University of Central Florida
School of Electrical Engineering & Computer Science
COP 3402: System Software
Fall 2023

Homework #2 (Lexical Analyzer)

Due Friday, September 29th, 2023 by 11:59 p.m.

Goal:

In this assignment your team have to implement a lexical analyzer for the programming language PL/0. Your program must be capable to read in a source program written in PL/0, identify some errors, and produce, as output, the source program, the source program lexeme table, and the token list. ***For an example of input and output refer to Appendix A.*** In the next page we show you the grammar for the programming language PL/0 using the extended Backus-Naur Form (EBNF).

You will use the given Context Free Grammar (see next page) to identify all symbols the programming language provides you with. These symbols are shown below:

Reserved Words: **const, var, procedure, call**, **begin**, **end**, **if**, **then**, **ifel, else**, **while**, **do**, **read**, **write**.
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 declaration of the token symbols that may be useful.*

In this assignment, you will not check syntax.

Example1: program written in PL/0:

```
var x, y;
begin
        x := y * 2;
end.
```

Use these rules to read PL/0 grammar expressed in EBNF.

1.- [ ] means an optional item,
2.- { } means repeat 0 or more times.
3.- Terminal symbols are enclosed in quote marks.
4.- Symbols without quotes are called no-terminals or a syntactic class.
5.-A period is used to indicate the end of the definition of a syntactic class.
6.-The symbol '::=' is read as 'is defined as'; for example, the following syntactic class:

program ::= block "**.**".

must be read as follows:
a program **is defined as** a block followed by a dot.
program **::=** block "**.**".

Context Free Grammar for PL/0 expressed in EBNF.

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
| "**ifel**" condition "**then**" statement "**else**" statement
| "**while**" condition "**do**" statement
| "**read**" ident
| "**write**" ident
| empty] **.**
empty ::=

condition ::= expression rel-op expression**.**

rel-op ::= "**=**"|"**<>**"|"**<**"|"**<=**"|"**>**"|"**>=**"**.**
expression ::= ["**+**"|"**-**"] term { ("**+**"|"**-**") term}**.**
term ::= factor {("*"|"**/**") factor}**.**
factor ::= ident | number | "**(**" expression "**)**"**.**

In this assignment, you will identify valid PL/0 symbols and then translate them into an internal representation called "Tokens".

Lexical Grammar for PL/0 expressed in EBNF.

ident ::= letter {letter | digit}.
letter ::= "**a**" | "**b**" | ... | "**y**" | "**z**" | "**A**" | "**B**" | ... | "**Y**" | "**Z**".
number ::= digit {digit}.
digit ::= "**0**" | "**1**" | "**2**" | "**3**" | "**4**" | "**5**" | "**6**" | "**7**" | "**8**" | "**9**".

Lexical Conventions for PL/0:

A numerical value is assigned to each token (internal representation) as follows:

skipsym = 1, identsym = 2, numbersym = 3, plussym = 4, minussym = 5,
multsym = 6, slashsym = 7, ifelsym = 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.

Example2: program written in PL/0:

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

Remember, in this assignment, you will not check syntax.

For the scanner
x := y + 7; and + 7 ; x y :=. are valid inputs

Constraints:

Input:

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:

1. The token separator in the output's Lexeme List (Refer to Appendix A) can be either a space or a bar ('|').
2. In your output's Lexeme List, identifiers must show the token and the variable name separated by a space or bar.
3. In your output's Token list, numbers must show the token and the value separated by a space or bar. The value must be transformed into ASCII Representation (as discussed in class)
4. Be consistent in output. Choose either bars or spaces and stick with them.
5. The token representation of the Token list will be used in the Parser (HW3). So, PLAN FOR IT!

Detect the Following Lexical Errors:

1. Number too long.
2. Name too long.
3. 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 Webcourse:

1. Source code.
2. Instructions to use the program in a **readme document**.
3. One run containing the input file (Source Program), and output in a file (Source, Lexeme Table(lexeme-token), Token List)

Appendix A:

If the input is:

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

The output will be:

Source Program:

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

Lexeme Table:

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

Token List:

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 {
skipsym = 1, identsym, numbersym, plussym, minussym,
multsym,  slashsym, ifelsym, 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 1 17 2 2 18 21 2 1 20 2 2 4 3 56 18 22 19"

Is Equivalent:

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 x =1 then z := z+a;
        a := 2*a;
        b := b/2;
   end
 end;

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

Find out the output for this example!