1 The problem

You have been selected to write a compiler for the PL/0 language. In this assignment you 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's lexeme table, and a list of lexemes. *For an example of input and output refer to Appendix A.* The scanner must **not** generate the Symbol Table, which contains all of the variables, procedure names and constants within the PL/0 program. As follows we show you the grammar for the programming language PL/0 using the Extended Backus-Naur Form (EBNF).

Table 1: EBNF definition 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 .
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".
EBNF->Wirth's 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.

1.1 PL/0 lexical conventions

- 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.

Table 2: PL/0 lexical words & symbols

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	/* . . . */

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

1.2 Output format specifications

The code used to generate the test cases described in the following section use the following ***printf*** formats:

```
// Print input file header
    fprintf(stdout, "Source Program:%s\n", argv[1]);
// Read in character by character from input code, and print input file
    fprintf(stdout, "%c", code[codeIndex]);
// Error messages
    printf("Error: Identifier too long.");
    printf("Error: Invalid identifier.");
    printf("Error: Number too large.");

// Print lexeme table
   fprintf(stdout, "\nLexeme Table:\n");
   fprintf(stdout, "lexeme\t\ttoken type\n");

   for(k = 0; k < lexTableIndex; k++)
    fprintf(stdout, "%s\t\t%d\n", lex.name, lex.token);

// Print lexeme list
   fprintf(stdout, "\nLexeme List:\n");
   for(k = 0; k < lexTableIndex; k++){
    fprintf(stdout, "%d ", lexeme_table[k].token);
    // If an identifier, print variable name
    if(lex.token == 2)
        fprintf(stdout, "%s ", lex.name);
    // If number, print its ascii number value
    else if(lex.token == 3)
        fprintf(stdout, "%d ", lex.val);
    }
```

Note: *The print formats above should be used as the format specification used for testing. The variable names and structure are bogus and will need to be adjusted for your usage.*

2 Example input/outputs

Given the input text is the contents of *inputLex.txt*:

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

The output would contain the following elements:

- The actual text in the input file, line by line.
- The *lexeme* table.
- The *lexeme* list.

```
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

Lexeme 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
```

3 Test Cases

There are seven input test files specifically created to verify the functions in the following table. *Note that the test script is shown below in the **Test Script** section.*

Table 3: PL/0 Test cases

Test name	Input filename	Purpose
lexical-1	inputLex.txt	Simple case for valid but simple program
lexical-2	symbIn.txt	Accepted symbol tests
lexical-3	err1In.txt	Variable does not start with a letter
lexical-4	err2In.txt	Number more than five digits
lexical-5	err3In.txt	Name too long
lexical-6	err4In.txt	Invalid symbols
lexical-7	trkTestIn.txt	Mixed mode exceptions (aka Tricky Test)

3.1 Test Script

The test script `hw2Testing.sh` uses the seven input files to produce the seven student log files and compares the student log files to a well known base file supplied in the ZIP file desribed below.

```
echo "Start main tests"
gcc lexical.c -o lex

echo " Begin lexical-1 testing (inputLex.txt)"
./lex inputLex.txt > lexical-stu1.log
diff lexical-stu1.log lexBase1.log

echo " Begin lexical-2 testing (symIn.txt) "
./lex symbIn.txt > lexical-stu2.log
diff lexical-stu2.log lexBase2.log

echo " Begin lexical-3 testing (err1In.txt)"
./lex err1In.txt > lexical-stu3.log
diff lexical-stu3.log lexBase3.log

echo " Begin lexical-4 testing (err2In.txt)"
./lex err2In.txt > lexical-stu4.log
diff lexical-stu4.log lexBase4.log

echo " Begin lexical-5 testing (err3In.txt)"
./lex err3In.txt > lexical-stu5.log
diff lexical-stu5.log lexBase5.log

echo " Begin lexical-6 testing (err4In.txt)"
./lex err4In.txt > lexical-stu6.log
diff lexical-stu6.log lexBase6.log

echo " Begin lexical-7 testing (trkTestIn.txt)"
./lex trkTestIn.txt > lexical-stu7.log
diff lexical-stu7.log lexBase7.log
```

A successful run of the testing script would produce the following output:

```
Begin lexical-1 testing (inputLex.txt)
Begin lexical-2 testing (symIn.txt)
Begin lexical-3 testing (err1In.txt)
Begin lexical-4 testing (err2In.txt)
Begin lexical-5 testing (err3In.txt)
Begin lexical-6 testing (err4In.txt)
Begin lexical-7 testing (trkTestIn.txt)
```

In the event there are discrepancies in the outputs, that will be considered an error and graded per the **Grading Rubric** shown below.

4 Usage Notes

The files supplied in the ZIP file are:

- Input test files consisting of raw instructions as described in the assignment.
- The expected *output* files, all with the ****Output.txt*** in the filename.
- The **shell script** named *hw2Testing.sh* will run the program with seven input files described above. (This script is invoked at the command line as follows: **./hw2Testing.sh**.)

Download the ZIP file to your machine, then upload it to your homework 2 working directory. The **shell script** named **hw2Testing.sh** expects the source code to be in a file named ***lexical.c*** located in the **same** directory with the all of the **unzipped** contents of the ZIP file. *Yes, the may be other ways to get the contents of the ZIP file uploaded to Eustis for testing.*

5 Grading Rubric

Scoring will be based on the following rubric:

Table 4: Grading Rubric

Deduction	Description
-100	Code does not compile on *Eustis*
-100	Code does not accept the input filename from the command line
- 15	Code does not show an error message and/or does not exit safely when there is a file I/O problem
- 20	crashed on inputLex.txt, or output does not match
- 20	crashed on symbIn.txt, or output does not match
- 20	crashed on trkTestIn.txt, or output does not match
- 10	crashed on err1In.txt, or output does not match
- 10	crashed on err2In.txt, or output does not match
- 10	crashed on err3In.txt, or output does not match
- 10	crashed on err4In.txt, or output does not match

6 Resources

The symbol values can be defined as follows:

```
// 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;
```

The error codes that might be useful in the ***lexer*** problem are shown below. Please note that only the lexigraphical errors shown in the Output print specifications will be tested in this assignment.

```
Error messages for the PL/0 Compiler:

 1. Use = instead of :=.
 2. = must be followed by a number.
 3. Identifier must be followed by =.
 4. const, var, procedure must be followed by identifier.
 5. Semicolon or comma missing.
 6. Incorrect symbol after procedure declaration.
 7. Statement expected.
 8. Incorrect symbol after statement part in block.
 9. Period expected.
10. Semicolon between statements missing.
11. Undeclared identifier.
12. Assignment to constant or procedure is not allowed.
13. Assignment operator expected.
14. call must be followed by an identifier.
15. Call of a constant or variable is meaningless.
16. then expected.
17. Semicolon or end expected.
18. do expected.
19. Incorrect symbol following statement.
20. Relational operator expected.
21. Expression most not contain a procedure identifier.
22. Right parenthesis missing.
23. The preceding factor cannot begin with this symbol.
24. An expression cannot begin with this symbol.
25. This number is too large.
```