LULEÅ UNIVERSITY OF TECHNOLOGY

Final exam in **Declarative programming**
Number of problems: 7
Teacher: Håkan Jonsson, 491000, 073-8201700
The result will be available: ASAP.

Course code	D7012E
Date	2017-05-26
Total time	4 tim

Apart from general writing material, you may use: Dictionary.

---

1 List processing [Haskell]

Implement the two somewhat related functions `prefix :: Int -> [a] -> [a]` and `suffix :: Int -> [a] -> [a]` that, given a length i and a list ℓ, return parts of ℓ. A prefix is a first part while a suffix is a last part. The length i specifies how long the prefix/suffix should be as shown in the examples below. Note that negative as well as much too large lengths are OK. (6p)

```
prefix -1 [1,2,3,4] --> []          suffix -1 [1,2,3,4] --> []
prefix  0 [1,2,3,4] --> []          suffix  0 [1,2,3,4] --> []
prefix  3 [1,2,3,4] --> [1,2,3]     suffix  3 [1,2,3,4] --> [2,3,4]
prefix 99 [1,2,3,4] --> [1,2,3,4]   suffix 99 [1,2,3,4] --> [1,2,3,4]
```

2 Trees [Haskell]

(a) Declare an algebraic data type `Tree a` for binary trees with empty inner nodes and leaves with values of type `a`. (2p)

(b) Write the `Tree a`-expression that corresponds to the tree in the figure on the last page. (2p)

(c) Write a function `flatten :: Tree a -> [a]` that turns a tree T of type `Tree a` into a list such that if x is a value in the left subtree of an inner node n in T and y is a value in the right subtree of n, x ends up before (left of) y in the list[1]. If the tree on the last page is flattened, we get the list `[1,2,3,4,5,6]`. (4p)

(d) Write a function `balance :: Tree a -> Tree a` that returns a balanced version of a tree using `flatten` and the method outlined below. You may use `prefix` and `suffix` from Problem 1 even if you do not solve that problem. (4p)

- First, flatten the tree into a list containing all values.
- Then build a balanced tree of the list as follows:
  1) If the list contains just one value x, construct a leaf that contains x.
  2) Otherwise, for lists with more than one element, divide the list roughly in half, compute balanced trees of the two halfs, and then construct a node with the two balanced trees as subtrees. By "roughly in half" we mean that the lengths of the lists differ at most by one.

  Note that because of how `Tree a` is defined, there is no need to build trees of empty lists.

[1] This corresponds to an *inorder* traversal of the tree.

3 I/O and some monadic programming [Haskell]

Implement a simple cash register in Haskell: The cash register should repeatedly wait for number inputs as long as numbers are entered and, when an empty line is entered, the sum of the previous numbers should be printed. When an "e" is entered, the program should exit. (6p)

Example of session:

```
Main> cashreg
5
7
3

15
-------
3
8

11
-------
e
```

Observe that the entered numbers are printed as they are entered by Haskell and need not be printed by other means.

Helpful functions (can be used freely):

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

4 Higher-order functions [Haskell]

These are two common higher-order functions we learned about in the course:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
map :: (a -> b) -> [a] -> [b]
```

Show how to implement `map` with (just) `foldr` and a lambda expression. (4p)

5 Membership issues [Prolog]

(a) In Prolog, declare a predicate `member(X,L)`, that determines whether or not `X` is part of the list `L` or not. (2p)

You are allowed to write helper functions that performs the actual work, but you are not allowed to use built-in predicates that performs the same or similar operations.

(b) Using your implementation of `member`, what would happen if you call

```
?- member(A, [a,b,c,a,d,a]).
```

from a Prolog prompt?

Show what would be printed on screen, including attempts to perform backtracking until the Prolog system prints "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). (2p)

6 Logical equivalence [Prolog]

What is the logical equivalent of the following programs? State a logical (boolean) expression equivalent to `p` in each of the six cases. (6p)

(a)

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

(b)

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

(c)

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

(d)

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

(e)

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

(f)

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

7 Money exchange [Prolog]

Design a predicate, `cashExchange(ValueList, Sum, ResultList)`, that, in `ResultList`, provides a list of values such that the sum of all elements in `ResultList` is `Sum`. Legal values are present in `ValueList`. Each value from `ValueList` is positive and can be used as may times as appropriate. `ResultList` should have minimum length (as few values as possible). If no solution is possible, `cashExchange` should fail. You may declare helper predicates as you like. (6p)