CSE 4713 / 6713 — Programming Languages
Assignment 1

In this course, we will design and implement an interpreter for a string substitution language called *super-sub*. Our first task is to write a lexical analyzer for *super-sub*. 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 should use C++ for this assignment.

Here are the lexemes in *super-sub*:

Keywords	**Token Identifier Value**	**Token Constant**
combinator	1001	TOK_COMBINATOR
evaluate	1002	TOK_EVALUATE
dictionary	1003	TOK_DICTIONARY
set	1004	TOK_SET
Datatype Specifiers	**Token Identifier Value**	**Token Constant**
int	1100	TOK_INT
float	1101	TOK_FLOAT
string	1102	TOL_STRING
Punctuation	**Token Identifier Value**	**Token Constant**
;	2000	TOK_SEMICOLON
(	2001	TOK_OPENPAREN
)	2002	TOK_CLOSEPAREN
[	2003	TOK_OPENBRACKET
]	2004	TOK_CLOSEBRACKET
{	2005	TOK_OPENBRACE
}	2006	TOK_CLOSEBRACE
,	2007	TOK_COMMA
Operators	**Token Identifier Value**	**Token Constant**
\	3000	TOK_SLASH
.	3001	TOK_DOT
+	3002	TOK_PLUS
-	3003	TOK_MINUS
*	3004	TOK_MULTIPLY
/	3005	TOK_DIVIDE
:=	3006	TOK_ASSIGN
==	3007	TOK_EQUALTO
<	3008	TOK_LESSTHAN
>	3009	TOK_GREATERTHAN
<>	3010	TOK_NOTEQUALTO
and	3011	TOK_AND
or	3012	TOK_OR
not	3013	TOK_NOT
length	3014	TOK_LENGTH

Useful Abstractions	Token Identifier Value	Token Constant
identifier	4000	TOK_IDENTIFIER
c-identifier	4001	TOK_CIDENTIFIER
integer literal	4002	TOK_INTLIT
floating-point literal	4003	TOK_FLOATLIT
string	4004	TOK_STRINGLIT
End of file	5000	TOK_EOF
Unknown lexeme	6000	TOK_UNKNOWN

An identifier is defined by <letter> { <letter> | <digit> | ‘_’ }, where <letter> is any upper- or lower-case letter in the English alphabet, <digit> is any numeral 0..9, and _ is the underscore character. Therefore, this_is_an_identifier is a valid identifier whereas _this_is_not and 1more_bad_example are not a valid identifiers. Identifiers may not be keywords, however case is significant and keywords are always composed of lowercase characters. An identifier ends when a character that is not legal for an identifier is encountered. The c-identifier is a special case identifier that is defined by ‘$’ <letter> { <letter> | <digit> | ‘_’} where the c-identifier is just like the identifier except it is preceded with a ‘$’ symbol.

An integer literal consists of a sequence of digits without a decimal point.

A floating-point literal is a sequence of digits containing an embedded decimal point, or ending with a decimal point.

A string literal is a sequence of characters within double quotation marks.

Also, the ‘#’ character is the comment delimiter. Whenever the comment delimiter is identified all characters between this and then end-of-line (denoted by ‘\n’) is ignored by the lexical analyzer and discarded.

Regular expression for identifiers: [a-zA-Z][a-zA-Z0-9_]*

Regular expression for c-identifiers: [$][a-zA-Z][a-zA-Z0-9_]*

Regular expression for integer literal: [0-9][0-9]*

Regular expression for float literal: and [0-9][0-9]*[.][0-9]*

Regular expression for a string literal: \”.*\”

Lexeme terminators are whitespace characters, such as space, tab, or new-line (or comments). Other than separating lexemes, whitespace and comments should be ignored by your lexical analyzer.

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

getChar - gets next input character and stores in nextChar
must also determine the character class and store in charClass

addChar - add nextChar to lexeme

Your lexical analyzer should present the following interface to the calling program:

A. Your analyzer uses these global variables:

1. Input stream `yyin`

2. Output stream `yyout`

3. Integer `yyleng` contains the length of the currently identified lexeme. `int lexlen = yyleng`

4. character array `yytext` contains the identified lexeme `int lexlen = yyleng`

5. Integer `yyLine` that contains the current line number in the input file. Increment this number whenever a `'\n'` is parsed (if it terminates a comment line or otherwise.)

B. Your analyzer is composed of the following functions:

`yylex();` no parameters, returns an integer token that holds the value of the identified lexeme (the Token Identifier Value in the chart above).

Organize your program into two separate source files: `lexer.cpp` and `driver.cpp`. `lexer.cpp` should contain your lexical analyzer code in function `yylex()` and manage the global variables `yyleng` and `yytext`. `driver.cpp` declares and initializes the global variables, opens the input and output streams, and initializes the stream variables. `driver.cpp` then repeatedly calls `yylex()` until `yylex()` returns `TOK_EOF`.

For each lexeme, `driver.cpp` prints out the lexeme text, followed by the token identifier. Print out one lexeme per line, as illustrated below. It is highly recommended that you use the provided driver program without change as the verification of the program will be accomplished by doing a textual comparison between your program when provided a test input. Small deviations in the output will yield an invalid comparison. At the end of this program is an incomplete input test set and the expected output. You can use this to start your development, but you will need to develop a more comprehensive test set to test the full lexicon of the lexical analyzer.

Once your solution is complete and has been demonstrated to the Project Instructor, Daniel Rayborn dcr101@msstate.edu. This may be accomplished through emailing your driver.cpp, lexer.cpp and lexer.h, and Daniel will run it through the test input. If it fails he will give you some guidance on the tokens that it wasn't doing proper analysis so you can revise your testing of your program. Once you get a correct program certification from the project instructor, then you will submit your files through mycourses. You only need to submit the files `driver.cpp`, `lexer.cpp`, and `lexer.h`.

Sample input stream:

```
# The input can also admit comments
# comments are all characters after a '#' symbol to the end-of-line
Ab123("hello world") @ 321ab
forever set int  < <>
\ / $set.7 setting set ;
#This is a comment in the middle
float 12.34 int 234
string  or &
"I'm a string, I'm also a string"
```

Resulting output stream:

```
lexeme: |Ab123|, length: 5, token: 4000, line=3
lexeme: |(|, length: 1, token: 2001, line=3
lexeme: |"hello world"|, length: 13, token: 4004, line=3
lexeme: |)|, length: 1, token: 2002, line=3
lexeme: |@|, length: 1, token: 6000, line=3
   ERROR: unknown token
lexeme: |321|, length: 3, token: 4002, line=3
lexeme: |ab|, length: 2, token: 4000, line=3
lexeme: |forever|, length: 7, token: 4000, line=4
lexeme: |set|, length: 3, token: 1004, line=4
lexeme: |int|, length: 3, token: 1100, line=4
lexeme: |<|, length: 1, token: 3008, line=4
lexeme: |<>|, length: 2, token: 3010, line=4
lexeme: |\|, length: 1, token: 3000, line=5
lexeme: |/|, length: 1, token: 3005, line=5
lexeme: |$set|, length: 4, token: 4001, line=5
lexeme: |.|, length: 1, token: 3001, line=5
lexeme: |7|, length: 1, token: 4002, line=5
lexeme: |setting|, length: 7, token: 4000, line=5
lexeme: |set|, length: 3, token: 1004, line=5
lexeme: |;|, length: 1, token: 2000, line=5
lexeme: |float|, length: 5, token: 1101, line=7
lexeme: |12.34|, length: 5, token: 4003, line=7
lexeme: |int|, length: 3, token: 1100, line=7
lexeme: |234|, length: 3, token: 4002, line=7
lexeme: |string|, length: 6, token: 1102, line=8
lexeme: |or|, length: 2, token: 3012, line=8
lexeme: |&|, length: 1, token: 6000, line=8
   ERROR: unknown token
lexeme: |"I'm a string, I'm also a string"|,  length: 33, token: 4004, line=9
```