Project #1
Lexical Analyzer

Develop the C or C++ source code required to solve the following problem.

Problem

Develop a lexical analyzer in C or C++ that can identify lexemes and tokens found in a source code file provided by the user. Once the analyzer has identified the lexemes of the language and matched them to a token group, the program should print each lexeme / token pair to the screen.

The source code file provided by the user will be written in a new programming language called "DCooke" and is based upon the following grammar (in BNF):

P ::= *S*
S ::= V=E | read(V) | write(V) | do { S } while (C) | S;S
C ::= *E* < *E* | *E* > *E* | *E* == *E* | *E* != *E* | *E* <= *E* | *E* >= *E*
E ::= *T* | *E* + *T* | *E* - *T*
T ::= *F* | *T* * *F* | *T* / *F*
F ::= (*E*) | *O* | *N* | *V*
O ::= *V*++ | *V*--
V ::= a | b | … | y | z | a*V* | b*V* | … | y*V* | z*V*
N ::= 0 | 1 | … | 8 | 9 | 0*N* | 1*N* | … | 8*N* | 9*N*

Your analyzer should accept the source code file as a required command line argument and display an appropriate error message if the argument is not provided or the file does not exist. The command to run your application will look something like this:

Form: dcooke_analyzer <path_to_source_file>
Example: dcooke_analyzer test_file.dc

Lexeme formation is guided using the BNF rules / grammar above. Your application should output each lexeme and its associated token. Invalid lexemes should output **UNKNOWN** as their token group. The following token names should be used to identify each valid lexeme:

Lexeme	Token
=	ASSIGN_OP
<	LESSER_OP
>	GREATER_OP
==	EQUAL_OP
!=	NEQUAL_OP
<=	LEQUAL_OP
>=	GEQUAL_OP
;	SEMICOLON

Lexeme	Token
+	ADD_OP
-	SUB_OP
*	MULT_OP
/	DIV_OP
++	INC_OP
--	DEC_OP
(	LEFT_PAREN
)	RIGHT_PAREN

Lexeme	Token
{	LEFT_CBRACE
}	RIGHT_CBRACE
read	KEY_READ
write	KEY_WRITE
while	KEY_WHILE
do	KEY_DO
<variable name>	IDENT
<integer>	INT_LIT

Additional Solution Rules

Your solution must conform to the following rules:

1) Your solution should be able to use whitespace, tabs, end of line characters, and end of file characters as delimiters between lexemes, however your solution should ignore these characters and not report them as lexemes, nor should it **require** these characters to delimit lexemes of different types.
    a. Example: “while i<=n do”
        i. This line will generate 5 lexemes “while”, “i”, “<=”, “n”, and “do”.
        ii. This means the space between “while” and “i” separated the two lexemes but wasn’t a lexeme itself.
        iii. This also means that no space is required between the lexemes “i”, “<=”, and “n”.
2) Your solution should print out “DCooke Analyzer :: R<#>” on the first line of output. The double colon “::” is required for correct grading of your submission.
3) Your solution must be tested to ensure compatibility with the GNU C/C++ compiler version 5.4.0.
4) Lexemes that do not match to a known token should be reported as an “UNKNOWN” token. This **should not** stop execution of your program or generate an error message.
5) Code that fails to compile for **any** reason will be marked at 0% correctness. Ensure what you turn in compiles and works, it is not the responsibility of the grader to correct your code or makefile.

Hints

1) Draw inspiration by looking at the lexical analyzer code discussed and distributed in class.
    a. Utilizing this code will not be considered plagiarism.
2) Start by focusing on writing the program in your usual C/C++ development environment.
3) Once your solution is correct, then work on testing it in Linux using the appropriate version of the GNU compiler (gcc).
4) Helpful Links:
    a. Linux Video walkthrough
    b. Linux Text walkthrough
    c. Makefile tutorial
    d. Command Line Arguments in C/C++

What to turn in to BlackBoard

A zip archive (.zip) containing the following files:

- makefile
    - A makefile for compiling your C/C++ file.
    - This makefile must work in the HPCC environment to compile your source code file and output an executable named dcooke_analyzer.
- All source code files required to compile your program.
    - Failure to include all source code files will result in your program failing to compile – which will ensure you receive a 0% for correctness.

Example Execution

The example execution below was run on Quanah, one of the HPCC clusters. It shows all the commands used to compile and execute my analyzer. Bolded text is text from the Linux OS, text in red are the commands I typed and executed, and the text in blue represents the output from each step.

```
quanah:/project_1$ make clean
rm -f dcooke_analyzer

quanah:/project_1$ make
gcc -o dcooke_analyzer Eric_Rees_R123456_Project1.c

quanah:/assignment_3$ ./dcooke_analyzer test.dc
DCooke Analyzer :: R123456

f       IDENT
=       ASSIGN_OP
15      INT_LIT
;       SEMICOLON
i       IDENT
=       ASSIGN_OP
11      INT_LIT
;       SEMICOLON
read    KEY_READ
(       LEFT_PAREN
n       IDENT
)       RIGHT_PAREN
;       SEMICOLON
do      KEY_DO
{       LEFT_CBRACE
n       IDENT
=       ASSIGN_OP
i       IDENT
+       ADD_OP
f       IDENT
;       SEMICOLON
f       IDENT
--      DEC_OP
;       SEMICOLON
i       IDENT
++      INC_OP
}       RIGHT_CBRACE
while   KEY_WHILE
(       LEFT_PAREN
f       IDENT
==      EQUAL_OP
i       IDENT
)       RIGHT_PAREN
```