Regular Expressions

REMINDER: This is an INDIVIDUAL assignment. You may not share / look at anybody else's code. Code similarity will be checked and the honor policy will be invoked if necessary.

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 expressions can be applied to implement a regular expression recognizer.
- To see the connection between our proof that regular expressions are equivalent to NFAs 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.

Regular expression matchers are a commonly used tool in computer science, and they directly relate to the concept of a DFA/NFA from class. You have probably used a website before that forced you to type in an email address in a proper format, and complained if your entry was "not a valid email address".

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. In addition, we give you a good amount of code to start with that will make the assignment much less cumbersome for you. 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 (System.in). 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 (and thus the * operator will always appear after every right paren). Parentheses will never appear in any other context. So, you might see a(dda)* but you will never see (ad)d*.
- Notice that the star operator will always appear after a right paren, but it can also appear after a single character *a* or *d*, such as *aad** or *aa*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 expression 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. In other words, the union operator will never be inside of parentheses.
- There will never be nested parentheses. You might see something like (dd)*(aa)* but you will never see (dd(aa)*)*.

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 am providing starter code that will help get your started and you will be asked to implement a specific few of the empty methods provided.

- **Step 1: Understand the provided NFA object**: The first thing you should do is read the NFA class that is provided annd understand its methods (yes, simulating non-determinism will make this much easier). This object contains fields for each part of an NFA (list of states in the machine, the start state, list of final states, list of transitions). You should understand the provided 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: Implement the empty NFA methods that apply the *, U, and concatenation operators to a given machine**: In the NFA class, you will see three methods that given a current NFA object (which represents some regular expression), applies one of the three respective operators to that machine. Your next task is to implement these methods.
- **Step 3: Implement the build NFA method**: In Main.java, there is a method that takes the regular expression and turns it into an NFA. We have provided comments that summarize how we approached this method if you would like to use that as a guide.
- **Step 4: Implement the acceptsString method in NFA.java**: This method should take a string and simulate the NFA moving from state(s) to state(s) given each input character. If the machine ends in a final state, you should return true (false otherwise).

Running the Code

The sample code provided contains a simple *Makefile*. You can compile the code by simply opening a terminal, navigating to the project directory, and type *make*. To run the code, simply type *make run*.

Submission

You should submit four files (Main.java, NFA.java, QSig.java, Makefile). The second two of those four do not need to be altered to complete the assignment.

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. This includes having nested parentheses.