Project #2
Syntax Parser

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

Problem

Using your lexical analyzer developed in Project #1 and the parser code provided in class as a guide, develop a recursive descent syntax parser in C or C++ that can validate the syntax of a source code file provided by the user.

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 parser 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_parser <path_to_source_file>
Example: dcooke_parser 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

Your parser should read the given DCooke source code file then, using your lexical analyzer, identify lexemes/tokens one at a time to ensure they can be aligned with known BNF rules for the language. For this language, P::=S would be the root of the parse tree. Your parser may print out the steps it takes during execution if you wish (similar to the provided parser) but that is not required. The following is the only required output and exit codes from your parser:

1) Your solution must print out "DCooke Parser :: R<#>" as the first line of output with <#> being replaced by your R#. The double colon "::" is required for correct grading of your submission.
2) If the provided user file is free of syntax errors:
    a. Your solution must print out "Syntax Validated" as the last line of output.
    b. Your solution must return with an exit code of 0.
3) If the provided user file contains syntax errors:
    a. Your solution must print out "Error encounter: The next lexeme was <lexeme> and the next token was <token>"
        i. Where <lexeme> is the lexeme that caused the problem (examples: "x", "<>")
        ii. Where <token> is the uppercase name of the token (examples: "IDENT", "UNKNOWN")
    b. Your solution must return with an exit code of 1.
4) If the user did not provide a file as input:
    a. Your solution must display an appropriate error message.
    b. Your solution must return with an exit code of 2.
5) If the user did provide a file as input but the file does not exist:
    a. Your solution must display an appropriate error message.
    b. Your solution must return with an exit code of 3.

Additional Solution Rules

Your solution must conform to the following rules:

1) If a user input file contains multiple syntax errors, your solution is only required to find and report the first syntax error.
2) Your solution should be able to use whitespace, tabs, and end of line 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)"
        i. This line will generate 6 lexemes "while", "(", "i", "<=", "n", and ")".
        ii. This means the space between "while" and "(" 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".
3) Your solution must be tested to ensure compatibility with the GNU C/C++ compiler version 5.4.0.
4) 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.

Extra Credit Opportunity (Optional)

There are two extra credit opportunities for this project that you may attempt to do. Keep in mind that extra credit is only granted if you meet all the specifications of the opportunity, and your program runs correctly. For ECO #1, partial credit is given if your solution can find multiple errors but doesn't find them all. For ECO #2 Partial extra credit is not given for solutions that only partly meet the specifications – extra credit is all or none.

Extra credit is optional. If you opt to work on the extra credit you may select to do either #1, #2 or both.

Extra Credit Opportunity #1: Finding all syntax errors in a file (+15)

This EC opportunity requires that your solution report multiple syntax errors found in a file that contains multiple syntax errors – instead of only printing the first syntax error. Keep in mind this means making your program more fault tolerant and capable of "recovering" when invalid data has been encountered. Also note, this EC opportunity does not state you must find all syntax errors as some syntax errors may make finding others impossible. This opportunity will be graded based on whether your solution finds most of the syntax errors that exist in files crafted with multiple errors.

Extra Credit Opportunity #2: Solve using Shift-Reduce (+35)

A word of warning, this EC opportunity is valued so high because it carries a **significant risk**. Solving the problem using this method will require you to forgo using any of the code shown in class and instead devise a method of solving the problem using Shift-Reduce. You will also be responsible for either building the parser tables by hand or learning (on your own time) how to make use of the programs that help you generate one. The parser code given in class is a Recursive Descent parser and the two methods are mutually exclusive – meaning they share almost no code. This mean once you select to solve the problem using Shift-Reduce or Recursive Descent then you would have start completely over to switch to the other.

Do not select this if you are not willing to spend upwards of 2x the time necessary to solve the problem.

Hints

1) Draw inspiration by looking at the parser code discussed and distributed in class.
2) Depending on the type of solver you choose, the language's BNF rules **may** need to be updated.
3) Start by focusing on writing the program in your usual C/C++ development environment.
4) Once your solution is correct, then work on testing it in Linux using the appropriate version of the GNU compiler (gcc).
5) Linux/Makefile tutorials:
    a. Linux Video walkthrough: http://www.depts.ttu.edu/hpcc/about/training.php#intro_linux
    b. Linux Text walkthrough: http://www.ee.surrey.ac.uk/Teaching/Unix/
    c. Makefile tutorial: https://www.tutorialspoint.com/makefile/index.htm
6) Keep in mind that this is to be solved on your own without working in groups. Also be wary of using REPL or other online compilers to ensure they meet the Academic Integrity rules discussed at the beginning of the course.

What to turn in to BlackBoard

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

- makefile
  - A makefile for compiling your C/C++ file(s).
  - This makefile must work in the HPCC environment to compile your source code file(s) and output an executable named dcooke_parser.
- 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 the parser for DCooke. 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_2$ make clean
rm -f dcooke_parser

quanah:/project_2$ make
gcc -o dcooke_parser Eric_Rees_R123456_Project_2.c

quanah:/project 2$ ./dcooke_parser example_1.dc
DCooke Parser :: R123456

Syntax Validated

quanah:/project_2$ ./dcooke_parser example_2.dc
DCooke Parser :: R123456

Error encounter:  The next lexeme was ++ and the next token was INC_OP
```