CS 3500 – Programming Languages & Translators
Homework Assignment #1

- This assignment is **due by 8 p.m. on Friday, August 31, 2018**.
- This assignment will be worth **2%** of your course grade.
- You are to work on this assignment **by yourself**.
- You should **take a look at the sample input and output files** posted on the Canvas website **before** you actually submit your assignment for grading.

Basic Instructions:

For this assignment you are to use ***flex*** to create a C++ program that will perform **lexical analysis** for a small programming language called MFPL (described below). If your *flex* file is named **mfpl.l**, you should be able to compile and execute it on one of the campus Linux machines (such as rc***nn***ucs213.managed.mst.edu where ***nn*** is **01-32**) using the following commands (where *inputFileName* is the name of some input file):

```
flex mfpl.l
g++ lex.yy.c -o mfpl_lexer
mfpl_lexer < inputFileName
```

Your program should **output information about each token** that it encounters in the input source program. You will need a token type **UNKNOWN** for any tokens that cannot be properly categorized as an operator, keyword, identifier, etc. (effectively, these are lexical errors). Sample input and output are given at the end of this document.

Your program should **continue processing tokens** from the input file **until end-of-file is detected**. Note that your program should **NOT** do anything other than recognize tokens (e.g., no syntax checking, etc.), as that is the only purpose of lexical analysis.

MFPL Programming Language:

For now, all you need to be concerned with are the **tokens** in the MFPL programming language.

An **identifier** in MFPL must start with a letter or underscore, followed by any number of letters, digits, and/or underscores.

An **integer constant** in MFPL is a sequence of one or more digits, optionally preceded by **+** or **-**. Don't worry about a size limit on integer constants.

A **string constant** in MFPL is the same as a valid string constant in C++. Note that a string constant that begins on one line must be terminated with an ending double quote character " on that same line. Don't worry about doing any special processing for characters preceded with a backslash character like \n, \t, or \".

The only **keywords** in MFPL are the following: **let***, **if**, **lambda**, **print, input, and, or, not, t, nil**. The language is case-sensitive, so those keywords must be in **lowercase** (otherwise, they should be recognized as identifiers).

The only **operator symbols** are the following: **+**, -, *****, **/, <, >, <=, >=, =, /=**

This programming language also uses **parentheses**, so you need to recognize **(** and **)**. Don't check for matching parentheses, etc. as that is **not** the responsibility of a lexical analyzer.

Comments in this programming language are similar to the C++ // style of comments, except that **; is used instead of //**. Comments simply should be scanned over and ignored (i.e., **not** included in your output!).

In summary, your program should report the following types of tokens (and their lexemes):

> **LETSTAR, LAMBDA, INPUT, PRINT, IF, LPAREN, RPAREN,**
> **ADD, MULT, DIV, SUB, AND, OR, NOT, LT, GT, LE, GE, EQ, NE,**
> **IDENT, INTCONST, STRCONST, T, NIL, UNKNOWN**

Because we are using an automated script (program) for grading, you **MUST** use exactly these token names; **otherwise, you will receive a ZERO for this assignment!!!**

Sample Input and Output:

You should output the **token and lexeme information for every token** processed in the input file even if the lexeme is not unique for the token (for example, the lexeme for every **LAMBDA** token will be **lambda**).

Given below is some sample input and output (also posted as files on Canvas). With the exception of whitespace, the output produced by **your** program should be **identical** for this input!

Input:

```
let* ( some lambda input + -1234 ;what about this?
*/- 0123 99 + x _underscore_this) &&^
;;; yet another comment
print if flex  let 203978 -2 + "30x^2" % !
1+2
3 + 4 > t
-5+ 6
"a
bc"
7
5 >= nil and 4 or not toBe1 <<==78
/= -42
```

Output:

```
TOKEN: LETSTAR      LEXEME: let*
TOKEN: LPAREN       LEXEME: (
TOKEN: IDENT        LEXEME: some
TOKEN: LAMBDA       LEXEME: lambda
TOKEN: INPUT        LEXEME: input
TOKEN: ADD          LEXEME: +
TOKEN: INTCONST     LEXEME: -1234
TOKEN: MULT         LEXEME: *
TOKEN: DIV          LEXEME: /
TOKEN: SUB          LEXEME: -
TOKEN: INTCONST     LEXEME: 0123
TOKEN: INTCONST     LEXEME: 99
TOKEN: ADD          LEXEME: +
TOKEN: IDENT        LEXEME: x
TOKEN: IDENT        LEXEME: _underscore_this
TOKEN: RPAREN       LEXEME: )
TOKEN: UNKNOWN      LEXEME: &
TOKEN: UNKNOWN      LEXEME: &
TOKEN: UNKNOWN      LEXEME: ^
TOKEN: PRINT        LEXEME: print
TOKEN: IF           LEXEME: if
TOKEN: IDENT        LEXEME: flex
TOKEN: IDENT        LEXEME: let
TOKEN: INTCONST     LEXEME: 203978
TOKEN: INTCONST     LEXEME: -2
TOKEN: ADD          LEXEME: +
TOKEN: STRCONST     LEXEME: "30x^2"
TOKEN: UNKNOWN      LEXEME: %
TOKEN: UNKNOWN      LEXEME: !
TOKEN: INTCONST     LEXEME: 1
TOKEN: INTCONST     LEXEME: +2
TOKEN: INTCONST     LEXEME: 3
TOKEN: ADD          LEXEME: +
TOKEN: INTCONST     LEXEME: 4
TOKEN: GT           LEXEME: >
TOKEN: T            LEXEME: t
TOKEN: INTCONST     LEXEME: -5
TOKEN: ADD          LEXEME: +
TOKEN: INTCONST     LEXEME: 6
TOKEN: UNKNOWN      LEXEME: "
TOKEN: IDENT        LEXEME: a
```

```
TOKEN: IDENT        LEXEME: bc
TOKEN: UNKNOWN      LEXEME: "
TOKEN: INTCONST     LEXEME: 7
TOKEN: INTCONST     LEXEME: 5
TOKEN: GE           LEXEME: >=
TOKEN: NIL          LEXEME: nil
TOKEN: AND          LEXEME: and
TOKEN: INTCONST     LEXEME: 4
TOKEN: OR           LEXEME: or
TOKEN: NOT          LEXEME: not
TOKEN: IDENT        LEXEME: toBe1
TOKEN: LT           LEXEME: <
TOKEN: LE           LEXEME: <=
TOKEN: EQ           LEXEME: =
TOKEN: INTCONST     LEXEME: 78
TOKEN: NE           LEXEME: /=
TOKEN: INTCONST     LEXEME: -42
```

You might find it helpful to use the *diff* command to compare your output with the sample output posted on Canvas. To do this, first *flex* and compile your program, and run it on the sample input file **hw1_mfpl.txt** that is posted on Canvas, redirecting the output to a file named **myOutput.out** using the following commands:

```
flex mfpl.l
g++ lex.yy.c -o mfpl_lexer
mfpl_lexer < hw1_mfpl.txt > myOutput.out
```

Assuming there were no errors in that process, you can now compare your output (which should be in file **myOutput.out**) with the output file posted on Canvas (**hw1_mfpl.txt.out**), ignoring differences in spacing, using the following command (typed all on one line):

```
diff myOutput.out hw1_mfpl.txt.out --ignore-space-change --side-by-side
     --ignore-case --ignore-blank-lines
```

To learn more about the *diff* command, see http://ss64.com/bash/diff.html

What to Submit for Grading:

Name your *flex* file using your last name followed by your first initial with the .l extension (e.g., Homer Simpson would name his file **simpsonh.l**). Do **NOT** submit your file using the name **mfpl.l** or **you will receive a ZERO on the assignment** (since no one's last name in this class is Mfp).

Submit **only** your *flex* file (**not** your lex.yy.c file) via Canvas. You can submit multiple times before the deadline; only your last submission will be graded.

The grading rubric is given below so that you can see how many points each part of this assignment is worth. Note that the next assignment builds upon this one, so **it is critical that this assignment works properly in all respects!**

	Points Possible	Mostly or completely incorrect (0% of points possible)	Needs improvement (70% of points possible)	Adequate, but still some deficiencies (80% of points possible)	Mostly or completely correct (100% of points possible)
Comments correctly processed and ignored	10				
Identifiers correctly recognized and identified (IDENT)	10				
Signed integers recognized and identified (INTCONST)	5				
Unsigned integers recognized and identified (INTCONST)	5				
Arithmetic operators (+, -, *, /) correctly recognized and identified (ADD, SUB, MULT, DIV)	5				
Relational operators (<, >=, etc.) correctly recognized and identified (LT, GE, etc.)	10				
Keywords correctly recognized and identified (IF, INPUT, PRINT, etc.)	15				
Parentheses correctly recognized and identified (LPAREN, RPAREN)	5				
String constants correctly recognized and identified (STRCONST)	10				
Unknown tokens correctly processed and identified as UNKNOWN	5				
Program processes an entire input file and correctly handles end-of-file	5				
Program outputs both token type and lexeme information for all tokens	10				
Whitespace correctly processed and ignored	5				