# LULEÅ UNIVERSITY OF TECHNOLOGY

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

| | |
|---|---|
| Course code | D7012E |
| Date | 2017-10-18 |
| Total time | 4 tim |

Apart from general writing material, you may use: A dictionary. Motivate and explain your solutions. Note that half the exam contains predefined functions and operators you may use freely, if not otherwise is stated.

---

## 1 Turning lists into lists [To be solved using Haskell]

(a) Implement the functions `tail` and `init` on page 5. Do so without using any of the predefined operators and functions in Appendix A *except* the list constructor `:`. (3p)

(b) Implement functions `suffixes :: [a] -> [[a]]` and `prefixes :: [a] -> [[a]]` that compute all suffixes/prefixes of a list. Examples:

- `suffixes []` results in an error.
- `suffixes [1,2,3]` returns `[[1,2,3],[2,3],[3]]`.
- `prefixes []` results in an error.
- `prefixes [1,2,3]` returns `[[1,2,3],[1,2],[1]]`.

You may use `tail` and `init` from (a), even if you fail to implement them, but no other predefined function or operator *except* the list constructor `:`. (3p)

## 2 Binary trees [To be solved using Haskell]

(a) Declare an algebraic data type `Tree a` for binary trees with empty inner nodes and leaves containing values of type `a`. (2p)

(b) Write the `Tree Int`-expression for the tree in Fig. 1. (2p)

(c) Write a function `sumTree :: Type a -> int` that computes the sum of all integers stored in a tree. Fig. 1 shows an example in which the sum is $\sum_{i=1}^{8} i = 36$. (2p)

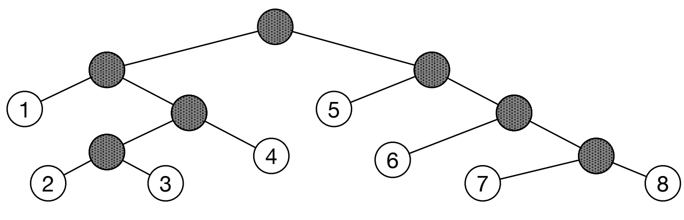


**Figure 1:** A `Tree Int`.

## 3 Types [To be solved using Haskell]

Below you find 8 expressions. State, for each valid expression, its type. If an expression is invalid, state instead this fact and explain what is wrong.

(a) `"x"` (0,5p)

(b) `(1,'x',[True])` where `1` is an integer. (0,5p)

(c) `["x":[]]` (0,5p)

(d) `not.not.isLetter` where `isLetter` has the type `Char -> Bool`. (0,5p)

(e) `map not` (0,5p)

(f) `(+1).(0<)` (0,5p)

(g) `\x -> x + 1` where `1` is an integer. (0,5p)

(h) `[1,'2',"3"]` where `1` is an integer. (0,5p)

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

Implement a user interface for adding numbers in Haskell. The interface should prompt the user for numbers, one at a time. When nothing but return is pressed, numbers entered so far are printed (in order) along with their sum and then the program continues as if no numbers have been entered. If no numbers have been entered, `(none)` should be printed along with the sum 0. (6p)

The example below shows how the output should be. Here, 1, 2, and 3 are first entered. Then no number is entered before we just press return (and that is why `(none)` is printed). Finally, 4 followed by 5 is entered. Note that once started, the program keeps asking for user input "forever".

```
Main> calculator
Enter number: 1
Enter number: 2
Enter number: 3
Enter number:
Numbers entered: 1, 2, 3
Accumulated sum: 6
Sum reset.
Enter number:
Numbers entered: (none)
Accumulated sum: 0
Sum reset.
Enter number: 4
Enter number: 5
Enter number:
Numbers entered: 4, 5
Accumulated sum: 9
Sum reset.
Enter number:
```

Observe that the entered numbers are printed as they are entered by Haskell and need not be printed by other means.

Two helpful functions that can be used freely:

```
putStrLn :: String -> IO ()
getLine :: IO String
```

## 5 Higher-order functions [To be solved using Haskell]

(a) Show how to implement the higher-order functions `map` and `filter` with list comprehensions but without predefined functions or operators. (2p)

(b) Write a *1-case* function (a "one-liner") `evenCubeSum :: Int -> Int` that, given an integer $n > 1$, returns the sum $2^3 + 4^3 + 6^3 + \cdots + m^3$ where $m$ is the largest even integer such that $m \leq n$. Do so using only the higher-order functions `foldr`, `filter`, and `map` together with lambda expressions and list comprehensions. (2p)

*Hint: An integer is even if the remider is 0 when divided by 2. Once a list with the terms have been computed, use* `foldr` *to sum it up.*

## 6 Extracting elements [To be solved using Prolog]

(a) Declare a predicate `select(X,F,R)` that removes once occurance of `X` from the list `F` and returns the result as the list `R`...

1) ...so that it fails if `X` is not part of `F`. (2p)
2) ...so that, instead, `R=F` if `F` does not contain any occurances of `X`. (2p)

Solve these subproblems without predefined predicates.

(b) Declare a predicate `pickTwoDifferent(L,E1,E2)` that returns, as `E1` and `E2`, two different elements of `L`. Applying the predicate on a list that does not contain two different elements should fail (give `false`).

You may use the first version of `select` from subproblem (a), the one that fails, even if you have not solved that problem, but no other predefined predicate. (2p)

## 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, b.
p :- c.
```

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

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

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

## 8 Suitable combinations [To be solved using Prolog]

(a) Declare a predicate `subLists(L,R)` that computes all possible sublists of `L` and returns them in the list `R`:

```
?- subLists([1,2,3],R).
R = [[1, 2, 3], [1, 2], [1, 3], [1], [2, 3], [2], [3], []].
```

Note that each sublist contains elements taken from `L` in order and that `R` contains all of them including the empty list. You can assume the size of `L` is small (at most 20-30 elements long). (4p)

(b) Declare another predicate `keep(L,Max,R)` that goes through a list `L` with integer lists, discards those whose sum is larger than `Max`, and returns the remaining ones in `R`.

```
?- keep([[1, 2, 3], [1, 2], [1, 3], [1], [2, 3], [2], [3], []],3,R).
R = [[1, 2], [1], [2], [3], []].
```

In this example, those summing up to more than 3 are not kept. (4p)

# A List of predefined functions and operators

## A.1 Haskell

### A.1.1 Arithmetics and mathematics in general

| | |
|---|---|
| `+` | The sum of two integers. |
| `*` | The product of two integers. |
| `^` | Raise to the power; `2^3` is 8. |
| `-` | The difference of two integers, when infix: `a-b`; the integer of opposite sign, when prefix: `-a`. |
| `div` | Whole number division; for example `div 14 3` is 4. This can also be written ``14 `div` 3``. |
| `mod` | The remainder from whole number division; for example `mod 14 3` (or ``14 `mod` 3``) is 2. |
| `abs` | The absolute value of an integer; remove the sign. |
| `negate` | The function to change the sign of an integer. |

Note that `` `mod` `` surrounded by **backquotes** is written between its two arguments, is an **infix** version of the function `mod`. Any function can be made infix in this way.

| | | |
|---|---|---|
| `+ - *` | `Float -> Float -> Float` | Add, subtract, multiply. |
| `/` | `Float -> Float -> Float` | Fractional division. |
| `^` | `Float -> Int -> Float` | Exponentiation `x^n` = $x^n$ for a natural number `n`. |
| `**` | `Float -> Float -> Float` | Exponentiation `x**y` = $x^y$. |
| `==,/=,<,>, <=,>=` | `Float -> Float -> Bool` | Equality and ordering operations. |
| `abs` | `Float -> Float` | Absolute value. |
| `acos,asin atan` | `Float -> Float` | The inverse of cosine, sine and tangent. |
| `ceiling floor round` | `Float -> Int` | Convert a fraction to an integer by rounding up, down, or to the closest integer. |
| `cos,sin tan` | `Float -> Float` | Cosine, sine and tangent. |
| `exp` | `Float -> Float` | Powers of e. |
| `fromInt` | `Int -> Float` | Convert an `Int` to a `Float`. |
| `log` | `Float -> Float` | Logarithm to base e. |
| `logBase` | `Float -> Float -> Float` | Logarithm to arbitrary base, provided as first argument. |
| `negate` | `Float -> Float` | Change the sign of a number. |
| `pi` | `Float` | The constant pi. |
| `signum` | `Float -> Float` | `1.0`, `0.0` or `-1.0` according to whether the argument is positive, zero or negative. |
| `sqrt` | `Float -> Float` | (Positive) square root. |

### A.1.2 Relational and logical

Operators `&&`, `||`, and `not` to compute *and*, *or*, and *not*. Also:

| | |
|---|---|
| `>` | greater than (and not equal to) |
| `>=` | greater than or equal to |
| `==` | equal to |
| `/=` | not equal to |
| `<=` | less than or equal to |
| `<` | less than (and not equal to) |

### A.1.3 List processing

| | | |
|---|---|---|
| `:` | `a -> [a] -> [a]` | Add a single element to the front of a list.<br>`3:[2,3]` ⇝ `[3,2,3]` |
| `++` | `[a] -> [a] -> [a]` | Join two lists together.<br>`"Ron"++"aldo"` ⇝ `"Ronaldo"` |
| `!!` | `[a] -> Int -> a` | `xs!!n` returns the nth element of `xs`, starting at the beginning and counting from 0.<br>`[14,7,3]!!1` ⇝ `7` |
| `concat` | `[[a]] -> [a]` | Concatenate a list of lists into a single list.<br>`concat [[2,3],[],[4]]` ⇝ `[2,3,4]` |
| `length` | `[a] -> Int` | The length of the list.<br>`length "word"` ⇝ `4` |
| `head,last` | `[a] -> a` | The first/last element of the list.<br>`head "word"` ⇝ `'w'`<br>`last "word"` ⇝ `'d'` |
| `tail,init` | `[a] -> [a]` | All but the first/last element of the list.<br>`tail "word"` ⇝ `"ord"`<br>`init "word"` ⇝ `"wor"` |
| `replicate` | `Int -> a -> [a]` | Make a list of n copies of the item.<br>`replicate 3 'c'` ⇝ `"ccc"` |
| `take` | `Int -> [a] -> [a]` | Take n elements from the front of a list.<br>`take 3 "Peccary"` ⇝ `"Pec"` |
| `drop` | `Int -> [a] -> [a]` | Drop n elements from the front of a list.<br>`drop 3 "Peccary"` ⇝ `"cary"` |
| `splitAt` | `Int -> [a] -> ([a],[a])` | Split a list at a given position. |
| `reverse` | `[a] -> [a]` | Reverse the order of the elements.<br>`reverse [2,1,3]` ⇝ `[3,1,2]` |
| `zip` | `[a]->[b]->[(a,b)]` | Take a pair of lists into a list of pairs.<br>`zip [1,2] [3,4,5]` ⇝ `[(1,3),(2,4)]` |
| `unzip` | `[(a,b)] -> ([a],[b])` | Take a list of pairs into a pair of lists.<br>`unzip [(1,5),(3,6)]` ⇝ `([1,3],[5,6])` |
| `and` | `[Bool] -> Bool` | The conjunction of a list of Booleans.<br>`and [True,False]` ⇝ `False` |
| `or` | `[Bool] -> Bool` | The disjunction of a list of Booleans.<br>`or [True,False]` ⇝ `True` |
| `sum` | `[Int] -> Int`<br>`[Float] -> Float` | The sum of a numeric list.<br>`sum [2,3,4]` ⇝ `9` |
| `product` | `[Int] -> Int`<br>`[Float] -> Float` | The product of a numeric list.<br>`product [0.1,0.4 .. 1]` ⇝ `0.028` |

### A.1.4 General higher-order functions and operators

```
(.) :: (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
```

## A.2 Prolog

### A.2.1 Mathematical operators

Common arithmetic operators like `+`, `-`, `*`, and `/`.

### A.2.2 List processing

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

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

- ```
  % conc(L1,L2,List) - adds list L2 to L1 and returns L3
  % example: conc([b,c],[a,b,e],X). X = [b,c,a,b,e]
  conc([],L,L).
  conc([X|L1],L2,[X|L3]):-  conc(L1,L2,L3).
  ```

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

- ```
  % insert(X,L,BL) - inserts X to a custom position in list and returns BL
  insert(X,List,BL):-  del(X,BL,List).
  ```

### A.2.3 Procedures to collect all solutions

- `findall`
- `setof`
- `bagof`

### A.2.4 Other operators

| | |
|---|---|
| `!` | cut |
| `<`, `>`, `>=`, `=<` | 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. |
  ```