University of Central Florida
Department of Electrical Engineering & Computer Science
COP 3402: System Software

Homework #2 (Lexical Analyzer)

Goal

In this assignment you will implement a lexical analyzer for the programming language PL/0. More precisely, you are going to implement a lexical analyzer function which has the following signature:

```
LexerOut lexicalAnalyzer(char* sourceCode);
```

Your function must be capable to do lexical analysis on the given source program written in PL/0, identify some errors, and give the source program's lexeme table (*LexerOut*). ***For an example of input and output refer to Appendix A***. In addition, the scanner must **NOT** generate the Symbol Table, which contains all the variables, procedure names and constants within the PL/0 program.

As follows we show you an example source code in PL/0 and the grammar for the programming language PL/0 using the Extended Backus-Naur Form (EBNF).

Example of a program written in PL/0:

```
var x, w;
begin
   read w;
   x:= 4;
   if w > x then
      w:= w + 1
   else
      w:= x;
   write w;
end.
```

Based on Wirth's definition for EBNF we have the following rules:

[ ] means **an optional item**,

{ } means **repeat 0 or more times**.
Terminal symbols are enclosed in quote marks.
A period is used to indicate the end of the definition of a syntactic class.

EBNF of PL/0:

```
program ::= block "." .
block ::= const-declaration  var-declaration proc-declaration statement.
constdeclaration ::= [ "const" ident "=" number {"," ident "=" number} ";"].
var-declaration  ::= [ "var" ident {"," ident} ";"].
proc-declaration::= {"procedure" ident ";" block ";" } statement .
statement   ::= [ ident ":=" expression
                    | "call" ident
                | "begin" statement { ";" statement } "end"
                | "if" condition "then" statement ["else" statement]
               | "while" condition "do" statement
                    | "read" ident
                    | "write" ident
                | e ] .

condition ::= "odd" expression
            | expression  rel-op  expression.

rel-op ::= "=" | "<>" | "<" | "<=" | ">" | ">=".
expression ::= [ "+" | "-" ] term { ( "+" | "-" ) term}.
term ::= factor {( "*" | "/" ) factor}.
factor ::= ident | number | "(" expression ")".
number ::= digit {digit}.
ident ::= letter {letter | digit}.
digit ;;= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
letter ::= "a" | "b" | … | "y" | "z" | "A" | "B" | ... | "Y" | "Z".
```

PL/0 LexicalConventions:
A numerical value is assigned to each token (internal representation) as follows:
nulsym = 1, identsym = 2, numbersym = 3, plussym = 4, minussym = 5, multsym = 6, slashsym = 7, oddsym = 8, eqlsym = 9, neqsym = 10, lessym = 11, leqsym = 12,
gtrsym = 13, geqsym = 14, lparentsym = 15, rparentsym = 16, commasym = 17, semicolonsym = 18, periodsym = 19, becomessym = 20, beginsym = 21, endsym = 22, ifsym = 23, thensym = 24, whilesym = 25, dosym = 26, callsym = 27, constsym = 28, varsym = 29, procsym = 30, writesym = 31, readsym = 32, elsesym = 33.

Reserved words: const, var, procedure, call, begin, end, if, then, else, while, do, read, write.
Special symbols: '+', '-', '*', '/', '(', ')', '=', ',' , '.', ' <', '>', ';' , ':' .
Identifiers: identsym = letter (letter | digit)*
Numbers: numbersym = $(digit)^+$
Invisible characters: tab, white spaces, newline

Comments denoted by: /* . . . */

Input

The input to your program is a PL/0 source code. The source code is passed as a null-terminated string (a character array which ends with null character '\0') to the function *lexicalAnalyzer()*, which you should be implementing inside *lexical_analyzer.c* file.

The input source may **NOT** be grammatically valid PL/0 code. The followings are the only constraints that you should consider and check for.

1. Identifiers can be a maximum of 11 characters in length.
2. Numbers can be a maximum of 5 digits in length.
3. Comments should be ignored and not tokenized.
4. Invisible Characters should be ignored and not tokenized.

Output

You should return a *LexerOut* struct from the function *lexicalAnalyzer()*, which is defined and should be implemented by you in the file *lexical_analyzer.c*. ***For the type definition of LexerOut and the lexer error enumeration refer to Appendix B***. If the given PL/0 source code is grammatically valid, you should fill the token list field, ie. *LexerOut::tokenList*, of the *LexerOut* structure with tokens in correct order. In case of a lexical error, which are defined in this assignment text, you should fill the lexer error and error line information, ie. *LexerOut::lexerError* and *LexerOut::errorLine*, with the type of error you encountered and the line of the error. Take the first line of the code as 0^{th} line.

Detect the following lexical errors:

1. **LexErr::NONLETTER_VAR_INITIAL**: Variable does not start with letter.
2. **LexErr::NUM_TOO_LONG**: Number too long.
3. **LexErr::NAME_TOO_LONG**: Name too long.
4. **LexErr::INV_SYM**: Invalid symbols.

You could create a transition diagram (DFS) to recognize each lexeme on the source program and once accepted generate the token, otherwise emit an error message.

Notice that you are not going to print anything in this assignment. After you return well-formed instance of *LexerOut* from *lexicalAnalyzer()*, the skeleton code given to you will process and print your output. For further information on how to fill the *LexerOut* struct, you may refer to the documentation inside *lexical_analyzer.h* file.

Appendix A:

If the input is:

```
/* An example PL/0 program */

const m = 7, n = 85;
var  i,x,y,z,q,r;
procedure mult;
  var a, b;
  begin
     a := x;  b := y; z := 0;
     while b > 0 do
     begin
        if odd x then z := z+a;
           a := 2*a;
           b := b/2;
     end
  end;

/* The main function assigns the value
   .. stored in m to x. Then, proceeds... */
begin
  x := m;
  y := n;
  call mult;
end.
```

The output will be:

```
SUCCESS: No errors found the source code.

Token Type        Lexeme
     28          const
      2              m
      9              =
      3              7
     17              ,
      2              n
      9              =
      3             85
     18              ;
```

```
29      var
 2       i
17       ,
 2       x
17       ,
 2       y
17       ,
 2       z
17       ,
 2       q
17       ,
 2       r
18       ;
30  procedure
 2     mult
18       ;
29      var
 2       a
17       ,
 2       b
18       ;
21    begin
 2       a
20       :=
 2       x
18       ;
 2       b
20       :=
 2       y
18       ;
 2       z
20       :=
 3       0
18       ;
25    while
 2       b
13       >
 3       0
26      do
21    begin
23      if
 8     odd
 2       x
```

24	then
2	z
20	:=
2	z
4	+
2	a
18	;
2	a
20	:=
3	2
6	*
2	a
18	;
2	b
20	:=
2	b
7	/
3	2
18	;
22	end
22	end
18	;
21	begin
2	x
20	:=
2	m
18	;
2	y
20	:=
2	n
18	;
27	call
2	mult
18	;
22	end
19	.

Appendix B:

```
/**
* LexerOut struct: the return value of lexicalAnalyzer() func
* */
typedef struct {
/**
* The list of tokens to be filled by lexicalAnalyzer(). The essential
* .. field that should be filled after successful execution of lexical
* .. analysis, ie., when encountered no errors.
* */
TokenList tokenList;


/**
* Should be filled when an error is encountered while lexical analysis.
* Otherwise, should be kept as NONE to signal success.
* Be careful that the errorLine field should also be filled with the
* .. Number of the line the error was encountered when LexErr is
* .. different than NONE.
* */
LexErr lexerError;


/**
* Should be filled when LexErr is encountered. Indicates the number of
* .. line that the LexErr is encountered.
* */
int errorLine;

} LexerOut;
```

```
/**
* Enumeration for possible lexer errors.
* */
typedef enum {
   NONE = 0,
   NONLETTER_VAR_INITIAL,
   NAME_TOO_LONG,
   NUM_TOO_LONG,
   INV_SYM,
   NO_SOURCE_CODE
} LexErr;
```