COSC 4785
Compiler Construction I
Fall 2022
Programming Project 1, Flex

1 Introduction

This is a *get-your-feet-wet* assignment using *flex*. The assignment is worth 50 points. *flex* generates a C or C++ code file named (by default) either "lex.yy.c" or "lex.yy.cc," because that is the name. I do not know why the *yy*. We will change that behavior and and OURS will be named exactly "program1_lex.cpp." The C++ lexer we create will require that you include <FlexLexer.h> **only in the main program**.

When using *flex* for C++ input files and desiring C++ compatible output, just use *flex++*. Additionally, use the *--warn* option. That means that to generate "program1_lex.cpp" from the command line we do

```
$> flex++ --warn program1.lpp
```

And if there are any warnings, FIX THEM!!!!! That means figure out the problem and correct your input. It does not mean just putz around until something good happens, it means understand the warning, and do something specific and correct your file to fix the problem. This means that you might have to (gasp) READ the documentation on *flex* or even ask questions.

In order to change the output file name you need to include an option in the declaration section. Make sure that it is NOT enclosed in a %{ %} block.

```
%option outfile="program1_lex.cpp"
```

This needs to start at the left margin.

2 Instructions

You will be writing a very simple program, all contained in just a few files. It will do a simple lexical analysis of a text input file. This will read from *standard input* and write to *standard output*. If you do not know what that means, ask. You MUST test this and make sure that it compiles, that it works correctly, that there are no endless loops, and MOST ESPECIALLY that there are **no** core dumps. You will have to use *make* for this assignment. It will make your life easier in the future and I insist.

1. You will use **C++** code for the scanner. The *flex* input file will be named "program1.lpp." This is to differentiate it from C code flex input files which usually use the plain ".l" file extension. This will have NO "user code" section. I do not care what you find online or in other documentation.

2. You will create a *main* program in the file "program1.cpp." Think of this as a driver. In it, you must declare a variable of type *yyFlexLexer*. This will be your "scanner" and one of its members is the function *yylex()*.

   Use *g++* to compile this program and use the following "flags"

   ```
   -ggdb -Wall
   ```

   These options are not to be ignored, not to be changed, not to be added to.

   - The first option causes the compiler to include debugging information for the default debugger, *gdb*. Regardless of what you read, do NOT use plain **-g**.
   - The next option turns on all the "normal" warnings. A note on this, Just like with *flex* FIX ALL THE WARNINGS!!! If you do not understand them, ask. Especially, ask in class. You may eventually encounter warnings that are caused by code *flex*, or later on *bison*, generates and we will discuss handling of those when the occur.

   Beginning on the next page is a starting point for the driver program.

```
// program1.cpp
// Kim Buckner
// COSC 4785 Fall 2032
// Program 01
// August 34, 2032

#include<FlexLexer.h>

int main()
{
  yyFlexLexer myScanner;
  int rtn;

   // any needed initialization code

  while((rtn=myScanner.yylex()) != 0) {
    // loop code goes here
  }

  //more code if needed

  return 0;

}
```

3. The program will output information in the following manner.

```
Line    Column  Type    Length  Value
1       1       6       2       if
1       4       6       3       else
1       8       7       3       the
1       11      11      1
2       1       1       2       ==
```

The first column heading starts at column 1 (left margin). Each of the following headings are separated by a tab set to 8 spaces. There will be one row for every lexeme (token) recognized. Each column will be positioned just like the headers. DO NOT print blank lines, extra characters, garbage, fancy things, anything but what I have listed.The two solid lines are for clarity and not to be included in your output. Note that the column, type, and length may be more than one digit so do NOT let that change the alignment of the data.

You must use standard C++ operators, objects, and functions for this. DO NOT combine C and C++ standard libraries.

4. The "type" (there are 11) in column 3 will be one of the following:

   - 1 = compare_op: `== > < >= <= !=`
   - 2 = logical_op: `|| &&`
   - 3 = math_op: `+ - * /`
   - 4 = enclosing_op: `{ } [ ] ( )`
   - 5 = punctuation: `. , ;` (just period, comma, semicolon)
   - 6 = keyword: `this if else while`
   - 7 = identifier: starts with an _ (underscore) or letter and followed by any number of _ or digits or letters
   - 8 = number: a series of digits (an integer)
   - 9 = float: a series of digits, followed by
     - a period and one or more digits.
     - a period, one or more digits, either 'E' or 'e', an optional '+' or '-', and one or more digits.
     - either 'E' or 'e', an optional '+' or '-', and one or more digits.
     - Examples
       * 42.0E01
       * 1.0e17
       * 42.7e09
       * 9e+08
       * 2.3E-45
   - 10 = unrecognized character
   - 11 = newline

5. Note that the return value from the *.yylex()* function that indicates it has reached the end of the input is **0** so you might have to do something to make sure you do not return 0 accidentally.

6. The length is the number of characters that were matched for this lexeme. The line and column are where, in the input, the match started. If you read the *flex* documentation it will discuss how to get these.

7. A *value* (the characters actually matched) is printed for everything BUT space, tab, and newline. Spaces and tabs are ignored completely, no match, no output, no yylex() return, just thown into a black hole. But you may need to count them to keep track of starting columns.

   For tab you will need to increase the column count like

   $$\text{next\_column} = \text{current\_column} + ((\text{current\_column} + 1) \ \% \ 8)$$

   In other words after a tab, the next lexeme starts on a column which is a multiple of 8, plus 1. Like 9, 17, 15, etc.

   Do not match multiple newlines at once, it is not faster and can cause problems. You have to be careful because the *flex* matching is always greedy, that is it matches as much as possible when using $*$ or $+$ operators.

8. **Required for Graduate Students, for Undergraduates 10 points extra credit** Add the ability to **correctly** match C-style comments. These are comments that can be enclosed between `/* and */`. This MUST include correctly accounting for lines and columns. That is worth half the points. These comments can occur anywhere in the input and can span multiple lines. The comments will NOT be printed out, and there will be no indication in the output (other than changes in line and/or column numbers) that they were matched.

   DO NOT match C++/Java style comments for this assignment!

9. You really must ask questions. Do not come to me the day this assignment is due and tell me that you could not figure how to start the assignment, that you could not log into a Linux machine, that you could not figure out how to compile the program, that *stack**overflow*** or Stack**Exchange** did not have anything like this, that *make* or *g++* are broken, or that you cannot figure out *tar*.

3 Submission

You will submit the files for this on WyoCourses. There must be at least three files contained in the *tar* archive:

- A "makefile" named exactly **Makefile**, note the capital M.
- The C++ "driver" program in a file named exactly **program1.cpp**.
- The *flex* input in a file named exactly **program1.lpp**.
- If you have some header file(s) you created and are required to compile the program, submit those as well. These MUST use the **.hpp** file extension, because I said so. The filenames must NOT contain spaces or any characters other than _, letters, or digits.

- The archive will be named either program1.tar or program1.tgz.
  - The "archive", also called a "tarfile" or sometimes a "tarball" is created with GNU *tar*. BUT you must NOT include **any** directories, ONLY files.
    Use a command line something like

    ```
    $> tar cf program1.tar program1.cpp program1.lpp Makefile
    ```

    or compress it with

    ```
    $> tar zcf program1.tgz program1.cpp program1.lpp Makefile
    ```
  - Read the man page on tar for usage.

You can then submit just the tarfile to wyocourses. I will allow only the following file extensions on the submission

- **.tar** – *tar* uncompressed archive
- **.tgz** – *tar* gzip compressed archive