Fragen

$$
\begin{aligned}
f\ [\,]\ &=\ [\,] \\
f\ (x : y : ys)\ &=\ x * y\ :\ f\ ys \\
f\ (x : xs)\ &=\ f\ (x : x : xs)
\end{aligned}
$$

Wozu evaluiert $f\ [1, 2, 3]$ (Abkürzung für $f\ (1 : 2 : 3 : [\,])$)?

Wozu evaluiert $[f\ [\,],\ f[\,]]$?

Weiteres Beispiel:

$$f\ [\,]\ y\ =\ [\,]$$
$$f\ (x : xs)\ y\ =\ x\ :\ f\ xs\ y$$
$$g\ [x]\ =\ x\ :\ g\ [x]$$

Wozu evaluiert $f\ [5]\ (g\ [3])$?

Evaluierung von links nach rechts

$$f\ [\,]\ x\ =\ [\,]$$
$$g\ =\ [\,]$$
$$h\ =\ 1\ :\ h$$

Wozu evaluiert der Ausdruck $f\ g\ h$?

- h läßt sich nicht evaluieren. Es ergäbe sich die unendliche Liste $[1, 1, 1, ...]$.
- $f\ g\ h$ läßt sich mit keiner Definition matchen.
- Daher wird erst einmal g einmal evaluiert.
- Daraus ergibt sich der Ausdruck $f\ [\,]\ h$, welcher jetzt zur Definition von f paßt.
- Jetzt erhalten wir die rechte Seite $[\,]$ als Ergebnis.
- Es ist wichtig, daß Hakell von links nach rechts evaluiert.

Funktionales Morsedekodieren

Zum Vergleich mit Java dekodieren wir denselben Morsecode nun mit Haskell.

Dies ist eine Aufgabe, die sich gut mit einem funktionalem Programm lösen läßt.

Wir gehen ähnlich vor:

```
main = do
  contents ← readFile "audio.txt"
  (print . decode2 . decode1 . read) contents
```

Wir dekodieren in zwei Stufen:

```
decode1 = (cummulate 0) . threshold . (average 200) . absolute
decode2 = demorse . (crack "") . (foldl (++) "") . (map symbolize)
```

Absolute

Wir berechnen die Absolutwerte einer Liste. Am einfachsten können wir dies mit *map* erledigen und einer Funktion, die den Absolutwert eines *Double* berechnet.

```
absolute :: [Double]→[Double]
absolute = map (\x→if x > 0 then x else -x)
```

Ohne Funktionen höherer Ordnung würden wir alternativ schreiben:

```
absolute [] = []
absolute (x : xs) = absx : absolute xs
  where absx = if x > 0 then x else -x
```

Die Folgende Funktion berechnet die Durschnittswerte von allen Unterlisten einer gegebenen Länge.

Zum Beispiel: $average\ 2\ [1,2,3,4,5,6]$ soll das Ergebnis $[1.5,\ 2.5,\ 3.5,\ 4.5,\ 5.5]$ liefern.

```
average :: Double → [Double] → [Double]
average k l = map (\x→x/k) (listdiff (chop (k − 1) ll) (0 : ll))
  where ll = prefixsum l 0
```

Wenn die Fensterlänge k ist, dann berechnen wir zuerst die Präfixsummen p, von diesen eine neue Liste, von der nur die ersten $k-1$ Elemente fehlen und ziehen von ihr elementweise $0 : p$ ab. Am Ende muß noch durch k geteilt werden.

Beispiel: $p = [1,3,6,10,15,21]$

$[3,6,10,15,21]$ minus $[0,1,3,6,10,15,21]$ ergibt $[3,5,7,9,11]$.

Die Hilfsfunktionen *prefixsum*, *chop* und *listdiff* sind einfach:

$$
\begin{aligned}
&\mathit{prefixsum} :: [\mathit{Double}] \to \mathit{Double} \to [\mathit{Double}] \\
&\mathit{prefixsum}\ [\,]\ a\ =\ [\,] \\
&\mathit{prefixsum}\ (x : xs)\ a\ =\ (x + a)\ :\ \mathit{prefixsum}\ xs\ (x + a)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{chop} :: \mathit{Double} \to [\mathit{Double}] \to [\mathit{Double}] \\
&\mathit{chop}\ 0\ l\ =\ l \\
&\mathit{chop}\ k\ (x : xs)\ =\ \mathit{chop}\ (k - 1)\ xs
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{listdiff} :: [\mathit{Double}] \to [\mathit{Double}] \to [\mathit{Double}] \\
&\mathit{listdiff}\ [\,]\ l\ =\ [\,] \\
&\mathit{listdiff}\ (x : xs)\ (y : ys)\ =\ x - y\ :\ (\mathit{listdiff}\ xs\ ys)
\end{aligned}
$$

In *prefixsum* nutzen wir einen Akkumulator, um lineare Laufzeit zu erhalten.

$$
\begin{aligned}
&\mathit{threshold} :: [\mathit{Double}] \rightarrow [\mathit{Int}] \\
&\mathit{threshold}\ [\,] = [\,] \\
&\mathit{threshold}\ (x : xs) = (\textbf{if}\ x > 0.2\ \textbf{then}\ 1\ \textbf{else}\ -1) : (\mathit{threshold}\ xs)
\end{aligned}
$$

Der Kürze wegen übersetzt *threshold* alle Werte in einer Liste zu -1 oder 1, je nachdem ob sie größer als 0.2 sind.

In Java hatten wir einen Clustering-Algorithmus verwendet, um diesen Schwellwert automatisch zu bestimmen. Darauf verzichten wir jetzt.

```
cummulate :: Int→[Int]→[Int]
cummulate 0 (x : xs)  =  cummulate x xs
cummulate n [ ]  =  [n]
cummulate n (x : xs)  =
  if (n > 0) == (x > 0)
  then cummulate (n + x) xs
  else n  :  cummulate 0 (x : xs)
```

Die Funktion *cummulate* berechnet die Länge von Blöcken, die nur aus -1 oder 1 bestehen.

Beispiel:

$$cummulate\,[1, 1, 1, -1, -1, 1, 1, -1, -1, -1, -1] = [3, -2, 2, -4]$$

```
symbolize :: Int→String
symbolize x
        | x <  -20000 = "  "
        | x <  0      = ""
        | x >  2000   = "-"
        | otherwise   = "."
```

In *symbolize* verwenden wir *guarded commands*, welche bisher nicht vorkamen. Mit ihnen lassen sich gut viele verschiedene Fälle behandeln.

Wir übersetzen kurze Signale in einen Punkte und lange in einen Strich. Lange Pausen werden zu einem Leerzeichen übersetzt.

```
crack :: String→String→[String]
crack a [ ] = [a]
crack a (x : xs) =
  if x == ' '
  then a : crack "" xs
  else crack (a++[x]) xs
```

Diese Funktion bricht einen String in eine Liste von Teilstrings auf, wobei die Teilstrings von Leerzeichen getrennt wurden.

Beispiel:

```
crack "" "-- . --- .-." = ["--", ".", "---", ".-."]
```

```
demorse :: [String]→String
demorse = map (\s → zeichen s tab "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
```

demorse übersetzt eine Liste von Morsecodes in ihre jeweiligen Klartextsymbole.

Die Hilfsfunktion *zeichen s l1 l2* findet einen String *s* in der Liste *l1* auf Position *p* und gibt dann das *p*te Zeichen aus *l2* aus.

```
zeichen :: String→[String]→String→Char
zeichen _ [ ] _ = '?'
zeichen z (x : xs) (y : ys) =
  if z == x
  then y
  else zeichen z xs ys
```

```
tab = [".-","-...","-.-.","-..",".","..-.","--.",
       "....","..",".---","-.-",".-..","--","-.",
       "---",".--.","--.-",".-.","...","-","..-",
       "...-",".--","-..-","-.--","--.."]
```

Überblick

Neues Kapitel

Überblick

Logische Programmierung

Eine Weise der logischen Programmierung:

- Definition von Fakten
- Definition von Regeln

Wir definieren eine „Welt“ mittels Fakten und Regeln.

Dann stellen wir dem System Fragen über diese Welt.

Das System überlegt selbst, wie es diese Fragen beantworten kann.

Familienbeziehungen

Einige einfache Fakten und eine Schlußregel:

$parent(peter, petik).$
$parent(peter, kenny).$
$parent(dieter, peter).$
$parent(dieter, martin).$
$parent(martin, martinek).$
$grand(X, Y) :- parent(X, Z), parent(Z, Y).$

Die letzte Regel besagt, daß wenn X Elternteil von Z und Z von Y ist, dann ist X ein Großelternteil von Y.

$:-$ ist ein stylisierter Pfeil nach links. Die linke Seite „folgt“ von der rechten.

Variablen werden groß geschrieben.