COP4020 Programming Assignment 1
Lexical Analyzer
Due February 10, 2016

Purpose

This project is intended to give you experience in using a scanner generator (Lex) to create a lexical analyzer for the CALC language.

Summary

Your task is to write a lexical analyzer for a simple calculator whose programming language, CALC, contains basic language constructs such as variables and assignment statements. You must write the Lex specification file that will be used to generate the scanner source code, lex.yy.c. The scanner source code is used in conjunction with a provided driver file (driver.c) to tokenize a given CALC source file. For example, when you have finished creating your lex specification file, you can test the scanner on linprog like so:

```
$ flex lexer.l
$ gcc driver.c –lfl
$ ./a.out < p0.cal
```

where p0.cal is a file containing CALC code. The output will be information about which tokens were identified and, if applicable, what the image of the token was.

Note that we can do all of this within the rules specified in the lexer.l file, but typically scanners are not used by themselves! They are called continually by parsers. So, we will simulate a parser continually calling our scanner code, except our "parser" isn't doing anything fancy – it's just printing out which token was found.

A sample executable, proj1_linprog, is provided in the project materials. Please use this to verify your output. You may try the sample executable like so:

```
$ ./proj1_linprog < p0.cal
```

All submissions must work on linprog with the command sequence given above.

Lexical Specification

The table below defines the tokens that must be recognized, with their associated symbolic names. Comments are enclosed in (* ... *), cannot be nested, and do not span multiple lines. An identifier is a sequence of (at least one) upper or lower case letters followed by zero or more digits. There is no limit on the length of identifiers. However, you may impose limits on the total number of characters in all distinct identifiers. There should be no other limitation on the number of lexemes that the lexical analyzer will process. An integer constant is an unsigned sequence of digits representing a base-10 number. Integer constants must start with a non-zero digit, unless the constant is simply zero itself.

The tokens in bold are shown exactly as they are to be found in the file. The non-bolded tokens represent a class of tokens that have an "image" – that is, they are found in many different ways. For example, the keyword "var" (VARnumber) will always be picked up as is, while an integer constant (ICONSTnumber) may have the image "2" or "10" or "572".

Token	Symbolic Name
;	SEMInumber
(	LPARENnumber
integer constant	ICONSTnumber
begin	BEGINnumber
program	PROGRAMnumber
-	MINUSnumber
*****	TIMESnumber
var	VARnumber
<<EOF>>	EOFnumber
,	COMMAnumber
)	RPARENnumber
identifier	IDnumber
end	ENDnumber
is	ISnumber
+	PLUSnumber
div	DIVnumber
print	PRINTnumber
=	EQnumber

Token attributes

A unique identification of each token must be returned by the lexical analyzer. In addition, the lexical analyzer must pass extra information for identifier and integer constant tokens to the driver. For identifier and integer constant tokens, the extra information is passed to the

driver as a single value of the integer type, through a global variable as described below (yylval). For integer constants, the numeric value of the constant is passed. In order to allow other phases of the compiler to access the original identifier lexeme, the lexical analyzer passes an integer uniquely identifying the identifier. The unique identifying number for identifiers should be an index into a string table created by the lexical analyzer to record the lexemes.

Implementation

I recommend that you start by checking out the contents of driver.c. The main function in driver.c simply continually calls yylex until yylex returns an EOF token. What this should be telling you is that your scanner should return the tokens as it picks them up, using the predefined symbolic names in driver.c. If comments or whitespace are found, they should be ignored (i.e. not returned to the driver – you'll still have to do something when you find them!).

Also note the use of *yylval* and *string_table*. The variable yylval, typically defined in a parser, is a vehicle for passing information about the images of the token. In our case, yylval should be an int type. We expect yylval to contain the integer value for ICONSTnumber tokens, and to contain an index into string_table for IDnumber tokens. The string_table is simply a character array which stores identifier images as strings in the following way (assuming we have three identifiers myint, a, and b):

m	y	i	n	t	\0	a	\0	b	\0	…

Therefore, the expression string_table+yylval should allow us to access the first position of the string in memory if yylval is the index of the string's first character. I recommend writing a function called which takes the string's image and length as arguments, places the string in the table in the next available position, and returns the index of its first character.

The central routine of the scanner is yylex, an integer function that returns a token number, indicating the type (identifier, integer constant, semicolon, etc...) of the next token in the input stream. In addition to the token type, yylex must set the global variables yyline and yycolumn to the line and column number at which that token appears (hint: modify yyline when you see a newline character and modify yycolumn when you see anything else), and in the case of identifiers and constants, put the extra information needed, as described above, into a global integer variable yylval. Lex will write yylex for you, using the patterns and rules defined in your lex input file (which should be called lexer.l). Your rules must include the code to maintain yyline, yycolumn and yylval.

Whenever an unknown character is found, print a message indicating the position of the error. You need only print the line and column number to indicate the position.

The #define mechanism is used in the driver.c file to allow the lexical analyzer to return token numbers symbolically. In order to avoid using token names that are reserved or significant in C/C++ or in the parser, the token names have been specified for you in the table above.

Answers to some common questions:
- You do not have to define main() in your .l file. The reason we defined main() in class is that our lexer was intended to be executed on its own. However, we will be using driver.c to drive the scanning process instead.

- Note the fact that lex.yy.c is included in driver.c. This means that the symbolic token names (i.e. ICONSTnumber) are available to the lex code. Feel free to use these in your actions for an associated regular expression. Notice how the driver captures the return value from yylex and compares it against these symbolic definitions. This should tell you that our actions will need to return a value to the driver.c.

- The variable yylval is typically predefined in a lex/YACC suite, however we are not using YACC this time so we will need to define yylval ourselves as an integer.

Assignment Submission

Submissions are due on February 10, 11:59 PM, before which you should submit your lexer.l file to Blackboard.

- Recognize correct tokens in the right order. (60).
- Report errors with line and column numbers (10).
- Correctly report integer images (10).
- Correctly report identifier images (20).
- First person to find errors in the sample executable (+5).

Your program will be tested with a series of CALC programs. Some of the testing programs are provided in the project package. You are encouraged to modify the programs to further test your scanner. If your lexer.l file cannot be used to create a legitimate lex.yy.c, 0 points will be given.