**Alexandria University**
**Faculty of Engineering**
**Computer and Systems Engineering Department**

**C321: Programming languages and compiler**
**Assigned: Saturday 24th Apr, 2021**
**Due: Saturday 8th May, 2021**

# ***Assignment 1***

*Assignment will be into 3 phases.*

### *Objective of Whole Assignment*

- *It is required to develop a suitable Syntax Directed Translation Scheme to convert Java code to Java bytecode, performing necessary lexical, syntax and static semantic analysis (such as type checking and Expressions Evaluation).*
- *Generated bytecode must follow the standard bytecode instructions defined in Java Virtual Machine Specification*
  *http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html*
  *http://en.wikipedia.org/wiki/Java_bytecode*

## ***Phase1: Lexical Analyzer Generator***

### *Objective*

*This phase of the assignment aims to practice techniques for building automatic lexical analyzer generator tools.*

### *Description*

*Your task in this phase of the assignment is to design and implement a lexical analyzer generator tool.*

1- *The lexical analyzer generator is required to automatically construct a lexical analyzer from a regular expression description of a set of tokens. The tool is required to construct a nondeterministic finite automata (NFA) for the given regular expressions, combine these NFAs together with a new starting state, convert the resulting NFA to a DFA, minimize it and emit the transition table for the reduced DFA together with a lexical analyzer program that simulates the resulting DFA machine.*

2- *The generated lexical analyzer has to read its input one character at a time, until it finds the longest prefix of the input, which matches one of the given regular expressions. It should create a symbol table and insert each identifier in the table. If more than one regular expression matches some longest prefix of the input, the lexical analyzer should break the tie in favor of the regular expression listed first in the regular specifications. If a match exists, the lexical analyzer should produce the token class and the attribute value. If none of the regular expressions matches any input prefix, an error recovery routine is to be called to print an error message and to continue looking for tokens.*

3- *The lexical analyzer generator is required to be tested using the given lexical rules of tokens of a small subset of Java. Use the given simple program to test the generated lexical analyzer.*

4- *Keep in mind that the generated lexical analyzer will integrate with a generated parser which you should implement in phase 2 of the assignment such that the lexical analyzer is to be called by the parser to find the next token.*

## *Lexical Rules*

1- *The tokens of the given language are: identifiers, numbers, keywords, operators and punctuation symbols.*

2- *The token id matches a letter followed by zero or more letters or digits.*

3- *The token num matches an unsigned integer or a floating- point number. The number consists of one or more decimal digits, an optional decimal point followed by one or more digits and an optional exponent consisting of an E followed by one or more digits.*

4- *Keywords are reserved. The given keywords are: int, float, boolean, if, else, while.*

5- *Operators are: +, -, *, /, =, <=, <, >, >=, !=, ==*

6- *Punctuation symbols are parentheses, curly brackets, commas and semicolons.*

7- *Blanks between tokens are optional*

## *Lexical Rules Input File Format*

1- *Lexical rules input file is a text file.*

2- *Regular definitions are lines in the form LHS = RHS*

3- *Regular expressions are lines in the form LHS: RHS*

4- *Keywords are enclosed by { } in separate lines.*

5- *Punctuations are enclosed by [ ] in separate lines*

6- *\L represents Lambda symbol.*

7- *The following symbols are used in regular definitions and regular expressions with the meaning discussed in class: - | + * ( )*

8- *Any reserved symbol needed to be used within the language, is preceded by an escape backslash character.*

**Input file example for the above lexical rules:**

```
letter = a-z | A-Z
digit = 0 - 9
id: letter (letter|digit)*
digits = digit+
{boolean int float}
num: digit+ | digit+ . digits ( \L | E digits)
relop: \=\= | !\= | > | >\= | < | <\=
assign: =
{ if else while }
[; , \( \) { }]
addop: \+ | -
mulop: \* | /
```

## *Lexical Analyzer Output File Format*

*Your program should output the transition table of the generated lexical analyzer in a format of your choice as well as recognized token classes one per line (like the following format).*

Output file example for the following test program:

```
int
id
,
id
,
id
,
id
;
while
(
id
relop
num
)
....to be continued
```

### Test Program

```
int sum , count , pass , mnt; while (pass !=
10)
{
        pass = pass + 1 ;
}
```

## *Bonus Task*

*Submit a report about using tools (e.g. Lex, Flex) to automatically generate a lexical analyzer for the given regular expressions. Your report should include a detailed description of the required steps together with screenshots for using the tool. References:*

- *http://dinosaur.compilertools.net/*
- *Flex for Windows: http://gnuwin32.sourceforge.net/packages/flex.htm*

## *Notes*

1. *Implement the project using C++.*
2. *Each group consists of 4 students.*
3. *Requirements:*
    1- *Your executables and source code.*
    2- *A project report: make sure that your report contains at least the following:*
        a. *A description of the used data structures.*
        b. *Explanation of all algorithms and techniques used*
        c. *The resultant transition table for the minimal DFA.*
        d. *The resultant stream of tokens for the example test program.*
        e. *Any assumptions made and their justification.*

**<u>Good luck</u>**