CS 3500 – Programming Languages & Translators
Homework Assignment #1

- This assignment is **due by 11:59 p.m. on Wednesday, Sep. 2, 2020**
- This assignment will be worth **3%** 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 Mini-R (described below). If your *flex* file is named **minir.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 minir.l
g++ lex.yy.c -o minir_lexer
minir_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.

Tokens for the Mini-R Programming Language

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

An **identifier** in Mini-R must start with a letter (upper or lower case), followed by any number of letters (upper or lower case), digits, and/or underscores.

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

A **float constant** in Mini-R is a sequence of zero or more digits, followed by a period, followed by one or more digits; all of that optionally can be preceded by **+** or **-**. Don't worry about a size limit on float constants.

A **string constant** in Mini-R 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 Mini-R are the following: **if, else, while, function, for, in, TRUE, FALSE, quit, print, cat, read, list**. The language is case-sensitive, so those keywords must be in the case we have specified; otherwise, they should be recognized as identifiers.

The only **operator symbols** are the following: **+ - * / %% ^ < > <= >= == != ! & | =**

This programming language also uses **semicolons**, **commas**, **parentheses**, **brackets**, and **curly braces**, so you need to recognize: **; , ( ) [ ] { }**. 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!).

Because we are using an automated script (program) for grading, you **MUST** use **EXACTLY** the same token names as we use; **otherwise, you will receive a ZERO for this assignment!!! See the sample output for the token names you are to use.**

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 **WHILE** token will be **while**).

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:

```
+1234 |"hello" $what ;;
while TRUE "nice" ^99*
# how about a comment
print if else 7.8245 >= 7
-999 /== FALSE )]& list("one", 2, 3)
{ " hello" > 3 +}
"a in
bc" .097 next next_please <
!= identifier_two - 5 +6
for ( break ) function =!
!TRUE cat =B0Jack "read" true then read
%%45 : i=2, <= 9 # comment time
[quit -000000000.1234567890
```

```
"BoJack""Horseman"
```

Output:

```
TOKEN: INTCONST        LEXEME: +1234
TOKEN: OR              LEXEME: |
TOKEN: STRCONST        LEXEME: "hello"
TOKEN: UNKNOWN         LEXEME: $
TOKEN: IDENT           LEXEME: what
TOKEN: SEMICOLON       LEXEME: ;
TOKEN: SEMICOLON       LEXEME: ;
TOKEN: WHILE           LEXEME: while
TOKEN: TRUE            LEXEME: TRUE
TOKEN: STRCONST        LEXEME: "nice"
TOKEN: POWER           LEXEME: ^
TOKEN: INTCONST        LEXEME: 99
TOKEN: MULT            LEXEME: *
TOKEN: PRINT           LEXEME: print
TOKEN: IF              LEXEME: if
TOKEN: ELSE            LEXEME: else
TOKEN: FLOATCONST      LEXEME: 7.8245
TOKEN: GE              LEXEME: >=
TOKEN: INTCONST        LEXEME: 7
TOKEN: INTCONST        LEXEME: -999
TOKEN: DIV             LEXEME: /
TOKEN: EQ              LEXEME: ==
TOKEN: FALSE           LEXEME: FALSE
TOKEN: RPAREN          LEXEME: )
TOKEN: RBRACKET        LEXEME: ]
TOKEN: AND             LEXEME: &
TOKEN: LIST            LEXEME: list
TOKEN: LPAREN          LEXEME: (
TOKEN: STRCONST        LEXEME: "one"
TOKEN: COMMA           LEXEME: ,
TOKEN: INTCONST        LEXEME: 2
TOKEN: COMMA           LEXEME: ,
TOKEN: INTCONST        LEXEME: 3
TOKEN: RPAREN          LEXEME: )
TOKEN: LBRACE          LEXEME: {
TOKEN: STRCONST        LEXEME: " hello"
TOKEN: GT              LEXEME: >
TOKEN: INTCONST        LEXEME: 3
TOKEN: ADD             LEXEME: +
TOKEN: RBRACE          LEXEME: }
TOKEN: UNKNOWN         LEXEME: "
TOKEN: IDENT           LEXEME: a
TOKEN: IN              LEXEME: in
TOKEN: IDENT           LEXEME: bc
TOKEN: UNKNOWN         LEXEME: "
TOKEN: FLOATCONST      LEXEME: .097
TOKEN: IDENT           LEXEME: next
```

```
TOKEN: IDENT          LEXEME: next_please
TOKEN: LT             LEXEME: <
TOKEN: NE             LEXEME: !=
TOKEN: IDENT          LEXEME: identifier_two
TOKEN: SUB            LEXEME: -
TOKEN: INTCONST       LEXEME: 5
TOKEN: INTCONST       LEXEME: +6
TOKEN: FOR            LEXEME: for
TOKEN: LPAREN         LEXEME: (
TOKEN: IDENT          LEXEME: break
TOKEN: RPAREN         LEXEME: )
TOKEN: FUNCTION       LEXEME: function
TOKEN: ASSIGN         LEXEME: =
TOKEN: NOT            LEXEME: !
TOKEN: NOT            LEXEME: !
TOKEN: TRUE           LEXEME: TRUE
TOKEN: CAT            LEXEME: cat
TOKEN: ASSIGN         LEXEME: =
TOKEN: IDENT          LEXEME: B0Jack
TOKEN: STRCONST       LEXEME: "read"
TOKEN: IDENT          LEXEME: true
TOKEN: IDENT          LEXEME: then
TOKEN: READ           LEXEME: read
TOKEN: MOD            LEXEME: %%
TOKEN: INTCONST       LEXEME: 45
TOKEN: UNKNOWN        LEXEME: :
TOKEN: IDENT          LEXEME: i
TOKEN: ASSIGN         LEXEME: =
TOKEN: INTCONST       LEXEME: 2
TOKEN: COMMA          LEXEME: ,
TOKEN: LE             LEXEME: <=
TOKEN: INTCONST       LEXEME: 9
TOKEN: LBRACKET       LEXEME: [
TOKEN: QUIT           LEXEME: quit
TOKEN: FLOATCONST     LEXEME: -000000000.1234567890
TOKEN: STRCONST       LEXEME: "BoJack"
TOKEN: STRCONST       LEXEME: "Horseman"
```

DO NOT OUTPUT THE NUMBER OF LINES PROCESSED!!!

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_input.txt** that is posted on Canvas, redirecting the output to a file named **myOutput.out** using the following commands:

```
flex minir.l
g++ lex.yy.c -o minir_lexer
minir_lexer < hw1_input.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_input.txt.out**), ignoring differences in spacing, using the following command (typed all on one line):

diff myOutput.out hw1_minir.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 **minir.l** or **you will receive a ZERO on the assignment** (since no one's last name in this class is Mini).

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. Note that if you submit more than once, **Canvas appends a number to your submission filename**; don't worry about that.

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	5				
Identifiers correctly recognized and identified (IDENT)	5				
Signed and unsigned integers correctly recognized and identified (INTCONST)	5				
Unsigned and unsigned floats correctly recognized and identified (FLOATCONST)	5				
String constants correctly recognized and identified (STRCONST)	5				
Assignment (=) and arithmetic operators (+, -, *, /, ^, %%) correctly recognized and identified (ADD, SUB, MULT, DIV, POW, MOD)	15				

Relational and logical operators (<, >=, etc.) correctly recognized and identified (LT, GE, etc.)	10				
Keywords correctly recognized and identified (IF, WHILE, PRINT, etc.)	15				
Parentheses, brackets, semicolons, curly braces, and comma correctly recognized and identified	6				
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	14				
Whitespace correctly processed and ignored	5				