# Musterlösung zur 2. Aufgabe der 4. Übung

Da viele von Euch anscheinend noch Probleme mit dem Entfalten haben, gibt es für diese Aufgabe eine Beispiellösung von uns. Als erstes wollen wir uns noch einmal die Entfaltung ansehen, wir hatten die Funktion unfold doch wie folgt definiert

```
unfold :: (b → Bool) → (b → a) → (b → b) → b → [a]
unfold p f g x
  | p x         = []
  | otherwise   = f x : unfold p f g (g x)
```

Um nun eine Funktion h :: b → [a] als Entfaltung darzustellen, müssen wir uns also jeweils überlegen, was p, f, g und x jeweils sind.

Bevor wir zu den Beispielen kommen, vielleicht noch kurz etwas zur Entfaltung im Allgemeinen: Die Entfaltung verwaltet (rekursiv natürlich) einen Zustand (vom Typ b), in jedem Schritt wird anhand des aktuellen Zustands (das ist das x) der Folgezustand (das macht die Funktion g) und eine Ausgabe (mit Hilfe von f) bestimmt, die Ausgaben werden in einer Liste „gesammelt“. Diese ist die Ausgabe der Funktion. Doch nun zu den Beispielen:

(1) *Unendliche Liste der Fibonaccizahlen*

Zunächst wollen wir uns einmal überlegen, wie wir die Fibonaccizahlen überhaupt rekursiv berechnen können, der naive Ansatz ist, einfach die Definition der Fibonaccizahlen[1] in Haskell zu übersetzen, wir erhielten:

```
fibNaiv :: Int → Integer
fibNaiv 0 = 1
fibNaiv 1 = 1
fibNaiv n = fibNaiv (n−1) + fibNaiv (n−2)
```

Soweit, sogut, klappt auch alles, das Problem ist nur, das schon die Berechnung von $f_{30}$ mit dieser Definition unverhältnismäßig lange dauert. Das Problem ist, dass wir fibNaiv bei der Rekursion zu oft aufrufen, zur Berechnung von $f_{20}$ zum Beispiel berechnen wir $f_{19}$ und $f_{18}$, um $f_{19}$ zu berechnen, berechnen wir erneut $f_{18}$ und $f_{17}$, $f_{18}$ also sogar zweimal, etc. Eine bessere Idee, die wir auch gleich für die Entfaltung verwerten können, scheint zu sein, ein Paar von Fibonaccizahlen zu berechnen, das Paar $(f_n, f_{n+1})$ liefert doch sofort das nächste Paar $(f_{n+1}, f_n + f_{n+1})$, das $n$-te Fibonaccipaar berechnet also

```
fibPaar :: Int → (Integer, Integer)
fibPaar 0 = (0,1)
fibPaar n = (k, m+k)
  where
    (m, k) = fibPaar (n−1)
```

D. h. die $n$-te Fibonaccizahl erigbt sich durch die Funktion

```
fib :: Int → Integer
fib = fst . fibPaar
```

[1] Zur Erinnerung: Es ist $f_0 := 1$, $f_1 := 1$ und $f_n := f_{n-1} + f_{n-2}$ für $n \geq 2$.

wir lesen also einfach den ersten Eintrag unseres Tupels aus, so können wir ohne größere Probleme sogar $f_{2000}$ berechnen.

Soweit also zu den Fibonaccizahlen, aber was hat das jetzt alles mit Entfaltung zu tun? Denkt an die Idee der Entfaltung: Wir haben einen Zustand, aus dem wir in jedem Schritt einen Ausgabewert produzieren, diese Werte werden gesammelt, und dann einen Zustand weiterschalten, bis uns unser Prädikat aufzuhören befiehlt. Das können wir hier doch auch machen: Als Zustand dient uns ein Fibonaccipaar, ausgegeben wird immer der erste Eintrag (so wie oben zur Berechnung der $n$-ten Fibonaccizahl), also haben wir unser f, es ist nämlich fst, weiterhin ist die Zustandsüberführung g einfach die Funktion, die uns aus einem Fibonaccipaar das nächste macht, d. h.

```
gFib :: (Integer, Integer) → (Integer, Integer)
gFib (m, k) = (k, m+k)
```

Fehlt noch das Prädikat. Wann wollen wir aufhören, wenn wir eine unendliche Liste wollen? Nie, also

```
pFib :: (Integer, Integer) → Bool
pFib _ = False
```

Als Startwert nehmen wir unser erstes Fibonaccipaar $(0, 1)$, wir erhalten also

```
fibList :: [Integer]
fibList = unfold pFib fst gFib (0, 1)
```

(2) map

Rufen wir uns die Funktion map noch einmal in Erinnerung, wir hatten doch

```
map :: (a → b) → [a] → [b]
map h []     = []
map h (x:xs) = h x : (map h xs)
```

Wir suchen nun also Funktionen fMap, gMap und pMap, so dass gerade

```
map h xs     = unfold pMap fMap gMap xs
```

gilt. Dazu betrachten wir zunächst noch einmal map: Wir wollen aufhören mit der Rekursion, wenn unsere Liste leer ist, also ist die leere Liste der Stop, d. h. doch gerade

```
pMap :: [a] → Bool
pMap [] = True
pMap _ = False
```

Ansonsten wollen wir h auf das erste Element der Liste anwenden und uns das merken, d. h. unsere Ausgabefunkion ist die Einsetzung unseres h in

```
fMap :: (b → a) → [b] → a
fMap h []     = error "Dieser Fall sollte nicht eintreten!"
fMap h (x:_) = h x
```

Weitergearbeitet wird nun mit dem Rest der Liste, d. h. wir können als gMap einfach die Prelude-Funktion tail nehmen, die uns das erste Element einer Liste entfernt. Also wäre damit

```
map' :: (a → b) → [a] → [b]
map' h xs = unfold pMap (fMap h) tail xs
```

Auch pMap und fMap lassen sich direkt in einer Zeile definieren, doch ich wollte hier noch einmal aufzeigen, was man sich überlegen muss, wir hätten auch gleich schreiben können (und dabei die Liste implizit übergeben)

```
map'' :: (a → b) → [a] → [b]
map'' h = unfold ((==0) . length) (h . head) tail
```

(Das wir nicht gleich auf Gleichheit mit [] testen, liegt daran, dass wir nicht Eq b fordern wollen, was sonst nötig wäre).

(3) *Umdrehen einer Liste*

Als erstes wollen wir wieder überlegen, wie wir drehUm ohne unfold definiert hatten, es war doch einfach

```
drehUm :: [a] → [a]
drehUm []     = []
drehUm (x:xs) = drehUm xs ++ [x]
```

Hm, hier ist es schon schwieriger, das Entfaltungsmuster „unterwegs Zeug sammeln“ zu sehen. Dazu müssen wir uns zunächst überlegen, das wir drehUm doch anstelle von „Das erste Element kommt hinten an die umgedrehte Restliste“ auch als „Das letzte Element kommt vorne an die Umgedrehte Liste, wo das letzte Element felht“ formulieren können, was wir mit Hilfe der Prelude–Funktionen last und init als

```
drehUm :: [a] → [a]
drehUm [] = []
drehUm xs = last xs : drehUm (init xs)
```

formulieren können. Hier ist das Entfaltungsmuster klarer zu erkennen: Wir sammeln die letzten Elemente der Listen auf, und zum nächsten Zustand kommen wir, in dem wir das letzte Element der Liste streichen, also ist gerade f die Funktion last und g ist init , als Prädikat wählen wir wieder (vgl. map'') den Vergleich mit der leeren Liste — beachte insbesondere die Bemerkung zu der Ein-Zeilen-Version —, wir schreiben also

```
drehUm' :: [a] → [a]
drehUm' = unfold ((==0) . length) last  init
```

wobei wir die Liste implizit übergeben.

(4) *Ziffern einer Zahl*

Hier tritt das Entfaltungsmuster wieder klarer hervor: Solange, wie unsere Zahl nicht 0 ist, sammeln wir die erste Ziffer auf und machen mit dem Rest der Zahl weiter. Doch wie kommen wir an die erste Ziffer einer Zahl? Das ist für einstellige Zahlen einfach: Wir geben die Zahl zurück. Ist die Zahl nicht einstellig, so teilen wir solange durch 10, bis sie es ist:

```
ersteZiffer :: Integer → Integer
ersteZiffer n
  | 0 ≤ n && n ≤9 = n
  | otherwise     = ersteZiffer (n `div` 10)
```

Wie kommen wir an den „Rest“hinter der ersten Ziffer? Nun, wir können doch einfach unser Konzept für die erste Ziffer umsetzen. Für einstellige Zahlen geben wir Null zurück, ansonsten addieren wir zum Rest der Division durch 10 einfach unsere letzte Ziffer:

```
letzteZiffern :: Integer → Integer
letzteZiffern n
  | 0 ≤ n && n ≤9 = 0
  | otherwise     = 10 ∗ letzteZiffern (n `div` 10) + (n `mod` 10)
```

Damit können wir nun definieren

```
ziffern' :: Integer → [Integer]
ziffern' = unfold (==0) ersteZiffer letzteZiffern
```

und alles wäre gut, oder? Wenn wir diese Funktion in Hugs aufrufen, geschieht folgendes:

```
Main> ziffern' 123
[1, 2, 3]
Main> ziffern' 100
[1]
Main> ziffern' 101
[1,1]
```

Hm, das ist nicht ganz das, was wir wollten, Nullen werden von unserer Funktion verschluckt, wir überlegen uns das mal am Beispiel der 101:

```
ziffern' 101 = unfold (==0) ersteZiffer letzteZiffern 101
             = ersteZiffer 101 : unfold (==0) ersteZiffer letzteZiffern (letzteZiffern 101)
```

Nun ist

```
ersteZiffer 101 = ersteZiffer 10
                = ersteZiffer 1
                = 1
```

und

```
letzteZiffern 101 = 10 ∗ letzteZiffern 1 + 1
                  = 10 ∗ 0 + 1
                  = 1
```

Also erhalten wir oben weiter

```
ziffern' 101 = 1 : unfold (==0) ersteZiffer letzteZiffern 1
             = 1 : 1 : []
             = [1, 1]
```

Das Problem ist, dass führende Nullen beim Streichen der ersten Ziffer entstehen können, die gleich verschluckt werden.

Wir haben nun zwei Möglichkeiten: Entweder wir sammeln die Ziffern von hinten auf und drehen unsere Liste dann um oder wir zählen mit, wie viele Ziffern noch übrig sind. Hier wollen wir die zweite Möglichkeit nutzen: Um die Ziffern von hinten aufzusammeln, brauchen wir Funktionen, die die letzte bzw. die führenden Ziffern bestimmen, das ist aber leicht durch ganzzahlige Division machbar:

```
letzteZiffer  :: Integer → Integer
letzteZiffer  n = n `mod` 10

ersteZiffern  :: Integer → Integer
ersteZiffern  n = n `div` 10
```

Wir bekommen dann durch

```
ziffernR  :: Integer → [Integer]
ziffernR  = unfold (==0) letzteZiffer ersteZiffern
```

eine Liste der Ziffern, nur rückwärts. Nun bleibt nur noch, die Liste umzudrehen, also

```
ziffern  :: Integer → [Integer]
ziffern  = drehUm . ziffernR
```

Wer sich jetzt fragt, ob das Ganze wegen der einfachen Form der Funktionen auch in einer Zeile geht, kann beruhigt werden, das Problem ist, dass (mod 10) leider nicht funktioniert, da wir ja durch 10 teilen wollen und nicht 10 durch etwas. Zum Glück gibt es die Preludefunktion flip :: (a → b → c) → b → a → c, die die Argumente einer zweistelligen Funktion vertauscht, wir könnten also auch schreiben

```
ziffern ' :: Integer → [Integer]
ziffern ' = drehUm . (unfold (==0) (flip mod 10) (flip div 10))
```

(5) *Tupelerzeugung aus Listen*

Wenn wir uns machTupel noch einmal ansehen, so war doch

```
machTupel :: [a] → [b] → [(a,b)]
machTupel []     _      = []
machTupel _      []     = []
machTupel (x:xs) (y:ys) = (x,y) : machTupel xs ys
```

Da wir jetzt ja schon etwas Übung haben, erkennen wir das Entfaltungsmuster, zwischendurch sammeln wir die Anfangswerte der beiden Listen in einem Tupel auf, als Zustand können wir hier das Paar der beiden Listen nehmen, für die Zustandsüberführung bilden wir von beiden den Schwanz, für die Sammelliste von beiden den Kopf, angehalten wird, wenn mindestens eine Liste leer ist:

```
machTupelFast :: ([a], [b]) → [(a, b)]
machTupelFast = unfold eineLeer doppelKopf doppelSchwanz
  where
    eineLeer       (xs, ys) = (length xs == 0) || (length ys == 0)
    doppelKopf     (xs, ys) = (head xs, head ys)
    doppelSchwanz (xs, ys) = (tail xs, tail ys)
```

Das ist machTupel, doch nur fast, die Eingabe ist jetzt ein Tupel von zwei Listen, nicht zwei Listen, zum Glück gibt es die Funktion (,) :: a → b → (a, b) die aus zwei Werten ein Tupel macht, wir können also definieren (man erinnere sich an die Vorlesung, in der wir uns überzeugt haben, dass (.) . (.) die richtige Komposition für zweistellige Funktionen ist):

```
machTupel' :: [a] → [b] → [(a, b)]
machTupel' = ((.) . (.)) machTupelFast (,)
```

Das wars.