COMPX301-25A Assignment 3: Pattern Search

Due: Friday, 16 May 2025, 11:59pm

This assignment is intended to give you experience building a regular expression (regexp) FSM compiler and corresponding pattern searcher. Your implementation is expected to be written in Java, and compile and run on a Linux machine such as those in the R Block labs.

Students must work in pairs to complete this assignment (otherwise we can't complete marking in time).

By default, one partner will be assigned to complete the compiler while the other is assigned the searcher. The compiler takes in a regexep as a commandline argument and then outputs the corresponding FSM expressed in a plaint-text format (detailed below), and the searcher accepts that FSM as standard input and uses it to search a plain-text file whose name is given to the searcher as a commandline argument, outputting [once] each line of that file that contains a substring recognised by the FSM.

These are roughly equally difficult tasks. For testing, a sample FSM output expected from the compiler for a regexp will be provided, along with a text file and the results expected for that file given that regexp.

To that end, the partners hardly need to coordinate except to check the extent to which each half works; and if one partner fails to deliver a working program then we can always test the submitted half with our working solution. And it is anyway possible to create hand-crafted FSMs before the compiler is finished, so work on the searcher does not depend on having a working compiler first.

I am creating partnerships and assigning the two programs in this way: if you had a partner for assignment two then I will maintain it for this assignment, although either of those partners can ask me to reassign them to another if they want (n.b. a change is a good educational experience and encouraged, but if you prefer to stick with someone you know then that's fine). The remainder of the class will be assigned one half of the assignment and then be paired up arbitrarily before the end of the teaching recess, unless I am informed of a new voluntary partnership before the last friday of the teaching recess, Friday 25 April 2025 5pm.

A list of partnerships and task assignments will be posted in Moodle under the assignment three link, and it will evolve as partnerships are formed. By the end of the teaching recess, the partnerships will all be established.

Once a partnership is formed, you can agree to split up the work anyway you like, but by default my allocation of tasks stands (so if you can't agree to changes then just stick with what I assign). Note that it is not a waste of time to have a go at working on the other half at least a bit so that you can familiarise yourself with potential problems that may affect your part of the problem.

Partners are expected to respond promptly to each other's communications (no hiding), and apply due diligence in completing their part and cooperating with their partner. As I say, if you can't get your part working, you might see if your partner can assist, or just submit whatever you have by the due date. Let me know of any communication/cooperation difficulties so that I can help keep partnerships/individuals on track. Try to enjoy the process!!

Overview: Implement a regexp pattern searcher using the FSM, deque and compiler techniques outlined in lectures. Your solution must consist of two programs: one called **REcompile.java** and the other called **REsearch.java**, which compile and run together in the same directory/folder.

The first of these programs accepts a regexp pattern as a command-line argument (typically enclosed within quotes—see "**Note**" below), and produces as **standard output** a description of the corresponding FSM, such that each line of output describes one state and thus includes four things (i.e. four fields, comma separated):

1. the state-number,
2. the state type (i.e. either a single literal symbol to be matched, or the character pair BR as a branch-state indicator, or the character pair WC as a wildcard indicator—see below),
3. an integer indicating one possible next state, and
4. another integer indicating a possible next state.

State Zero is the start state of the entire machine and should branch to whichever state your compiler builds that is the actual start state (as used in lecture).

The second program must accept, as **standard input** , the output of the first program, then it must execute a search for matching patterns within the text of a file whose name is given as a command-line argument. Each line of the text file that contains a substring that matches the regexp is output just once, regardless of the number of times the pattern might be satisfied in that line. (Note also we are just interested in searching plain text files, like The Brown Corpus, and no pattern crosses an end-of-line.)

Regexp specification: For this assignment, a wellformed regexp is specified as follows (n.b. these are a little different than in lecture):

1. any symbol that does not have a special meaning (as given below) is a literal that matches itself
2. . is a *wildcard* symbol that matches any literal

3. adjacent regexps are concatenated to form a single regexp
4. * indicates closure (zero or more occurrences) on the preceding regexp
5. ? indicates that the preceding regexp can occur zero or one time
6. + indicates one or more repetitions of the preceding regexp
7. | is an infix alternation operator such that if *r* and *e* are regexps, then *r*|*e* is a regexp that matches one of either *r* or *e*
8. (and) may enclose a regexp to raise its precedence in the usual manner; such that if *e* is a regexp, then *(e)* is a regexp and is equivalent to *e*. *e* cannot be empty.
9. \ is an escape character that matches nothing but indicates the symbol immediately following the backslash loses any special meaning and is to be interpretted as a literal symbol
10. operator precedence is as follows (from high to low):
    - escaped characters (i.e. symbols preceded by \)
    - parentheses (i.e. the most deeply nested regexps have the highest precedence)
    - repetition/option operators (i.e. * and ? and +)
    - concatenation
    - alternation (i.e. |)

You must implement your own parser/compiler, and your own FSM (simulating two-state machines and three-state branching machines) similar to how it was shown to you in lectures, and you must implement your own dequeue to support the search. You do not have to use the exact cluster of arrays as we did in lecture for representing your FSM, but it is probably not a bad choice. And *char* primitives can sometimes be a nuisance in Java so it is okay, if you wish, to just use a string with one character in it.

Note that you should make sure you have a good grammar before you start programming, so take time to write out the phrase structure rules that convince you that your program will accept all and only the regular expressions you deem legal. Anything not explicitly covered by this specification may be handled any way you want. For example, you can decide if a** is a legal expression or not. And it is okay to preprocess the expression prior to compiling it, if such preprocessing simplifies what you are trying to do. For example, you could decide to replace any ** with just * if you want ** to be legal. I also suggest you build and test a parser first before turning it into a compiler.

Observe that REsearch can be developed prior to, or in parallel with, REcompile simply by working out the states of a valid FSM by hand and *cat* it into your searcher for testing.

Note: Command shells typically parse command-line arguments as regular expressions, and some of the special characters defined for this assignment are also special characters for the command-line interpreter of various operating system shells. This can make it hard to pass your regexp into the argument vector of your program. You can get around most problems by simply enclosing your regexp command-line argument within double-quote characters, which is probably what you should do for this assignment. To get a double-quote character into your regexp, you have to escape it by putting a backslash in front of it, and then the backslash is removed by the time the string gets into your program's command-line argument vector. There is only one other situation where Linux shells remove a backslash character from a quoted string, and that is when it precedes another backslash. For this assignment, it is the string that gets into your program that is the regexp—which may entail some extra backslashes in the argument. We are not going to be testing on difficult characters, though ... just typical plaintext stuff. (N.b. Windows command prompt shell has a different syntax for parsing regexps than does Linux, so if you develop on a windows box, make sure you make the necessary adjustments for it to run under a linux bash shell on the lab machines.)

Submission: Place only copies of your well-documented, well-formatted source code and a README text file in an otherwise empty folder/directory whose name is your student ID number (and, if you want to do a single joint submission, the student ID number of your partner separated with an underscore) then compress it and submit through Moodle as per usual. Source code should plainly state (at the top) the name of the author, the date this code was last modified, and cite any resources used that suggest someone else deserves at least some credit for this work. The first line of your README should be your name and student ID# and the second line should be the name and ID of your assigned partner. You should also include instructions for usage, and any comments about bugs or features or troubles encountered along the way (e.g. partnership problems), or anything else you want the marker/lecturer to know about. For REcompiler, I recommend including your grammar as a set of phrase structure rules in the README file and in the header of your compiler to assist in marking. See your tutor for details.

Tony C Smith, April 2025