Lab 1: A Calculator Compiler Front-end

Lab 1 is due Wednesday March 13. Your lab report and source code must be submitted by **12:20PM before the class**. The late policy applies to this lab project.

This lab is an individual project. Get started early! The required format for lab reports can be found on the resource page.

Objective: The goal of this homework is to acquaint yourself with two very powerful tools for the generation of compilers: lex and yacc. When a program is presented to a compiler, the compiler has to divide the program into meaningful units, called tokens, and then discover the relationship among these units. The algorithm that divides the program into units is called a lexical analyzer, scanner, or lexer.

Once the tokens are identified, the compiler needs to find the expressions, statements, declarations, blocks, and procedures in the program. This task is the function of a parser. The parser requires a list of rules that define the relationships among the tokens. This list of rules is called a grammar.

Lex is a program that takes a set of descriptions of possible tokens and produces a C routine that implements a scanner. Yacc takes a concise description of a grammar and produces a C routine that implements a parser for the grammar.

Lex and yacc are tools which can be used to solve a variety of interesting problems, even outside the realm of compilers. We are sure that once you understand how they function, you will find that the use of lex and yacc will save you many programming hours. It is strongly recommended that you read pages 1-17 of the Lex & Yacc Tutorial by Tom Nieman to benefit the most from this exercise.

Equipment and software:

- For this assignment, you can use any Linux machine that has lex and yacc (or flex and bison) installed.
- You may use a Windows 10 machine provided that you install "windows subsystem for linux", which is basically an Ubuntu Linux system.
- You may use a Mac OS X machine provided that you install Xcode.

Tasks:

All the files needed for this assignment can be found in the attachments. An example lex specification cal.l. You can build a standalone scanner starting from this file to see how the scanner works. Follow these instructions to build the scanner:

1. Invoke this command to convert a lex specification file into a C source file. "$ lex cal.l"

2. Now you must compile this C source file into an executable program using the following

command: “$ gcc -o scanner lex.yy.c”

3. Now you can run the executable scanner and play with it by typing in text and checking if it is recognized as a token. “$ ./scanner”

We recommend that you consult the cal.l file as you play with the scanner to understand how the tokens are specified. Make sure you try operators, words, parenthesis, etc. The scanner will output the type of symbol that it recognizes for the input that you type. You can use CTRL-D to terminate the scanner.

Task 1: For this task you are asked to develop an infix calculator. You should develop two files: infix.l and infix.y. The file infix.l is going to be the lex file written to match the patterns that are required to generate the tokens needed to implement a simple calculator. The file infix.y described the rules of the grammar that implement the calculator. You should customize the attached Makefile to generate the calculator. Make sure you run the make clean command to remove older versions of the files then type make:

```
$ make clean

$ make
```

The infix calculator should work like following. You invoke the calculator by running the program ”./infix”. If you type ”3 + 2” the calculator answers with ”= 5”; if you type ”x = 4” followed by “3 * x” the calculator responds with ”= 12”. Here is a demonstration:

```
$ ./infix

3+2

=5

x=4

=4

3*x

=12
```

You should be able to do the following operations:

1. Addition, subtraction, multiplication, and division.
2. Unary bitwise operator NOT. Use the following symbol: ‘!’. The operator should precede the operand.
3. The power operator (e.g 2 to the power of 4). Use the following symbol: **.
4. Support parentheses “(“ and “)”, and can warn unmatched parentheses.
5. Support variables whose name must start with lower case alphabet characters “a-z”.

Note: The parser should NOT recognize symbols consisting of two characters with spaces in between them as valid (e.g. “*␣*”).

We suggest you look at the cal.l and cal.y files to try to understand how the calculator was specified.

Task 2: Modify your infix calculator to print out the prefix form of infix expressions. In particular, all parentheses should have been removed in the prefix form. Also spaces should be inserted when needed to properly delimit expression terms. For examples:

- “2+2” => “+ 2 2”
- “x=3” => “= x 3”
- “!x” => “!x”

What to Turn In

For this lab project, turn in your description documents, source code including all .l and .y files, and also all Makefiles. You should submit all documents to CANVAS.