Exam in
Declarative Languages

Course code:	D7012E
Time:	4 hours, 9:00-13:00
Number of assignments:	6
Total number of points:	35
Date of exam:	2009-01-15
Teacher:	Fredrik Bengtsson, tel. 0920492431, 0738166670
Allowed aiding equipment:	None

Don't forget to fill in the course evaluation (attached last) and hand it in, in a separate bin.

Good luck!

Assignment 1: Priority Queue **8p**

A priority queue can, for example, be implemented with a variant of a heap. The variant that we will study in this assignment works as a normal heap, except that there is no requirement of maintaining the heap-tree balanced. A node in a heap contains a *value* and *two nodes*. The value is larger than any value in the sub-trees represented by the two nodes. That is, the value in a node is larger than the value in the two child-nodes and this property holds for all nodes. You are about to implement four operations on this heap: min, delete-min, insert and lookup.

The operation min returns the smallest element of the heap.

The operation delete-min removes the smallest element from the heap and replaces it with the smallest element from the child-nodes. In order to remove the smallest element from a child-node, delete-min is performed on that node.

The operation insert inserts a new element into the heap. If this new element is smaller than the topmost element, the topmost element is replaced with the new element and the topmost element is then inserted in the left or the right sub-tree, chosen arbitrary by the programmer. If the new element is smaller than the topmost element, it is inserted into either child-node (chosen arbitrary).

Declare an algebraic data type, `PQHeap a`, for this kind of heaps Haskell.

Implement a function
```
max :: Ord a => PQHeap a -> a
```
that returns the maximum element of the heap.

Implement a function
```
delete-min :: Ord a => PQHeap a -> PQHeap a
```
that deletes the maximum element from the heap.

Implement a function
```
insert :: Ord a => PQHeap a -> a -> PQHeap a
```
that inserts an element into the heap. You may assume that all elements are distinct.

Implement a function
```
lookup :: Eq a => PQHeap a -> a -> Bool
```
that searches for an element in the heap, and returns true if the element is present in the heap. Observe that the heap is not ordered in the same way as a search tree. The lookup function does not need to be particularly efficient (this is not possible).

Assignment 2: Cash Exhange **5p**

In this assigment, we are to study the problem of exhanging money. The problem is that, if we have money of one large value, we want to exhange it for several smaller values. Depending on what values are available, the problem can be non-trivial.

Declare, in prolog, a predicate

```
exchange(ValueList, Value, ResultList)
```

where `ValueList` is a list of integer values and `Value` is an integer value. The predicate should compute one possible way to compose the value `Value` from the values in `ValueList` by summing values from `ValueList`. The result should be returned in `ResultList`. That is, the sum of all values in `ResultList` should be equal to `Value`. Each value in `ResultList` should be choosen from `ValueList`. It is ok to choose the same value several times. The predicate should be declared in such a way that all possible ways of obtaining the value `Value` could be generated throught backracking over the predicate. If it is not possible to choose values such that their sum becomes `Value`, the predicate should fail.

Assignment 3: Monads **3p**

Implement a user interface for sorting numbers in Haskell. The interface should prompt the user for numbers, one at a time, util the user enters the number 0. Then, all the numbers entered, except 0, should be sorted and output on the screen. You can assume there is a function `sort :: Ord a=> [a] -> [a]` for sorting the numbers.

Assignment 4: Logic **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 5: Higher order functions **6p**

In this assignment, you are not allowed to declare recursive functions directly. Instead, you are expected to use higher order functions to build your functions.

a: Declare a function in haskell,

```
comp :: [a -> a] -> a -> a
```

, that takes a list of functions `a -> a` and returns a composed function `a -> a`, where functions from the list are successively applied to the argument, from right to left. That is, given list `[f1, f2, f3, ... ]`, `comp` should return a function computing `f1(f2(f3(...)))`.

b: A repeat-loop is a construction that can perform the same actions repeatedly, each time updating a state. Declare a function

```
repeat :: (a -> a) -> Int -> a -> a
```

, where `a` is any type, but can be thought of as a state of the computation. The integer is the number of iterations in the repeat-loop. The semantics of the repeat-loop is to compose the function `(a -> a)` repeatedly (as in exercise a) as many times as the Int specifies. The `Int` is a natural number (starting from 0). You are required to use the function `comp`, declared in a. Even if you did not solve problem a, you may use comp in the solution to this problem.

c: Declare a function

```
pow :: Floating a => a -> Int -> a
```

that computes the first argument raised to the power of the second argument. That is, if the first argument is `a` and the second is `b`, `pow a b` is a^b. You have to use the repeat-loop from exercise b. Even if you did not solve problem b, you may use the for-loop in the solution to this problem.

Assignment 6: Negation **10**

a: In prolog, declare a predicate `not(P)`, which succeeds if and only if `P` fails.

b: The declaration of `not(P)` includes a red cut. What would happen if we removed this cut? Would the predicate work? Why/Why not?

Consider the following code:

```
notmember(X,L):-not(member(X,L)).

member(X,[X,_]).
member(X, [_|Rest]):-
  member(X, Rest).
```

In the following, assume no other bindings than the above two predicates.

c: Now, will `notmember(a, [b, c, d, e])` succeed?
State, for each subgoal of execution if that particular subgoal succeeds and explain why. You don't need to state if subgoals of the recursive member function succeeds or not. (start with the subgoal of `notmember` and continue from there. Stop with member.)

d: Will `notmember(A, [b,c,d,e])` succeed? (as in the 5c-assigment).