OBS! DENNA SIDA INGÅR INTE I ORIGINALET SOM SKA KOPIERAS UPP!

Uppgifter till tryckeriet

- Kurs: **D7012E Declarative languages**
- Tentamensdatum: **2018-08-22**
- Examinator/lärare: **Håkan Jonsson**

- Projektnr: **341980**
- Upplaga: **?**
- Antal sidor: **7** (förutom denna sida som inte ingår i tentan)
- Dubbel- eller enkelsidigt: **Enkelsidigt**

OBS! DENNA SIDA INGÅR INTE I ORIGINALET SOM SKA KOPIERAS UPP!

LULEÅ UNIVERSITY OF TECHNOLOGY

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

Course code	D7012E
Date	2018-08-22
Total time	4 tim

General information

I. Predefined functions and operators Note that Appendices A and B – roughly half the exam – list predefined functions and operators you may use freely, if not explicitly stated otherwise.

II. The Prolog database If not explicitly stated otherwise, solutions may *not* be based on the direct manipulation of the database with built-in procedures like `asserta`, `assertz`, `retract`, etc.

III. Helper functions It is allowed to add helper functions, if not explicitly stated otherwise. (Maybe needless to write but, of course, all added helpers must also be written in accordance with the limitations and requirements given in the problem description.)

IV. Explanations You must give **short** explanations for all declarations. Haskell declarations must include types. For a function/procedure, you must explain what it does and what the purpose of each argument is.

Solutions that are poorly explained might get only few, or even zero, points. This is the case regardless of how correct they otherwise might be.

Explain with at most a few short and clear (readable) sentences (not comments). Place them next to, but clearly separate from, the code. Use arrows to point out what is explained. Below is one example, of many possible, with both Haskell and Prolog code explained.

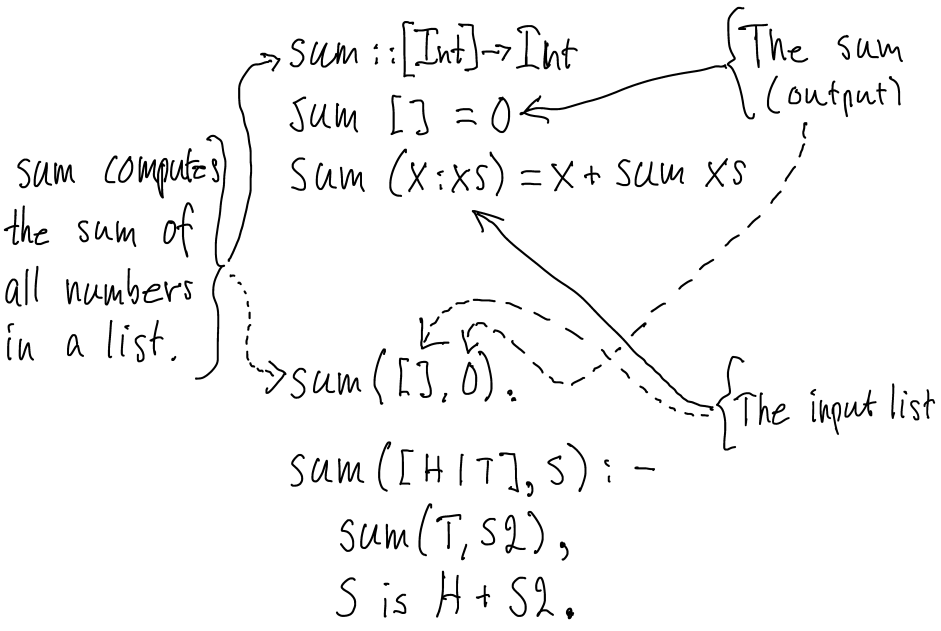

Figure 1: Example showing how to explain code.

1 Types and higher-order functions [To be solved using Haskell]

(a) State for each valid (correctly declared) function below its type. If a function is not valid, explain instead what is wrong. (2p)

a)
```
fa x = length x : x
```

b)
```
fd f xs = [f x | x <- xs]
```

c)
```
fb [] ys = ys
fb (x:xs) ys = x : fb xs ys
```

d)
```
fc (x:y:z) = x < y && fc (y:z)
fc _ = True
```

(b) Implement the operator `(.)`, *function composition*, listed in section A.4 of Appendix A. Do so without any predefined functions or operators. (2p)

2 Recursion over numbers and trees [To be solved using Haskell]

(a) Declare an algebraic type `T a` for binary tree with values of type `a` in both leaves and nodes. (3p)

(b) Write a function `transform :: T a -> (a -> a) -> T a` that, given a tree `t` and a function `f`, returns a tree with the same branching structure as `t` but in which each value `x` of type `a` is exchanged for `f x`. (4p)

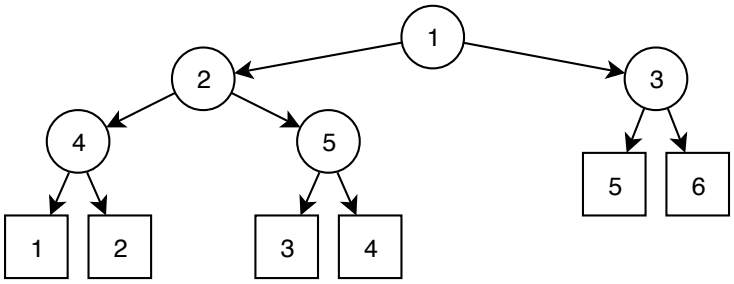

Figure 2: A tree `T Int`.

(c) Figure 2 shows a tree of type `T Int` in which circles denote inner nodes and squares denote leaves. Draw the tree that is returned if `transform` is called with the tree in the figure and the function `g` defined: (3p)

```
g n = if n<3 then n else g(n-1)+g(n-2)+g(n-3)
```

3 Escaping from a matrix [To be solved using Haskell]

A two-dimensional integer matrix A is *escapable* from a given element e_1 of A if either $e_1 = 1$ or a sequence of $n > 1$ elements $e_1, e_2, \ldots, e_n$ exists such that each $e_i \in \{0, 1\}$, $e_n = 1$, and e_i lies next to e_{i+1} either on the same row or the same column. The matrix below is, for instance, escapable from $(2, 2)$. One sequence that establishes this fact is $a_{2,2}, a_{3,2}, a_{3,3}, a_{3,4}, a_{3,5}, a_{2,5}, a_{2,6}, a_{2,7}, a_{3,7}$, $a_{4,7}, a_{4,8}$, and $a_{5,8}$ (highlighted in **bold**).

$$\begin{pmatrix} 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\ 8 & \mathbf{0} & 8 & 8 & \mathbf{0} & \mathbf{0} & \mathbf{0} & 8 & 0 & 8 \\ 8 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 8 & \mathbf{0} & 8 & 0 & 8 \\ 8 & 0 & 0 & 8 & 0 & 0 & \mathbf{0} & \mathbf{0} & 0 & 8 \\ 8 & 8 & 8 & 8 & 8 & 8 & 8 & \mathbf{1} & 8 & 8 \end{pmatrix}$$

We represent a matrix with a list `[[Int]]`, in which sublists are rows of the matrix. All matrices are assumed to have at least one element ($a_{1,1}$). There is no need to check this.

(a) A pair (i, j) is *valid* in a matrix A if $a_{i,j}$ is an element of the matrix.

Write a function `valid :: [[Int]] -> (Int,Int) -> Bool` that, given a matrix A and a pair (i, j), decides if (i, j) is valid in A. (1p)

(b) Write a function `a :: [[Int]] -> (Int,Int) -> Int` that, given a matrix A and a valid pair (i, j), returns the element $a_{i,j}$ of A. (1p)

(c) Write a function `escapable :: [[Int]] -> (Int,Int) -> Bool` that, given a matrix A and a valid pair (i, j), decides if A is escapable from $a_{i,j}$. (4p)

Hint: A is also escapable from an element $a_{i,j}$ if either $a_{i,j} = 1$ or $a_{i,j} = 0$ and A is escapable from one of $a_{i,j-1}, a_{i,j+1}, a_{i-1,j}$, or $a_{i+1,j}$. Make sure your function avoids going in cycles, so it always terminates properly.

4 The factorial function [To be solved using Prolog]

(a) The *factorial function* is defined as follows:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & n > 0. \end{cases}$$

Write a procedure `factorial(N,FN)` that returns $N!$ as `FN`. Write `factorial` so it does not backtrack in attempts to find more than one solution. (4p)

(b) Write a procedure `factorials(N,L)` that returns `L`$= [0!, 1!, \ldots, (N-1)!, N!]$. You may use `factorial` from (a) even if you do not solve that problem. (3p)

Example:

```
?- factorials(7,L).
L = [1, 1, 2, 6, 24, 120, 720, 5040].
```

> In this subproblem you are allowed, if you like, to use the built-in procedure `sort(0, @=<, In, Out)` that sorts the list `In` in ascending order, while keeping duplicate elements, and returns it as `Out`.

5 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 :- A.
```

b)
```
P :- A, !, B.
P :- A.
```

c)
```
P :- A, !, B.
P :- B.
```

d)
```
P :- A, B, !.
P :- B.
```

6 Relative relatives [To be solved using Prolog]

In this problem the Prolog database contains parent relations of the type `parent(X,Y)`, that states that `X` is the parent of `Y`.

(a) Write a procedure `siblings(X,Y)` that is true if, and only if, `X` and `Y` have a parent in common. (1p)

(b) Write a procedure `cousins(X,Y)` that is true if, and only if, `X` and `Y` are cousins, that is if a parent of `X` and a parent of `Y` are siblings. (2p)

7 Frequency counting [To be solved using Prolog]

In this two-part problem, the final objective is to compute how frequent elements are in a list.

(a) Write a procedure `f(X,L,N)` that returns, in `N`, the number of times `X` occurs in the list `L`. (3p)

(b) A *frequency table* states how often each element of a list occurs in the list. For `[a,c,b,a,a,c,a,b]` it is a list containing `[a,4]`, `[b,2]`, and `[c,2]]` (in some order).

Write a procedure `fL(L,FT)` that computes a frequency table `FT` for the list `L`. (3p)

A List of predefined Haskell functions and operators

NB! If `$` is a binary operator, `($)` is the corresponding two-argument function. If `f` is a two-argument function, `` `f` `` is the corresponding binary infix operator. Examples:

```
    [1,2,3] ++ [4,5,6] ⟺ (++) [1,2,3] [4,5,6]
map (\x -> x+1) [1,2,3] ⟺ (\x -> x+1) `map` [1,2,3]
```

A.1 Arithmetics and mathematics in general

```
For integers: +  -  *  div  mod  ^
              abs, negate

For floats:   +  -  *  /  **
              cos, acos, sin, asin, tan, atan, abs, negate,
              exp, log, ceiling, floor, round, fromInt, sqrt
```

A.2 Relational and logical

```
(==), (!=)           :: Eq t => t -> t -> Bool
(<), (<=), (>), (>)  :: Ord t => t -> t -> Bool
(&&), (//)           :: Bool -> Bool -> Bool
not                  :: Bool -> Bool
```

A.3 List processing (from the course book)

```
(:)         :: a -> [a] -> [a]        1 : [2,3] = [1,2,3]
(++)        :: [a] -> [a] -> [a]      [2,4] ++ [3,5] = [2,4,3,5]
(!!)        :: [a] -> Int -> a        (!!) 2 (7:4:9:[]) = 9
concat      :: [[a]] -> [a]           concat [[1],[2,3],[],[4]] = [1,2,3,4]
length      :: [a] -> Int             length [0,-1,1,0] = 4
head, last  :: [a] -> a               head [1.4, 2.5, 3.6] = 1.4
                                      last [1.4, 2.5, 3.6] = 3.6
tail, init  :: [a] -> [a]             tail (7:8:9:[]) = [8,9]
                                      init [1,2,3] = [1,2]
reverse     :: [a] -> [a]             reverse [1,2,3] = 3:2:1:[]
replicate   :: Int -> a -> [a]        replicate 3 'a' = "aaa"
take, drop  :: Int -> [a] -> [a]      take 2 [1,2,3] = [1,2]
                                      drop 2 [1,2,3] = [3]
zip         :: [a] -> [b] -> [(a,b)]  zip [1,2] [3,4] = [(1,3),(2,4)]
unzip       :: [(a,b)] -> ([a],[b])   unzip [(1,3),(2,4)] = ([1,2],[3,4])
and, or     :: [Bool] -> Bool         and [True,True,False] = False
                                      or [True,True,False] = True
```

A.4 General higher-order functions, operators, etc

```
(.) :: (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
curry :: ((a,b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> ((a,b) -> c)
fst :: (a,b) -> a
snd :: (a,b) -> b
```

B List of predefined Prolog functions and operators

B.1 Mathematical operators

Parentheses and common arithmetic operators like `+`, `-`, `*`, and `/`.

B.2 List processing functions (with implementations)

`length(L,N)`	returns the length of `L` as the integer `N`
	`length([],0).`
	`length([H|T],N) :- length(T,N1), N is 1 + N1.`

`member(X,L)`	checks if `X` is a member of `L`
	`member(X,[X|_]).`
	`member(X,[_|Rest]):- member(X,Rest).`

`conc(L1,L2,L)`	concatenates `L1` and `L2` yielding `L` ("if")
	`conc([],L,L).`
	`conc([X|L1],L2,[X|L3]):- conc(L1,L2,L3).`

`del(X,L1,L)`	deletes `X` from `L1` yielding `L`
	`del(X,[X|L],L).`
	`del(X,[A|L],[A|L1]):- del(X,L,L1).`

`insert(X,L1,L)`	inserts `X` into `L1` yielding `L` `insert(X,List,BL):- del(X,BL,List).`

B.3 Procedures to collect all solutions

`findall(Template,Goal,Result)`	finds and always returns solutions as a list
`bagof(Template,Goal,Result)`	finds and returns all solutions as a list, and fails if there are no solutions
`setof(Template,Goal,Result)`	finds and returns *unique* solutions as a list, and fails if there are no solutions

B.4 Relational and logic operators

`<`, `>`, `>=`, `=<`	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.

B.5 Other operators

`!`	cut
`\+`	negation
`->`	conditional ("if")
`;`	"or" between subgoals
`,`	"and" between subgoals

Errata (D7012E, 2018-08-22)

No errors found so far.

Existing suggested solutions (D7012E, 2018-08-22)

```
%% Suggested solution
%% Declarative languages D7012E 2018-08-28
%% /Håkan Jonsson

-- 180822: 1a
fa x = length x : x

fb [] ys = ys
fb (x:xs) ys = x : fb xs ys

fc (x:y:z) = x < y && fc (y:z)
fc _ = True

fd f xs = [f x | x <- xs]

-- *Main> :type fa
-- fa :: [Int] -> [Int]
-- *Main> :type fb
-- fb :: [a] -> [a] -> [a]
-- *Main> :type fc
-- fc :: Ord a => [a] -> Bool
-- *Main> :type fd
-- fd :: (t -> a) -> [t] -> [a]

-- 180822: 1b

(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)

-- 180822: 2a
data T a = Leaf a | Node (T a) a (T a)

-- 180822: 2b
transform (Leaf x) f = Leaf (f x)
transform (Node lt x rt) f = Node (transform lt f) (f x) (transform rt f)

g n = if n<3 then n else g (n-1) + g (n-2) + g (n-3)

m :: [[Int]]
m = -- to do test runs (note: not the example in the exam)
  [
   [8,8,8,8,8,8,8,8,8,8],
   [8,0,8,0,0,0,0,0,0,8],
   [8,0,8,0,8,8,0,8,8,8],
   [8,0,8,0,8,8,0,0,0,8],
   [8,0,0,0,0,0,0,8,0,8],
   [8,0,8,8,8,0,8,0,0,8],
   [8,0,8,0,8,0,8,0,8,8],
   [8,0,8,0,8,0,0,0,0,8],
   [8,8,8,8,8,8,8,1,8,8]
  ]


-- 180822: 3a
```

```
valid m (x,y) = -1 < x && -1 < y && x<length m && y< (length(head m))

-- 180822: 3b

a :: [[Int]] -> (Int,Int) ->  Int
a m (x,y) =  (m !! x) !! y

-- 180822: 3c

escapable :: [[Int]] -> (Int,Int) -> Bool
escapable m p = escapable2 m p []

escapable2 :: [[Int]] -> (Int,Int) -> [(Int,Int)] -> Bool
escapable2 m (sx,sy) visited
 | elem (sx,sy) visited = False
 | start == 1 = True
 | start == 0  =  valid m (sx,sy) &&
                  (
                    escapable2 m (sx+1,sy) newV ||
                    escapable2 m (sx-1,sy) newV ||
                    escapable2 m (sx, sy+1) newV ||
                    escapable2 m (sx,sy-1) newV
                  )
 | otherwise   = False
  where
   elem2 x [] = False
   elem2 x (a:as) = x==a || elem2 x as
   start = a m (sx,sy)
   newV = (sx,sy):visited

-- 180822: 4

calc :: IO ()
calc = do
         putStr "Enter number of positive integers to add: "
         line <- getLine
         sum <- calc' (read line) 0
         putStr ("Total sum: " ++ (show sum) ++ "\n")

calc' :: Int -> Int -> IO Int
calc' 0 sum = return sum
calc' n sum = do
                putStr "Enter an integer: "
                next <- getLine
                calc' (n-1) (sum + read next)
```

```
%% Suggested solution
%% Declarative languages D7012E 2018-08-28
%% /Håkan Jonsson

%% 180822: 4
```

```
% (a)

factorial(0,1) :- !.
factorial(N,R) :-
    N3 is N-1,
    factorial(N3,R3),
    R is R3*N.

% (b)

factorials(N,L) :-
    N >= 0,
    factorial(N,FN),
    N2 is N-1,
    factorials(N2,List),
    L2 = [FN | List],
    sort(0, @=<, L2, L), !.
factorials(_,[]) :- !.

%% 180828:6

parent(tom,nils).
parent(nils,maria).
parent(maria,karin).
parent(tom,per).
parent(per,anna).
parent(catrin,nils).

% (a)

siblings(X,Y) :- parent(Z,X), parent(Z,Y).

% (b)

cousins(X,Y) :- parent(Z1,X), parent(Z2,Y), siblings(Z1,Z2).


%% 180828:7

% (a)

freq(_,[],0) :- !.
freq(X,[X|T],Num) :- freq(X,T,N), Num is N+1,!. % ! prevents backtracking into next case
freq(X,[_|T],Num) :- freq(X,T,Num),!.

% (b)

freqList(L,FL) :- setof([S,D],(member(S,L),freq(S,L,D)),FL).
```