# Exam in
# Declarative Languages

| | |
|---|---|
| Course code: | D7012E |
| Time: | 4 hours, 9:00-13:00 |
| Number of assignments: | 7 |
| Total number of points: | 31 |
| Date of exam: | 2008-08-20 |
| Teacher: | Fredrik Bengtsson,<br>tel. 0920492431, 0738166670 |
| Allowed aiding equipment: | None |

**Assignment 1: Subset range** **4p**

In the subset-range problem, we are given a sequence of integers and two parameters m and n. The problem is to decide if there exists a subset of integers from the sequence such that their sum is between (inclusive) m and n (greater than or equal to m but smaller than or equal to n).

Declare a predicate

```
subsetrange(List, M, N)
```

which succeeds if an only if there exists a solution.

**Assignment 2: Unbalanced Heap** **6p**

An unbalanced heap 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 three operations on an unbalanced heap: max, delete-max and insert.

The operation max returns the largest element of the heap.

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

The operation insert inserts a new element into the heap. If this new element is larger 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, `UHeap a`, for unbalanced heaps Haskell.

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

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

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

**Assignment 3: Insertionsort** **3p**

Insertionsort is a sorting algorithms that works as follows: Take the first element in the unsorted list and put it in the right place in the sorted list. The right place is found by traversing the sorted list and inserting the element into the correct position. This is repeated until the unsorted list is empty. Done!

Implement insertionsort as a function

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

in Haskell. You may not use the built-in function `insert`.

**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: Negation** **6p**

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

**b:** 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.
Now, will `notmember(a, [b, c, d, e])` succeed? Explain why.
Will `notmember(A, [b,c,d,e])` succeed? Explain why.

**Assignment 6: Monads** **3p**

Implement a user interface for sorting numbers in Haskell. The interface should prompt the user for numbers, one at a time, and then sort the numbers. The input is ended by entering the number 0. After sorting, the sorted sequence should be printed on screen. You should use the function `insertionsort` implemented in assignment 3.

**Assignment 7: Higher implicit recursion** **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 assignment 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 repeat-loop in the solution to this problem.