors that you need to support:
  - **Invalid Number**: any tokens that begin as valid numbers, but that do not actually match the pattern of numbers (*e.g.*, **1.** and **1.1r**)
  - **Invalid String**: any tokens that begin as valid strings, but that do not actually match the pattern of strings (*e.g.*, **"hello**, **"_**)

Additional Notes

The provided test cases are examples only, and we may run your code with different test cases.

Submission:

- This is a group assignment of maximum 2 student size.
- You are required to turn in a Zip file that contains your complete project (all source files with whatever other files are needed to compile them; Screenshot of your output). Add proper cover page that shows information of your group members.
- The name of your Zip file should be roll number of all students in the group as follows:

RollNumber1_RollNumber2.zip.

(Note one group members is required to submit the assignment in Google Classroom.

****END OF ASSIGNMENT****