ASSIGNMENT 4 SOLUTION

Qs 1) \( R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)} R_{kk}^{(k-1)^*} R_{kj}^{(k-1)} \). It seems a rather mysterious statement, at first sight. Give the English explanation of what exactly this equation means. (5 marks)

Ans) \( R_{ij}^{(k)} \) means the number of ways to go from state i to j, while passing through no intermediate state with label greater than k. Now, when inductively building \( R_{ij}^{(k)} \), we have two choices. If we do not visit state k, in going from state i to state j, then this path can be represented by \( R_{ij}^{(k-1)} \). If on the other hand, we do visit state k, then we can break the path from i to j as follows: from state i to k (\( R_{ik}^{(k-1)} \)), then from state k to itself any number of times (\( R_{kk}^{(k-1)^*} \)), then finally from state k to state j (\( R_{kj}^{(k-1)} \)).

Qs 2) Given that we calculate all possible \( R_{ij}^{(k)} \) for an automata with n states, how many expressions will we end up with at the end(give an upper-bound)? Give some justification for your answer. Hint: For an automata with two states, we may have a total of 12 different \( R_{ij}^{(k)} \) including \( R_{00}^{(0)} \), \( R_{11}^{(1)} \), \( R_{21}^{(2)} \) etc. (5 marks)

Ans) It would simply be \( n(n)(n+1) \). When writing \( R_{ij}^{(k)} \), we have n choices for i, n choices for j and n+1 choices for k.

Qs 3) Suppose that you are given all possible \( R_{ij}^{(k)} \) for a 3 state automata that accepts a language \( L \) (assume that state 2 is the only accepting state). In terms of these expressions, give the regular expression for \( L^2 \). (2.5 marks)

Ans) It would simply be \( R_{12}^{(3)} \). \( R_{12}^{(3)} \)

Qs 4 a) Page no 121, Exercise 3.4.1 e) (5 marks)

Ans) We have to verify that \( (R+S)T = RT + ST \).
First, we replace R,S and T with alphabets to get the regular expressions:
\( (a+b)c = ac + bc \).
According to theorem 3.14, if the above law holds when we simply take R,S and T to be the alphabet a, b and c, then it will also hold if we substitute any other complex regular expression for them.
The language of L.H.S is \( \{ac, bc\} \).
The language of R.H.S is \( \{ac, bc\} \).
Since these two languages are equal, we can say this law of regular expressions is verified.

Qs 4 b) Page no 121, Exercise 3.4.2 d)
Ans) We have to check whether \((R+S)^*S = (R*S)^*\) is true or not.
We convert this law to a regular expression to get:
\((a+b)^*b = (a*b)^*\)
The language of L.H.S consists of all strings ending with a b.
Yet the language of R.H.S contains a string that does not end with a b, namely lambada, the empty string.
Hence, these two languages are not equivalent, and the law does not hold.

Qs 6) In class we showed right-linear grammars are regular by giving a method for converting a right-linear grammar to a NFA. You are required to give a construction showing that for every left-linear grammar there is an equivalent regular expression and vice versa, i.e, left-linear grammar \(\Rightarrow\) regex and regex \(\Rightarrow\) left-linear grammar.
Note: You need to give a direct conversion; your method should not be of the form left-linear grammar \(\Rightarrow\) NFA \(\Rightarrow\) regular expression.

(15 marks)

Ans 6)
**regex \(\Rightarrow\) left-linear grammar**

Read the regular expression from right to left one symbol at a time; for each operator you see introduce a new variable in your grammar.

Take an example \((a+ b.c)^*a^* + a.b\). You’ll need 7 new non-terminals for the 7 operators used here

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>(\Rightarrow Ab ) (A for the concatenation between a and b)</td>
</tr>
<tr>
<td>A</td>
<td>(\Rightarrow Ba) , S (\Rightarrow C) (B for the + between ((a+ bc)^<em>a^</em>) and ab )</td>
</tr>
<tr>
<td>B</td>
<td>(\Rightarrow \epsilon) (C for the * on a*)</td>
</tr>
<tr>
<td>C</td>
<td>(\Rightarrow Ca \mid D) (over here, we complete the recursive loop for getting a*)</td>
</tr>
<tr>
<td>D</td>
<td>(\Rightarrow E) (D for the concatenation between ((a+ b.c)^<em>) and a</em>)</td>
</tr>
<tr>
<td>E</td>
<td>(\Rightarrow Fc \mid \epsilon) (E for the * on ((a+bc)^*). Recursive loop will be closed on E later.)</td>
</tr>
<tr>
<td>F</td>
<td>(\Rightarrow b) , E (\Rightarrow G) (F for the concatenation between b and c)</td>
</tr>
<tr>
<td>G</td>
<td>(\Rightarrow a \mid E) (G (\Rightarrow E) completes the recursive loop)</td>
</tr>
</tbody>
</table>

At any one step, when reading from right to left, we need only consider the current symbol(operator) we are on. Wherever we encounter operators ‘+’ or ‘*’ or terminal symbol of a RE, we need to introduce a null production (\(\epsilon\)) for the corresponding new non-terminal . The only tricky part is keeping track of * operator, for closing the recursion. Just remember which non-terminal has been introduced specifically for keeping track of *, and whenever you reach the end of the parenthesis which are undergoing * operation, close the recursive loop.

**Left-linear grammar \(\Rightarrow\) regex**
This is a comparatively easier conversion. Wherever you see a production of from
\[ A \Rightarrow Xw \] (where \( w \) is a string of symbols), you can write it as:
\[ (\text{regex for } X).w \]

Similarly, wherever you have productions of the form \[ A \Rightarrow C|D \], you can introduce “+” operator. Again though, the tricky bit is keeping track of recursive loops and introducing * (or +) operator whenever you see them.

Example: So when you have
\[ S \Rightarrow Ab | C \] \([L = R.b + T, \text{ where } R \text{ is regex for } A, T \text{ is regex for } C]\)
\[ A \Rightarrow Ba \] \([R = S.a, \text{ where } S \text{ is regex for } B]\)
\[ B \Rightarrow \varepsilon \] \([S = \varepsilon]\)
\[ C \Rightarrow Ca | D \] \([T = U.a^*, \text{ where } U \text{ is regex for } D]. \text{This is not } a^* + U, \text{ because the recursion on } C \text{ can’t end till it goes through } D, \text{ as it has no terminal productions.}\)
\[ D \Rightarrow E \] \([U = (V), \text{ where } V \text{ is regex for } E]\)
\[ E \Rightarrow Fc | G | \varepsilon \] \([V = Wc + X + \varepsilon]\)
\[ F \Rightarrow b \] \([W = b]\)
\[ G \Rightarrow a | E \] \([X = a, \text{ and the loop closing gives us } U = (V)^*]\)

Note: If the loop closes without an \( \varepsilon \), it gives us something of form \( L^+ \), rather than \( L^* \).
So, the regex becomes (by substituting recursively)
\[ X = a, W = b, V = a + bc + \varepsilon, U = (a + bc)^*, T = (a + bc)^*, a^* \]
\[ S = \varepsilon, R = a, L = ab + T \text{ and the final Regex is } L = ab + (a + bc)^*.a^* \]

**Qs 7) Page 147, question 4.2.6 b).**

Ans) Consider the DFA for \( L \), which we know to exist since \( L \) is regular. Let us term it as \( D_1 \).
Now, consider all outgoing transitions from the final states of \( D_1 \). Let us construct an NFA \( N_2 \), identical to \( D_1 \), except that all of these outgoing transitions from the final states of \( D_1 \) have been removed. I propose that \( N_2 \) would represent \( \text{max}(L) \). To give this assertion solidity, here is a somewhat informal proof.

1) Consider a string \( w \) of size \( n \), and \( w \) is accepted by \( D_1 \).
There are two cases:

Case 1:
With some portion of the string \( w \) still not read, we also reach an accepting state. That is, some prefix of the string \( w \) is also part of the language \( L \).
Now, according to the definition of \( \text{max}(L) \), this string should not be acceptable to \( N_2 \). Does this actually hold true? Well, we have removed ALL outgoing transitions from accepting states. So, if the automata reaches an accepting state, and some
portion of the string remains unconsumed, then the automata will “hang” and the string will be rejected, as it should be.

Case 2:
We reach an accepting state for the first time when the nth and final symbol of w is read by automata. That is, no prefix of w is part of language L, as \( \delta (q_0, x) \) is not equal to an accepting state for any x (where x is a prefix of w, and \( \delta \) is the extended transition function). Hence, w belongs to max(L). Now, the transition function for N2 is same as the transition function for D1, as long as accepting states are not involved. Also, for the computation on w in this case, transition function of accepting states are not involved. Hence, if D1 accepts w then so does N2, as it should.

2) Consider a string w of size n, and w is rejected by D1
Here also, there are two cases.
Case 1) At no point in computation of w do we visit an accepting state.
Since computations for D1 and N2 are the same where accepting states not involved, rejection of w by D1 automatically implies rejection of w by N2.

Case 2) We reach an accepting state during computation of w, with some portion of w left unconsumed.
Since there are no outgoing transitions from accepting states, the automata will hang and string will be rejected as it should be.

Having considered all possible cases, we can see that the language of N2 and max(L) are the same.

**Qs 8)** Let \( L \) be a language containing only strings of even length. The language half-L is defined as:

\[
\text{Half-L} = \{ u \mid uv \in L, \text{ and } u \text{ and } v \text{ are equal in length} \}
\]

Informally, half-L is the set of strings representing “first halves” of strings in L. So, for instance, if \( L \) was \( \{00, 011111\} \) then half-L would be \( \{0, 011\} \). Prove that if \( L \) is a regular language containing only strings of even length, then half-L is regular. No vague arguments please………find a way to construct an NFA, a DFA or a regex for half-L from an arbitrary NFA/DFA/regex for L (you may choose any one of the conversions)

**Hint:** You might find it easiest to construct an NFA for half-L from the DFA for L.

**Extra credit**
Ans 8) First consider what does the problem statement mean in terms of the n-state DFA for L? To determine whether to accept a string w, it means having read the whole length of w and reached a state q_i in the DFA for L, is there any computation of the same length from q_i to a final state of the automata. The construction is as follows.

Take the DFA that represents L, D. Now over all the non-null transition arcs (assuming each label is a single symbol) place a new label of all the alphabet. For instance if alphabet is (a,b,c); each transition arc will carry label a, b, c. Try and convince yourselves that this modified DFA, D_1, will accept any x-length string w, but only if there is some string of length x that is acceptable in the language.

Now, we modify D_1 to give n-different DFAs, such that the starting state is q_1, q_2, q_3, q_4, q_5, q_6, ..., q_n (its q_1 in D_1, q_2 in D_2 and so on), and the final states are the same as in D. What language does D_i accept? It's the language of all x-length strings, on the condition that there is a computation of length x from q_i to the final state.

Similarly, we construct D’ to give n-different DFAs, such that the starting state is as in D, but the final state is q_1, q_2, q_3, q_4, ..., q_n respectively. D'_i accepts a string w simply if the DFA ends in state q_i after completing the computation on w.

For the sake of clarification, consider what the language D_3 intersection D_3' gives. D_3 accepts a string of length x, if there is some computation of length x from q_3 to the final state. Similarly, D'_3 accepts the same string if we end up in state q_3 after reading this string of length x. So, D_3 intersection D_3' gives us the condition that the string of length x leads the automata to state q_3 and there then exists a computation of length x from q_3 to the final state.

Now, look at what strings half-L should contain. The possible conditions are:

1) After the string of length x is read by the DFA for L, we end up in a state q_i, AND there should be a computation of length x from q_i to the final states.

So, the language half-L can be written as:

(D_1 intersection D_1') union (D_2 intersection D_2') ... union (D_n intersection D_n')

Since the union and intersection of a finite number of DFAs is also a DFA, hence we can say half-L is regular.