Regular Expressions

In this assignment, you will implement a simple regular expression parser (with a very simplified input format). The goals of this assignment are to:

- Gain an appreciation for how simply our theoretical understanding of regular exprressions can be applied to implement a real regular expression recognizer.
- To see the connection between our proof that regular expressions are equivalent to DFAs by implementing some code that converts between them.
- To see the connection between our proof that regular expressions are closed under common operations by implementing some code that applies those ideas.
- To appreciate the (relative) lack of memory that is necessary to implement a regular expression parser.

Regular expression matchers are a commonly used tool in computer science, and they directly relate to the concept of a DFA/NFA from class. For this assignment we will give you a simple regular expression and a series of strings that may or may not match that expression. For each string, you will output *YES* if the string matches the expression, and *NO* if it does not.

There are many different ways to implement such features, but I would like you to use the implementation described below so that your work best reflects our discussion from class. This is despite the fact that there are more memory efficient implementations. The rest of this document will describe the input you can expect, the output you should provide, along with some details on how you should implement this.

Input

All input will be provided to standard input (cin, input(), System.in, etc.). The first line of input is the regular expression that needs to be parsed. It has a very simplified format. Each character in the string will be one of the following:

- **d**: The lower-case character d matches to any single numeric digit 0-9.
- **a**: The lower-case character a matches to any lower-case alphabetical letter a-z.
- *****: The star symbol means the previous regular expression can be repeated 0 or more times. For example, the expression (dda)* would match with the empty string, 32z, 34z76b, etc.
- **U**: The capital letter U denotes union. The string matches the expression if it matches the left side of this operator OR it matches the right side of this operator. For example, aaUdd would match for string ba or 97, but not for a7.

We will adhere to a couple of rules when formatting the input in order to make the implementation a little bit easier for you. The input is gauranteed to follow the following rules:

- Parentheses will only appear when grouping an expression for use with the * operator. Parentheses will never appear in any other context. So, you might see a(dda)* but you will never see (ad)d*.
- In addition, no other operator (* or U) will be inside those parentheses if present.
- The union operator U will always apply to the entire expression to the left and right of the operator (up to but not including the nearest U to the left or right). For example, aaddU(da)*a is the entire expressoin aadd unioned with the entire expression (da)*a. You will never see something like aa(ddUdaa)* because the parentheses mean that the union does not apply to everything to its left and this violates one of the rules above.
- There will never be nested parentheses. You might see something like (dd)*(aa)* but you will never see (dd(aa)*)*. No need to worry about using recursion to handle nested parentheses!
-

The next line will contain a single number n, which is the number of strings that will be given to match with. The next n lines after that will each give a single string that is a potential match to the regular expression.

Output

For each of the n example strings. Output *YES* on a single line if the string matches the regular expression, and *NO* if it does not.

Getting Started

Even with such a simple format, this is not a trivial implementation. I would like you to proceed using the following approach. As stated earlier, there are several ways to implement this, but I would like everybody to use the approach below.

- **Step 1: Implement an NFA object**: The first thing you should do is create an object (class) that represents an NFA (yes, simulating non-determinism will make this much easier). This object will contain fields for each part of an NFA (list of states in the machine, the start state, list of final states, list of current states the machine is in at any one time, list of transitions). You should create methods that allow you to add states, add transitions between states based on characters (numeric digits or alphabetic characters), change states to be final or start or back to normal, etc..
- **Step 2: Create methods that apply the * and U operators to a given object**: Create a method that given a current NFA object (which represents some regular expression), applies the * operator to it. How might you do this? Think about the proof we did in class? How did we construct this new NFA in our proof? Then, do this for the union operator as well. Given two NFA object, return a new one that represents the union of the two machines. How might you go about this?
- **Step 3: Parse the input into tokens**: Given a regular expression string (e.g., *aa(da)*Udd*), split the expression into tokens. Each token will be string of character (a, d, ad, aad, etc.), or an operator (* or U). For example, *aa(da)*Udd* would be divided into a, a, da, *, U, dd.

- **Step 3: For each token, instantiate an NFA object**: Instantiate an NFA object for each token that does not contain operators. Then, for each operator take the relevant objects and call the methods from step 2 to combine the machines into one (applying the operator to them). Once this process is done, you should have a single NFA object that represents the entire expression.
- **Step 4: Simulate the NFA on each string**: Have you NFA start with only the start state in the list of current states. Read in the characters from the input one character at a time. For each character, loop through each state the machine is in and transition each to a new state if possible. If you end up in any final state, print *YES*, otherwise print *NO*.

Submission

You should submit your code in Java, C++, or Python. You should configure your makefile properly to run your code. See the course website for details on how to configure submissions for Gradescope

Sample Input

```
aadaUaaadaUaaadaa
5
mrf8t
mk5sc
lab2d
bea3ch
lbh1

(a)*dd(a)*
4
23
a56
bas98edd
as98f7f
```

Sample Output

```
YES
NO
YES
YES
NO

YES
YES
YES
NO
```

Optional Challenge

Want an optional challenge? Try implementing this such that parentheses can appear to group the expressions in any way and Union and start do not necessarily have associated parentheses (they apply to only one character if no parentheses are present).