LULEÅ UNIVERSITY OF TECHNOLOGY

Final exam in **Declarative languages**
Number of problems: 8
Teacher: Håkan Jonsson, 491000, 073-8201700
The result will be available: 2019-06-17.

Course code	D7012E
Date	2019-05-31
Total time	4 tim

Apart from general writing material, you may use: A dictionary. Motivate and explain your solutions.

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.

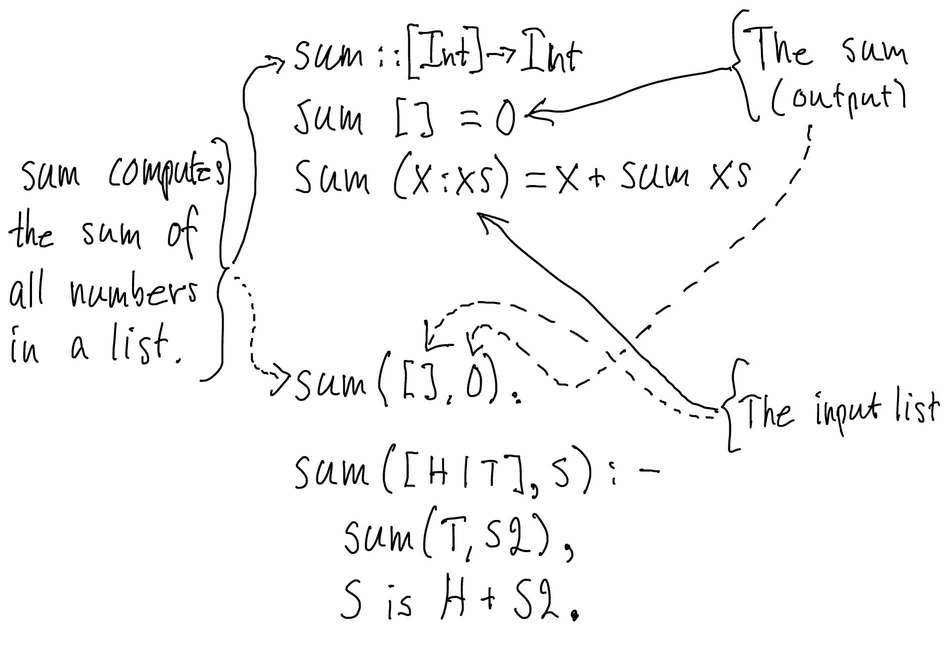

Figure 1: Example showing how to explain code.

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

(a) Use nothing else pre-defined than `foldr` to write a function `totalSum :: [Int] -> Int` that returns the sum of a list containing integers. (2p)

(b) What are the types of the functions `f1`, `f2`, and `f3` below? (3p)

```
a) f1 x y = y x    b) f2 = map (.)    c) f3 f x y = f (f x) y
```

2 Transpose [To be solved using Haskell]

In this problem we use nonempty lists of nonempty sublists (of equal lengths) to represent matrices. Each sublist makes up a row while the n^{th} element of all sublists make up the n^{th} column. The matrix

$$M = \begin{pmatrix} 1 & 7 & 5 & \cdots & 11 \\ 2 & 3 & 10 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 6 & 12 & 4 & \cdots & 8 \end{pmatrix}$$

is, for instance, represented by the list $[[1,7,5,\ldots,11],[2,3,10,\ldots,0],\ldots,[6,12,4,\ldots,8]]$. To *transpose* a matrix means to "flip" the matrix over its diagonal; that is, the row and column indices of the matrix are switched to form another matrix. The transpose M^T of M is, for instance,

$$\begin{pmatrix} 1 & 2 & \cdots & 6 \\ 7 & 3 & \cdots & 12 \\ 5 & 10 & \cdots & 4 \\ \vdots & \vdots & \ddots & \vdots \\ 11 & 0 & \cdots & 8 \end{pmatrix},$$

which is represented by $[[1,2,\ldots,6],[7,3,\ldots,12],[5,10,\ldots,4],\ldots,[11,0,\ldots,8]]$. Note how the top-most row of M has become the left-most column in M^T.

Define a function `tr :: [[a]] -> [[a]]` that computes the transpose of a matrix. (5p)

```
*Main> tr [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
[[1,5,9],[2,6,10],[3,7,11],[4,8,12]]
*Main>
```

3 Triangle IO – Monadic programming [To be solved using Haskell]

Write a Haskell program `triangle :: Int -> IO ()` that prints a triangle on the screen as shown in the examples. (5p)

```
*Main> triangle 4         *Main> triangle 3         *Main> triangle 5
****                      ***                       *****
***                       **                        ****
**                        *                         ***
*                         *Main>                    **
*Main>                                              *
                                                    *Main>
```

4 Hamming numbers [To be solved using Haskell]

A *Hamming number* is an integer of the form $2^i 3^j 5^k$, where i, j, and k are all nonnegative integers. For $i = j = k = 0$ we get the smallest Hamming number: 1. Given one Hamming number, we get another by multiplying it with either 2, 3, or 5.

Define the infinite list `ham` of all Hamming numbers listed in ascending order and without duplicates. Neither list comprehensions nor list expressions for infinite lists (like `[1..]` or `[3,6..]`) are allowed. To merge, you may (without writing it) use a Haskell function `mergeUnq :: [Int] -> [Int] -> [Int]` that works like the Prolog predicate in Problem 5. (5p)

Hint: Suppose `ham` has already been computed. The list of Hamming numbers in ascending order is then 1 followed by the merge (in ascending order without duplicates) of

- the numbers in `ham` times 2 (in ascending order without duplicates),
- the numbers in `ham` times 3 (in ascending order without duplicates), and
- the numbers in `ham` times 5 (in ascending order without duplicates).

```
*Main> take 20 ham
[1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36]
*Main>
```

5 Merging [To be solved using Prolog]

Define a predicate `mergeUnq(L1,L2,L)` that merges two lists `L1` and `L2` with elements in ascending order into a single list `L` without duplicates. The elements must occur in ascending order in the resulting list. (5p)

```
?-  mergeUnq([2,4,6,8],[1,2,5,6,9],L).
L = [1, 2, 2, 4, 5, 6, 6, 8, 9].
?-
```

6 Swapping [To be solved using Prolog]

(a) Write a predicate `swap(t(A,B),t(C,D))` that is true if, and only if, `A` equals `D` and `B` equals `C`. (2p)

(b) Use `swap` to write a predicate `swapAll(L1,L2)` that swaps all arity 2 structures with functor `t` in a list `L2` and returns them as `L2`. (3p)

7 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.
p :- b.
```

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

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

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

8 Domino sequences [To be solved using Prolog]

In this problem we will study a simple variant of *Domino*, a game where players take turn placing flat rectangular tiles. Each tile has two halves. Each half is marked with a number of dots. Tiles are placed "half to half" and must form a sequence $T_1, T_2, \ldots, T_n$ such that, for any consecutive tiles T_i and T_{i+1}, the halves placed towards each other have the same number of dots. An exmple is shown in Fig. 2.

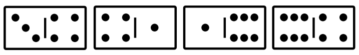

Figure 2: A sequence of domino tiles.

To get dominos into Prolog, we use the structure `t(N,M)` to represent a tile with `N` dots on one half and `M` dots on the other. Then, `t(3,4)`, `t(4,1)`, `t(1,6)`, and `t(6,4)` denote the tiles in Fig. 2.

Note that `t(N,M)` is the same (rotated) tile as `t(M,N)` and that to form a sequence, it might be necessary to rotate tiles. Also, each tile can (of course) only be used once in forming a sequence.

Write a predicate `path(S,E,List)` that given a start tile `S`, an end tile `E`, and a list of tiles `List`, determines if `List` contains tiles that can be used to form a sequence that starts with `S` and ends with `E`. You may use `swap` from problem 6 even if you have not written it. (6p)

Example: If `List = [t(4,3),t(1,2),t(6,4),t(1,4),t(2,2),t(1,6)]`, it is possible to form the sequence in Fig. 2. To do that, two tiles (`t(4,3)` and `t(1,4)`) must be rotated.

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