

Prof. Dr. Jürgen Giesl
René Thiemann

Exercises *Functional Programming* – Sheet 2

Solutions will be collected until Thursday, Oct 31, 2002 in the exercise course.

Exercise 1 (2 points)

Look at the functions `take` and `from`, which take some elements from a list and create a list.

```
take :: Int -> [a] -> [a]
take 0 xs = []
take (n+1) (x:xs) = x:take n xs
```

```
from :: Int -> [Int]
from n = n:from (n+1)
```

Show every reduction step in the (lazy) evaluation of `take 3 (from 8)`.

Exercise 2 (2.5 + 1.5 points)

- (i) Give the most general types for the functions `f`, `g`, `h`, `j`, `k` defined as follows (Here we assume that `5.0` has type `Float`):

```
data DualTree a b = Nil | Leaf a | Node (DualTree a b) b (DualTree a b)
```

```
f Nil = Leaf 5.0
f (Node l "Hallo" r) = f l
```

```
g Nil = Nil
```

```
h (Node l x r) = if True then (Node l x r) else h r
```

```
j = \x y -> (:y)
```

```
k = \ (x:xs) -> \ y -> x++y
```

- (ii) Define the data structure `HetList` with the constructors `Empty` and `Cons` such that the following declarations do not return a type error.

```

hetlength Empty = 0
hetlength (Cons _ x) = 1 + hetlength x

heterogeneous = Cons 1 (Cons True (Cons 'X' Empty))

```

Give the most general types for `hetlength` and `heterogeneous`.

Exercise 3 (1+2+2 points)

In the file *Poly.hs* on the webpage you will find a data structure for polynomials, defined as follows:

```

data Operator = Add | Mul | Sub
data Poly     = C Float | V String | Op Poly Operator Poly

```

So a polynomial is a constant value, a variable an expression built with an operator. In *Poly.hs* one example is given with `testpoly` ($\cong 4 - x + 3x^2$). There is also an evaluation function `eval` for polynomials without variables given by:

```

eval :: Poly -> Float
eval (C x) = x
eval (Op p1 o p2) = let val1 = eval p1
                       val2 = eval p2
                     in interpret o val1 val2

```

- (i) Define the `interpret` function, which should interpret the operator symbols by their corresponding functions. Use the following form:

```

interpret :: Operator -> (Float -> Float -> Float)
interpret Add = \ ...
interpret Mul = ...

```

- (ii) Define a function `derive :: String -> Poly -> Poly` which computes the (symbolic) derivation. For example `derive "x" testpoly` should return a polynomial which corresponds to $-1 + 6x$.
- (iii) Define a function `simplify :: Poly -> Poly` to simplify polynomials in various ways. You can use the evaluator and rules like $0 + x \equiv x$. For example, `simplify (derive "x" testpoly)` should return a result at least as simple as $-1 + 3 * (x + x)$.

Exercise 4 (1+2+2+2 points)

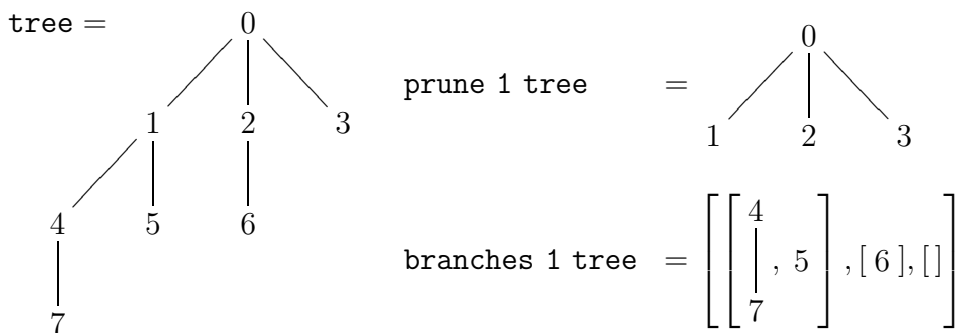
For any list there exist the predefined functions `take`, `drop :: Int -> [a] -> [a]`. For any $n \in \mathbb{N}$ and any list `xs` we have:

```
take n [x1, ..., xm] = [x1, ..., xmin(n,m)]
drop n [x1, ..., xm] = [xn+1, ..., xm]
(take n xs) ++ (drop n xs) == xs
```

Define a data structure for finite trees with arbitrary many children per node and the functions

```
prune    :: Int ->    Tree a    -> Tree a
branches :: Int ->    Tree a    -> [[Tree a]]
joinT    :: Tree a -> [[Tree a]] -> Tree a
```

that works in the same way on trees: `prune n` cuts off all nodes that are deeper than `n`. If `prune n` has k leaves, then `branches n` returns a list $[l_1, \dots, l_k]$ where l_i is the list of children of the i -th leaf of `prune n`. Look at the following example.



`joinT` reconstructs pruned trees, i.e., for all $n \in \mathbb{N}$ and for any tree `tree` we have:

```
joinT (prune n tree) (branches n tree) == tree
```

Hint: You can write `deriving Show` behind your data definition to look at your trees in HUGS (`data Tree a = ... deriving Show`).