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	2017-08-16
Total time	4 tim

Apart from general writing material, you may use: A dictionary. Motivate and explain your solutions. Note that half the exam contains predefined functions and operators you may use freely if not otherwise stated.

---

1 Turning lists into lists [To be solved using Haskell]

(a) Implement a function `append :: [a] -> [a] -> [a]` that works like the operator `++` on page 5. Do so without using any of the predefined operators and functions in Appendix A *except* the list constructor `:`. (3p)

(b) Implement the function `concat` listed on page 5. You may use `append` above, even if you fail to implement it, but no other predefined function or operator. (3p)

2 Ternary trees [To be solved using Haskell]

(a) Declare an algebraic data type `TT a` for trees with inner nodes and leaves. An inner node should contain a value of type `a` and three subtrees of types `TT a`. A leaf should not contain anything. (2p)

(b) Write the `TT Int`-expression for the tree in Fig. 1a. (1p)

(c) Write a function `prune :: TT a -> TT a` that substitutes every inner node whose subtrees are all leaves for a single leaf. Fig. 1 shows an example. (2p)

(d) Write a function that counts how many times a `TT a` can be pruned until what remains is just a leaf. Although it is not required, you may use `prune` above even if you fail to implemented it. The tree in Fig. 1a can be pruned 3 times. (3p)

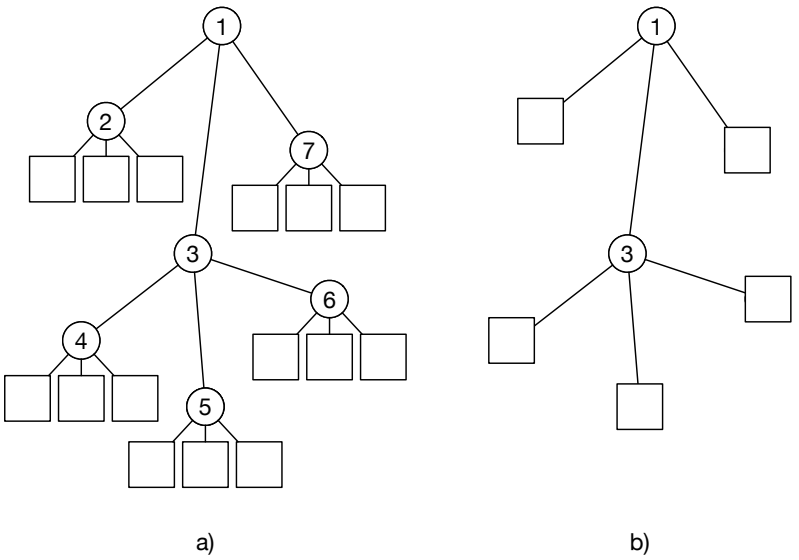

Figure 1: a) A `TT Int`. b) The tree in a) pruned.

3 I/O and some monadic programming [To be solved using Haskell]

Implement a user interface for adding numbers in Haskell. The interface should prompt the user for numbers, one at a time. When nothing but return is pressed, numbers entered so far are printed (in order) along with their sum and then the program continues as if no numbers have been entered. If no numbers have been entered, `(none)` should be printed along with the sum 0. (6p)

The example below shows how the output should be. Here, 1, 2, and 3 are first entered. Then no number is entered before we just press return (and that is why `(none)` is printed). Finally, 4 followed by 5 is entered. Note that once started, the program keeps asking for user input "forever".

```
Main> calculator
Enter number: 1
Enter number: 2
Enter number: 3
Enter number:
Numbers entered: 1, 2, 3
Accumulated sum: 6
Sum reset.
Enter number:
Numbers entered: (none)
Accumulated sum: 0
Sum reset.
Enter number: 4
Enter number: 5
Enter number:
Numbers entered: 4, 5
Accumulated sum: 9
Sum reset.
Enter number:
```

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

Two helpful functions that can be used freely:

```
putStrLn :: String -> IO ()
getLine :: IO String
```

4 Higher-order functions [To be solved using Haskell]

(a) Show how to implement the higher-order function `filter` without any predefined functions or operators. (3p)

(b) Write a function `squareSum :: Int -> Int` that, given an integer $n > 0$, returns the sum $1^2 + 2^2 + 3^2 + \cdots n^2$. Do so using only the higher-order functions `foldr` and `map` together with lambda expressions and list comprehensions. (3p)

Hint: Use `map` to transform the list of numbers $1, 2, 3, \ldots, n$ to a list with the terms in the final sum. Compute the sum of the list with the terms with a summation function based on `foldr`.

5 Turning lists into lists [To be solved using Prolog]

(a) In Prolog, declare a predicate `append(L1,L2,L3)` that computes the concatenation of `L1` and `L2` and returns it as `L3`. (This is Prolog's version of Haskell's `++`). Do this without the use of any predefined predicate. (3p)

(b) Implement the predicate `concat(LL,L)` that concatenates (in order) all lists in `LL` into one list and returns this list as `L`. You may use `append` from subproblem (a), even if you fail to implement it, but no other predefined predicate. (3p)

(c) Using your implementation of `append`, what would happen if you call

```
?- append(X, Y, [a,b,c,d]).
```

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 [To be solved using Prolog]

What is the logical equivalent of the following programs? State a logical (boolean) expression equivalent to `p` in each of the four cases. (4p)

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

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

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

(d)
```
p :- !, c.
p :- a, b.
```

7 Packing a suitcase [To be solved using Prolog]

Imagine we have a number of (deformable) bags filled with different amounts of liquid and are asked to pack them into a suitcase. Since all bags contain the same liquid, it does not matter which bags we pack. However, we like to pack as much liquid as possible. Unfortunately, there is not room for all bags in the suitcase and bags can not be opened or split. So which bags should we then pack?

Write a predicate `pack(AvailableBags,SuitcaseVolume,BagsPacked)` that finds an optimal packing. The list `AvailableBags` contains the *volume* of each of the bags while, as the name suggests, `SuitcaseVolume` is the volume of the suitcase. (To simplify, we only consider the volumes of bags and ignore their identities.) The list `BagsPacked` is the result and contains a list of volumes of bags taken from `AvailableBags`. You can assume that `AvailableBags` is non-empty, but has at most 20 elements, and that `SuitcaseVolume` is more than 0. You may declare helper predicates as you like. (6p)

Hint: Generate all subsets of `AvailableBags` whose total volumes are not too large. The result is then the subset with the largest volume. Be sure not to pack a bag more than once.

A List of predefined functions and operators

A.1 Haskell

A.1.1 Arithmetics and mathematics in general

`+`	The sum of two integers.
`*`	The product of two integers.
`^`	Raise to the power; `2^3` is 8.
`-`	The difference of two integers, when infix: `a-b`; the integer of opposite sign, when prefix: `-a`.
`div`	Whole number division; for example `div 14 3` is 4. This can also be written `14 'div' 3`.
`mod`	The remainder from whole number division; for example `mod 14 3` (or `14 'mod' 3`) is 2.
`abs`	The absolute value of an integer; remove the sign.
`negate`	The function to change the sign of an integer.

Note that `'mod'` surrounded by **backquotes** is written between its two arguments, is an **infix** version of the function `mod`. Any function can be made infix in this way.

`+ - *`	`Float -> Float -> Float`	Add, subtract, multiply.
`/`	`Float -> Float -> Float`	Fractional division.
`^`	`Float -> Int -> Float`	Exponentiation `x^n` = x^n for a natural number `n`.
`**`	`Float -> Float -> Float`	Exponentiation `x**y` = x^y.
`==,/=,<,>, <=,>=`	`Float -> Float -> Bool`	Equality and ordering operations.
`abs`	`Float -> Float`	Absolute value.
`acos,asin atan`	`Float -> Float`	The inverse of cosine, sine and tangent.
`ceiling floor round`	`Float -> Int`	Convert a fraction to an integer by rounding up, down, or to the closest integer.
`cos,sin tan`	`Float -> Float`	Cosine, sine and tangent.
`exp`	`Float -> Float`	Powers of e.
`fromInt`	`Int -> Float`	Convert an `Int` to a `Float`.
`log`	`Float -> Float`	Logarithm to base e.
`logBase`	`Float -> Float -> Float`	Logarithm to arbitrary base, provided as first argument.
`negate`	`Float -> Float`	Change the sign of a number.
`pi`	`Float`	The constant pi.
`signum`	`Float -> Float`	`1.0`, `0.0` or `-1.0` according to whether the argument is positive, zero or negative.
`sqrt`	`Float -> Float`	(Positive) square root.

A.1.2 Relational

`>`	greater than (and not equal to)
`>=`	greater than or equal to
`==`	equal to
`/=`	not equal to
`<=`	less than or equal to
`<`	less than (and not equal to)

A.1.3 List processing

`:`	`a -> [a] -> [a]`	Add a single element to the front of a list. `3:[2,3]` ⇝ `[3,2,3]`
`++`	`[a] -> [a] -> [a]`	Join two lists together. `"Ron"++"aldo"` ⇝ `"Ronaldo"`
`!!`	`[a] -> Int -> a`	`xs!!n` returns the nth element of `xs`, starting at the beginning and counting from 0. `[14,7,3]!!1` ⇝ `7`
`concat`	`[[a]] -> [a]`	Concatenate a list of lists into a single list. `concat [[2,3],[],[4]]` ⇝ `[2,3,4]`
`length`	`[a] -> Int`	The length of the list. `length "word"` ⇝ `4`

`head,last`	`[a] -> a`	The first/last element of the list.
		`head "word"` ⇝ `'w'`
		`last "word"` ⇝ `'d'`

`tail,init`	`[a] -> [a]`	All but the first/last element of the list.
		`tail "word"` ⇝ `"ord"`
		`init "word"` ⇝ `"wor"`

`replicate`	`Int -> a -> [a]`	Make a list of n copies of the item. `replicate 3 'c'` ⇝ `"ccc"`
`take`	`Int -> [a] -> [a]`	Take n elements from the front of a list. `take 3 "Peccary"` ⇝ `"Pec"`
`drop`	`Int -> [a] -> [a]`	Drop n elements from the front of a list. `drop 3 "Peccary"` ⇝ `"cary"`
`splitAt`	`Int -> [a] -> ([a],[a])`	Split a list at a given position.
`reverse`	`[a] -> [a]`	Reverse the order of the elements. `reverse [2,1,3]` ⇝ `[3,1,2]`
`zip`	`[a]->[b]->[(a,b)]`	Take a pair of lists into a list of pairs. `zip [1,2] [3,4,5]` ⇝ `[(1,3),(2,4)]`
`unzip`	`[(a,b)] -> ([a],[b])`	Take a list of pairs into a pair of lists. `unzip [(1,5),(3,6)]` ⇝ `([1,3],[5,6])`
`and`	`[Bool] -> Bool`	The conjunction of a list of Booleans. `and [True,False]` ⇝ `False`
`or`	`[Bool] -> Bool`	The disjunction of a list of Booleans. `or [True,False]` ⇝ `True`
`sum`	`[Int] -> Int` `[Float] -> Float`	The sum of a numeric list. `sum [2,3,4]` ⇝ `9`
`product`	`[Int] -> Int` `[Float] -> Float`	The product of a numeric list. `product [0.1,0.4 .. 1]` ⇝ `0.028`

A.1.4 General higher-order functions and operators

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)          (Function composition)
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldr :: (a -> b -> b) -> b -> [a] -> b
```

A.2 Prolog

A.2.1 Mathematical operators

Common arithmetic operators like `+`, `-`, `*`, and `/`.

A.2.2 List processing

- ```
  % length(List,N) - returns the length as the integer N
  length([],0).
  length([H|T],N) :- length(T,N1), N is 1 + N1.
  ```
- ```
  % member(X,List) - checks if X is a member of a List
  member(X,[X|_]).
  member(X,[_|Rest]):- member(X,Rest).
  ```
- ```
  % conc(L1,L2,List) - adds list L2 to L1 and returns L3
  % example: conc([b,c],[a,b,e],X). X = [b,c,a,b,e]
  conc([],L,L).
  conc([X|L1],L2,[X|L3]):-  conc(L1,L2,L3).
  ```
- ```
  % del(X,L,L1) deletes element X from list L
  del(X,[X|L],L).
  del(X,[A|L],[A|L1]):-  del(X,L,L1).
  ```
- ```
  % insert(X,L,BL) - inserts X to a custom position in list and returns BL
  insert(X,List,BL):-  del(X,BL,List).
  ```

A.2.3 Procedures to collect all solutions

- `findall`
- `setof`
- `bagof`

A.2.4 Other operators

`!`	cut
`<`, `>`, `>=`, `=<`	relational operations
`=`	unification (doesn't evaluate)
`\=`	true if unification fails
`==`	identity
`\==`	identity predicate negation
`=:=`	arithmetic equality predicate
`=\=`	arithmetic equality negation
`is`	variable on left is unbound, variables on right have been instantiated.
  ```