University of Central Florida
Department Computer Science
COP3402: Systems Software
Almalki, Spring 2022

Homework #2 Lexical Analyzer

MAKE SURE YOU HAVE READ THE PROJECT OVERVIEW, ESPECIALLY THE PROJECT INSTRUCTIONS AND GUIDELINES SECTION

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 using PL/0 tokens, identify some errors, and produce, as output, the source program lexeme table.

Lexical Conventions for PL/0:

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

periodsym = 1, varsym = 2, lbracketsym = 3, procsym = 4, rbracketsym = 5,
callsym = 6, commasym = 7, beginsym = 8, semicolonsym = 9, endsym = 10,
assignsym = 11, ifsym = 12, questionsym = 13, dosym = 14, colonsym = 15,
readsym = 16, lparenthesissym = 17, writesym = 18, rparenthesissym = 19, identsym = 20,
eqlsym = 21, numbersym = 22, neqsym = 23, modsym = 24, lsssym = 25,
divsym = 26, leqsym = 27, multsym = 28, gtrsym = 29, subsym = 30,
geqsym = 31, addsym = 32, whilesym = 33

Reserved Words ::= var, procedure, call, begin, end, if, do, while, read, write
Special Symbols **::= . [ ] , ; := ? : ( ) == <> < <= > >= + - * / %**
Identifiers: identsym **::=** letter (letter | digit)*
Numbers: numbersym **::=** $(digit)^{+}$
Invisible Characters: control characters like tab and newline (can be detected with iscntrl()) and whitespaces (can be detected with isspace())
Comments begin with // and end with a newline OR EOF (‘\0’). Comments should be ignored and not tokenized Please note that newline on Macs and Linux are ‘\n’ while newlines on Windows are ‘\r\n’ by default. Your program should work for either.

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

Detect the Following Lexical Errors:

1. Invalid Identifier (identifiers should not begin with digits)
2. Number Length (at most 5 digits)
3. Identifier Length (at most 11 letters or digits)
4. Invalid Symbol

Submission Instructions:

Submit to Web-courses:

1. Source code named lex.c (if you alter driver.c or compiler.h, please also submit the altered versions and explain changes in your readme)
2. Instructions to use the program and author names in a readme text file. (If you didn't alter the package structure, your readme will just be author names)
3. Only one submission per team; 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. (If your partner receives a grade but you don't just send us a message/email)
4. Include comments in your program.
5. Output should print to the screen and should follow the format in Appendix A. If your output does not follow this format, you will experience a delay in grading.

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.
- Don't try to detect grammar errors in the lexical analyzer, and don't expect all test cases to be grammatically correct.
- 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 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 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, you must implement the function lexeme *lexanalyzer(char *input) in lex.c. The Project Overview contains an explanation of testing and the skeleton. 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. The primary function marks the end of the lexeme list before returning. This is necessary for the parser to function and should be preserved. 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. 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 compiler.h which includes the lexeme struct and token type enumeration.

Rubric

15	Compiles
20	Produces some entries to list/table (which clearly change depending on the input file) before segfaulting, looping infinitely, or erroring on all inputs
5	Reading Instructions (readme.txt indicates if provided support files were changed, author names are in the header comment, readme, and submission comment)
10	Prints out message and stops execution after encountering error
20	Supports all four errors (5 points each)
5	Is not context dependent (“var x := 5” is recognized as varsym identsym assignsym numbersym; “5 x := var” is recognized as numbersym identsym assignsym varsym)
5	Is not whitespace dependent (“; x < <=“ and “;x<<=“ are both recognized as semicolonsym identsym lsssym leqsym)
10	Supports identifier, number, and reserved word symbols
5	Supports special symbols
5	Supports comments

Appendix A:

If the input is:

```
var x, [3];
procedure func
     // this program should generate a parser error
     if <> ? problem : bueno
     y(2) := 4 * 8 % 11
     end
begin
     call
     do
     while > < == candy : /**/
     read write <===>=-+
.
```

The output should be:

```
Lexeme Table:
lexeme          token type
        var     2
          x     20
          ,     7
          [     3
          3     22
          ]     5
          ;     9
  procedure     4
       func     20
         if     12
         <>     23
          ?     13
    problem     20
          :     15
      bueno     20
          y     20
          (     17
          2     22
          )     19
         :=     11
          4     22
          *     28
          8     22
          %     24
         11     22
        end     10
```

```
      begin    8
       call    6
         do    14
      while    33
          >    29
          <    25
         ==    21
      candy    20
          :    15
          /    26
          *    28
          *    28
          /    26
       read    16
      write    18
         <=    27
         ==    21
         >=    31
          -    30
          +    32
          .    1

Parser Error: Symbol names must be identifiers
```

Appendix B:

Declaration of Token Types:

```
typedef enum token_type {
     periodsym = 1, varsym, lbracketsym, procsym, rbracketsym,
     callsym, commasym, beginsym, semicolonsym, endsym,
     assignsym, ifsym, questionsym, dosym, colonsym,
     readsym,  lparenthesissym,  writesym,  rparenthesissym,
identsym,
     eqlsym, numbersym, neqsym, modsym, lsssym,
     divsym, leqsym, multsym, gtrsym, subsym,
     geqsym, addsym, whilesym
} token_type;
```

Token Definitions:

periodsym	1	.
varsym	2	var
lbracketsym	3	[
procsym	4	procedure
rbracketsym	5	]
callsym	6	call
commasym	7	,
beginsym	8	begin
semicolonsym	9	;
endsym	10	end
assignsym	11	:=
ifsym	12	if
questionsym	13	?
dosym	14	do
colonsym	15	:
readsym	16	read
lparenthesissym	17	(
writesym	18	write
rparenthesissym	19	)
identsym	20	*identifiers*
eqlsym	21	==
numbersym	22	*numbers*
neqsym	23	<>
modsym	24	%
lsssym	25	<
divsym	26	/
leqsym	27	<=
multsym	28	*
gtrsym	29	>
subsym	30	-

```
geqsym              31   >=
addsym              32   +
whilesym            33   while
```