LULEÅ UNIVERSITY OF TECHNOLOGY

Final exam in **Declarative languages**
Number of problems: 7
Teacher: Håkan Jonsson, 491000, 073-8201700
Result available: 2 weeks after I have gotten the student solutions

Course code	D7012E
Date	2021-05-31
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 – 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.

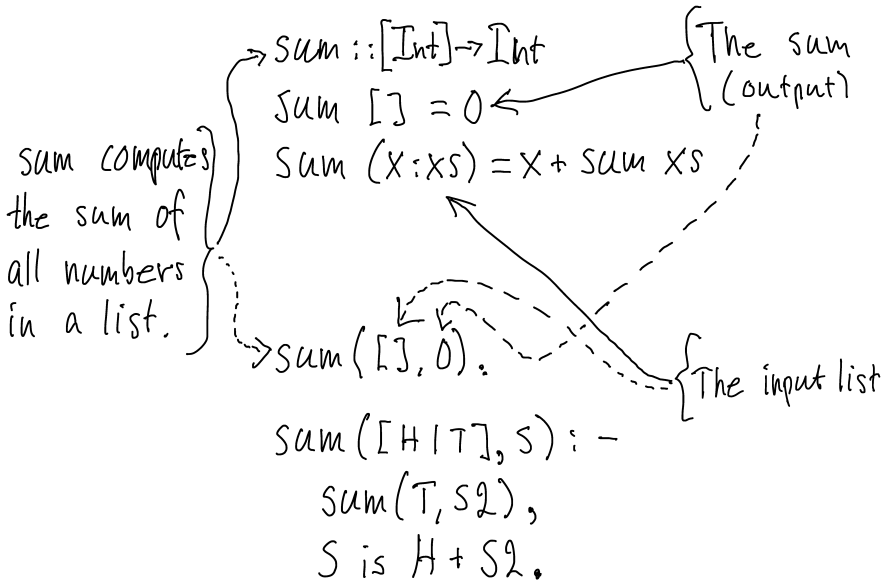


Figure 1: Example showing how to explain code.

---

1 Algebraic data types [To be solved using Haskell]

This is an algebraic data type for binary trees with empty leaves and inner nodes containing integers:

```
data BT = Leaf | Node BT Int BT
```

Write a function `count :: BT -> (Int, Int) -> Int` that, given a tree t and a pair (a, b) of integers, returns the number of integers x in t such that $a \leq x \leq b$. (5p)

2 Types and infinite lists [To be solved using Haskell]

(a) Consider the following numbered expressions and functions:

1) `\g -> g "ABC"` 2) `length.filter (==0)` 3) `map.map`
4) `head "ABC"` 5) `f x = [[x]:[]]:[]` 6) `h a b c = c (b c) a`

State, for each expresion and each function, its type or, if there is something wrong, instead explain what. (3p)

(b) Define the infinite list `stars` containing all strings `"*"`, `"**"`, `"***"`, `"****"`, ... that contain 1 or more asterisks. Do this without using brackets (symbols `[` and `]`). (2p)

3 Reverse Polish Notation [To be solved using Haskell]

Reverse Polish Notation (RPN) is a way to write artithmetic expressions without using paranthesis. For example, an RPN-expression corresponding to the arithmetic expression $2 * (3 + 4) - 5$, where $*$ denotes multiplication, is the sequence `2 3 4 + * 5 -`. Likewise,

- $6 - 9$ corresponds to `6 9 -`
- $1 + (2 - 3)$ corresponds to `1 2 3 - +`
- $(1 + 2) - 3$ corresponds to `1 2 + 3 -` and
- $(1 + (2 - 3) * 4) * 5$ corresponds to `1 2 3 - 4 * + 5 *`

A well-formed RPN is evaluated by processing its symbols in order from left to right, using an initially empty stack and these two rules:

- If the next symbol is a number, push it onto the stack.
- If, instead, the symbol is an operator, pop the stack twice, apply the operator on the numbers popped, and push the result back onto the stack.

When there are no more symbols to process, the stack should contain one item only, and this is then the result. Otherwise, the RPN is not well-formed.

(a) Declare an algebraic data type `Symbol` for representing a symbol that can occur in an RPN with integers, multiplications, additions, and subtractions. (2p)

(b) Write a function `rpn :: [Symbol] -> Int` that takes a list of symbols representing a well-formed RPN, evaluates it, and returns the result.

Hint: Use a list as a stack and let a recursive helper function go through the sumbols while accumulating the answer in a parameter. (3p)

4 Lists [To be solved using Prolog]

(a) Write a procedure `larger(X,List)` that is true if, and only if, there is no element in `List` that is at larger than or equal to `X`. (2p)

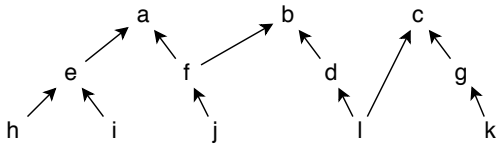
(b) Write a procedure `removeAll(X,List,Result)` that returns, in `Result`, what remains of `List` after all occurances of `X` have been removed. (3p)

5 Same same but different [To be solved using Prolog]

In this problem we assume the Prolog database contans facts (that something *is a* something else) in the form of structures `isA(_,_)` that together form a directed acyclic graph in which the components are nodes and each fact contributes with an arc. If structures

```
isA(h,e).   isA(i,e).   isA(j,f).   isA(k,g).   isA(l,c).   isA(l,d).
isA(e,a).   isA(f,a).   isA(f,b).   isA(g,c).   isA(d,b).
```

are in the database, they form the directed acyclic graph



Write a procedure `same(X,Y)` that is true if, and only if, `X=Y`, or `X` and `Y` have a common ancestor in the graph, or one of them is an ancestor of the other. (5p)

Here, a node $n^\star$ is an ancestor of a node n if it lies on the path *upwards* from n. It is immediate if `isA(`n`,`$n^\star$`)` is true. It is a common ancestor to two nodes if it lies on paths going upwards from both of the two.

In the example, `same(h,j)` and `same(c,k)` are both true, because `a` is a common ancestor of `h` and `j` while `c` is an ancestor of `k`. However, `same(e,d)` is false.

Hint: Going up one step, that is going in the direction of an arrow in the graph, means going to an immediate ancestor.

6 Logical relations [To be solved using Prolog]

Write, for each procedure `P` below, a logical (boolean) expression for when `P` is satisfied in terms of the goals `A`, `B`, and `C`. (5p)

Use $\wedge$ for logical "and", $\vee$ for "or", and $\neg$ for "not". Insert parentheses everywhere, so your answers are crystal clear, except around goals and the final expression. Example: The expression that corresponds to "`A` and not `B` or `C`" should be written $(A \wedge (\neg B)) \vee C$[†].

1)
```
P :- A, B.
P :- C.
```

2)
```
P :- A, B, !.
P :- C.
```

3)
```
P :- \+A, ! \+B
P :- C
```

4)
```
P :- A, !, B.
P :- C.
```

5)
```
P :- !, A, B.
P :- C.
```

6)
```
P :- A, !, B.
P :- \+B, !, C.
P :- \+C.
```

[†]Yes, it is the same as $A \wedge \neg B \vee C$ without parantheses, because of the precedence rules of logical operators, but you should still answer with the paranteses to be clear.

7 Safe passage [To be solved using Prolog]

(This text streches over 3 pages, but the last two pages just contain a large example.)

In this problem, we consider the task of finding a safe path among monsters in a dense forest. The forest is far to dangerous to walk around in freely, so we have to walk on forest paths only. Each such path starts and ends at glades*. While forest paths are all safe, glades might not be. Some monsters unfortunately prefer to live not just in the deep dense forest outside the forest paths but also in the actual glades.

Fortunatelly, we have a map that shows not only the forest paths and glades but also what monsters reside in each glade. A safe path goes along forest paths and only via glades that lacks monsters or glades with monsters that are weaker than us (so we can successfully defeat all monsters, one at a time, during your walk). Figure 2 below shows an example of a map.

Problem: Write a procedure (5p)

```
safe(OurStrength, Start, Dest, Glades, ForestPaths, Monsters, SafePath)
```

that returns, in `SafePath`, a safe path, if one exists, or otherwise fails. You do not need to, but you may, make use of `larger` and `removeAll` from previous problems even if you have not solved them. Use the following representations in your solution:

- A *glade* is an integer.
- A *forest path* is a structure `fp(`g`,`g'`)`, where $g \neq g'$ are glades.
- A *monster* is a structure `m(`g`,`s`)`, where g is the glade in which the monster resides and s, an integer, is the strength of the monster.

The parameters of `safe` can now be described as follows:

- `OurStrength` is the strength we have (an integer).
- `Start` is the glade where we start.
- `Dest` is the destination glade.
- `Glades` is a list of glades.
- `ForestPaths` is a list of forest path structures.
- `Monsters` is a list of monster structures.
- `SafePath`, a safe solution, is a list of glades starting with `Start` and ending with `Dest`. For each pair of consecutive glades g_i and g_{i+1} in `SafePath`, there must be an edge `fp(`g_i`,`g_{i+1}`)` or `fp(`g_{i+1}`,`g_i` )` in `ForestPaths` (forest paths can be walked along in both directions). To be safe, `OurStrength` must be larger than the strength of each individual monster in the glades in `SafePath`.

One more thing. Please make sure your procedure really returns a safe path when one exists and preferably fairly fast. Why? Well, someone inside the forest during the day, and running your code to compute a safe path out, really likes to get out before the sun sets; you see, during the night, as rumour has it, all the monsters roam around *the entire forest including the forest paths and all glades* in hunt for fresh meat...

Hint: Search, extend possible paths, one glade at a time, from the start. If safe paths exist, there exists one that does not enter the same glade twice. This means that each glade visited during a search can be disregarded for the reminder of that search.

* "skogsglänta" in Swedish, an opening in the forest with virtually no trees.

Extended example

Consider the map in Figure 2. It shows 9 glades numbered $1-9$ and a number of forest paths going between the glades. Glade 3 is where we start and the task is to get to glade 7 in a safe manner (and then leave the forest).

The are monsters in 7 of the glades. We immediately realize that a strength of at least 71 is needed to get out safely, because a monster with strength 70 happen to live in glade 7, the glade from which safety is reached(!) In glade 9 there is another monster, and in glade 1 a real beast (much stronger than the other monsters)! In glade 4, there are 2 monsters, one of which is a baby monster (only strength 20).

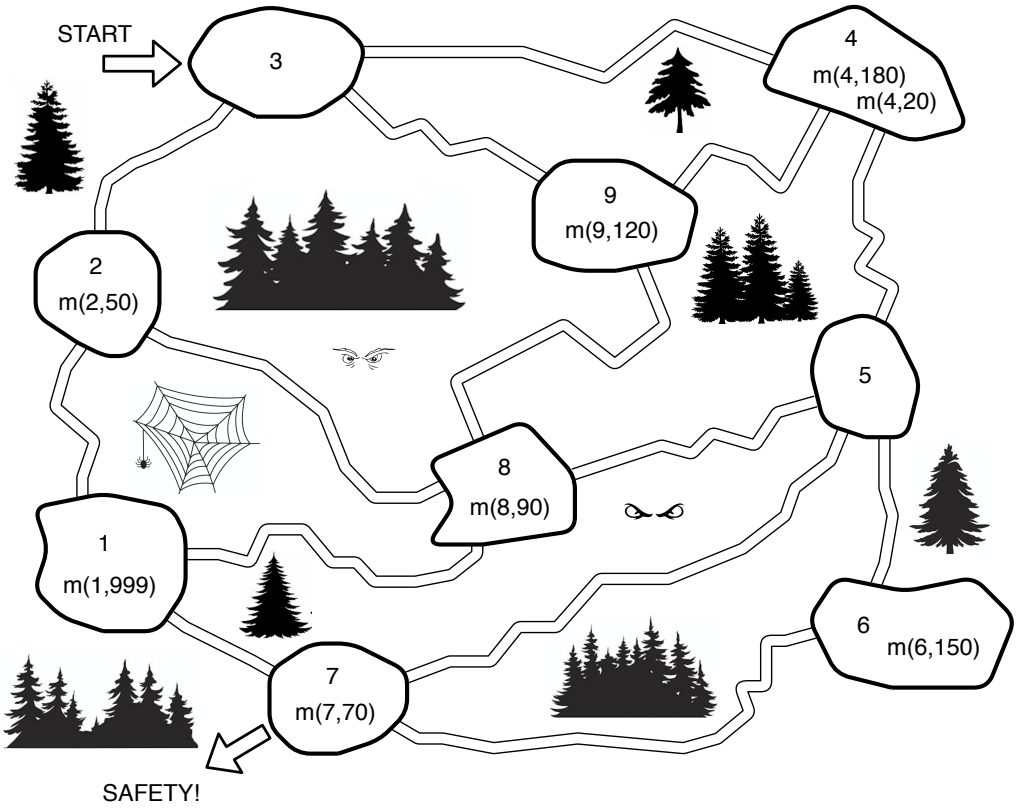


Figure 2: . An example of a map (the evil eyes, spider, and web just illustrate dangers in the forest outside forest paths and glades).

The existence of a safe path depends heavily on how strong we are. In fact, studying the map reveals that strength 71 is not enough; at least 91 is needed, and then the path would be $3 \to 2 \to 8 \to 5 \to 7$. If we had strength 121, we could start by going to glade 9 instead of 2. And if we even had strength 181, we could instead start by going to glade 4, then to glade 5 and further either directly to glade 7 or via glade 6 (slashing yet another monster). And so on.

Let's define a test procedure that defines arguments to `safe` based on Figure 2 and then calls it:

```
test(SafePath) :- OurStrength = 100, Start = 3, Dest = 7,
  Glades = [1,2,3,4,5,6,7,8,9],
  ForestPaths =
    [e(1,2), e(1,8), e(1,7), e(2,3), e(2,8), e(3,9), e(3,4),
     e(4,9), e(4,5), e(5,8), e(5,7), e(5,6), e(6,7), e(8,9)],
  Monsters =
    [m(1,999), m(2,50), m(4,180) ,m(4,20) ,m(6,150), m(7,70), m(8,90),
     m(9,120)],
  safe(OurStrength, Start, Dest, Glades, ForestPaths, Monsters,  SafePath).
```

We then get:

```
?- test(SafePath).
SafePath = [3, 2, 8, 5, 7] ;
false.

?-
```

Indeed this is a safe path. The two monster encountered have strengths 50 and 70, so they are no match for us. However, decreasing our strength to 42, by setting `OurStrength = 42` in `test`, makes us so weak there is no safe path:

```
?- test(SafePath).
false.

?-
```

Note that with strength 42, we would win over the baby monster in glade 4. But we would not be able to deafeat the other monster, so we would anyhow not be able to pass through the glade.
If we instead increase our strength to 121 we get two options:

```
?- test(SafePath).
SafePath = [3, 9, 8, 5, 7] ;
SafePath = [3, 2, 8, 5, 7] ;
false.

?-
```

As discussed, paths through glade 9 are now also safe. If we would even increase our strength to 181, we would get quite a few alternative paths:

```
?- findall(SafePath, test(SafePath), SafePaths); true.
SafePaths = [[3, 9, 4, 5, 7], [3, 9, 4, 5, 6, 7], [3, 9, 8, 5, 7],
             [3, 9, 8, 5, 6, 7], [3, 4, 9, 8, 5, 7], [3, 4, 9, 8, 5, 6, 7],
             [3, 4, 5, 7], [3, 4, 5, 6, 7], [3, 2, 8, 9, 4, 5, 7],
             [3, 2, 8, 9, 4, 5, 6, 7], [3, 2, 8, 5, 7], [3, 2, 8, 5, 6, 7]] ;
true.
?-
```

Note that now there is a safe path among the solutions that passes through all glades, except glade 1 of course (because in there, the beast lurks around).

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