LULEÅ UNIVERSITY OF TECHNOLOGY

"Exam" in **Declarative languages**
Number of problems: 4
Teacher: Håkan Jonsson, 491000, 073-8201700
The result will be available: After the final exam has been given.

Course code	D7012E
Date	2019-04-25
Total time	2 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.

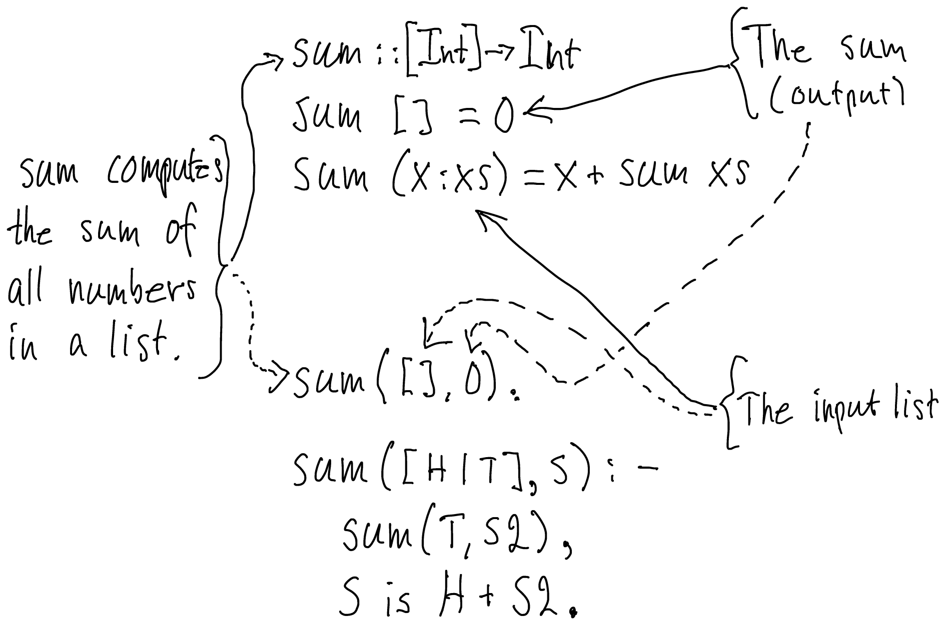

Figure 1: Example showing how to explain code.

1 Goldbach's Conjecture [To be solved using Haskell]

In a letter sent to Leonard Euler on June 7, 1742, the german mathematician Christian Goldbach discusses a general relation between even integers and prime numbers. The relation, called *Goldbach's Conjecture* since it has still not been proved, reads as follows:

Every even integer greater than two is the sum of two prime numbers[1].

Define a function `goldbach :: Int -> Bool` that decides if Goldbach's Conjecture is true for a given number $n > 2$. In solving this problem you may assume there is a function `primes :: Int -> Int -> [Int]` for you to use freely that given two integers a and b returns a list with all primes p such that $a \leq p \leq b$. (3p)

2 Laziness [To be solved using Haskell]

Define the function `periodise :: [a] -> [a]` that given a list $[a_1, a_2, \ldots, a_n]$ returns the infinite list $[a_1, a_2, \ldots, a_n, a_n, \ldots, a_2, a_1, a_1, a_2, \ldots, a_n, a_n, \ldots, a_2, a_1, a_1, a_2, \ldots]$. (3p)

3 Recursion over trees [To be solved using Haskell]

(a) Declare an algebraic type `BranchingTree a` for trees in which leafs contain a value of type `a` and inner nodes (or "branches") contain a list of trees of type `BranchingTree a`. (3p)

(b) Write a function `mapBT :: (a -> b) -> BranchingTree a -> BranchingTree b` that does for branching trees what `map` does for lists. In essence, `mapBT f t` returns a tree with the same branching structure as `t` but in which each value `v` in a leaf has been replaced by what `f` yields applied to `v`. (4p)

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

(a) The function `concatMap` behaves such that `concatMap f` is the same as `concat . map f`. Write this function in terms of `foldr`. (3p)

(b) Suppose the functions `const`, `subst` and `fix` are defined by the equations:

```
a) const x y = x    b) subst f g x = f x (g x)    c) fix f x = f (fix f) x
```

What are their types? You do not need to show how you derived the answers.
Hint for c): Note that `fix` and `f` both "take two arguments", return the same type of result, and take `x` as their second argument. (4p)

[1]For instance, $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5$, $10 = 7 + 3$, and $12 = 5 + 7$. In fact, by the use of computers the relation has been shown correct for integers well above 10^{18} but no one has been able to prove it.

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)

<table>
<tr><td><code>length(L,N)</code></td><td>returns the length of <code>L</code> as the integer <code>N</code>
<code>length([],0).</code>
<code>length([H|T],N) :- length(T,N1), N is 1 + N1.</code></td></tr>
<tr><td><code>member(X,L)</code></td><td>checks if <code>X</code> is a member of <code>L</code>
<code>member(X,[X|_]).</code>
<code>member(X,[_|Rest]):- member(X,Rest).</code></td></tr>
<tr><td><code>conc(L1,L2,L)</code></td><td>concatenates <code>L1</code> and <code>L2</code> yielding <code>L</code> (“if”)
<code>conc([],L,L).</code>
<code>conc([X|L1],L2,[X|L3]):- conc(L1,L2,L3).</code></td></tr>
<tr><td><code>del(X,L1,L)</code></td><td>deletes <code>X</code> from <code>L1</code> yielding <code>L</code>
<code>del(X,[X|L],L).</code>
<code>del(X,[A|L],[A|L1]):- del(X,L,L1).</code></td></tr>
<tr><td><code>insert(X,L1,L)</code></td><td>inserts <code>X</code> into <code>L1</code> yielding <code>L</code>
<code>insert(X,List,BL):- del(X,BL,List).</code></td></tr>
</table>

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