Problem 1 [30 points]: Regular Expressions and Finite Automata

Develop a regular expression (RE) that detects the longest string over the alphabet \{a-z\} with the following properties:

1. The string begins with an ‘a’ character and ends with a ‘z’ character;
2. After the first ‘a’, the string can include a alternation of two subsequences, namely a sequence that begins with a ‘b’ character followed by zero or more ‘s’ characters ending with an ‘e’ character and a subsequence that begins with a ‘c’ character but does not have any ‘s’ character until a terminating ‘t’ character.
3. If at any point in the subsequence there is a ‘x’ character, that specific sequence is considered terminated, i.e., the ‘x’ acts as the ‘e’ character in the first type of sequence and as the ‘t’ character in the second type of sequence.
4. The two subsequences cannot be nested but can be repeated in any alternating order.

As an example the string “absssecaaabbaafghtbsez” is to be accepted as well as the string “abssxcefztz”

Questions:

a) [05 points] Develop a regular expression that captures the structure of the acceptable strings described above. Use short-hands to capture subsets of characters in the alphabet so that your description and the corresponding FA are kept short.

b) [10 points] Using the regular expression define in section a) devise the corresponding Non-Deterministic Finite Automaton (NFA) using the Thompson construction described in class.

c) [10 points] Convert the NFA in section b) to a DFA using the subset construction. Show the mapping between the states in the NFA and the resulting DFA.

d) [05 points] Minimize the DFA derived in section c) (or show it is already minimal) using the iterative refinement algorithm described in class.

Solution:

a) Among the many possible solutions for a regular expression the one below seems to be the most intuitive. Here we use the short-hand “notSTX” for all characters in the alphabet except ‘s’ and ‘t’ i.e., notSTX = \{a-r, u, v, y, z\}.

\[ \text{RE} = a \cdot \left( b \cdot \left( s^* \cdot \left( x|e \right) \right) \right) \cdot \left( c \cdot \left( \text{notSTX}^* \cdot \left( x|t \right) \right) \right)^* \cdot z \]

b) The NFA below reflects the structure of the RE presented above where we have also “compressed” some \(\varepsilon\)-transitions for brevity, in particular the ones that reflect Kleene closure and alternation.
c) The subset construction of this NFA is as shown below where again we have merged transitions on character with analogous behavior.

d) The minimization of this DFA using the iterative refinement algorithm is summarized below. We begin by a simple partition between the accepting states and the non-accepting states. This immediately allows us to “isolate” the accepting state 17. The second partition used the character ‘z’ as the discriminative token showing that states “1,2,3,4,11,13,15,16” and “1,2,3,4,12,14,15,16” may be equivalent. The last partition diagrams reflect the fact that all states are disjoint.
Problem 2 [20 points]: Translation from DFA to Regular Expressions

Given the DFA below over the alphabet \{0,1\} determine the following:

a. [15 points] Use the dynamic-programming Kleene algorithm to derive the regular expression that denotes the language accepted by it. Make sure your labelling of the DFA states is correct and that the DFA is completely specified, i.e., each state has transitions on all the alphabet characters.

b. [05 points] Describe succintely what are the words accepted by this DFA?

Solution:

a. [15 points] Use the dynamic-programming Kleene algorithm to derive the regular expression that denotes the language accepted by it. Make sure your labelling of the DFA states is correct and that the DFA is completely specified, i.e., each state has transitions on all the alphabet characters.

Expressions for \(k = 0\)

\[
R_0^{11} = (\varepsilon) \\
R_0^{12} = (0) \\
R_0^{13} = (1) \\
R_0^{21} = \emptyset \\
R_0^{22} = (0 \mid \varepsilon) \\
R_0^{23} = \emptyset \\
R_0^{31} = \emptyset \\
R_0^{32} = (1) \\
R_0^{33} = (0 \mid \varepsilon)
\]

Expressions for \(k = 1\)

\[
R_1^{11} = R_0^{11} (R_0^{11})^* R_0^{11} \mid R_0^{11} = (\varepsilon)(\varepsilon)^* (\varepsilon) \mid (\varepsilon) = (\varepsilon) \\
R_1^{12} = R_0^{11} (R_0^{11})^* R_0^{12} \mid R_0^{12} = (\varepsilon)(\varepsilon)^* (0) \mid (0) = (0) \\
R_1^{13} = R_0^{11} (R_0^{11})^* R_0^{13} \mid R_0^{13} = (\varepsilon)(\varepsilon)^* (1) \mid (1) = (1) \\
R_1^{21} = R_0^{21} (R_0^{11})^* R_0^{21} \mid R_0^{21} = \emptyset \\
R_1^{22} = R_0^{21} (R_0^{11})^* R_0^{22} \mid R_0^{22} = R_0^{22} = (0 \mid \varepsilon) \\
R_1^{23} = R_0^{21} (R_0^{11})^* R_0^{23} \mid R_0^{23} = \emptyset \\
R_1^{31} = R_0^{31} (R_0^{11})^* R_0^{31} \mid R_0^{31} = R_0^{31} = \emptyset
\]
R_{12}^2 = R_{12}^1 (R_{12}^2) | R_{12}^1 = (0) . (0 | \varepsilon)^* . \emptyset | (\varepsilon) = (\varepsilon)
R_{12}^3 = R_{12}^2 (R_{12}^3) | R_{12}^2 = (0) . (0 | \varepsilon)^* . (0 | \varepsilon) | (0) = (0^*)
R_{13}^2 = R_{12}^1 (R_{12}^2) | R_{13}^1 = (0) . (0 | \varepsilon)^* . \emptyset | (1) = (1)
R_{21}^2 = R_{22}^1 (R_{22}^2) | R_{21}^1 = (0 | \varepsilon) (0 | \varepsilon)^* \emptyset | \emptyset = \emptyset
R_{22}^2 = R_{22}^1 (R_{22}^2) | R_{22}^1 = (0 | \varepsilon) (0 | \varepsilon)^* (0 | \varepsilon) | (0 | \varepsilon) = (0^*)
R_{23}^2 = R_{22}^1 (R_{22}^3) | R_{23}^1 = (0 | \varepsilon) (0 | \varepsilon)^* \emptyset | \emptyset = \emptyset
R_{31}^2 = R_{32}^1 (R_{32}^2) | R_{31}^1 = (1) . (0 | \varepsilon)^* . \emptyset | \emptyset = \emptyset
R_{32}^3 = R_{32}^2 (R_{32}^3) | R_{32}^2 = (1) . (0 | \varepsilon)^* . (0 | \varepsilon) | (1) = (1) . (0^*)
R_{33}^3 = R_{32}^2 (R_{32}^3) | R_{33}^2 = (1) . (0 | \varepsilon)^* . \emptyset | (0 | \varepsilon) = (0 | \varepsilon)

Expressions for \( k = 3 \)

R_{11}^3 = R_{13}^2 (R_{33}^3) | R_{11}^2 = (1) . (0)^* . \emptyset | (\varepsilon) = \emptyset
R_{12}^3 = R_{13}^2 (R_{33}^3) | R_{12}^2 = (1) . (0)^* . (1) . (0)^* | (0^*)
R_{13}^3 = R_{13}^2 (R_{33}^3) | R_{13}^2 = (1) . (0)^* . (0 | \varepsilon) | (1) = (1) . (0^*)
R_{21}^3 = R_{23}^2 (R_{33}^3) | R_{21}^2 = \emptyset . (0)^* . \emptyset | \emptyset = \emptyset
R_{22}^3 = R_{23}^2 (R_{33}^3) | R_{22}^2 = \emptyset . (0)^* . (1) . (0)^* | (0^*) = (0^*)
R_{23}^3 = R_{23}^2 (R_{33}^3) | R_{23}^2 = \emptyset . (0)^* . (0 | \varepsilon) | \emptyset = \emptyset
R_{31}^3 = R_{33}^2 (R_{33}^3) | R_{31}^2 = (0 | \varepsilon) . (0)^* . \emptyset | \emptyset = \emptyset
R_{32}^3 = R_{33}^2 (R_{33}^3) | R_{32}^2 = (0 | \varepsilon) . (0)^* . (1) . (0)^* | (1) . (0^*) = (0^*) . (1) . (0^*)
R_{33}^3 = R_{33}^2 (R_{33}^3) | R_{33}^2 = (0 | \varepsilon) . (0)^* . (0 | \varepsilon) | (0 | \varepsilon) = (0)^*

The language recognized by this DFA can be expressed by the regular expression \( R_{13}^3 \).

\[ L = R_{13}^3 = (1) . (0 | \varepsilon)^* . (0 | \varepsilon) = (1) . (0 | \varepsilon)^* = 1.0^* \]

b. [05 points] Describe succinctly what are the words accepted by this DFA?

This automaton recognizes all binaries strings that denote integers that are non-negative powers of two and as such can be described by the compact regular expression: \( 10^* \).
Problem 3 [50 points]: Predictive Top-Down Parsing

Consider the CFG grammar \( G = (N=\{S, A, B\}, T=\{a \, b\}, P, S) \) where the set of productions \( P \) is given below:

\[
\begin{align*}
S & \rightarrow A \, a \, A \, b \mid B \, b \, B \, a \\
A & \rightarrow a \mid \varepsilon \\
B & \rightarrow b \mid \varepsilon 
\end{align*}
\]

Questions:

a) [05 points] Can this grammar be used as presented for parsing using a predictive (backtrack-free) algorithm? Why or why not?

b) [10 points] Devise an alternative (but equivalent) grammar for the same language that has the LL(1) property.

c) [15 points] Compute the FIRST and FOLLOW sets for each production’s RHS and the non-terminal symbol respectively. Use these to show that the grammar has in fact the LL(1) property.

d) [10 points] Derive the LL(1) parsing table as described in the lectures and show that in fact the grammar is parseable using the LL(1) parsing algorithm.

e) [10 points] Show the sequence of parsing steps and the corresponding parse tree for this algorithm and the two inputs \( w_1 = “aab” \) and \( w_2 = “ba” \).

Solution:

a) [05 points] Can this grammar be used as presented for parsing using a predictive (backtrack-free) algorithm? Why or why not?

While at first sight this grammar does appear not to suffer from the issues of left-recursion and common prefixes in fact it is not LL(1). This is because the productions for either the non-terminals \( A \) and \( B \) allow for the derivation of the empty string and the characters ‘a’ and ‘b’ below to the FOLLOW sets of the respective non-terminals.

b) [10 points] Devise an alternative (but equivalent) grammar for the same language that has the LL(1) property.

A way to avoid this issues, and hence derive an equivalent LL(1) grammar will be to eliminate the \( \varepsilon \)-transitions as shown in the revised grammar below. Notice that this leads to an increase in the number of productions in the grammar as all combinations of transitions for the two occurring \( A \) and \( B \) non-terminals in the productions of \( S \) need to be explored.

\[
\begin{align*}
S & \rightarrow A \, a \, A \, b \mid aAb \mid ab \mid Aab \\
S & \rightarrow B \, b \, B \, a \mid bBa \mid ba \mid Bba \\
A & \rightarrow a \\
B & \rightarrow b 
\end{align*}
\]

Now we have to deal with the issue if left-factorization in both sets of productions for \( S \).

\[
\begin{align*}
S & \rightarrow A_3 \, b \\
S & \rightarrow B_3 \, a \\
A_3 & \rightarrow a \, A_2 \\
A_2 & \rightarrow a \, A_1 \mid \varepsilon \\
A_1 & \rightarrow a \mid \varepsilon \\
B_3 & \rightarrow b \, B_2 \\
B_2 & \rightarrow b \, B_1 \mid \varepsilon \\
B_1 & \rightarrow b \mid \varepsilon 
\end{align*}
\]
Notice now that the FOLLOW of any non-terminal that derive the empty string does not conflict with its FIRST set and so the grammar is LL(1).

c) [15 points] Compute the FIRST and FOLLOW sets for each production’s RHS and the non-terminal symbol respectively. Use these to show that the grammar has in fact the LL(1) property.

\[
\begin{align*}
\text{FOLLOW}(S) &= \{\} \\
\text{FOLLOW}(A_3) &= \{b\} \\
\text{FOLLOW}(A_2) &= \{b\} \\
\text{FOLLOW}(A_1) &= \{b\} \\
\text{FOLLOW}(B_3) &= \{a\} \\
\text{FOLLOW}(B_2) &= \{a\} \\
\text{FOLLOW}(B_1) &= \{a\}
\end{align*}
\]

FIRST(A_3 b) = \{a\}, FIRST(B_3 a) = \{b\} \Rightarrow FIRST^+(A_3 b) \cap FIRST^+(B_3 a) = \emptyset \text{ as FOLLOW}(S) = \{\}$

FIRST(a A_2) = \{a\}, FIRST(\varepsilon) = \{\varepsilon\} \Rightarrow \text{because } A_2 \text{ derives the empty string, FIRST}'(A_2 \rightarrow a A_1) = \{a,b\}

Similarly, for $B$’s productions we get:

FIRST(b B_2) = \{b\}
FIRST(b B_1) = \{b\}, FIRST(\varepsilon) = \{\varepsilon\} \Rightarrow \text{because } B_2 \text{ derives the empty string, FIRST}'(B_2 \rightarrow b B_1) = \{b,a\}
FIRST(b) = \{b\}, FIRST(\varepsilon) = \{\varepsilon\} \Rightarrow \text{because } B_1 \text{ derives the empty string, FIRST}'(B_1 \rightarrow b) = \{b,a\}

d) [10 points] Derive the LL(1) parsing table as described in the lectures and show that in fact the grammar is parseable using the LL(1) parsing algorithm.

\[
\begin{array}{c|c|c|c}
\text{Stack} & \text{Input} & \text{Prod.} & \text{Stack} & \text{Input} & \text{Prod.} \\
\hline
S & S \rightarrow a b & S \rightarrow b a & S & ba S & S \rightarrow B_3 a \\
S & A_3 & A_3 \rightarrow a A_2 & S & a B_3 & S \rightarrow B_3 a \\
S & b A_3 & b A_3 & S & a B_3 & S \rightarrow a B_3 \\
S & b & b & b & a B_3 & S \rightarrow a B_3 \\
S & \varepsilon & a & a & a B_3 & S \rightarrow a B_3 \\
S & b & b & b & S & S \\
\end{array}
\]

e) [10 points] Show the sequence of parsing steps and the corresponding parse tree for this algorithm and the two inputs $w_1 = \text{“aab”}$ and $w_2 = \text{“ba”}$.