Tutorium_2

November 23, 2020

1 Tutorium 2

Gruppen 2 und 4
Samuel Teuber
propa@teuber.dev
https://teuber.dev/propa

2 Feedback

Danke für das wertvolle Feedback!

- Ton: Sollte jetz besser sein
- Gemeinsames Whiteboard: Schwierig
- "Code Platform"
- Folien vorher

3 Organisatorisches

- Bitte schaut regelmäSSig ins ILIAS Forum (**Benachrichtigungen!**)
- TGI

4 Übungsblatt

- **Probleme?**
- Bitte nutzt **Spaces** und nicht Tabs! -> Probleme beim kompilieren
- Lest die Aufgabenstellung **genau**
- Achtet auf **Randfälle** (root: Was ist z.B. mit 0 und 1 Fällen?)
- Probleme mit dem **Praktomat**?

UMFRAGE

4.1 Aufgabe 1 - 1

Schreiben Sie eine rekursive Funktion pow1, die die Basis b und den Exponent e als Parameter nimmt und b^e naiv über die Gleichungen $b^0 = 1$ und $b^{e+1} = bb^e$ berechnet.

```
In [18]: pow1 b e
             | e < 0 = error "Negativer Exponent"
             | e == 0 = 1
             | otherwise = b * pow1 b (e - 1)

         pow1 2 10
```

```
1024
```

5 Aufgabe 1 - 2

Wesentlich effizienter als die naive Implementierung ist es, bei jedem Rekursionsschritt den Exponenten zu halbieren und die Basis zu quadrieren: $b^{2e} = (b^2)^e$ bzw. $b^{2e+1} = b(b^2)^e$. Schreiben Sie eine weitere Funktion pow2, die die Potenz auf diese Weise effizienter berechnet. Wie viele Aufrufe braucht pow2 im Vergleich zu pow1?

```
In [19]: pow2 b e
             | e < 0 = error "Negativer Exponent"
             | e == 0 = 1
             | e `mod` 2 == 0 = pow2 (b * b) (e `div` 2)
             | otherwise = b * pow2 (b * b) (e `div` 2)

         pow2 2 10
```

```
1024
```

6 Aufgabe 1 - 3

Transformieren Sie nun pow2 in eine endrekursive Version pow3. pow3 soll weiterhin nur zwei Parameter, Basis und Exponent, erwarten, aber mit einer Hilfsfunktion mit Akkumulator die Potenz berechnen. Fügen Sie auch noch eine Überprüfung hinzu, die bei negativen Exponenten mittels error einen Fehler mit aussagekräftiger Fehlermeldung auslöst.

```
In [5]: pow3 b e
            | e < 0 = error "Negativer Exponent"
            | otherwise = pow3Acc 1 b e
            where
                pow3Acc acc b e
                    | e == 0 = acc
                    | e `mod` 2 == 0 = pow3Acc acc (b * b) (e `div` 2)
                    | e `mod` 2 == 1 = pow3Acc (b * acc) (b * b) (e `div` 2)

        pow3 2 10
```

```
1024
```

`pow2` benötigt $O(log(n))$ Aufrufe

7 Aufgabe 1 - 4

Implementieren Sie nun eine Funktion root e r, die die ganzzahlige, e-te Wurzel von r berechnet, d.h. root e r errechnet die grösSte nichtnegative ganze Zahl x, für die $x^e \leq r$ gilt.

```
In [6]: root e r
            | e < 1 = error "Nicht-positiver Wurzelexponent"
            | r < 0 = error "Negativer Radikant"
            | otherwise = searchRoot 0 (r + 1)
            where
                searchRoot low high
                    | low + 1 == high = low
                    | pow3 half e <= r = searchRoot half high
                    | otherwise = searchRoot low half
                    where half = (low + high) `div` 2

        root 10 (pow3 3 10 - 1)
```

```
2
```

Schönes Beispiel für `where`

8 Aufgabe 1 - 5

Schreiben Sie eine Funktion isPrime, die eine natürliche Zahl auf ihre Primzahleigenschaft testet. Für n 2 soll dazu getestet werden, ob n durch eine Zahl zwischen 2 und $\sqrt{n}$ teilbar ist

```
In [8]: isPrime n
            | n < 2 = error "Zu kleine Zahl fuer Primzahltest"
            | otherwise = nHasNoDivisorGreaterThan 2
            where
                upto = root 2 n
                nHasNoDivisorGreaterThan k
                    | k > upto = True
                    | n `mod` k == 0 = False
                    | otherwise = nHasNoDivisorGreaterThan (k + 1)
        isPrime 7
```

```
True
```

9 Aufgabe 2

Erzeugen Sie ein Modul Sort und implementieren Sie darin Insertionsort.

Häufigeres Problem: Nicht stabil

```
In [23]: insert x [] = [x]
         insert x (y : ys)
             | x <= y = x : y : ys
             | otherwise = y : insert x ys

         insertSort [] = []
         insertSort (x : xs) = insert x (insertSort xs)

         insertSort [10,9,8,4,3,65,4,2,1,53,32,231,42]
```

```
Line 6: Use foldr
Found:
insertSort [] = []
insertSort (x : xs) = insert x (insertSort xs)
Why not:
insertSort xs = foldr insert [] xs


[1,2,3,4,4,8,9,10,32,42,53,65,231]
```

10 Aufgabe 3

Erweitern Sie Ihr in der vorigen Aufgabe begonnenes Modul Sort um eine Mergesort-Implementierung!

```
In [24]: merge xs [] = xs
         merge [] ys = ys
         merge (x : xs) (y : ys)
             | x <= y = x : merge xs (y : ys)
             | otherwise = y : merge (x : xs) ys

         mergeSort [] = []
         mergeSort [x] = [x]
         mergeSort xs = merge
             (mergeSort (take (length xs `div` 2) xs))
             (mergeSort (drop (length xs `div` 2) xs))

         mergeSort [10,9,8,4,3,65,4,2,1,53,32,231,42]
```

```
[1,2,3,4,4,8,9,10,32,42,53,65,231]
```

11 B Seite

11.1 Umfrage

```
a -> a
```

```
\x -> x
```

```
   (a -> b) -> a -> b
```

```
\x -> x
```

```
   (a -> b) -> (a -> b)
```

```
\x -> x
```

```
   a -> b -> a
```

```
\x y -> x
```

Alle wohldefinierten, semantisch unterschiedlichen Funktionen des Typs
`a -> a -> a`

```
\x y -> x
```

```
\x y -> y
```

Leere Liste vom Typ `[a]`

```
[]
```

Einelementige Liste vom Typ `[Int]`

```
[42]
```

Leere Liste vom Typ `[[a]]`

```
[]
```

Liste vom Typ `[[a]]`, die nur die *leere Liste* enthält.

```
[[]]
```

Warum gibt es keinen Wert, der für alle Typen `a` eine einelementige Liste von Typ `[a]` darstellt?
Enthaltener Wert? Es gibt kein `null` in Haskell
Gibt es einen Wert, der für alle Typen `a` eine einelementige Liste vom Type `[[a]]` darstellt?
Ja, nämlich:

```
[[]]
```

Geben Sie den Term vom Typ `[[[[[[[[[[a]]]]]]]]]]` mit der kürzesten Schreibweise an

```
[]
```

12 Nächstes Übungsblatt

Macht unbedingt die B-Seite!

13 Currying

13.1 Reprise: Funktionstypen

- Jede Funktion hat einen Typ: Parametertypen + Rückgabetyp
- Typen können spezifisch sein (z.B. `Int`) oder eine Variable `a`
- Schreibweise `Parameter1 -> Parameter2 -> Ausgabe`
- Rechtsassoziativ: `Parameter1 -> Parameter2 -> Ausgabe` entspricht `Parameter1 -> (Parameter2 -> Ausgabe)`

Beispiel: `a -> b -> a`
Genaue Typen egal, nur wichtig, dass Ausgabetyp dem Typ des ersten Parameters entspricht

14 Currying

14.1 Lambda Notation

$g(x,y) = x - \frac{y}{2}$ ist das selbe wie $g = \lambda x, y.x - \frac{y}{2}$
Ermöglicht **anonyme** Funktionen

14.2 Lambda Notation in Haskell

```
In [25]: g = \x y -> x - (y/2)

         g 1 2
```

```
Line 1: Redundant lambda
Found:
g = \ x y -> x - (y / 2)
Why not:
g x y = x - (y / 2)


0.0
```

15 Currying

15.1 In Lambda Notation

Aus $g(x,y) = x - \frac{y}{2}$ wird $g(x) = \lambda y.x - \frac{y}{2}$

15.2 In Haskell

Aus...

```
g x y = x - (y/2)
```

...wird...

```
g x = \ y -> x - (y/2)
```

...und das völlig automatisch!

16 Unterversorgung

```
In [13]: (+) 5 10
```

```
15
```

```
In [14]: -- oder:
         add5 = (+) 5

         add5 10
```

```
15
```

```
In [1]: -- oder:
        ((+) 5) 10
```

```
15
```

```
In [16]: -- oder:

         (+) 5 10
```

```
15
```

17 Funktionen höherer Ordnung vol. 2

Von Kombinatoren zu List Comprehensions

17.1 Funktionen als Parameter

- `map :: (s -> t) -> [s] -> [t]`: Wende eine gegebene Funktion auf alle Elemente der Liste an

```
map ((+) 5) [1,2,3,4,5]
```

- `filter :: (t -> Bool) -> [t] -> [t]`: Filtere eine Liste nach einem gegebenen Kriterium

```
filter ((==) 4) [1,2,4,6,4,6,3,44]
```

Was machen diese Ausdrücke? -> Umfrage

- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`: Kombiniert zwei Listen mithilfe der übergebenen Funktion

```
In [28]: zipWith (==) [1,2] [1,1]

[True,False]
```

17.2 Funktionen als Ausgabe

- `(.) :: (b -> c) -> (a -> b) -> a -> c`: Funktionskomposition

```
add10 = ((+) 5) . ((+) 5)
```

```
In [21]: add10 = (+) 5 . ( 5 + )
         add10 10

20
```

17.3 Kombinatoren

- `sum :: [Int] -> Int`
- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

Von rechts nach links:

```
foldr op i [] = i
foldr op i (x:xs) = op x (foldr op i xs)
```

Von links nach rechts:

```
foldl op i []  = i
foldl op i (x:xs) = foldl op (op i x) xs
```

```
In [3]: :t foldl
        :t foldr
```

```
foldl :: forall (t :: * -> *) b a. Foldable t => (b -> a -> b) -> b -> t a -> b


foldr :: forall (t :: * -> *) a b. Foldable t => (a -> b -> b) -> b -> t a -> b
```

```
In [22]: sentenceLength :: [String] -> Int
         sentenceLength = foldr (\l n -> length l + n) 0
         sentenceLength ["How long is this sequence", "of totally unrelated", "sentences?"]
```

```
55
```

```
In [1]: -- Übung 1: Listenkonkatenation mit foldr (2min)
        app xs ys = foldr (:) ys xs
        app [1,2,3,4,5] [6,7,8,9,10]
```

```
[1,2,3,4,5,6,7,8,9,10]
```

```
In [2]: -- Übung 2: Listenumkehrung mit foldl (3min)
        -- Tipp: Nutzt cons (:) und flip :: (a -> b -> c) -> b -> a -> c
        rev = foldl (flip (:)) []
        rev [1,2,3,4,5]
```

```
[5,4,3,2,1]
```

```
In [1]: -- Übung 3: Addiere Listen Elementweise (2min)
        addLists = zipWith (+)
        addLists [1,2,3,4,5,6,7] [1..20]
```

```
[2,4,6,8,10,12,14]
```

17.4 List Comprehensions

```
[e|q1, ..., qn]
```

`q1,...,qn` sind **Prädikate** oder **Generatoren** der Form `p <- list`

Aufgabe Schreibt einen haskell Ausdruck, der die Menge $\{(x+1)^2 \mid x \in \{0 \ldots 10\} \wedge x \texttt{ mod } 2 = 0\}$ als Liste ausgibt

```
In [4]: -- Übung 4 (2min)
        mySet = [ (x+1)^2 | x <- [0..10], x `mod` 2 == 0, y<-[0..5] ]
        mySet
```

```
[1,1,1,1,1,1,9,9,9,9,9,9,25,25,25,25,25,25,49,49,49,49,49,49,81,81,81,81,81,81,121,121,121,121
```

18 Namensbindung

Festlegung der Bedeutung und des Geltungsbereichs von Variablen
Definitionen:

```
f x = x*x
pi = 3.14159
```

Bindung von x **im Rumpf von** f
Globale Bindung von f und pi
In der Definition von g ist Variable x gebunden, Variable y ist frei

```
g x = y + x*x
```

19 Namensbindung

Innere Bindungen verdecken auSSere:

```
f = (\x -> ((\x -> x*x) 3)+x)
```

Was kommt bei f 1 raus?

```
In [35]: f = (\x -> ((\x -> x*x) 3)+x)
         f 1

Line 1: Redundant lambda
Found:
f = (\ x -> ((\ x -> x * x) 3) + x)
Why not:
f x = ((\ x -> x * x) 3) + xLine 1: Redundant bracket
Found:
(\ x -> ((\ x -> x * x) 3) + x)
Why not:
\ x -> ((\ x -> x * x) 3) + xLine 1: Redundant bracket
Found:
((\ x -> x * x) 3) + x
Why not:
(\ x -> x * x) 3 + x


10
```

20 Lokale Namensbindung

- **where**

```
f x = g (x+1)
    where g x = x
```

- **let**

```
f' x = let g x = x in g (x+1)
```

```
In [36]: f x = g (x+1)
             where g x = x
         f' x = let g x = x in g (x+1)
         f 1
         f' 1
```

```
2


2
```

21 Lazy Evaluation

Daniel P. Friedman: > Cons should never evaluate its arguments

22 Lazy Evaluation

Idee: Werte Ausdrücke nur aus, wenn benötigt

```
choose n m x y z
    | n==1 = x
    | m==1 = y
    | otherwise = z
```

- Auswertung `n`: Immer
- Auswertung `x`: Wenn `n==1` wahr
- Auswertung `m`: Wenn `n==1` falsch
- Auswertung `y`: Wenn `n==1` falsch und `m==1` wahr
- Auswertung `z`: Wenn `n==1` falsch und `m==1` falsch

->Umfrage

22.1 Streams: Unendliche Listen

```
In [39]: powerTwos = 2 : map (*2) powerTwos
         take 10 powerTwos
```

```
[2,4,8,16,32,64,128,256,512,1024]
```

22.1.1 Unendliche Listen mit `iterate`

`iterate :: (a -> a) -> a -> [a]`
Mit Anfangswert und `next` Funktion: Erzeugung einer unendlichen Liste

```
In [40]: dividableBy5 = iterate (+5) 0
         take 10 dividableBy5
```

```
[0,5,10,15,20,25,30,35,40,45]
```

```
In [5]: -- Übung 5: Baut mit iterate eine unendliche Liste der Form [[0],[1,0],[2,1,0],[3,2,1,
        myList = iterate f [0]
            where f list@(x:xs) = (x+1) : list
        take 10 myList
```

```
[[0],[1,0],[2,1,0],[3,2,1,0],[4,3,2,1,0],[5,4,3,2,1,0],[6,5,4,3,2,1,0],[7,6,5,4,3,2,1,0],[8,7,
```

23 Weitere Übungen

24 Fibonacci

Definieren Sie eine unendliche Liste `fibs :: [Integer]` aller Fibonacci-Zahlen

```
In [2]: fibs = 0:1:zipWith (+) fibs (tail fibs)
        take 15 fibs
```

```
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377]
```

24.0.1 Tipps:

- `zipWith`
- `fibs = 0 : 1 : -- ????`

24.1 Primzahltest vol. 2

`isPrime` wie in Übungsblatt 1, aber jetzt mit Listenfunktionen

Hinweis

Relevante Funktionen sind: - `null`: Ist die Liste leer? - `filter cond l`: Filtert l nach `cond`

```
In [7]: isprime n
                | n < 2 = error "error"
                | otherwise = null (filter (\x -> n `mod` x == 0) [2..root 2 n])

        isprime 8
```

```
Line 3: Use any
Found:
null (filter (\ x -> n `mod` x == 0) [2 .. root 2 n])
Why not:
not (any (\ x -> n `mod` x == 0) [2 .. root 2 n])


False
```

24.1.1 Kleiner Hinweis vor der nächten Aufgabe:

```
cycle :: [a] -> [a]
cycle [] = errorEmptyList "cycle"
cycle xs = xs' where xs' = xs ++ xs'
```

24.2 Vigenère-Chiffre

```
vigenere :: [Int] -> String -> String
vigenere key input
```

Ann.: `key` ist k Zeichen lang
Eingabe `input` an jeder Stelle i = n*k+j um `key !! j` verschieben
Tipp: `ord::Char->Int` gibt für `Char` die ASCII Zahl zurück, `chr::Int->Char` ist die Inverse

```
In [8]: import Data.Char

        translate :: Int -> Char -> Char
        translate j c = chr (a + (ord c - a + j) `mod` 26)
            where a = ord 'A'

        vigenere key input = zipWith translate (cycle key) input

        vigenere [0,1,2,3,4,5] "ABCDEF"
```

```
Line 7: Eta reduce
Found:
vigenere key input = zipWith translate (cycle key) input
Why not:
vigenere key = zipWith translate (cycle key)


"ACEGIK"


In [ ]:
```