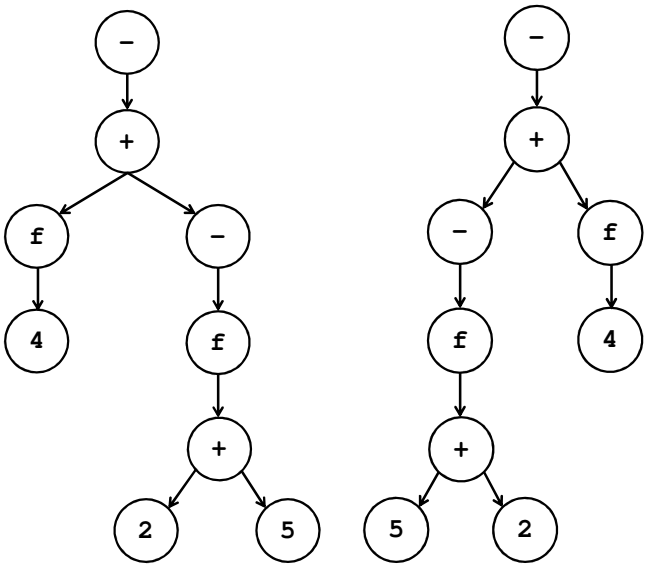
Exam in
Declarative Languages

Course code:	D7012E
Time:	4 hours, 9:00-13:00
Number of assignments:	6
Total number of points:	33
Date of exam:	2012-05-28
Teacher:	Fredrik Bengtsson, tel. 0920492431, 0738166670
Allowed aiding equipment:	Dictionary
Result announced:	2012-06-18

Good luck!

Assignment 1: Data structure (8p)

a (2p): Declare a data type `ETree`, in Haskell that are able to represent a special type of expressions. Our expressions are much like arithmetic expressions, but contains additions, unary minus (a single minus sign in front of an expression), a function application, and numbers. Only one type of function can be represented.
The data type should represent the expression in a tree-like fashion (sometimes called parse-tree or abstract syntax tree). The operations should be called `Add`, `Minus`, `Function` and `Number` in the tree. The function takes only one argument. An example of such an expression is "-(f(4)+-f(2+5)) and its corresponding tree would look like the *left* tree below:



b (1p): State the haskell expression that would correspond to the above tree.

c (2p): Declare a function that takes an `ETree` and a function from `int` to `int` as argument and evaluates the expression represented by the `ETree` and returns the result of the evaluation. The function should be used for all function evaluations in the tree.

d (3p): Declare a function, `mirror`, that use the fact that addition is commutative, in order to change the operand order for all additions while maintaining the arithmetic value of the represented expression. For example, all `a+b` chould be changed to `b+a`. The example from the *left* figure in (a) would look like the *right* figure in (a).

Assignment 2 (3p):

What is the logical equivalent of the following programs:

a:
```
p :- a,b.
p :- c.
```

b:
```
p :- a,!,b.
p :- c.
```

c:
```
p :- c.
p :- a,!,b.
```

State a logical (boolean) expression equivalent to p.

Assignment 3 (6p):

a: In prolog, declare a predicate `count(+E,+L,-N)`, that counts the number of occurrences of the element `E` in the list `L` and binds the result to `N`. Make sure the predicate always binds the correct number of elements to `N`, even when backtracking over the predicate. You are allowed to write a helper function that performs the actual work, but you are not allowed to use built-in predicates that performs the same or similar operations.

b: Using a correct implementation of count, what would happen if you call

```
count(A, [a,b,a,a,c,d,a], N).
```

Show what would be printed on screen, including attempts to perform backtracking until "false". (exact what characters are printed is not important, but you need to show what names are bound and to what value they are bound).

Assigment 4 (4p):

Quicksort is a sorting algorithm that works as follows: Pick a pivot element (can be chosen as first element of the sequence), partition the elements around the pivot (smaller than the pivot respective larger than the pivot). Sort each partition using quicksort. Put together the sorted sequence of smaller numbers, the pivot, and the sorted sequence of larger numbers, in that order. Done!
You are allowed to use built-in helper functions that you find suitable (but you are not allowed to use built-in functions that implements the same function as quicksort).

Implement quicksort as a function

```
quicksort :: Ord a => [a] -> [a]
```

in Haskell.

Assignment 5 (6p):

Suppose we have two buckets with clearly specified volumes and a supply of water. Our task is to measure a different volume by pouring water from one bucket to another, pouring the water out on the ground or filling a bucket full of water. This can be done repeatedly. If we pour water from a large bucket into a small bucket, there will still be water left in the large bucket. An example of such a problem instance could be: "You have buckets of 3 and 5 liters. Pour up 4 liters in one of the buckets." The task is to compute the sequence of water pouring.

a: Declare a prolog predicate `pour_(B1Vol, B2Vol, Vol, PourList, N)`, that pours up `Vol` amount of water in one of the buckets in at most N steps. One step can consist of pouring water from one bucket to another, filling a bucket of water or emptying a bucket. If you like, changing order of buckets can also count as a step. `PourList` should store the order of water-pouring according to the following format: `pour(From,To)`, where `From` is the volume (total volume of bucket, not current volume of water) of the bucket to pour from, `To` is the volume of the bucket to pour into. The last item of `PourList` should be the first pour action (`PourList` is in reversed order). Use `0` for `To` to represent that we pour everything from `From` on the ground and use `0` for `From` to represent that we fill water in `To` (we always fill to the top).

b: Declare a predicate `pour(B1Vol, B2Vol, Vol, PourList)`, that uses `pour_` and successively increases `N` until a solution is found.

If the problem instance is not possible to solve (for example, given 1 and 2 liter buckets, pour up 200 liter), the behaviour of `pour` is undefined (it is ok not to terminate).

Assignment 6: Higher order functions (6p)

a: Declare a function in haskell,

```
collapse :: BinTree a -> (a -> b -> b -> b) -> b -> b
```

, that takes as arguments a binary tree, a function and a value. Also, declare a datatype for a binary tree. `collapse` "combines" the values, of type `a`, from a tree into a single value of type `b`. The function argument `(a->b->b->b)` is used for the actual combining of values. The first argument to this function is the value from the node, whereas the second and third arguments are combined results from the left and right subtree, respectively. The last argument to collapse is the start value for the computation used at leaf nodes.

b: Define a function

```
treetolist :: BinTree a -> [a]
```

, that returns the node-values of a tree according to a pre-order traversal of the tree. Pree-order traversal is when you visit the node first, then all nodes in the left subtree then all nodes in the right subtree. Here, you *must* use collapse from (a) in order to perform most of the work.