CSE 311: Foundations of Computing I

Implement Grep!

In this question, you will implement the `grep` program using the constructions we have learned in class. The general idea is the following:

> Given (1) a regular expression, and (2) a file, your program will find all the lines in the file that match the regular expression.
> For instance, if your regular expression were "$bb \cup (aa^*)$", and the file had the following lines:
>
> ```
> 1  aaaaa
> 2  bb
> 3  ccaacc
> 4  bbbbbbbb
> 5  abc
> 6  cc
> 7  aaaaaaa
> ```
>
> Then, your program would ultimately return lines 1, 2, 3, 4, 5 and 7. In particular, note that if *any* part of the line matches the regular expression, you should include that line.

These instructions will guide you through approaching this program in several separate phases:

(a) Write a CFG for our brand of regular expressions. Your program will use this CFG to parse the regular expression input!

(b) Read the existing parts of the NFA class to understand how it works, and then implement simulation of a string on NFAs.

(c) Construct NFAs for the regular expression that the user gives using the construction from lecture.

(d) Finally, put it all together by reading in the regex and file and outputs the right lines.

Note: You will need to type in "command-line arguments" for this assignment. To do this in JGrasp, open the Build menu and click "Run Arguments". Then, type the necessary arguments into the "Run Arguments" box at the top of the screen.

There are *many* files in the distributed code which you can *completely* ignore. In fact, it's easier to list the files which you will (eventually) need to modify:

- `regex-grammar.cfg`
- `NFA.java`
- `Grep.java`

There are two files which we provide as examples of a similar problem (e.g., instead of parsing regexps and evaluating them, we provide code that parses arithmetic expressions and evaluates them). These files should be *extremely helpful* as an example of what to do:

- `arithmetic-grammar.cfg`
- `ArithmeticExpressionEvaluator.java`

Task 1: Writing A CFG For Regular Expressions

Your first task is to write a CFG for the following grammar for regular expressions:

- `'` is a regular expression (it means the same as $\varnothing$).
- `-` is a regular expression (it means the same as ε).
- $a, b, \ldots, z, 0, 1, \ldots, 9$ are all regular expressions.
- If A and B are regular expressions, then `A|B` is a regular expression (this is union).
- If A and B are regular expressions, then `AB` is a regular expression.
- If A is a regular expression, then `A*` is a regular expression.
- If A is a regular expression, then `(A)` is a regular expression.

(Note that this syntax is different from class, but it is made up of one-for-one substitutes. We choose this alternate syntax, because it's easier to type!)

We have provided a sample grammar for *arithmetic expressions* in the `arithmetic-grammar.cfg` file; this example will be helpful as you construct a CFG for the regular expressions above. You should put your regular expression CFG in the file `regex-grammar.cfg`. To enter the regular expression in the way the program is set up to parse it, you should be aware of the following:

- Omit arrows; they are unnecessary.
- Capital letters will be interpreted as variables; everything else will be interpreted as terminals.
- You do not need to put bars between the terms of the CFG; a space is enough to indicate that it is a choice.
- You may not use ε in your CFG.

You should make sure your grammar is not ambiguous. The grammar given for arithmetic expressions is set up to "encode" the precedence order for arithmetic expressions. That precedence order is (from highest to lowest): digit concatenation, parentheses, multiplication/division, addition/subtraction.

For reference, precedence for regular expressions works as follows (from highest to lowest): parentheses, star, concatenation, union. You should use a similar idea as is used in the arithmetic expression grammar for your regular expression grammar.

The Java program `src/RegexCFGTester.java` will let you test your grammar file.

Once you're relatively sure you have it working, move on to Task 2.

Task 2: Finishing the NFA Class

Our Java classes that implement NFAs are the following: `NFA.java`, `FSMTransition.java`, and `FSMState.java`.

The biggest (`NFA`) is the one you will be primarily editing. `FSMTransition` represents a *single* arrow in the NFA; `FSMState` represents a *single* state in the NFA.

First, you should read the existing parts of the classes to familiarize yourself with them. Then, you should implement the `read(String S)` method of `NFA` which should return true if and only if the NFA accepts the string S.

You will find it easier to have your `read` method assume that there are no ε-transitions in your NFA. As long as you implement Task 3 as explained below, this should not be a problem.

You may edit the `NFA` class by adding/removing/changing methods as you please. The only requirements for the class are that the constructor signature and `read` signature remain the same. Also, you should not edit `FSMTransition` or `FSMState`.

The Java program `src/NFAReadTester.java` will let you test your implementation of `read`.

Task 3: Regular Expression to NFA Conversion

This is the most involved of the tasks, but, luckily, we've given you an example of an implementation of a similar method for the arithmetic expressions in the class `ArithmeticExpressionEvaluator` (which takes in an arithmetic expression and evaluates it to a number). Your task will be to take in the parsed regular expression a user entered, and convert it (using the construction from class) to an NFA (using the NFA class).

To do this, you should add new methods to the `NFA` class that deal with each regular expression operation. That is, you should add methods that create new NFAs by union, concatenation, and kleene star. To implement this, you should add the methods:

```
public static NFA union(NFA a, NFA b);
public static NFA concat(NFA a, NFA b);
public static NFA star(NFA n);
```

Each of these constructs *completely new* NFAs using the constructions from lecture.

The one major remaining "gotcha" is that the construction from lecture adds ε-transitions which we previously assumed would not exist. To deal with that, you should write a method which constructs the "ε-closure" of an NFA:

```
public static NFA epsilonClosure(NFA n, HashSet<FSMTransition> epsilonTransitions);
```

This method should take, as arguments, an NFA (with no epsilon transitions) and any epsilon transitions "to be added to the NFA". Your method should consist of two steps:

(1) Repeatedly add every possible transition that can be made by taking any number of (1) epsilon transitions and (2) edges added by only taking epsilon transitions.

(2) Replace all ε-transitions with normal transitions in such a way that the NFA is equivalent.

Another way to think about this is that you're replacing the ε-transitions in the NFA with "everywhere they go". This is the trickiest part of the program, but we've written a guide (see the `docs` folder) to help you out. Take a look!

Then, at the end of your `union/star/concat` functions, you should call your `epsilonClosure` method to *actually incorporate* the necessary epsilon transitions.

Finally, once you've written each of these methods, you will only have to edit the `makeNFAFromRegex` method in the `Grep` class. You will have a case for each of the parts of your grammar. The `ASTNode` class is just a convenient way of saying "we have part of a parsed regular expression". The methods of `ASTNode` let you figure out exactly what type of regular expression you have. Once you have cased on which type of regular expression the argument is, just call the corresponding method you just wrote.

The Java program `src/NFAConstructionTester.java` will let you test your implementation of `union`, `start`, and `concat`.

Task 4: Finishing Grep!

You've already done all of the difficult work. All that remains is to slightly modify your `Grep` class so that it takes a second argument (a file name) and runs the `NFA` you created in the previous part on each line of the file. Then, it should print *only the lines that the NFA matches any part of*.

Hint: We've implemented a "dot" method for you which will be useful here.

The Java program `src/GrepTester.java` will let you test your implementation of `union`, `start`, and `concat`.