**Exam in**

# Declarative Languages

| | |
|---|---|
| Course code: | D7012E |
| Time: | 4 hours, 9:00-13:00 |
| Number of assignments: | 6 |
| Total number of points: | 34 |
| Date of exam: | 2009-03-17 |
| Teacher: | Fredrik Bengtsson, tel. 0920492431, 0738166670 |
| Allowed aiding equipment: | None |
| Result announced: | 2009-04-07 |

**Assignment 1: Special heap** **10p**

In this assigment, we will study a variant of a heap. This variant works as a normal heap, with a few exceptions, as follows: A node in a heap contains a *value* and *two nodes*. The value is smaller than any value in the sub-trees represented by the two nodes. That is, the value in a node is smaller 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, max-depth, delete-min, insert.

The operation min returns the smallest element of the heap.

The operation max-depth returns the maximum depth in the tree representing the heap.

The operation delete-min removes the smallest element from the heap and replaces it with the element from the child-node where the maximum tree depth is the smallest of the two child-nodes. In order to remove the smallest element from a child-node, delete-min is performed on that node.
When implementing delete-min, you are allowed to check the tree depth in each node (you do not have to store the path to the deepest node, even though this would be more efficient).

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 as the subtree that does *not* contain the deepest node. If the new element is larger than the topmost element, it is inserted into the subtree *not* containing the deepest element.

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

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

Implement a function
```
max-depth :: Ord a => Heap a -> Int
```
that returns the maximum depth of any node in a heap.

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

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

**Assignment 2: Load a Truck** **4p**

We are posed with the problem of loading a truck with heavy boxes. The size of the boxes is not a problem, but we want to load as much weight as possible onto the truck. However, the truck can only hold a certain amount of weight and we want to be as close as possible to this weight.You should implement, in prolog, a predicate

```
load(WeightList, MaxWeight, ResultList)
```

where `WeightList` is a list of values of weights of the boxes and `MaxWeight` is a value bound to the maximum allowed weight on the truck. The predicate should compute the best possible combination of boxes to load onto the truck and bind the resulting list of box weights to `ResultList`. The predicate should fail if it is not possible to load the truck with any boxes.

**Assignment 3: Monads** **3p**

Implement a user interface for adding numbers in haskell. The interface should prompt the user for numbers, one at a time, until the user enters 0. The program should then print the sum of the numbers on screen.

**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 and backtracking** **8p**

**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 6c-assigment).