

> module Ueb7 where

### Aufgabe 7.1

=====

a) Der Anfangswert der foldx1 – Funktionen ist ein Listenelement, hat also den Basistyp der Liste, der damit auch Wertetyp sein muss.

b)  $((\text{map } f).(\text{filter } p)) \text{ xs} = [f \ x \mid x \leftarrow \text{xs}, p \ x]$

c)

> zweimal f = f . f

Typ: zweimal::(a->a) -> a -> a

Oder gleichbedeutend: zweimal:: (a->a) -> (a -> a)

d)

> iteriere:: Int -> (a->a) -> a -> a

> iteriere n f

> | n==1 = f

> | otherwise = f . (iteriere (n-1) f)

### Aufgabe 7.2

=====

a)

> removeEven :: [Int] -> [Int]

> removeEven [] = []

> removeEven (x:xs)

> | even x = removeEven xs

> | otherwise = (x:xs)

b)

> dropWhile' :: (a -> Bool) -> [a] -> [a]

> dropWhile' p [] = []

> dropWhile' p (x:xs)

> | p x = dropWhile' p xs

> | otherwise = (x:xs)

c)

> removeEven' = dropWhile' even

### Aufgabe 7.3

=====

```
> test f f' n m = foldl andT True [(f x, f' x) | x <- [n..m]]
> where andT x (a,b) = x && (a==b)
```

#### Aufgabe 7.4

=====

- a) `g str = [c | c <- str, c == "x"]`  
`g "xerox..."` liefert Typfehler, da im Filter als Basistyp eine Liste von Strings unterstellt wird. "xx" ist NICHT der Wert.  
 Antwort also:  
 keine der Alternativen.
- b) `(x:xs)` äquivalent zu  
`[x] ++ xs`
- c)
- (1) nein Cebel ist Variable, nicht typkorrekt, wenn anderer Basistyp als String
  - (2) `double :: [Char] -> [String]`
  - (3) falsch, concat auf Liste von Listen definiert, nicht Funktion
  - (4) korrekt
  - (5) korrekt (dasselbe wie (4))

#### Aufgabe 7.5

=====

a)

```
> insertBetween :: [a] -> a -> [a]
> insertBetween xs y = insertBetweenHilf (reverse xs) y []
> where
>   insertBetweenHilf :: [a] -> a -> [a] -> [a]
>   insertBetweenHilf [] y ys = ys
>   insertBetweenHilf (x:xs) y ys = insertBetweenHilf xs y (x:y:ys)
```

Die folgende Version fügt nur ZWISCHEN zwei Listenelementen ein. Da sieht man, wie wichtig eine Spezifikation ist. Glücklicherweise war die Aufgabenstellung eindeutig ;)

```
> insertBetween' :: [a] -> a -> [a]
> insertBetween' xs y = insB xs [] y
> where insB :: [a] -> [a] -> a -> [a]
>       insB [x] res y = res ++ [y,x]
>       insB (x:y:xs) res z = insB xs (x:z:y:res) z
```

... und die einfache Variante:

```
> insertBetween" :: [a] -> a -> [a]
> insertBetween" [] _ = []
> insertBetween" (x:xs) y = x:y:(insertBetween" xs y)
```

b)

```
> duplicate::[a] -> [a]
> duplicate xs = concat [[x,x] | x <- xs]
```

oder schöner:

```
> duplicate' :: [a] -> [a]
> duplicate' xs = [ x | x<-xs, y<-[1..2] ]
```

.. und ohne ZF:

```
> duplicate" :: [a] -> [a]
> duplicate" [] = []
> duplicate" (x:xs) = x:(x:(duplicate' xs))
```

### Aufgabe 7.6

=====

```
> replaceW badWord word
> | badWord == word = rep (length badWord) 'x'
> | otherwise      = word
> rep n x = [x | k <- [1..n]]
> s1 badWord = map (replaceW badWord)
> -- ist korrekt, (replace badWord) ist Funktion mit einem String-Argument. sie
> -- wird auf jedes Listenelement angewandt.

> -- s2 badWord = map . replaceW badWord falsch, heißt: map (replaceW badWord xs)
> --      aber das ist eine Liste, keine Funktion wie nötig.
> -- s3 badWord = replaceW badWord . map : falscher Typ
> --      replaceW badWord . map id hätte Typ String -> String
> --      löst aber nicht das Problem
> --s4 badWord = map BadWord . replaceW wendet map auf eine Zeichenkette
> -- (String) an, das ist gänzlich falsch, 1. ARGument von map muss eine
>-- Funktion sein.
```

b)

```
> multiSubW bWords xs = [repl w | w<- xs ]
> where repl w
>   | elem w bWords = rep (length w) 'x'
>   | otherwise     = w
>   elem y ys
>   | ys == [] = False
>   | y == head ys = True
>   | otherwise = elem y (tail ys)
```

### Aufgabe 7.7

=====

<adresse> ::= <Anrede> "\n" <Namensfeld> "\n" <OrtStrasse>  
<Anrede> ::= "Frau" | "Herr"  
<Namensfeld> ::= <Vorname> <Name> | <Name>  
<OrtStrasse> ::= <StrassePF> "\n" <Ortsangaben> | <Ortsangaben>  
<StrassePF> ::= <Strasse> <Nr> | "Postfach " <Nr>  
<Ortsangabe> ::= <PLZ> " " <Ort>  
  
<Vorname> ::= <Zeichenkette>  
<Name> ::= <Zeichenkette>  
<Strasse> ::= <Zeichenkette>  
<Nr> ::= <Zeichenkette> --z.B. 47a  
<Ort> ::= <Zeichenkette>  
<PLZ> ::= <Ziffer><Ziffer><Ziffer><Ziffer><Ziffer>

Syntaxdiagramm: entsprechend. Für eine richtige Lösung hätte ein Satz der Art "  
<Vorname>, <Name> usw. sind Zeichenketten."

#### Aufgabe 7.8

=====

a)

- Ein Übersetzer wandelt ein Programm in Maschinensprache um, so dass es auf einer Maschine ausgeführt werden kann. Ein Interpretierer ist ein auf einer Maschine ablauffähiges Programm, das ein Programm (manchmal zeilenweise) interpretiert. Die Reduktionen eines Haskellsystems sind keine Maschinenbefehle, es handelt sich also um einen Interpretierer. (Die beiden ersten Sätze sind für eine korrekte Antwort ausreichend.)
- Jeder Folgeschritt eines Algorithmus ist eindeutig festgelegt. Man hat also keine Wahlmöglichkeit, welcher Befehl als nächster auszuführen ist.
- Die Robustheit kennzeichnet die Art, in der ein Algorithmus (besser: ein Programm) mit Fehlern (z.B. Eingabefehlern) umgeht. Je mehr Fehler das programm antizipiert, desto robuster.
- partiell: nicht für jedes Element von A ist f definiert.  
surjektiv: zu jedem b aus B gibt es ein a aus A mit  $f(a) = b$
- ein links- und rechtsassoziativer Operator:  $(x \text{ op } y) \text{ op } z = x \text{ op } (y \text{ op } z)$

b)

$$3/2 = 1,5$$

$$1/6 = 0.1666..$$

Fehler: reverse [1,2,3,4] hat Liste als Wert, nicht mit (take 2 .) verknüpfbar  
[4,3]

[(1,2) (3,4)]

[(1,2),(1,4), (3,2), (3,4), (5,2), (5,4)]