Problem 1 [40 points]: Consider the alphabet $\Sigma = \{0,1\}$.

a) [05 points] Consider the Non-Deterministic Finite Automaton (NFA) depicted below. Why is this automaton non-deterministic? Explain the various source on indeterminacy.

b) [15 points] Do the sentences $w_1 = "1101"$ and $w_2 = "10"$ belong to the language generated by this FA? Justify.

c) [15 points] Convert the NFA in part a) to a DFA using the subset construction. Show the mapping between the states in the NFA and the resulting DFA.

d) [05 points] By inspection can you derive a regular expression (RE) this NFA detects?

Solution:

a) This FA is a NFA as, in addition to $\epsilon$-transitions some states have more than one transition on the same alphabet symbol. For example, states 7 and 8, have two outgoing edges labelled '1'.

b) Regarding the word "1101" there is a path from state 0 to the accepting state 10, namely: 0,1,3,5,6,8,10. Regarding the word "10" not advance beyond state 1 and so there is no path to state 10, the only accepting state in this NFA.

c) Using the subset construction we arrive at the following subsets and transitions listed below:

\[
\begin{align*}
S0 &= \epsilon\text{-closure (0)} = \{0,1\} - \text{this is not a final state.} \\
DFAedge(S0,0) &= \epsilon\text{-closure (goto(S0, 0))} = \{0,1,2\} = S1 \\
DFAedge(S0,1) &= \epsilon\text{-closure (goto(S0, 1))} = \{3\} = S2 \\
S3 &= DFAedge(S1,0) = \epsilon\text{-closure (goto(S1, 0))} = \{0,1,2,4,6\} = S3 \\
DFAedge(S1,1) &= \epsilon\text{-closure (goto(S1, 1))} = \{3\} \\
DFAedge(S2,0) &= \epsilon\text{-closure (goto(S2, 0))} = \{\} \text{ Error state} \\
S4 &= DFAedge(S2,1) = \epsilon\text{-closure (goto(S2, 1))} = \{5,6\} = S4 \\
S5 &= DFAedge(S3,0) = \epsilon\text{-closure (goto(S3, 0))} = \{0,1,2,4,6,7,8\} = S5 \\
DFAedge(S3,1) &= \epsilon\text{-closure (goto(S3, 1))} = \{3\} = S2 \\
S6 &= DFAedge(S4,0) = \epsilon\text{-closure (goto(S4, 0))} = \{7,8\} = S6 \\
DFAedge(S4,1) &= \epsilon\text{-closure (goto(S4, 1))} = \{\} \text{ Error state} \\
DFAedge(S5,0) &= \epsilon\text{-closure (goto(S5, 0))} = \{0,1,2,4,6,7,8\} = S5 \\
S7 &= DFAedge(S5,1) = \epsilon\text{-closure (goto(S5, 1))} = \{3,7,8,9,10\} = S7 \text{ (accepting state)} \\
DFAedge(S6,0) &= \epsilon\text{-closure (goto(S6, 0))} = \{\} \text{ Error state} \\
S8 &= DFAedge(S6,1) = \epsilon\text{-closure (goto(S6, 1))} = \{7,8,9,10\} = S8 \text{ (accepting state)} \\
DFAedge(S7,0) &= \epsilon\text{-closure (goto(S7, 0))} = \{\} \text{ Error state} \\
S9 &= DFAedge(S7,1) = \epsilon\text{-closure (goto(S7, 1))} = \{5,6,7,8,9,10\} = S9 \text{ (accepting state)} \\
DFAedge(S8,0) &= \epsilon\text{-closure (goto(S8, 0))} = \{\} \text{ Error state} \\
DFAedge(S8,1) &= \epsilon\text{-closure (goto(S8, 1))} = \{7,8,9,10\} = S8 \text{ (accepting state)} \\
\end{align*}
\]
This results in the DFA shown below with starting state $S_0 = \{0, 1, 6\}$.

d) A possible (but not unique) regular expression is: $((00) \cdot 0^+ \cdot (1 \mid 110 \mid 11) \cdot 1^+) \mid (110 \cdot 1^+)$

Problem 2 [60 points]: Predictive Top-Down Parsing

Consider the following CFG $G = (N=\{S, B, D\}, T=\{a, b, c, d\}, P, S)$ where the set of productions $P$ is given below:

$$
S \rightarrow B \cdot c \mid DB \\
B \rightarrow ab \mid cS \\
D \rightarrow d \mid \varepsilon
$$

a) [05 points] Is this grammar suitable to be parsed using the recursive descendent parsing method? Justify and modify the grammar if needed.

b) [15 points] Compute the FIRST and FOLLOW set of non-terminal symbols of the grammar resulting from your answer in a)

c) [15 points] Construct the corresponding set of mutually recursive parsing functions in a C-like imperative programming language (pseudo-code) as illustrated in class.

d) [15 points] Construct the corresponding parsing table using the predictive parsing LL method.

e) [10 points] Show the stack contents, the input and the rules used during parsing for the input $w =$ "cdabc"

Solution:

a) No because the grammar has productions that can generate sentential forms with common prefixes. The production $S \rightarrow B \cdot c$ and $S \rightarrow DB$ can both generate a sentential form that begin with an 'a' as the non-terminal D can generate the empty string. To eliminate this common prefix we first eliminate D's productions by folding them into the second production of S (see revised grammar below - left). As this revised grammar still suffers from the same problem we next create a new non-terminal 'C' for the tail end of S's productions resulting in the revised grammar below - right.

$$
S \rightarrow Bc \\
S \rightarrow dB \\
S \rightarrow B \\
B \rightarrow ab \\
B \rightarrow cS \\
B \rightarrow cS \\
C \rightarrow c \\
C \rightarrow \varepsilon
$$

Unfortunately, this grammar still does not have the LL(1) property as shown below in b).
b) We now compute the first of all the RHS of all the productions for this CFG. We also compute the Follow set of each non-terminal symbol. As can be seen this CFG does not have the LL(1) property as $\text{FOLLOW}(C) = \{ c, S \} \cap \text{FIRST}(c) \neq \emptyset$ ($C \rightarrow c$ is one of the productions in $P$).

\[
\begin{align*}
\text{FIRST}(Bc) &= \{ a, c \} & \text{FOLLOW}(S) &= \{ c, S \} \\
\text{FIRST}(B) &= \{ a, c \} & \text{FOLLOW}(B) &= \{ c, S \} \\
\text{FIRST}(dB) &= \{ d \} & \text{FOLLOW}(C) &= \{ c, S \} \\
\text{FIRST}(ab) &= \{ a \} & \\
\text{FIRST}(cS) &= \{ c \} & \\
\text{FIRST}(c) &= \{ c \} & \\
\text{FIRST}(\epsilon) &= \{ \epsilon \}
\end{align*}
\]

There is apparently no way to remove this conflict and the situation below illustrates why:

There are two possible derivation tree for the same input - the grammar is ambiguous.

c) The snippet of code below presents the set of mutually recursive procedures that implement a recursive-descendant parsing strategy for the revised grammar under the assumption that for an input 'c' we can choose the production $C \rightarrow c$ (in effect given precedence to an shift or expansion over a reduction):

```java
boolean parse_program(){
    if(parse_S() = true){
        if(current_token = EOF){
            return true; // success
        } else {
            error();
            return false;
        }
    } else {
        error();
        return false;
    }
}

boolean parse_S(){
    if((current_token = 'a') OR (current_token = 'c')){
        if(parse_B() = true){
            if(parse_C() = true){
                return true;
            }
        }
        if(current_token = 'd'){
            current_token = getNextToken();
            return parse_B();
        }
        error();
        return false;
    }
}

boolean parse_B(){
    if((current_token = 'a') OR (current_token = 'c')){
        if(parse_B() = true){
            if(parse_C() = true){
                return true;
            }
        }
        if(current_token = 'd'){
            current_token = getNextToken();
            return parse_B();
        }
        error();
        return false;
    }
}

boolean parse_C(){
    if(current_token = 'c'){
        current_token = getNextToken();
        return true;
    } else if(current_token = EOF){
        return true;
    }
    error();
    return false;
}
```
d) Under the assumption that for an input 'c' we can choose the production $C \rightarrow c$ (in effect given precedence to a shift or expansion over a reduction) the table below depicts the corresponding LL parsing table for the transformed grammar.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>$S \rightarrow BC$</td>
<td>$S \rightarrow BC$</td>
<td>$S \rightarrow dB$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>$B \rightarrow a b$</td>
<td>$B \rightarrow c S$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>$C \rightarrow c$</td>
<td>$C \rightarrow c$</td>
<td>$C \rightarrow \epsilon$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


e) The stack and input are as shown below using the predictive, table-driven parsing algorithm:

<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUT</th>
<th>RULE/OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>cdabc$</td>
<td>$S \rightarrow BC$</td>
</tr>
<tr>
<td>$S$ B</td>
<td>cdabc$</td>
<td>$B \rightarrow cS$</td>
</tr>
<tr>
<td>$S$ C</td>
<td>dabc$</td>
<td>match c</td>
</tr>
<tr>
<td>$S$ C</td>
<td>dabc$</td>
<td>$S \rightarrow dB$</td>
</tr>
<tr>
<td>$S$ C B</td>
<td>abc$</td>
<td>match d</td>
</tr>
<tr>
<td>$S$ C B</td>
<td>abc$</td>
<td>$B \rightarrow ab$</td>
</tr>
<tr>
<td>$S$ C B a</td>
<td>bcs$</td>
<td>match a</td>
</tr>
<tr>
<td>$S$ C B</td>
<td>c$</td>
<td>match b</td>
</tr>
<tr>
<td>$S$ C</td>
<td>c$</td>
<td>$C \rightarrow c$</td>
</tr>
<tr>
<td>$S$ c</td>
<td>$S$</td>
<td>match c</td>
</tr>
<tr>
<td>$S$</td>
<td>$S$</td>
<td>halt and accept</td>
</tr>
</tbody>
</table>