This document contains the solution for Tests 1. I have added additional comments, usually preceded by the keyword **comment**.

Test 1 - Solution

1. Regular expression: `c? a b* c`

cbbbbc	NO	**comment**: The a is missing.
cabc	YES	
abbc	YES	**comment**: optional c? not used
cac	YES	**comment**: zero length b*
abcccc	NO	**comment**: string must end with a single character c

2. The tokens generated by the lexical analyser are:

   - `x78 if 45 &` generates: ID(x78) IF NUMBER(45) BADCHAR(&)

     comment: The token definition `[]~` defines length-one strings made of any character. The `~` is similar to logical negation or the set complement operator. For example, it will match the first x of the input string but, since the lexical analyser is able to find a longer string i.e. x78, the latter is reported.

   - `78x ifelse 0` generates: NUMBER(78) ID(x) ID(`ifelse`) NUMBER(0)

     comment: ifelse matches two tokens, ID and IFELSE. The first is reported (ID) because it shows up first in the lexical specification.

   - `x$$98` generates: ID(x) BADCHAR($) BADCHAR($) NUMBER(98)

3. Given alphabet `{x,y}`, what regular expression defines the set of strings that always contain at least one character y?

   Answer: `(x|y)* y (x|y)*`

   comment: Regular expressions `y` and `x*yx*` define sets of strings that contain at least one character y. However, these sets are limited since they only define a fraction of the set of all possible strings that satisfy this condition. For example, the second regular expression does not contain strings xyxy and yyyy.

4. Regular expression that specifies odd numbers e.g. 1, 3, 227, 1001.

   Answer: `[0-9]* (1 | 3 | 5 | 7 | 9)`

   comment: The set of odd numbers defined by the regular expression above includes numbers that start with zero(es). For example, 0235 is part of the set. If we want to remove these numbers, the regular expression should be defined as follows:

   ```
   (1 | 3 | 5 | 7 | 9)  | ([1-9] [0-9]* (1 | 3 | 5 | 7 | 9))
   ```

   We have seen that multiples of 3 can't be represented this way (try). What other 'multiples of' can we represent using regular expressions?

5. Given the DFA defined in the question 5.

   - The strings accepted/rejected by the DFA are:

ca	ACCEPT(1,4,5) - **comment**: Does not use the transition labeled with b
cba	ACCEPT(1,4,4,5)
cb	REJECT - **comment**: DFA gets stuck at 4
ba	ACCEPT(1,2,3)
cbbbba	ACCEPT(1,4,4,4,4,5)
cab	REJECT - **comment**: DFA gets stuck at 5 with one character yet to be read.

comment: Execution of the DFA starts from initial state 1 (the state pointed by the arrow). Each character of the string must be consumed, from left to right, by a transition that matches the character. The string is accepted if ALL characters have been consumed and the current state is a **final** state (in this example, states 3 and 5).
Each of the ACCEPT cases above contain the sequence of states that lead to the final accepting state.

6. The DFA represents regular expression `(cb*a) | ba`.

   comment: The DFA contains two branches, the top branch defined by `ba` and the bottom branch defined by `cb*a`. Branching is represented by `|` (alternation), therefore the final regular expression is `(cb*a) | ba`. The transition labeled with b that starts and ends at state 4 is represented by b* (kleene closure), that is, it can accept the empty string, b , bb, bbb, etc.

7. I will not draw the DFA but will give its definition instead (note that you were asked to draw it).

   The DFA is made of the following components:

   - States: 1,2,3,4
   - Initial state: 1
   - Final states: 4
   - Trasitions: (1,'0',2), (2,'x',3), (3,[A-Z],4), (3,[0-9],4), (4,[A-Z],4), (4,[0-9],4)

   Transitions are represented by (s1,c,s2), where c is the character that labels the transition, s1 the starting state and s2 the target.

   comment: I have splitted '0x' into two transitions, as suggested by the instructions. The syntax of DFAs does not accept `|` (alternation), it has to be split into two transitions (branches). The transitions that go from 3 to 4 make sure that at least one character, [A-Z] or [0-9], is read. The transtions from 4 to 4 implement a kleene closure (the one implicit in +).

   A different solution uses an extra state 5 and two final states, 4 and 5:

   - States: 1,2,3,4,5
   - Initial state: 1
   - Final states: 4,5
   - Trasitions: (1,'0',2), (2,'x',3), (3,[A-Z],4), (3,[0-9],5), (5,[A-Z],4), (4,[0-9],5)

   Try with a few examples to check they actually work.