CMPILER Final Exam: Regular Expression Recognizer Instructor: Neil Patrick Del Gallego

[100 pts] General Instructions: In the study of compiler design, you have learned the phases of compilers, namely lexical, syntax, semantics, and runtime execution. For this final exam, you are going to apply what you have learned in designing your interpreter, by creating a recognizer for regular expressions, using a top-down parser generator. To create a regular expression accepter, you are going to design your grammar rules such that it is free from ambiguity. Your grammar must also be right-recursive, and left-factored.

Restrictions
In this hands-on activity, you are not allowed to use any scanner or parsing libraries. This includes ANTLR, Java/C++/C# RegEx utilities, and the like. Only your chosen programming language, native string operations such as concat, format, split, and similar, and a UI library (optional) is allowed. You must write your own scanner from scratch. Using any restricted library will automatically earn a 0 for this exam score.

Lexical Analyzer
Apply concepts you have learned from class, such as designing your own maximal munch algorithm for recognizing tokens.

Syntax Analyzer
Implement **LL (1) parsing** for this exam. Your grammar must not have left-recursion and must be left-factored.

Semantic Analyzer
You do not need a semantic analyzer when recognizing regular expressions. You are simply checking its syntax.

Regular Expression Notation
Your recognizer should notation accepted for regular expressions:

- Terminals are represented as small symbols or letters. In this exam, support only digits and characters.
- E – epsilon or the empty string. Represent the epsilon symbol as capital letter, 'E'.
- A U B – union of two regular expressions. In this exam, you may use the capital letter, 'U', to represent the union.
- AB – concatenation of two regular expressions.
- A* - 0 or more characters in A
- A+ - 1 or more characters in A.
- A? – optional
- () - parentheses

Input
The input will be the following:

- A text file containing **n** regular expressions, separated line-by-line.
- Regular expression. At least terminal symbols, 'U', 'E', '*', '+', and '?' should be recognized.
- The terminal symbols are, [a – z], [0 – 9]. Small characters and digits only.

Output
The recognizer should create a text file containing **n** outputs:

- ACCEPT, if the input string from the user, is a valid regular expression.
- REJECT, if it is an invalid regular expression.
- The input is also printed for cross-checking.

Sample Test Case
A regular expression accepter should print ACCEPT, or REJECT, for every input string in the text file. The following test cases you may use are given in the table below

Input	Output
ab	ab - **ACCEPT**
abbbbcdef	Abbbbcdef - **ACCEPT**
a U b*	a U b* - **ACCEPT**
U	U - **REJECT**
E	E - **ACCEPT**
(a U b) U 1	(a U b) U 1 - **ACCEPT**
(a) U (b)	(a) U (b) - **ACCEPT**
(aa U (b)	(aa U (b) - **REJECT**
(ab*c* U b??)	(ab*c* U b??) - **REJECT**
*(ab)	*(ab) - **REJECT**
a++	a++ - **REJECT**
a+ U b* U c? U E	a+ U b* U c? U E - **ACCEPT**
1234E	1234E - **REJECT**

1234? 1a* U 2*b U 3+c+ U 4+d?	1234? - **ACCEPT** 1a* U 2*b U 3+c+ U 4+d? - **ACCEPT**

Grading Scheme
The **actual** test case will be different from the sample provided. It will consist of 25 test cases, each worth 4 points each. 4 points are given if the actual result yields the expected result (ACCEPT or REJECT).

Deductions will be given if the following requirements were not met:

Lexical Analysis	**Syntax Analysis**
Does not use NFA/DFA/finite state accepter for recognizing tokens. Brute-force method.	Grammar rules are not clearly implemented using a data structure. Grammar rules are hardcoded.
No evidence of using tokens in code.	Does not support error recovery. Program halts if an invalid RE is present in the text file.
	Grammar rules are ambiguous. A different derivation can be produced.
	Not using LL(1) parsing.

Submission Details
You may prepare some of the requirements in advance.

The **actual** test case will be given during the final exam period. An online meeting will be held for proctoring. Within the period of the exam, you are to run the given test case. After you are done running the program using the test case provided, submit a GDrive URL containing the following:

- SOURCE – Contains your source code. Add a README.txt that has your name and instructions how to run your program. Also indicate the entry class file, where the main function is located. Alternative can be a Github link.
- PPT – A PowerPoint showing the following set of slides.
    - Title page – Your name + section + a video demo in MP4 format. The video contains a recording of your running program, showing the **actual** test case file as input, and showing the output listing down the results in detail. Should be in MP4 file. Recording must be seamless and not cut. Please ensure that your video is embedded in your PPT.
    - Lexical analysis – Discussion of your lexical analyzer implementation. Discuss token implementation. Provide code snippets.
    - Syntax analysis – Discussion of your syntax analyzer implementation. Show your grammar rules. Provide code snippets.
- FORM AND OUTPUT –
    - Text file produced by your program which should have a similar format as discussed in the specs.
    - Academic honesty agreement form declaration. See below.

APPENDIX A: Academic Honesty Form

==========Copy and paste the following section below and sign the form. Save in PDF file==========

ACADEMIC HONESTY AGREEMENT

I am answering this exam myself and to the best of my ability, without any assistance from other persons, materials, or resources that I am not allowed to access during the exam period. I declare that the software is fully written by me and without any assistance from my peers. I am fully aware and hereby agree to the clause that violation of this agreement is considered cheating and will result in a 0.0 for this course.

_____

Signature over Printed Name

==========END==========