These document contains the solutions for Tests 1,2 and 3. I have added additional comments, useally 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. 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))
   ```

Test 2 - Solution

Test 2 will cover the following topics:

- Grammars. You should be able to:
  - Determine if a given string belongs to the language defined by a grammar. In other words, you should be able to generate derivations from a grammar. In particular, leftmost and rightmost derivations.
  - Identify ambiguous and left-recursive grammars, and come-up with a new equivalent grammar that is not ambiguous or left-recursive.

- TPL. Understand the instruction set and be able to write short programs, or code fragments. Understand the main translations from SPL to TPL.
- SPL. Be familiar with the concrete syntax, JavaCC spec and abstract syntax of SPL. The last lab (and programming assignment) contains the whole JavaCC spec of SPL.
- SPL to TPL. Given the syntax of new SPL expressions or statements, you should be able to come up with new abstract syntax, JavaCC specifications and translations into TPL.

1 Derivations/Ambiguous grammars

Given the grammar:

(1) E → E + E
(2) E → E - E
(3) E → E * E
(4) E → E / E
(5) E → (E)
(6) E → num

and string **'6 / 3 + (2)'**. Answer the following questions:

1. Show the leftmost derivation that generates the string. Label each arrow with the appropriate rule number (production rule).

   $\underline{E} \xrightarrow{1} \underline{E} + E \xrightarrow{4} \underline{E} / E + E \xrightarrow{6} 6 / \underline{E} + E \xrightarrow{6} 6 / 3 + \underline{E} \xrightarrow{5} 6 / 3 + (\underline{E})$
   $\xrightarrow{6} 6 / 3 + (2)$

   A leftmost derivation is one that always expands the leftmost non-terminal. I have underlined the non-terminal being expanded in order to show that, in the derivation above, an expansion/pruduction rule is always applied to the leftmost remaining non-terminal.

2. Show the rightmost derivation.

   $\underline{E} \xrightarrow{1} E + \underline{E} \xrightarrow{5} E + (\underline{E}) \xrightarrow{6} \underline{E} + (2) \xrightarrow{4} E / \underline{E} + (2) \xrightarrow{6} \underline{E} / 3 + (2)$
   $\xrightarrow{6} 6 / 3 + (2)$

   It's the same idea, just replace 'leftmost' with 'rightmost'.

3. When is a grammar ambiguous?

   A grammar is ambiguous if it can generate two different parse trees for the same string.

4. Is the grammar above ambiguous? Show using your previous answer.

   The grammar above is ambiguous. For example, the leftmost derivation shown above, and the derivation shown below:

   $\underline{E} \xrightarrow{4} E / \underline{E} \xrightarrow{4} \underline{E} / E + E \xrightarrow{6} 6 / \underline{E} + E \xrightarrow{6} 6 / 3 + \underline{E} \xrightarrow{5} 6 / 3 + (\underline{E})$
   $\xrightarrow{6} 6 / 3 + (2)$

   produce two different parse trees:

```
         E
         |
   --------------
   |     |      |
   E     +      E
   |            |
-------     ---------
|  |  |     |   |   |
E  /  E     (   E   )
|     |         |
6     3         2
```

and

```
         E
         |
   --------------
   |     |      |
   E     /      E
   |            |
   6        -----------
            |    |    |
            E    +    E
            |         |
            3      ---------
                   |   |   |
                   (   E   )
                       |
                       2
```

2 Left-Recursion

Given the grammar:

(1) L $\longrightarrow$ L ; L
(2) L $\longrightarrow$ S

- Write down 3 strings defined by the grammar above:
  - String **S**, defined by: L $\xrightarrow{2}$ S
  - String **S ; S**, defined by L $\xrightarrow{1}$ L ; L $\xrightarrow{2}$ S ; L $\xrightarrow{2}$ S ; S.
  - String **S ; S ; S**, defined by L $\xrightarrow{1}$ L ; L $\xrightarrow{2}$ S ; L $\xrightarrow{1}$ S ; L ; L $\xrightarrow{2}$ S ; S ; L $\xrightarrow{2}$ S ; S ; S.
- Is the grammar left-recursive? If it is, write down an equivalent grammar that is not left-recursive. Compare with the rulw given in the lecture notes.
  - Yes, the grammar is left-recursive because the right-hand side of rule L $\longrightarrow$ L ; L has L as its leftmost non-terminal.
  - We can easliy see that the grammar above defines a non-empty sequence of S's separated by semi-colons. This can be expressed by:

    (3) L $\longrightarrow$ S LS
    (4) LS $\longrightarrow$; S LS
    (5) LS $\longrightarrow$ ϵ

    Recall that the rule that removes left-recursion takes a grammar of the form:

    X $\longrightarrow$ X γ
    X $\longrightarrow$ α

    and transforms it into:

$$\begin{aligned} X &\longrightarrow \alpha\, X' \\ X' &\longrightarrow \gamma\, X' \\ X' &\longrightarrow \epsilon \end{aligned}$$

which is exactly what happens with our transformation after making α = S and γ = ; L, and using LS instead of X'.

- Write down an equivalent grammar using EBNF:

  Answer: S (; S)*

3 TPL

- Suppose memory locations 2 and 4 are used to store variables x and y, with values 50 and 100. What does the following code do? Whats the output?

```
STORE $2, R1
STORE $4, $2
STORE R1, $4   // the first versio moved R1 to $4
WRITEI $2
WRITEI $4
```

  The code swaps the values of x and y and prints the result: 100 50

- Write code that prints the contents of x n times, where n is the value of y. Do not modify y.

```
STORE $4, R1   // R1 will be our counter
LABEL L1
GT R1, 0, R2   // R2 = (R1 > 0)
JMP0 R2, L2    // jump if R2 is false
WRITEI $2
SUBI R1, 1, R1  // decrement counter
JMP L1
LABEL L2
```

4 SPL to TPL

Suppose we want to add a short version of a For statement to SPL using the following syntax:

ForStm $\longrightarrow$ for (*id* := *Exp* ; *Exp*; *id* := *Exp*) *Body*

where *Body* is a non-empty sequence of statements.
For example, we should be able to write:

```
for(x := 0; x < 10; x := x + 1) {  // this is the body
  print(x);
  y := y + x;
}
```

- Write down the abstract syntax for *ForStm*.

  The abstract syntax gets rid of all elements of the grammar (concrete syntax) that are necessary for parsing (such as punctuation marks, reserved words) and keeps the elements that will be needed for the future passes of the compiler. For example, our For-statement uses the keyword 'for', left and right parenthesis, and semicolons as part of its concrete syntax but only needs to store the names of the ids, three expressions and a list of statements. The abstract syntax can be expressed by:

ForStm(Id x1, Id x2, Exp e1, Exp e, Exp e2, Statement+ body)

where e1 corresponds to the first assignment, e to the test and e2 to the second assignment.

- Write the JavaCC code that parses and creates the AST for the For statement.

  ```
  Stm ForStm :
  { Exp e1, e2, e; List<Stm>  body;
    Token t1, t2;
  }
  {
    "for" "(" t1=<ID> ":=" e1=Exp() ";" e = Exp() ";" t2=<ID> ":=" e2=Exp() ")"
    body = block()
    { return new ForStm(t1.image, t2.image, e1, e, e2, body); }
  }
  ```

  assuming that their is a constructor for the class ForStm that takes as arguments two strings, 3 objects of class Exp and a list of Stm.

- Translate ForStm to TPL. Given abstarct syntax ForStm(Id x1, Id x2, Exp e1, Exp e, Exp e2, Statement+ body), a for-statement is executed as follows:

  1. x1 := e1 is executed.
  2. Test e is executed. If it returns true, continue. If it returns false, end execution of statement.
  3. Execute each statement in body.
  4. Execute x2 := e2
  5. goto 2).

  Following the sequence above, the translation into TPL will be:

  ```
  t1 = stable.getTemp(e1)
  a1 = stable.getAddress(x1)
  t = stable.getTemp(e)
  t2 = stable.getTemp(e2)
  a2 = stable.getAddress(x2)

  Code:
  genCode(e1,stable)          // evaluate e1 - result stored in t1
  STORE t1, a1                // assign result of e1 into address a1
  LABEL L1
  genCode(e, stable)          // evaluate e - result stored in t
  JMP0 t, L2                  // if result of e is false, jump to L2
  genCode(body,stable)
  genCode(e2,stable)          // evaluate e2 - result stored in t2
  STORE t2, a2                // assign result of e2 into address a1
  JMP L1
  LABEL L2
  ```