CS 5500 – The Structure of a Compiler
HW #1

- This assignment is **due by 11:59 p.m. on Wednesday, August 28, 2019**.
- This assignment will be worth **4%** of your course grade.
- A grading rubric also is posted on Blackboard so that you can see approximately how many points each functional requirement is worth.
- You are to work on this assignment **by yourself**.
- You are strongly encouraged to **take a look at all of the sample input and output files** posted on the Blackboard website **before** you actually submit your assignment for grading.

Basic Instructions:

For this assignment you are to write a program that will perform **lexical analysis** for a small programming language called MIPL (partially described below). You are **NOT** allowed to use ***flex****;* you will receive **NO CREDIT** for this assignment if you do so – and we will be checking for that! It's recommended that you use C++ or Python because they can do regular expression processing, but you can use any programming language as long as **your program compiles/interprets and executes on the cslinux machines**.

Because we are allowing you to implement your lexical analyzer in any language, you will have to submit a ***make*** file. There is plenty of information/documentation available on the internet for creating *make* files for various languages – look it up! We will be testing your program on several different input files. Your program should expect **the name of the input file as a command line argument**. Your program should not require any other command line arguments. We use a script to grade the programs and it will not be providing any other command line arguments.

Your program should **output information about each token** that it encounters in the input source program. In later parts of this compiler project you will want to have the **ability to suppress** this output, so create an internal Boolean variable that you can set to turn on/off this functionality; do **NOT** use a command line argument for this purpose as our grading script will not be looking for that!

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.

MIPL Programming Language Description

The programming language for which you are to write a lexical analyzer is called MIPL (Mini Imperative Programming Language). For now, all you need to be concerned with are the **tokens** in the programming language. Valid token types include case-sensitive **keywords and operators**; specifically:

> '(', ')', '*', '+', ',', '-', '.', '..', ':', ':=', ';', '<', '<=', '<>', '=', '>', '>=', '[', ']', '"', 'and', 'array', 'begin', 'boolean', 'char', 'div', 'do', 'else', 'end', 'false', 'if', 'integer', 'not', 'of', 'or', 'procedure', 'program', 'read', 'then', 'true', 'var, 'while', 'write'

Note: Unlike many programming languages, **/** is **NOT** a valid operator in MIPL.

An **identifier** in MIPL is similar to a valid identifier in C/C++ **(i.e., it must start with a letter or underscore, followed by any number (including zero) of letters, digits, and/or underscores)**.

Integer constants (which can be signed or unsigned) are as in standard C/C++. If an integer constant is signed, you are to **treat the sign as a separate token**. For example, -12 should be processed as **two** tokens, - and 12. This is necessary for subsequent processing in our MIPL compiler.

Character constants are made up of a single quote, followed by a single character, and an ending single quote. There are no special backslashed characters in MIPL (e.g., you don't have to process character constants like '\n'; those will be considered invalid). There are no strings in MIPL.

MIPL **comments** are similar to the C and C++ **/* … */** style of comments, except that parentheses are used instead of **/**. Note that comments can continue over more than one line. **Comments should simply be scanned over and ignored (NOT included in the output!).** Note that processing comments might be trickier to implement than they initially appear to be. For example, consider the following input:

```
(* comment_1 *)
line of valid code
(* comment_2 *)
```

If your regular expression matched '(*', followed by anything (including newline), followed by '*)', then it would probably match the initial '(*' on the first line, followed by the entire line of valid code on the second line, and end the match with the '*)' at the end of the third line, thus making the entire input parse as a single comment. The correct thing to do is to have the 'anything' that is to be matched in the middle be 'anything that isn't an asterisk followed by a right parenthesis.' Additionally, a comment may span more than one line of the input file. You might find it easier to write a function to handle this rather than just

trying to come up with a single regular expression to do it. In general, when the patterns are particularly complex, a function gives the programmer better control than a regular expression alone might provide.

If you encounter input that cannot be classified into any of the above categories (e.g., integer constant, identifier, etc.), it should be classified as an ***unknown*** token. Do **NOT** flag it as a lexical error.

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

> **ASSIGN, MULT, PLUS, MINUS, DIV, AND, OR, NOT,**
> **LT, GT, LE, GE, EQ, NE, VAR, ARRAY, OF, BOOL, CHAR, INT,**
> **PROG, PROC, BEGIN, END, WHILE, DO, IF, THEN, ELSE,**
> **READ, WRITE, TRUE, FALSE, LBRACK, RBRACK, SCOLON,**
> **COLON, LPAREN, RPAREN, COMMA, DOT, DOTDOT,**
> **INTCONST, CHARCONST, IDENT, UNKNOWN**

In the output, you are to add the prefix **T_** to each of these token names to make it clear that they are, in fact, **T**okens. In later assignments the output will include other symbolic names that are not tokens.

Error Detection

There are two types of errors that your lexical analyzer must be able to detect and report: (1) **invalid character constant** (i.e., missing a closing quote), and (2) **invalid integer constant** (i.e., larger than 2147483647).

If programming in C++, do **not** use *atoi()* to distinguish between valid and out-of-range integers. Depending upon the actual input, this strategy may or may not work. The maximum C integer is machine-dependent, and is not always what you are trying to match. Instead write a function that parses the integer constant token as a string of digits. There are several C/C++ string functions that might be useful here. (Hint: after eliminating leading zeroes, what do you know about strings of over ten digits? What about strings with fewer than ten digits? Exactly ten digits?)

Your lexical analyzer should neither attempt to “fix” the input for a lexical error (e.g., “insert” a missing quote into the input), nor terminate processing if it encounters a lexical error. After printing an error message, invalid integer constants and invalid character constants **(internally)** should be designated as ***unknown*** tokens, and the lexical analyzer should continue processing the input file.

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 **READ** token will be **read**).

Sample input and output files are posted on Blackboard. Note that these files were created on a Windows PC, so you might need to run ***dos2unix*** on them before using them on a Linux/Unix machine. With the exception of whitespace and capitalization, the output produced by your program **MUST** be **IDENTICAL** to what our "grader" program has produced! If it doesn't, your program may **NOT** be graded; consequently, you may receive a **ZERO** on this assignment! Grading is done using a script (program) that runs your program and compares your output to ours; there simply isn't time to manually grade individual programs, visually inspecting your output for multiple test files, particularly when the output can be quite lengthy (which it will be as the semester goes on!).

You might find it helpful to use the *diff* command to compare your output with the sample output posted on Blackboard; *diff* is basically what our grading script uses. To do this, first run your program on a sample input file (for example, **input.txt)**, redirecting the output to a file named **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 corresponding expected output file posted on Blackboard (**input.txt.out**), ignoring differences in spacing and upper vs. lower case, using the following command (typed all on one line):

diff myOutput.out input.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:

You should submit via Canvas a single **zip** file containing **only source code files and a *make* file** for your homework submission (i.e., no data files!). Name your zip file using your last name followed by your first initial (e.g., Homer Simpson would name his **simpsonh.zip**).

***WARNING*: If you fail to follow all of these instructions, the automated grading script may reject your submission, in which case it will NOT be graded!!!** ☹