University of Central Florida
Department Computer Science
COP3402: Systems Software
Fall 2021

Homework #2 (Lexical Analyzer)
(Solo or Team assignment – Same team members who implemented HW1)

If you do not follow the specifications your grade will be zero

Goal:
In this assignment you have to implement a lexical analyzer for the programming language PL/0. Your program must be able to read in a source program written in PL/0, identify some errors, and produce, as output, the source program lexeme table and a list of tokens. ***For an example of input and output refer to Appendix A***.

Lexical Conventions for PL/0:
A numerical value is assigned to each token (internal representation) as follows:
constsym = 1, varsym = 2, procsym = 3, beginsym = 4, endsym = 5, whilesym = 6, dosym = 7, ifsym = 8, thensym = 9, elsesym = 10, callsym = 11, writesym = 12, readsym = 13, identsym = 14, numbersym = 15, assignsym = 16, addsym = 17, subsym = 18, multsym = 19, divsym = 20, modsym = 21, eqlsym = 22, neqsym = 23, lsssym = 24, leqsym = 25, gtrsym = 26, geqsym = 27, oddsym = 28, lparensym = 29, rparensym = 30, commasym = 31, periodsym = 32, semicolonsym = 33.

Reserved Words ::= const, var, procedure, call, if, then, else, while, do, begin, end, read, write, odd.
Special Symbols **::= == != < <= > >= % * / + - ( ) , . ; :=**
Identifiers: identsym **::=** letter (letter | digit)*
Numbers: numbersym **::=** (digit)$^+$
Invisible Characters: tab, white spaces, newline
Comments begin with // and end at the next newline. Multiline comments are not accepted. Take care as in Windows text files a new line is '\r\n' while in Linux and Mac systems a new line is '\n' There is no way to escape from a comment once started, except by reaching the newline.
*Refer to **Appendix B** for a declaration of the token symbols that may be useful.*

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.

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

Output:

1. In your output's Token List, identifiers must show the token and the variable name separated by a space.
2. In your output's Token List, numbers must show the token and the value separated by a space.
3. The token representation of the Token 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.

Submission Instructions:

Submit to Webcourse:

1. Source code named lex.c
2. Instructions to use the program in a readme text file.
3. This is a team assignment (the same team members who worked together in HW1) or a solo assignment.
4. Only one submission per team.
5. The name of all team members must be written in all source code header files, in a comment on the submission, and in the readme.
6. Include comments in your program.
7. Output should print to the screen and should follow the format in Appendix A. A deduction of 5 points will be applied to submissions that do not print to the screen.
8. The input file should be given as a command line argument. A deduction of 5 points will be applied to submissions that do not implement this.

Hints:

- 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.
- Use the C function iscntrl() to check for whitespace rather than hardcoding acceptable control characters. Your program should function regardless of what control characters are present in the input file. iscntrl() will not check for a standard space character (‘ ‘)
- Use the C functions isalpha() and isdigit() to check for letters and digits respectively.
- The only guaranteed whitespace is the whitespace that separates identifiers, numbers, and reserved words. All other whitespace is optional. Assuming no length errors:

    - If an identifier is followed by a number with no whitespace, it is an identifier.
    - If an identifier is followed by a reserved word with no whitespace, it is an identifier.
    - If an identifier is followed by an identifier with no whitespace, it is an identifier.
    - If a number is followed by an identifier with no whitespace, it is an invalid identifier error.
    - If a number is followed by a reserved word with no whitespace, it is an invalid identifier error.
    - If a number is followed by a number with no whitespace, it is a number.
    - If a reserved word is followed by an identifier with no whitespace, it is an identifier
    - If a reserved word is followed by a number with no whitespace, it is an identifier
    - If a reserved word is followed by a reserved word with no whitespace, it is an identifier.

Error Handling:

- When your program encounters an error, it should print out an error message and stop.
- When you are reading in a token that begins with a letter, read in characters until you reach one that is not alphanumeric (at which point you check for reserved word and move on to the next token) or you reach the twelfth consecutive alphanumeric character (at which point you print the Excessive Identifier Length error and stop).
- When you are reading in a token that begins with a number, read in characters until you reach one that is not numeric (at which point you tokenize the number and move on) OR you reach the sixth consecutive number (at which point you print the Excessive Number Length error and stop) OR you reach a letter (at which point you print the Invalid Identifier error and stop).
- When you are reading in a token that starts with an invalid symbol, print out the error message and stop.

For this assignment, we are providing you with a skeleton. You must implement the function lexeme *lexanalyzer(char *input) in lex.c. You may add as many helper functions and global variables as you desire. This function takes the input file contents as a string and should return the lexeme list unless there is an error, in which case it should return NULL. lex.c also includes two printing functions: printerror which will print the error message and free the lexeme list, and printtokens which will print the lexeme table and token list and mark the end of the lexeme list. Make sure to call one of these functions before you return from the lexanalyzer function. For these functions to work, you must have the list and list index as global variables. We also provide driver.c which handles reading in the input file and calling lexanalyzer as well as compiler.h which includes the lexeme struct and token type enumeration. Additionally a make file, two examples, and a bash script for testing are included.

Rubric
-100 – Does not compile
15 – Compiles
20 – Produces some entries to list/table before segfaulting or looping infinitely
5 – Follows IO specifications (takes command line argument for input file name and prints output to console)
5 – README.txt containing author names
10 – Prints both the list and table formats
15 – Prints out message and stops execution after encountering error
5 – Is not context dependent
5 – Is not whitespace dependent
10 – Supports all four errors
10 – Supports all thirty-three symbols

Appendix A: **(More examples can be found in the skeleton file on Webcourses)**

If the input is:

```
// this program multiplies two numbers given by the user
const one := 1;
var x, y, result;
procedure mult;
     begin
     if x > one then     // x number of recursive calls
          begin
               x := x - one;
               call mult;
          end;
     result := result + y; // after we make x recursive calls we
can start adding y
     end;
begin
     // initialize values
     read x;
     read y;
     result := 0;
     call mult;
     write result;
end.
```

The output should be:

```
Lexeme Table:
lexeme         token type
      const    1
        one    14
         :=    16
          1    15
          ;    33
        var    2
          x    14
          ,    31
          y    14
          ,    31
     result    14
          ;    33
  procedure    3
       mult    14
          ;    33
      begin    4
         if    8
          x    14
          >    26
        one    14
```

```
         then    9
        begin    4
            x    14
           :=    16
            x    14
            -    18
          one    14
            ;    33
         call    11
         mult    14
            ;    33
          end    5
            ;    33
       result    14
           :=    16
       result    14
            +    17
            y    14
            ;    33
          end    5
            ;    33
        begin    4
         read    13
            x    14
            ;    33
         read    13
            y    14
            ;    33
       result    14
           :=    16
            0    15
            ;    33
         call    11
         mult    14
            ;    33
        write    12
       result    14
            ;    33
          end    5
            .    32

Token List:
1 14 one 16 15 1 33 2 14 x 31 14 y 31 14 result 33 3 14 mult 33
4 8 14 x 26 14 one 9 4 14 x 16 14 x 18 14 one 33 11 14 mult 33 5
33 14 result 16 14 result 17 14 y 33 5 33 4 13 14 x 33 13 14 y
33 14 result 16 15 0 33 11 14 mult 33 12 14 result 33 5 32
```

Appendix B:

Declaration of Token Types:

```
typedef enum {
        constsym = 1, varsym, procsym, beginsym, endsym, whilesym, dosym, ifsym, thensym,
        elsesym, callsym, writesym, readsym, identsym, numbersym, assignsym, addsym,
        subsym, multsym, divsym, modsym, eqlsym, neqsym, lsssym, leqsym, gtrsym, geqsym,
        oddsym, lparensym, rparensym, commasym, periodsym, semicolonsym
} token_type;
```

Token Definitions:

constsym	1	const
varsym	2	var
procsym	3	procedure
beginsym	4	begin
endsym	5	end
whilesym	6	while
dosym	7	do
ifsym	8	if
thensym	9	then
elsesym	10	else
callsym	11	call
writesym	12	write
readsym	13	read
idensym	14	identifiers
numbersym	15	numbers
assignsym	16	:=
addsym	17	+
subsym	18	-
multsym	19	*
divsym	20	/
modsym	21	%
eqlsym	22	==
neqsym	23	!=
lsssym	24	<
leqsym	25	<=
gtrsym	26	>
geqsym	27	>=
oddsym	28	odd
lparensym	29	(
rparensym	30	)
commasym	31	,
periodsym	32	.
semicolonsym	33	;