COP 3402: System Software
Summer 2014

Programming Project Module #2 (Lexical Analyzer)

Due Sunday 06/14 by 11:59 PM

Lexical Analyzer

Your task is to implement a lexical analyzer (scanner) for the programming language PL/0. Your program must be capable to read a source program written in PL/0, identify some errors, and produce as output:

- the source program (without comments),
- the lexeme table, and
- the list of lexemes.

For an example of input and output refer to Appendix A. The grammar for the programming language PL/0 using the extended Backus-Naur Form (EBNF) is presented below.

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

[ ] means an optional item**, { }** means repeat 0 or more times**,** and **ε** denotes the empty string. **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.
const-declaration ::= [ "const" ident "=" number { "," ident "=" number } ";" ].
var-declaration   ::= [ "var" ident { "," ident } ";" ].
proc-declaration  ::= { "procedure" ident ";" block ";" }.
statement         ::= [ ident ":=" expression
                      | "call" ident
                      | "begin" statement { ";" statement } "end"
                      | "if" condition "then" statement [ "else" statement]
                      | "while" condition "do" statement
                      | "read" ident
                      | "write" ident
                      | ε ].


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".
```

Example PL/0 program (lexically correct, but not necessarily grammatically correct):

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

Lexical Conventions for PL/0:

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, odd.
```

Special Symbols:

```
"+", "-", "*", "/", "(", ")", "=", "," ,".", "<", ">", ";",
":"
```

Identifiers:

```
identsym = letter (letter | digit)*
```

Numbers:

```
numbersym = (digit)+
```

Invisible Characters:
tab, white spaces, newline

Comments denoted by:
`/* ... */`

Refer to **Appendix B** for a C declaration of the token symbols that may be useful.

Constraints:

Input:

1. Input filename must be called "input.txt". This is the file with the input PL/0 program.
2. Identifiers can be a maximum of 11 characters in length.
3. Numbers can be a maximum of 5 digits in length.
4. Comments should be ignored and not tokenized.
5. Invisible Characters (space, tab, newline) should not be tokenized.

Important Note: Input files may NOT be grammatically valid PL/0 code.

Output:

1. The source program without comments must be saved in a file named "cleaninput.txt".
2. The lexeme table must be saved in a file named "lexemetable.txt".
3. The lexeme list must be saved in a file named "lexemelist.txt".
4. The token separator in the output's lexeme list must be a space character (refer to Appendix A).
5. In your output's lexeme list, identifiers must show the token and the variable name separated by a space.
6. In your output's Lexeme List, numbers must show the token and the value separated by a space. The value must be transformed into ASCII Representation (as discussed in class).
7. The token representation of the Lexeme List will be used in the Parser (Project 3). So, PLAN FOR IT!

Detect the Following Lexical Errors:

1. Variable does not start with letter.
2. Number too long.
3. Name too long.
4. Invalid symbols.

Hint: 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.

Submission Instructions:

Submit to Webcourses (inside a single zip file):

1. Source code of your lexical analyzer.
2. Instructions to compile and use your program in a readme document.
3. One run containing the input file (Source Program), and output files (clean source, lexeme table and lexeme list).

Note: it is mandatory that you test your scanner on Eustis, given that the char reading and handling in Eustis may behave differently than it does for other platforms.

Appendix A:

If the input is:

input.txt

```
var x, y;
begin
  y := 3; /* This is a comment */
  x := y + 56;
end.
```

The output will be:

cleaninput.txt

```
var x, y;
begin
  y := 3;
  x := y + 56;
end.
```

lexemetable.txt

```
lexeme    token type
var       29
x         2
,         17
y         2
;         18
begin     21
y         2
:=        20
3         3
;         18
x         2
:=        20
y         2
+         4
56        3
;         18
end       22
.         19
```

lexemelist.txt:

29 2 x 17 2 y 18 21 2 y 20 3 3 18 2 x 20 2 y 4 3 56 18 22 19

Appendix B:

Declaration of Token Types:

```
typedef enum {
  nulsym = 1, identsym, numbersym, plussym, minussym,
  multsym,  slashsym, oddsym, eqsym, neqsym, lessym, leqsym,
  gtrsym, geqsym, lparentsym, rparentsym, commasym, semicolonsym,
  periodsym, becomessym, beginsym, endsym, ifsym, thensym,
  whilesym, dosym, callsym, constsym, varsym, procsym, writesym,
  readsym , elsesym
} token_type;
```

Example of Token Representation:

```
29  2  x  17  2  y  18  21  2  x  20  2  y  4  3  56  18  22
19
```

Is equivalent to:

```
varsym identsym x commasym identsym y semicolonsym beginsym
identsym x becomessym identsym y plussym numbersym 56
semicolonsym  endsym periodsym
```

Appendix C:

Example of a 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;

begin
  x := m;
  y := n;
  call mult;
end.
```