Theory of Computation: Programming Assignment 3 (Regex Pattern Recognition)

Arjun Chandrasekhar (borrowed from Dr. John Glick)

Due Monday, 03/28/2022 at 11:59 pm (40 points)

Assignment Description

For this program you will read an alphabet and a regular expression from a file, and then determine if a series of strings belong to the language of the regular expression. You will do this by implementing the following algorithm:

1. Convert the regular expression to an equivalent NFA, using the algorithm described in class and in the textbook. (More on this further down.)

2. Convert the NFA into an equivalent DFA, using the algorithm described in class and in the textbook. Here, you can use the code you wrote for PA2, but you will not write the DFA to a file – keep it in memory.

3. For each of the strings, determine if it is in the language of the DFA by simulating the DFA on the string. Here, you can use the code you wrote of PA1, but you will get the DFA from memory, and not from a file. You will write the results to a file, which will have one line per string, indicating if the string is ("true") or is not ("false") in the language of the regular expression.

Input/Output

The program should read input from a file that is specified as the first command line argument to the program, and write output to a file whose name is specified as the second command line argument to the program. The program should not prompt the user for interactive input.

You must implement the algorithm described in class and in the textbook (and given in more detail below), and you must write all of the code yourself. Do not use any classes/libraries that do lexical analysis or parsing, or any of the other tasks that are part of the algorithms.

Here is more detail:

- **The input file name is the first command line argument.**
- The alphabet of the language appears by itself on the first line of the input file. Every character in the line (not including the terminating newline character, or any space characters) is a symbol of the alphabet. The alphabet cannot include the letter e, the letter N, the symbol *, the symbol |, or the left or right parenthesis, as these have specific meanings in the regular expressions. See below for more on the format of regular expressions. (Note: in PA1 and PA2, the space character could be in the alphabet of the language, but here for this assignment we disallow it. This is so that spaces can be put into regular expressions, to improve readability, without being treated as symbols.)
- A regular expression appears by itself on the second line of the input file. See below for more on the format of regular expressions.
- Following the regular expressions are a sequence of strings, one per line for the remainder of the input file.
- **The output file name is the second command line argument.**
- **Your program should write true or false for each string in the input file, true if the string is in the language described by the regular expression, and false if not. The output file should contain one true or false value per line.**
- **If an invalid regular expression is encountered, your program should print "Invalid expression" to the output file on a single line. Nothing else should be printed to the file.**

.

Here is detail on the format of the regular expressions that your program should handle. Recall the formal definition of a regular expression from section 1.3 in our text.

- The letter e represents ϵ (epsilon, the empty string) in a regular expression.
- The letter N reperesents $\emptyset$ (the empty set) in a regular expression.
- The character | represents $\cup$ (the union operator) in a regular expression.
- The character $*$ represents the star operator in a regular expression.
- The concatenation operator is implied in a regular expression. For example, in the regular expression $(ab^*)(c|ba)$ is short for $(a \circ b^*) \circ (c|b \circ a)$.
- Spaces can be embedded in a regular expression to help readability. So, for example, the above regular expression could be written $(a\ b^*)\ (c \mid ba)$.
- Recall that the star operator has highest precedence, then concatenation, and finally union. Parentheses are used to change the order of operation.

Regex conversion algorithm

Here is how you will convert the regular expression into an equivalent NFA:

1. Parse the regular expression into an abstract syntax tree. In an abstract syntax tree, the interior nodes represent the operators, and the leaf nodes represent symbols in the alphabet. The children of the interior nodes are the operand(s) of the operator. Here is how you will construct the syntax tree:

   (a) Create two initially empty stacks: a operand stack that will contain references to nodes in the syntax tree; and an operator stack that will contain operators (plus the left parenthesis).

   (b) Scan the regular expression character by character, ignoring space characters.

      i. If a symbol from the alphabet is encountered, then create a syntax tree node containing that symbol, and push it onto the operand stack.

      ii. If a left paren is encountered, then push it onto the operator stack.

      iii. If an operator (star, union, or implied concatenation) is encountered, then, as long as the stack is not empty, and the top of the stack is an operator is precedence is greater than or equal to the precedence of the operator just scanned, pop the operator off the stack and create a syntax tree node from it (popping its operand(s) off the operand stack), and push the new syntax tree node back onto the operand stack. When either the stack is empty, or the top of the stack is not an operator with precedence greater than or equal to the precedence of the operator just scanned, push the operator just scanned onto the operator stack.

      iv. If a right parenthesis is encountered, then pop operators off the operator stack until the left parenthesis is popped off the operator stack. For each operator popped off the stack, create a new syntax tree node from it (popping its operand(s) off the operand stack), and push it onto the operand stack.

   (c) Empty the operator stack. For each operator popped off the stack, create a new syntax tree node from it (popping its operand(s) off the operand stack), and push it onto the operand stack.

   (d) Pop the root of the syntax tree off of the operand stack.

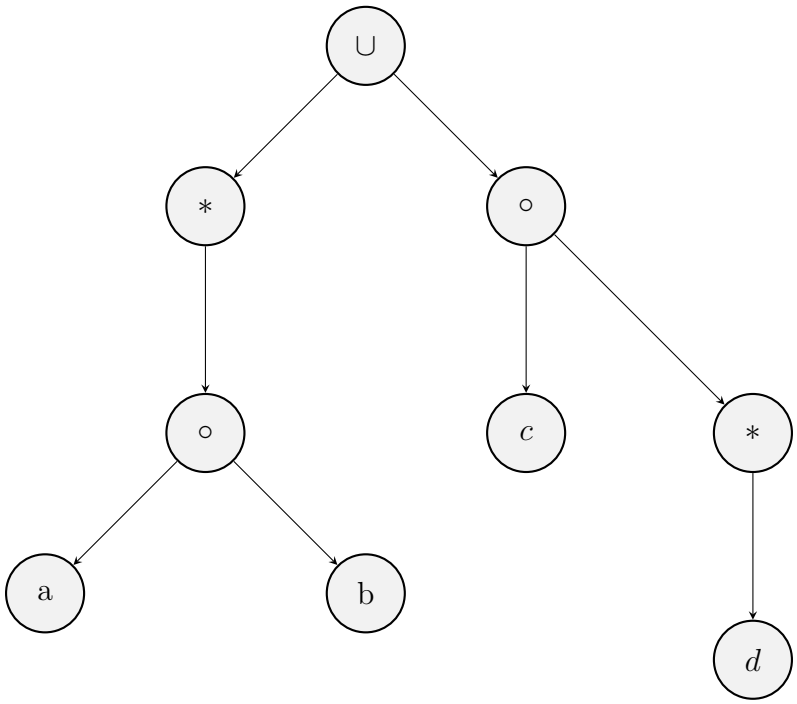
   (e) If any problems are encountered that indicate an invalid expression, then terminate parsing an print the error message to the output file as described above.

2. Create an NFA from the abstract syntax tree by doing a depth-first traversal of the syntax tree. (Remember here that each node of the syntax tree is the root of a subtree that represents a regular expression.) For each node, an NFA is created that is equivalent to the regular expression represented by the subtree rooted at the node. If the node is a leaf node, then we have a base case, and the NFA is straightforward to create. If the node is an interior node (representing an operator), then the NFA is

created from the NFA's of the child nodes using the constructions described in section 1.2 of the text (under "closure under the regular operations").

3. And now you have an NFA equivalent to the regular expression.

Here is what the syntax tree would look like for the regex $(ab)^* \cup cd^*$



You may program in Java or Python3. I strongly suggest using Python if possible; this assignment will be much more convenient to code in Python, and it will be easier for me to help you.

Completing the assignment

You may work on this assignment alone or with a partner. On Canvas you should join a group (even if you are working alone) before you submit - here are instructions for joining a group. By joining a group, you only have to make one submission for each group. If you work with a partner, you should use the pair programming software development technique to ensure that both of you are contributing to and learning from the assignment.

Testing your code

Test inputs and correct outputs are in a compressed testcases folder on the assignment webpage. Your program should work correctly on all of these tests. Input files have the form

re1In.txt, re2In.txt, and so on. The correct output files associated with the input files have names with the form re1Out.txt, re2Out.txt, and so on.

Additionally, I have uploaded a test script called pa3test.sh. This is a bash script that you can run in order to compare your outputs against the expected outputs. In order to run it, you need to create the following directory structure:

1. Create a folder called pa3
2. Within your pa3 folder, create two subfolders: test and submission
3. Place the test script inside of the test subfolder. Additionally, place all of the test case .txt files in this folder
4. Inside of the submission folder create another subfolder with your name. In my case, I made a folder called arjun
5. Put your source code inside the folder you made in the previous step
6. Finally, run the bash script from inside of the test folder. This will mimic the process of how I test your code, and it will tell you how many test cases you will pass when I run your code.

Note: The directory names that you choose are irrelevant, as long as the relative directory structure matches what I described. Additionally, you will need to be on a Unix-based OS in order to run the test script. If you are on a Windows machine, you *might* be able to run it using Cygwin or some other Terminal emulator, although I am not 100% sure.

Submitting your assignment

Submit your assignment on Canvas. Your program should be contained in one source file (with the appropriate extension for the language you are using). **Do not submit more than one file, or an entire folder; all of your code should be in one source code file.** If you are coding in java, please do not include any "package ..." statements at the top of your file; my directory names are different from yours, and I will get an error when I try to compile your code (see section 4 of this page for a more detailed explanation).

Grading rubric

Your work will be graded as follows:

- You will get 20 points for submitting a program that compiles and runs without errors, and passes at least one test.
- You will get 2 points for each test case that you pass

- If your program does not compile, does not run, does not follow the input/output directions (i.e. takes interactive input rather than a command line argument), or fails all test cases, you will receive 0 points.
- If your program follows the input/output directions correctly, and if your program passes at least one of the test cases, you will have one week to debug your program and re-submit for full points, starting from the time it is graded. If you fail to follow directions, e.g. interactive input or multiple files, or if your program does not pass any test cases, your submission will be treated as late and you will only be eligible for half of the points.