CSE 4713 / 6713 — Programming Languages
Project Part 1

Our first programming assignment for the project is to write a lexical analyzer for a subset of TIPS (which is a subset of the Pascal programming language). The job of a lexical analyzer is to return the lexemes (i.e., fundamental syntactical elements) in the input program to a parser for further analysis.

You will use C++ and flex for this assignment.

Example Lexical Analyzer Using Flex

Flex is a tool for generating scanners. A scanner, sometimes called a tokenizer, is a program that recognizes lexical patterns in text. The flex program reads user-specified input files for a description of a scanner to generate. It is traditional to end this file with "`.l`", for example `rules.l` . (The "l" stands for "lex", the name of the original Unix tool. Flex is the open-source replacement. Flex also uses a new algorithm that is much faster than the original lex.) The description is in the form of pairs of regular expressions and C code, called rules. Flex generates a C source file named `lex.yy.c` , which defines the function `yylex()` . The file `lex.yy.c` can be compiled and linked to produce an executable. When the executable is run, it analyzes its input for occurrences of text matching the regular expressions for each rule. Whenever it finds a match, it executes the corresponding C code.

Examine the `exp-rules.l` file in the `chapter4b_example` zip file :

- `exp-rules.l` contains the rules of how to recognize and return a token when reading each lexeme from the input. The example is recognizing the lexemes in the Arithmetic Expression Grammar from the textbook.
- Flex has its own format, go to https://westes.github.io/flex/manual/ and read sections 3 to 6 to get a basic idea of how a rules file is formatted / constructed.
- A good reference web site for flex is http://web.eecs.utk.edu/~bvz/cs461/notes/flex/

Using flex to generate a lexical analyzer:

- Type `flex exp-rules.l` in the Ubuntu terminal. Flex will then generate a `lex.yy.c` file in the current path.
- I have given you a `makefile` which will automatically compile this example, including calling flex, based on which files you have updated with an editor.
- To test the example program, use the `sample.in` file. You should get results similar to the textbook's hand-built lexical analyzer from Chapter 4.

Assignment

Use **flex** to generate the lexical analyzer using the language specification shown in the following table. A set of files that you will use as a starting point for your code is in the attached `Part_1_Starting_Point.zip` file.

Tables of the TIPS lexemes are shown on the next pages.

Keywords	Token Identifier Value	Token Constant
BEGIN	1000	TOK_BEGIN
BREAK	1001	TOK_BREAK
CONTINUE	1002	TOK_CONTINUE
DOWNTO	1003	TOK_DOWNTO
ELSE	1004	TOK_ELSE
END	1005	TOK_END
FOR	1006	TOK_FOR
IF	1007	TOK_IF
LET	1008	TOK_LET
PROGRAM	1009	TOK_PROGRAM
READ	1010	TOK_READ
THEN	1012	TOK_THEN
TO	1013	TOK_TO
VAR	1014	TOK_VAR
WHILE	1015	TOK_WHILE
WRITE	1016	TOK_WRITE

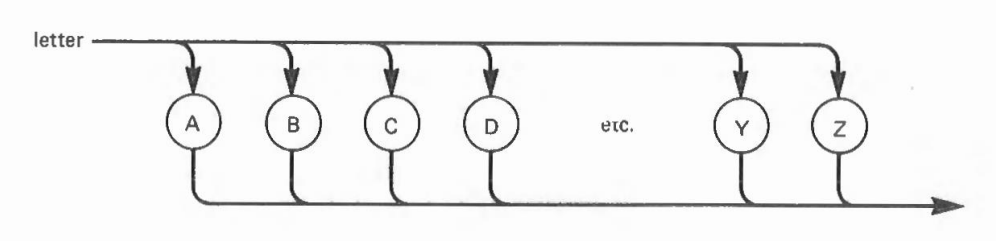
Datatype Specifiers	Token Identifier Value	Token Constant
INTEGER	1100	TOK_INTEGER
REAL	1101	TOK_REAL

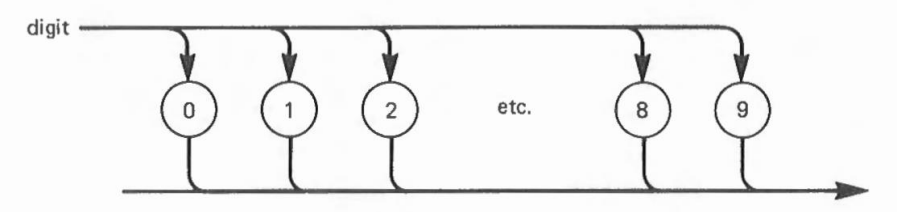
Punctuation	Token Identifier Value	Token Constant
;	2000	TOK_SEMICOLON
:	2001	TOK_COLON
(	2002	TOK_OPENPAREN
)	2003	TOK_CLOSEPAREN
{	2004	TOK_OPENBRACE
}	2005	TOK_CLOSEBRACE

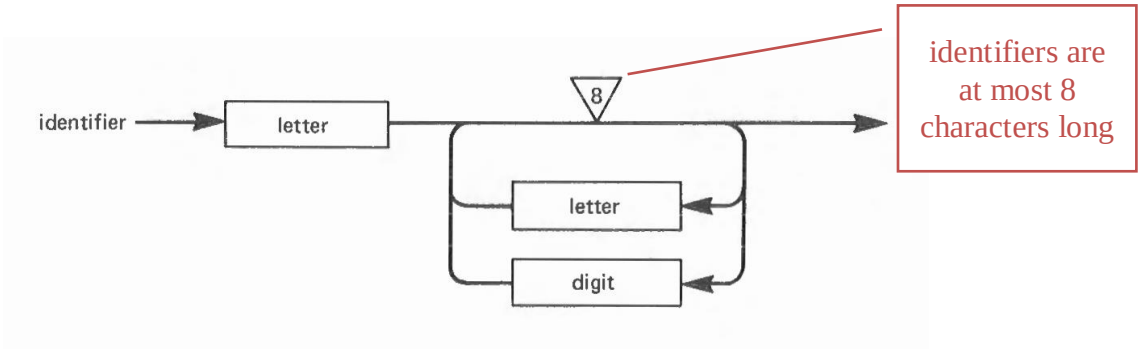
Operators	Token Identifier Value	Token Constant
+	3000	TOK_PLUS
-	3001	TOK_MINUS
*	3002	TOK_MULTIPLY
/	3003	TOK_DIVIDE
:=	3004	TOK_ASSIGN
=	3005	TOK_EQUALTO
<	3006	TOK_LESSTHAN
>	3007	TOK_GREATERTHAN
<>	3008	TOK_NOTEQUALTO
MOD	3009	TOK_MOD
NOT	3010	TOK_NOT
OR	3011	TOK_OR
AND	3012	TOK_AND

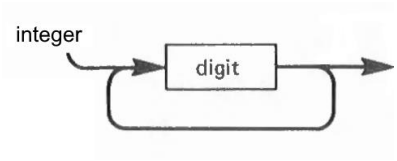
Useful Abstractions	Token Identifier Value	Token Constant
identifier	4000	TOK_IDENT
integer literal	4001	TOK_INTLIT
floating-point literal	4002	TOK_FLOATLIT
string literal	4003	TOK_STRINGLIT
end of file	5000	TOK_EOF
end of file while parsing a string literal	5001	TOK_EOF_SL
Unknown Lexeme	6000	TOK_UNKNOWN

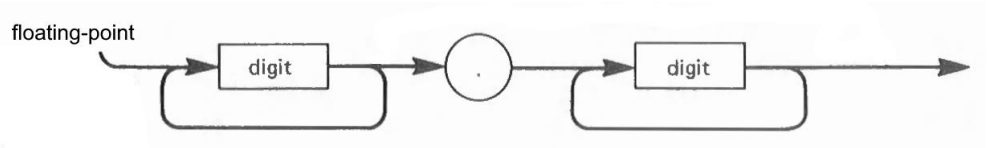
The following are syntax diagrams for some of the Useful Abstractions that we will recognize as lexemes.











Sequences of letters called strings (`TOK_STRINGLIT`) are enclosed between single quote marks. The maximum length of a string is 80 letters. Examples are:

```
'THE BANK BALANCE IS:'
'HAPPY BIRTHDAY!'
'ANY CHARACTERS OTHER THAN SINGLE QUOTE ARE PERMISSIBLE IN CHARACTER
STRINGS'
```

Other than separating lexemes, whitespace (space, tab, or newline) should be ignored by your lexical analyzer unless the whitespace occurs inside a string literal.

Ambiguity is resolved in favor of longer lexemes; therefore, the word `IFFINESS` in the input stream should create an identifier token, and not an `IF` keyword token followed by an identifier token.

Place the following header in your `rules.l` file. Modify the header to use your name, etc.

```
/*************************************************************************
  Name: Any Student             NetID: as3
  Course: CSE 4713              Assignment: Part 1
  Programming Environment: WSL C++
  Purpose of File: Contains the ....
*************************************************************************/
```

Deliverables

Place all of the source files (including files that you did not modify) needed to build your program using `make` in a zip file named *`yournetid`*`_part_1.zip` . For example, my submission would be named `jjc52_part_1.zip` .

Do NOT include object files (`*.o`), generated source code files (`lex.yy.c`), or executable files (`*.exe`) .

Upload your zip file to the assignment.

Grading

This is an individual assignment. Do not show / share your code with anyone else. You are responsible for debugging your own code. Most of programming is debugging. If you cannot debug your own code, you by definition have not written a program on your own.

The sample input / output shown below is a good starting point for testing your program. You are responsible for testing your program so that the lexical analyzer will function as expected with ***any*** input.

Program works with....	**Percentage**
Keywords and Datatype Specifiers	20
Operators	20
Useful Abstractions	20
Punctuation	10
Input 1	10
Input 2	10
Submitted Correctly	10

Submissions that are late will be penalized 10 points for each day past the due date/time. The timestamp of your last submission will be used to calculate the late penalty.

Any code that does not compile / run in the Windows Subsystem for Linux using `flex`, `make,` and `g++` will receive a grade of 0.

Sample input stream:

```
AB123('HELLO WORLD') # 321AB
FOREVER IF INTEGER FOR  + - ;
ELSE FOR PRINT WHILE
REAL 12.34 INT 234
STRING VAR OR :
'I AM A STRING, I AM ALSO A STRING'
'' 12. 34 AREALLYLONGNAME
```

Resulting output stream:

```
jjc52@CSE-CMSM6Y2:/mnt/c/proglang/Part1$ ./lex.exe < sample.in
INFO: Using stdin for input, use EOF to end input
      Windows EOF is Ctrl+z, Linux EOF is Ctrl+d
line: 1, lexeme: |AB123|, length: 5, token: 4000
line: 1, lexeme: |(|, length: 1, token: 2002
line: 1, lexeme: |'HELLO WORLD'|, length: 13, token: 4003
line: 1, lexeme: |)|, length: 1, token: 2003
line: 1, lexeme: |#|, length: 1, token: 6000
   ERROR: unknown token
line: 1, lexeme: |321|, length: 3, token: 4001
line: 1, lexeme: |AB|, length: 2, token: 4000
line: 2, lexeme: |FOREVER|, length: 7, token: 4000
line: 2, lexeme: |IF|, length: 2, token: 1007
line: 2, lexeme: |INTEGER|, length: 7, token: 1100
line: 2, lexeme: |FOR|, length: 3, token: 1006
line: 2, lexeme: |+|, length: 1, token: 3000
line: 2, lexeme: |-|, length: 1, token: 3001
line: 2, lexeme: |;|, length: 1, token: 2000
line: 3, lexeme: |ELSE|, length: 4, token: 1004
line: 3, lexeme: |FOR|, length: 3, token: 1006
line: 3, lexeme: |PRINT|, length: 5, token: 4000
line: 3, lexeme: |WHILE|, length: 5, token: 1015
line: 4, lexeme: |REAL|, length: 4, token: 1101
line: 4, lexeme: |12.34|, length: 5, token: 4002
line: 4, lexeme: |INT|, length: 3, token: 4000
line: 4, lexeme: |234|, length: 3, token: 4001
line: 5, lexeme: |STRING|, length: 6, token: 4000
line: 5, lexeme: |VAR|, length: 3, token: 1014
line: 5, lexeme: |OR|, length: 2, token: 3011
line: 5, lexeme: |:|, length: 1, token: 2001
line: 6, lexeme: |'I AM A STRING, I AM ALSO A STRING'|, length: 35, token: 4003
line: 7, lexeme: |''|, length: 2, token: 4003
line: 7, lexeme: |12|, length: 2, token: 4001
line: 7, lexeme: |.|, length: 1, token: 6000
   ERROR: unknown token
line: 7, lexeme: |34|, length: 2, token: 4001
line: 7, lexeme: |AREALLYL|, length: 8, token: 4000
line: 7, lexeme: |ONGNAME|, length: 7, token: 4000
```