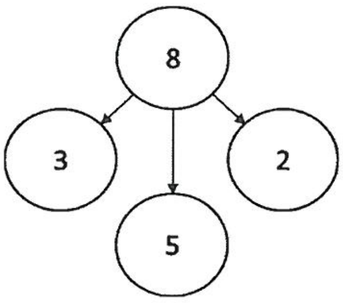
Exam in
Declarative Languages

Course code:	D7012E
Time:	4 hours
Number of assignments:	6
Total number of points:	31
Date of exam:	2016-08-17
Teacher:	Fredrik Bengtsson, tel. 0920492431, 0738166670
Allowed aiding equipment:	None

Good luck!

Assignment 1 (8p):
a (2p): Declare a data type that can represent a multi-branch tree. This is a tree where each node can have any number of sub-nodes (including 0). Each node also has a value which is the information content of the tree. Leaf-nodes (terminating nodes) should be represented by a regular node having zero child-nodes.

b (1p): What expression, using the type from (a), corresponds to the following tree:



c (3p): Implement a function `checkBin` that takes a multi-branch tree (from a) as a parameter and returns true if the tree is a binary tree (in a binary tree, each node has at most two children).

d (2p): Implement a function `flatten` that, given a multi-branch tree (from a), returns a list of all values for all nodes. The values from the nodes should be presented pre-order. That is, for each node, that value in that node should be placed before the values from the subnodes.

Assignment 2 (3p):
Assume three predicates, P1(X), P2(X) and P3(X). Declare a predicate P4(X) that corresponds to the following logic statement: (P1(X) and P2(X)) or (P3(X) and P2(X) and not P1(X))). You are not allowed to use the negation operators \+ or not or similar. Use cut.

Assignment 3 (4p):
Give the standard concatenation implementation:

```
conc([], L, L).
conc([X | L1], L2, [X | L3]):-
  conc(L1, L2, L3).
```

a: How many different solutions would the following goal yield on backtracking:

```
member(Z,[[1,2],[]]),conc(X,Y,Z).
```

State the binding for `X` and `Y` for each solution.

b: Same question as (a), but with the following goal:

```
member(Z,[[1,2],[]]),!,conc(X,Y,Z).
```

Assigment 4 (4p):

We want to implement a calculator in haskell. Our goal is to implement one of those shitty calculators that only have one accumulator and always applies the latest operator to the accumulator, regardless of operator precedence.
The calculator should ask for input and legal inputs will be "+", "*", "e" or an integer. The calculator maintains one internal value only (the accumulator) and whatever operation is performed is performed on this, and the entered value. The start value of the accumulator is 0. If "+" is entered, the next operation will be addition. If "*" is entered, the next operation will be multiplication. After an operation is entered, a numerical value may be entered and the operation specified previously should be performed on the accumulator and the new value. The accumulator value should be printed on screen. If "e" is entered, the program should quit. There are many special cases that can be implemented differently, but this is implementation specific. Only the above functionality is required. Behaviour in other cases are implementation specific.
Example of session (every second value is the value entered by the user, not the value printed from the program). User entered "3+4*2" with enter between each character.

```
Main> calc
0
3
3
+
3
4
7
*
7
2
14
e
```

Helpful functions (can be used freely):

```
putStrLn :: String -> IO ()
getLine :: IO String
read :: Read a => String -> a
print :: Show a => a -> IO ()
```

Assignment 5 (6p):
We want to be able to represent cash values given different values for notes (bills). Write a predicate `exchange(+L, +V, Ex)`, that given a list of values (notes values), `L`, and a value (cash value), calculates the shortest list `Ex`, such that the sum of all values in `Ex` is `V` and all values in `Ex` are taken from (also present in) `L`. Observe that the same value from `L` may be present several times in `Ex`. This is the classical cash exchange problem.

Assignment 6: Higher order functions (6p)

a (3p): Declare a function `foldl`

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

in Haskell that folds a list from the left, instead of from the right, as `foldr`. A direct implementation without auxiliary helper functions is required.

b (2p): Declare, using `foldl` from (a), a function `length` that computes and returns the length of a given list. The binary operator for `foldl` should be declared using lambda-function (anonymous function).

c (1p): Would `foldr` work for computing list length as `foldl` (possibly using a different binary operator)?