

Prof. Dr. Jürgen Giesl
René Thiemann

Exercises *Functional Programming* – Sheet 3

Solutions will be collected until Thursday, Nov 7, 2002 in the exercise course.

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

- (i) Define a higher-order function `filter :: (a->Bool) -> [a] -> [a]`. The expression `filter f xs` should return the list `xs` without those arguments that do not satisfy the predicate `f`. For example, we have `filter even [12,270,3,18,9,12] = [12,270,18,12]`.
- (ii) Look at the following data structure:

```
data BinTree a = Nil | Node (BinTree a) a (BinTree a)
```

Write functions `mapT :: (a->b) -> BinTree a -> BinTree b` and `filterT :: (a -> Bool) -> BinTree a -> BinTree a` that work on trees in the same way that `map` and `filter` work on lists.

You should use higher-order functions in part (iii) and (iv) whenever possible.

- (iii) Write a function `song :: Char -> String -> String` which replaces all vowels by the given char. So, `song 'a' "Drei Chinesen mit dem Kontrabass, die sassen ..."` should be evaluated to `"Draa Chanasan mat dam Kantrabass, daa sassan ..."`
- (iv) Write a function `depthLimit :: Int -> BinTree a -> BinTree (a,Int)` which labels all nodes with their depth and then cuts off the tree at level `n`, where `n` is the first argument of `depthLimit`.

Exercise 2 (2+3+7 points)

We consider a modified version of the Poly data structure.

```
data Operator = Add | Mul deriving Eq  
data Poly     = C Float | V String | Op Poly Operator Poly
```

Our aim is to put `Poly` in the classes `Eq`, `Num`, and `Ord`. Look at the definition of these classes in `Prelude.hs` to see which functions we have to define. (You can skip `abs` and `signum`).

- (i) Define functions `(.+.)`, `(.*.)` `:: Poly -> Poly -> Poly` and give infix-declarations so that we can write `C 5 .* V "x" .+. C 3 .* V "y"` instead of `Op (Op (C 5) Mul (V "x")) Add (Op (C 3) Mul (V "y"))`.
- (ii) Define `Poly` to be an instance of the classes `Eq`, `Num`, and `Ord`. Equality and orderings only have to be defined for constants and variables. Constants should be less than variables and variables should be compared by their name. For example, `C 5 < V "x" < V "y"`.
- (iii) In this part we want to compute equality for polynomials. Now symbolic comparison is not sufficient, because `V "x" .+. C 5` and `C 5 .+. V "x"` should be equal, but are syntactically different.

In order to compare polynomials we therefore have to compute a unique normal form and then do the comparison. You can find some useful functions and a skeleton implementation to compute a normal form in the file `Poly2.hs`. Use the given functions and the higher-order functions `map` and `filter` to fill in the missing parts in `nf` and write the `toSumOfProds`-function. To define `nf` you do not have to define other functions than those given in `Poly2.hs`. Every missing right-hand-side in the `let`-expression can be written in the remaining line.

Then you should modify your equality function so that it can deal with arbitrary polynomials. For example

```
(V "x" + C 1) * (C 2 + V "x") == V "x" * V "x" + C 3 * V "x" + C 2
```

should be evaluated to `True`.

Hint: Whenever types `a`, `a1`, `...`, `an` are members of the classes `Ord`, `Eq` then `[a]`, `(a1, ..., an)` are members of these classes, too. In the prelude, standard definitions are given for equality and ordering on these data types.