LULEÅ UNIVERSITY OF TECHNOLOGY

Final exam in **Declarative languages**
Number of problems: 7
Teacher: Håkan Jonsson, 491000, 073-8201700
The result will be available: 2020-09-10.

Course code	D7012E
Date	2020-08-19
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 – lists 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.

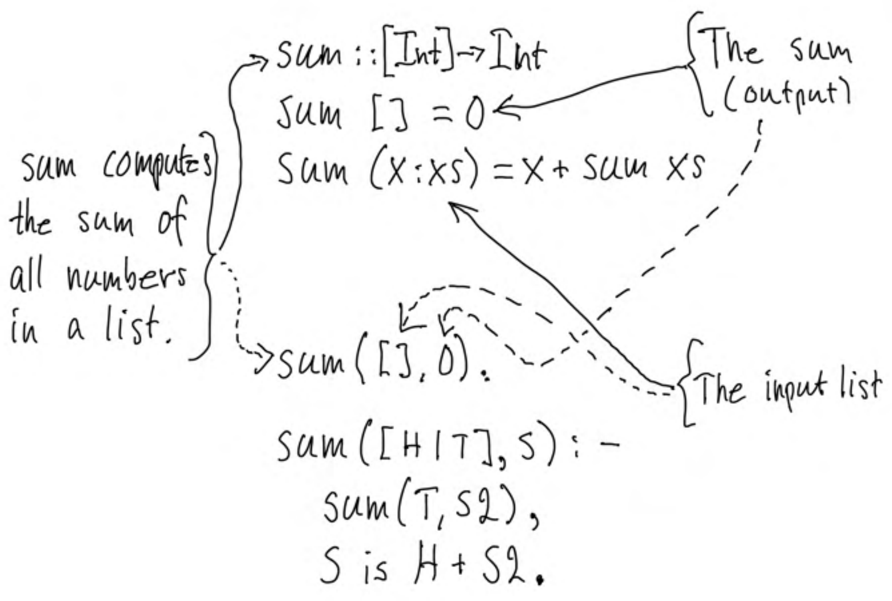

Figure 1: Example showing how to explain code.

1 Types and pattern matching

(a) Here are five expressions. State, for each valid expression, its type. If an expression is invalid, state instead this fact and explain what is wrong. (2,5p)

1) `tail "7"` 2) `[]:[]:[]` 3) `length.map (+1)` 4) `\h -> 'h' 0` 5) `map map`

(b) The *Ackermann function* is defined like this:

```
ack 0 n = n+1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))
```

Now, what does `ack 2 0`, `ack 0 2`, and `ack 1 1` evaluate to? (1,5p)

2 Lists

(a) Write, without using the operator `!!`, a function `middle :: [a] -> a` that returns the middle element of a list. "Middle" is here defined as element number $\lfloor (n+1)/2 \rfloor$, where n is the length of the list. If the list is empty, `error` should be called to stop the evaluation. (2p)

`middle [1.0]` and `middle [1.0,2.0]` return 1.0.
`middle ['1','2','3']` and `middle ['1','2','3','4']` return `'2'`.
`middle [1,2,3,4,5]` and `middle [1,2,3,4,5,6]` return 3.

(b) *Quicksort* sorts a list by first picking an element p (the *pivot*) from the list. Then the remaining elements are partitioned into those smaller than or equal to p and those larger then p. Each partition is then sorted using quicksort. Finally, the sorted sequence of smaller numbers, the pivot, and the sorted sequence of larger numbers are concatenated in that order and returned.

Write a function `qsort :: Ord t => [t] -> [t]` that implements quicksort and picks the middle element (see subproblem (a)) as pivot for non-empty lists. (3p)

(c) Write a function `mkList :: Int -> Int -> [Int]` that, given two integers `f` and `s`, creates the same infinite list as the built-in `[f,s..]` would produce. Do so without using neither `[f,s..]` nor `[f..]`. (3p)

`mkList 1 (-4)` returns `[1,-4,-9,-14,-19,-24,-29,-34,-39,-44,...]`
`mkList 1 1` returns `[1,1,1,1,1,1,1,1,1,1,...]`
`mkList 1 4` returns `[1,4,7,10,13,16,19,22,25,28,...]`

3 Algebraic data types

(a) Declare a recursive algebraic data type `ThreeTree a` for trees in which each leaf is empty and each inne node has three subtrees of type `ThreeTree a` and a pair of type `(a,Int)`. Figure 2 shows an example of such a tree. (2p)

(b) Write a function `topPart :: Int -> ThreeTree a -> [a]` that, given an integer `i` and a tree `t`, returns a list with all first components of nodes that lie at most at depth `i` (the order of the components in the list does not matter). The root node lies at depth 1. The depth then increases by one for each level down the tree. If `i` is zero or `t` is a leaf, the empty list should be returned.

If we apply `topPart 1` on the tree in Figure 2, we get `['X']` while `topPart 2` on the tree gives a list with not only `'X'` but also `'Z'` and `'H'`. (3p)

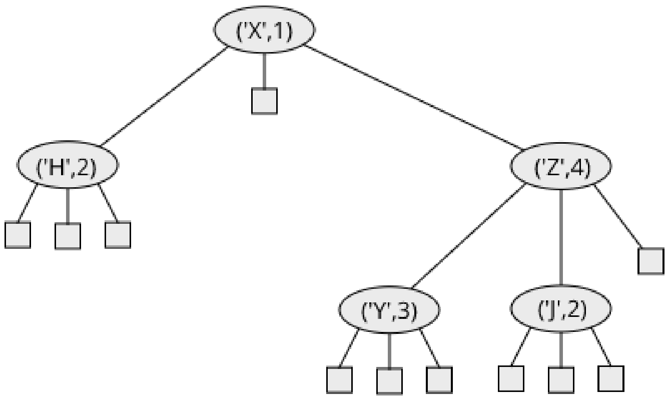

Figure 2: A `ThreeTree Char` containing the pairs `('Z',4)`, `('X',1)`, `('J',2)`, `('Y',3)`, and `('H',2)`.

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

Write a program `calc :: IO ()` that reads integers the user types in, and computes their sum. First, the program asks how many the numbers are and then each number is read one at a time. In the example below, the user has stated that 4 integers should be read and then typed in 3, -2, 4, and 1:

```
*Main> calc
Enter number of integers to add: 4
Enter an integer: 3
Enter an integer: -2
Enter an integer: 4
Enter an integer: 1
Total sum: 6
*Main>
```

Here, you are also allowed to use `putStr :: String -> IO ()`, `putStrLn :: String -> IO ()`, and `getLine :: IO String` (and helper functions). (3p)

5 Solutions [To be solved using Prolog]

One peculiar property of Prolog procedures is that they return answers that satisfies them one answer at a time.

(a) This procedure takes a list `L` of comparable elements and returns `A` and `B` that satisfy it:

```
select1(L,A,B) :-
    member(A,L),
    member(B,L),
    A<B.
```

If we ask the Prolog system

```
?- select1([1,2,3,4],A,B).
```

what, and in which order, would the output be? List the output as it occurs in steps while we enter semicolons (`;`) until the Prolog prompt appears again. (3p)

(b) Suppose we add a cut between lines 2 and 3:

```
select2(L,A,B) :-
    member(A,L),
    !,
    member(B,L),
    A<B.
```

What is now the output if we ask `select2([1,2,3,4],A,B)`? (2p)

(c) What would the output be if the cut in (b) is moved so it ends up between `member(B,L)` and `A<B`? (2p)

6 List generator [To be solved using Prolog]

In the lab part of the course, a list `[1,-2,3,-4,5,-6,...]` was used for test purposes. Write a predicate `generate(N,L)` that returns, in the list `L`, the first `N` elements of the list `[1,-2,3,-4,5,-6,...]`. If `N`$\leq$ 0, `[]` should be returned. Examples: (6p)

`generate(0,L)` returns `L=[]`.
`generate(7,L)` returns `L=[1,-2,3,-4,5,-6,7]`.
`generate(8,L)` returns `L=[1,-2,3,-4,5,-6,7,-8]`.

7 The Partition Problem [To be solved using Prolog]

Write a procedure `partition(L,S1,S2)` that solves the Partition Problem:

> Given a list of numbers `L`, is it possible to partition the numbers into two lists `S1` and `S2` so that the sum of the [numbers in the] lists are equal?

If so, one partition is returned in `S1` and `S2`. If not, the procedure should fail. (7p)

Note that there could be several ways to partition, but only one should be returned. Also, no fast solutions are known to this problem*. Existing methods basically test all possible partitions, which works for rather short lists. Do so in your implementation. Examples:

`partition ([1,2,3,2],S1,S2)` returns `S1=[2,2]` and `S2=[1,3]`.
`partition ([1,2,3,2,1],S1,S2)` and `partition ([],S1,S2)` fails.
`partition ([1,4,2,3,5,4,6,7],S1,S2)` returns `S1 = [1,2,3,4,6]` and `[4,5,7]`.

*It is NP-Complete.

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