Test Data for Non-Procedural Programming Project

Maxwell Reeser

11-6-16

This document is test data, showing that the DFA -> Regular Expression program and the Regular Expression -> DFA do in fact work correctly.

First of all, for the Regular Expression -> DFA, the proof is simply in the edge set generated and placed into the destination file. For example, if the input regular expression is (a+b+c)\* the edge set is the following:

%(a+b+c)\*

t(1,1,a).

t(2,1,a).

t(3,1,a).

t(1,2,b).

t(2,2,b).

t(3,2,b).

t(1,3,c).

t(2,3,c).

t(3,3,c).

t(0,1,a).

t(0,2,b).

t(0,3,c).

final(1,a).

final(2,b).

final(3,c).

final(0,l).

As can be seen here, this set of final transitions include state 0, the only start state, which the DFA is in before any input has been parsed. Since it is in the final state list, it means that the empty string is accepted, as it should be, since the top level group is an iteration group. From state 0, there are edges leading to every state, as there should be, and then from every state, to every other state except for 0. Then, every state is also a final state, also as they should be.

For another example, we can use a slightly more complicated regular expression, in this case (a+(bc)\*):

%a+(bc)\*

t(2,3,c).

t(3,2,b).

t(0,1,a).

t(0,2,b).

final(1,a).

final(3,c).

final(0,l).

Although the edge set here is substantially smaller than the previous one, it is a little more complex, in that not every state is a final state, and there is not a path from every state to every other state. But, it is correct, as we can see that the empty string is once again accepted, as a result of the (bc)\* option, which can be empty; also a1 and b2 are the first parsed elements, where a1 has no edges anywhere else, but which is a final state, as the regex would suggest. From b2 though, there is a path to state c3 and from there back to b2, etc. etc., with c3 the only final state, again, as the regex would suggest.

For another example, we can look at the edge set generated by the regular expression (a+(abc)\*+c)\*

%(a+(abc)\*+c)\*

t(2,3,b).

t(3,4,c).

t(4,2,a).

t(1,1,a).

t(4,1,a).

t(5,1,a).

t(1,2,a).

t(4,2,a).

t(5,2,a).

t(1,5,c).

t(4,5,c).

t(5,5,c).

t(0,1,a).

t(0,2,a).

t(0,5,c).

final(1,a).

final(4,c).

final(5,c).

final(0,l).

Obviously, this set is a little more difficult to analyze, but breaking it down will help. First of all, the empty string is once again accepted, as the top level group is an iteration group, and this is confirmed since state 0 is labeled as a final state. From there, there are three options, a single a, an iterated (abc), or a single c. State 0 leads to a1, a2, and c5, as might be expected. State a1 has edges leading to itself, a2, and c5, as the top level iteration group dictates. State a2 leads only to b3, which in turn only leads to c4. State c4 has the option to lead to a1, a2, or c5, as it essentially reenters the top level iteration group. State c5 then leads to itself, a1, and a2. This list does contain some redundancies, as a result of the nondeterministic way that c4 can lead to a2 as a result of the local iteration group, or the top level iteration group.

For another example, we can look at the differences between the above regular expression, and its slightly modified cousin, (a+(a+bc)\*+c)\*

%(a+(a+bc)\*+c)\*

t(3,4,c).

t(2,2,a).

t(4,2,a).

t(2,3,b).

t(4,3,b).

t(1,1,a).

t(2,1,a).

t(4,1,a).

t(5,1,a).

t(1,2,a).

t(2,2,a).

t(4,2,a).

t(5,2,a).

t(1,3,b).

t(2,3,b).

t(4,3,b).

t(5,3,b).

t(1,5,c).

t(2,5,c).

t(4,5,c).

t(5,5,c).

t(0,1,a).

t(0,2,a).

t(0,3,b).

t(0,5,c).

final(1,a).

final(2,a).

final(4,c).

final(5,c).

final(0,l).

%(a+(a+bc)\*+c)\*

Once again, the empty string is accepted. Starting states are a1, a2, b3, and c5, as they should be. State a1 has edges leading to itself, a2, b3, and c5. State a2 has edges leading to b3, itself, a1, and c5. State b3 has an edge leading only to c4. State c4 has edges leading to a1, a2, b3, and c5. State c5 has edges leading to a1, itself, a2, and b3.

For a final example, we can look at (a+b(a+bc)\*+c)

%(a+b(a+bc)\*+c)

t(4,5,c).

t(3,3,a).

t(5,3,a).

t(3,4,b).

t(5,4,b).

t(2,3,a).

t(2,4,b).

t(0,1,a).

t(0,2,b).

t(0,6,c).

final(1,a).

final(2,b).

final(3,a).

final(5,c).

final(6,c).

% (a1+b2^( (a3+b4^c5)\* )+c6)

Here we can see that the empty string is not accepted, as a result of the fact that there is no top level iteration group, or singular iteration group inside the primary union group. State a1 leads nowhere, but is a final state, like c6. State b2 leads either to a3 or b4. State a3 leads to itself, or b4. State b4 leads only to c5. State c5 leads to b4, or a3.

This completes the test data for the Regular Expression -> DFA program, program\_builder/1

For the DFA -> Regular Expression program, regex\_builder/0, we can examine the regular expressions generated from a few DFAs generated from some simpler regular expressions than what we have just been investigating.

For the first example, we can examine the result of the regex\_builder predicate called on the file generated by the regex (ab+(a+b)\*) which is below:

ab+ (a ((a)\*)+ (b+a ((a)\*)b) ((b+lam+a ((a)\*)b)\*)a ((a)\*)+ (b+a ((a)\*)b+ (b+a ((a)\*)b) ((b+lam+a ((a)\*)b)\*) (b+lam+a ((a)\*)b)+lam))

To begin with, color coding will help this be seen more clearly for what it is:

ab+ (a ((a)\*)+

(b+a ((a)\*)b) ((b+lam+a ((a)\*)b)\*)a ((a)\*)+

(b+a ((a)\*)b+

(b+a ((a)\*)b) ((b+lam+a ((a)\*)b)\*) (b+lam+a ((a)\*)b)+

lam)

)

Now that that’s done with, it becomes easier to see the important recurring elements which it will benefit us to reduce first.

For example (b+a ((a)\*)b) is a recurring structure, so reducing it will help us simplify the overall expression. As it happens, this regular expression expresses the same idea as:

(λ+a ((a)\*))b which has the substructure (λ+a ((a)\*)) which, as a result of the fact that it covers the empty string case, the single character case, and the as-long-as-you-want character sequence of a’s, is equivalent to ((a)\*), which means the rest of the expression is equal to ((a)\*)b. This will help us simplify the total expression.

Another recurring sequence it would benefit us to simplify would be the ((b+lam+a ((a)\*)b)\*) sequence. But this contains the structure we already analyzed, so that it can be rendered

((λ+((a)\*)b)\*)

So now we can rewrite our simplified regular expression as:

ab+ ( a ((a)\*)+

((a)\*)b ((λ+((a)\*)b)\*)a ((a)\*)+

((a)\*b + ((a)\*)b ((λ+((a)\*)b)\*) (λ+((a)\*)b)+λ)

)

This can further be transformed to:

ab+ ( (λ + ((a)\*)b (((a)\*)b)\*)a ((a)\*)+

((a)\*b + ((a)\*)b ((a)\*b)\* (λ+((a)\*)b)+λ)

)

Because ((λ+((a)\*)b)\*) is the same as (((a)\*)b)\* because the λ is superfluous in the iteration group.

This can then be transformed to:

ab+ ( ( ( ((a)\*)b )\* )a ((a)\*)+ (a)\*b [λ+ ((a)\*b)\* (λ+((a)\*)b)]+λ)

The element in the red parentheses then is actually equal to (a+b)\*. This is because the single ‘a’ character case is covered, as is the case of a single ‘b’ character, as is the empty string, as well as a string of infinite ‘a’s, or of infinite ‘b’s. Also, any string of ‘a’s, followed by a string of ‘b’s, or any string of ‘ba’s, or ‘ab’s, or anything in between, including a string of ‘b’s followed by a single, or multiple ‘a’s.

This is a fairly complicated case, but it is a good demonstration of how the algorithm is correct. Simpler demonstrations of the algorithm’s correctness also exist, such as the case of the regular expression (a+ab), which can be easily verified by running the regex through program\_builder/1 and the subsequently generated file through regex\_builder. Another simple regex which can be shown is any iteration group consisting of no union groups. Another example is a union group consisting of simple concatenation groups.