Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
29. April 2013

2. Musterlösung zur Vorlesung
Programmierung und Modellierung

A2-1 *Quicksort* Das Quicksort-Verfahren ist ein rekursiver Sortier-Algorithmus:

Schritt 1: Man wähle irgendein Element der sortierenden Liste, z.B. das erste Element.

Schritt 2: Man teilt die Restliste in zwei Teillisten auf: eine Liste, welche alle Elemente enthält, welche kleiner sind als das gewählte Element; und eine, dessen Element alle größer oder gleich sind.

Schritt 3; Sortieren Sie beide Teillisten durch rekursive Verwendung des Quicksort-Verfahrens.

Schritt 4: Fügen Sie die beiden sortierten Teillisten wieder zusammen, wobei Sie das anfangs ausgewählte Element in die Mitte dazwischen stecken.

Implementieren Sie diesen Algorithmus in Haskell, indem Sie folgende Funktionen definieren:

```
quicksort :: [Int] -> [Int]
splitBy   :: Int -> [Int] -> ([Int],[Int])
```

Die Funktion `quicksort` sortiert eine Liste. Die Funktion `splitBy` teilt eine Liste von Zahlen in zwei Teillisten auf, wobei die erste Liste des Ergebnispaares alle im Vergleich zum ersten Argument kleineren Elemente und die zweite Liste alle größeren oder gleichen Elemente enthält.

Beispiele:

```
> splitBy0 6 ([1..10] ++ [12,10..0])
([1,2,3,4,5,4,2,0],[6,7,8,9,10,12,10,8,6])

> quicksort ([1..10] ++ [12,10..0])
[0,1,2,2,3,4,4,5,6,6,7,8,8,9,10,10,12]
```

Verwenden Sie dazu keine Bibliotheksfunktionen außer `(:)`, `(++)`, `(>)`, `(>=)`, `(>)`, `(<=)`.

a) Implementieren Sie zuerst die Funktion `quicksort` gemäß des angegebenen Algorithmus, da Ihnen dies leichter fallen könnte.

   Verwenden Sie dabei bereits die Funktion `splitBy`, ohne diese gleich zu implementieren. Falls Sie Ihr Programm bereits jetzt auf Typfehler prüfen möchten, können Sie folgende temporäre Definition für `splitBy` verwenden:

```
splitBy :: Int -> [Int] -> ([Int],[Int])
splitBy _ _ = error "Code not written yet."
```

LÖSUNGSVORSCHLAG:

```
qsort :: [Int] -> [Int]
qsort   []  = []
qsort (h:t) = (qsort smaller) ++ h : (qsort bigger)
  where
    (smaller,bigger) = splitBy2 h t
```

b) Implementieren Sie nun `splitBy` und testen Sie anschließend beide Funktionen.

Hinweise: Auch wenn diese Aufgabe nichts mit benutzerdefinierten Datenstrukturen zu tun hat, könnten Sie Inspiration in Folie 5-30 finden. Wenn Ihnen dabei jedoch gar nichts einfällt, dann verwenden Sie doch einfach List-Comprehensions. Das ist nicht effizient, funktioniert aber trotzdem.

Zusatzfrage: Wenn Sie es andererseits wirklich effizient möchten, dann implementieren Sie `splitBy` mit endständiger Rekursion unter Verwendung eines Akkumulators für das Ergebnis. Beachten Sie, dass die Reihenfolge der Elemente in den beiden Ergebnislisten ja gar nicht spezifiziert wurde!

LÖSUNGSVORSCHLAG:

```
-- Wir gehen zweimal über die Eingabeliste:
splitBy0 :: Int -> [Int] -> ([Int],[Int])
splitBy0 n xs = ( [x | x<-xs, x<n] , [ x | x<-xs, x>=n] )

-- Wir gehen einmal rekursiv über die Eingabeliste:
splitBy1 :: Int -> [Int] -> ([Int],[Int])
splitBy1 _ []    = ([],[])
splitBy1 p (h:t) | p > h     = (h:smaller,   bigger)
                 | otherwise = (  smaller, h:bigger)
  where
    (smaller,bigger) = splitBy1 p t

-- Wir gehen einmal endrekursiv über die Eingabeliste:
splitBy2 :: Int -> [Int] -> ([Int],[Int])
splitBy2 pivot = splitBy_Aux ([],[])
  where
    splitBy_Aux :: ([Int],[Int]) -> [Int] -> ([Int],[Int])
    splitBy_Aux acc     []    = acc
    splitBy_Aux (sl,bg) (h:t)
      | h < pivot = splitBy_Aux (h:sl,   bg) t
      | otherwise = splitBy_Aux (  sl, h:bg) t
```

c) Löschen Sie die Typsignaturen aus Ihrem Programm heraus. Welche Typsignaturen inferiert GHC für Ihre beiden Funktionen? Was bedeuten diese Typen?

LÖSUNGSVORSCHLAG:

```
> :t quicksort
quicksort :: Ord a => [a] -> [a]
> :t splitBy
splitBy :: Ord a => a -> [a] -> ([a], [a])
```

Beide Funktionen sind generisch und können mit Listen irgendeines Typs der Typklasse `Ord` umgehen. D.h. wenn wir Werte eines Typs mit `(>)` vergleichen können, dann kann Ihr Code auch gleich Listen davon sortieren.

A2-2 ***Maybe*** Ein wichtiger Typ der Standardbibliothek ist `Maybe`:

```
data Maybe a = Nothing | Just a
```

a) In der Standardbibliothek gibt es eine Funktion `last :: [a] -> a`, welche das letzte Element eine Liste zurück liefert. Leider liefert diese Funktion manchmal eine Fehlermeldung zurück. Schreiben Sie eine totale Funktion `safeLast :: [a] -> Maybe a`, welche keine Fehlermeldung mehr produziert.

LÖSUNGSVORSCHLAG:

Die Funktion `last` liefert einen Fehler, wenn Sie auf die leere Liste angewendet wird. Die Funktion `safeLast` liefert in diesem Falle einfach den Wert `Nothing` zurück.

```
safeLast :: [a] -> Maybe a
safeLast [e]   = Just e     -- match nur mit 1er Liste
safeLast (h:t) = safeLast t -- match mit allen nicht-leeren Listen
safeLast []    = Nothing
```

Achtung, die Reihenfolge der Pattern-Match Zweige ist hier wichtig, da der erste Pattern-Match (matched nur ein-elementige Listen) ja vom zweiten Zweig (matched alle nicht-leeren Listen) subsumiert wird.

b) Schreiben Sie eine Funktion `mergeMaybe` welche aus zwei Maybe-Werten einen einzelnen macht. Bei zwei tatsächlichen Werten soll der erste zurückgegeben werden.

LÖSUNGSVORSCHLAG:

```
mergeMaybe :: Maybe a -> Maybe a -> Maybe a
mergeMaybe x@(Just _) _ = x
mergeMaybe _          y = y
```

c) Schreiben Sie eine Funktion `dividedBy :: Int -> [Maybe Int] -> [Maybe Double]` welche jede Zahl n der Eingabeliste durch m/n ersetzt, falls $n \neq 0$. Die Länge der Liste soll erhalten bleiben, d.h. `Nothing` und 0 werden im Ergebnis durch `Nothing` dargestellt. Verwenden Sie zur Division erneut die Funktion `intDiv` aus Aufgabe A1-4c.

```
> dividedBy 100 [Just 1000,Just 50,Nothing,Just 0,Just 2]
[Just 0.1,Just 2.0,Nothing,Nothing,Just 50.0]
```

LÖSUNGSVORSCHLAG:

```
dividedBy :: Int -> [Maybe Int] -> [Maybe Double]
dividedBy m xs = [ divAux x | x <- xs]
  where
    divAux (Just n) | n > 0 = Just (intDiv m n)
    divAux _ = Nothing

intDiv :: Int -> Int -> Double
intDiv z n = (fromIntegral z) / (fromIntegral n)
```

Anstatt List-Comprehension kann man natürlich auch eine rekursive Definition verwenden – letztendlich benutzten List-Comprehensions ja auch wieder (endständige) Rekursion.

A2-3 ***Termination*** Ein Riesenfass ist mit vielen Weißwürsten und Brezen gefüllt. Außerdem steht neben dem Fass ein Backofen, der beliebig viele Brezen erzeugen kann. Der folgende Vorgang soll nun solange ausgeführt werden, bis er sich nicht mehr wiederholen läßt:

1) Entnehmen Sie zwei beliebige Nahrungsmittel aus dem Fass.

2) Falls beide vom gleichen Typ sind, essen Sie sie auf und füllen eine Breze aus dem Backofen in das Fass.

3) Haben Sie eine Breze und eine Weißwurst genommen, dürfen Sie nur die Breze verputzen. Die Weißwurst muss wieder in das Fass zurück.

Beantworten Sie folgende Frage: Terminiert dieser Vorgang? Falls ja, wie viele Schmankerl verbleiben am Ende im Fass? Falls nein, wie entwickelt sich das Verhältnis von Weißwürsten zu Brezn? Begründen Sie Ihre Antwort!

LÖSUNGSVORSCHLAG: Es bezeichne B die Anzahl der Brezen und W die Anzahl der Weißwürste im Fass. Der Zustand des Fasses läßt sich also durch das Paar (B, W) beschreiben. Es gibt drei mögliche Ausgänge eines Schrittes, nämlich:

- Es werden zwei Weißwürste genommen. Dann wird W zu $W - 2$ geändert, und B zu $B + 1$, also der Zustand (B, W) zu $(B + 1, W - 2)$.
- Es werden zwei Brezen genommen. Dann wird (B, W) zu $(B - 1, W)$ geändert.
- Es werden zwei verschiedene Schmankerl genommen. Auch in diesem Fall wird (B, W) zu $(B - 1, W)$ geändert.

Nach jedem Schritt hat sich also die Summe $B + W$ um eins verringert, daher terminiert der Vorgang, wenn $B + W = 1$ ist, da dann kein Schritt mehr durchführbar ist. $B + W$ wäre also eine geeignete Abstiegsfunktion.

Ist W am Anfang ungerade, dann endet der Vorgang mit einer Weißwurst im Fass, andernfalls (W gerade) mit einer Breze.

Hinweis: Für die Bearbeitung der Aufgaben und Hausaufgaben sind bis auf weiteres die Verwendung von Funktionen der Standardbibliothek tabu, abgesehen von den Grundoperationen wie `(:)`, `(++)`, `(>)`, `(>=)`, `(>)`, `(<=)`, `div`, `mod`, etc. Von Ihnen implementierte Funktionen aus vorangegangenen Aufgaben sind ebenfalls zulässig.

Das soll keine Schikane sein: der Sinn der Aufgaben ist ja momentan, Grundprinzipien wie Rekursion, Endrekursion mit Akkumulatoren, usw. zu verstehen und zu erlernen. Dies lernen wir anhand der Standardbibliotheksfunktionen – diese dann direkt als Lösung zu verwenden `myConcat xs = concat xs` bringt natürlich keine tieferen Einsichten und auch keine Punkte.

H2-1 *Aufzählungen* (2 Punkte) (Geben Sie eine `.hs`-Datei als Lösung ab)
Betrachten Sie den Datentyp `Day` auf Vorlesungsfolie 05-15. Schreiben Sie eine Funktion `proceed :: Day -> Int -> Day` welche ermittelt, welche Wochentag in n Tagen es ist:

```
> proceed Mon 9
Wed
```

Die Verwendung der Funktion `next` von der Folie und von `pred` und `succ` sind erlaubt.

LÖSUNGSVORSCHLAG:

```
proceed :: Day -> Int -> Day
proceed d n
  | n > 0, n < 7 = proceed (next d) (pred n)
  | n == 0       = d
  | otherwise    = proceed d ((n+7) `mod` 7)

-- Alternativen:

proceed1 :: Day -> Int -> Day
proceed1 d n
  | n >= 0, n < 7 = toEnum n
  | otherwise     = proceed1 d ((n+7) `mod` 7)

prev :: Day -> Day
prev Mon = Sun
prev d   = pred d

proceed2 :: Day -> Int -> Day
proceed2 d n
  | n == 0        = d
  | n > 0, n <  7 = proceed2 (next d) (pred n)
  | n < 0, n > -7 = proceed2 (prev d) (succ n)
  | otherwise     = proceed2 d (n `mod` 7)
```

H2-2 *Rekursion* (3 Punkte) (Geben Sie eine `.hs`-Datei als Lösung ab)
Schreiben Sie eine Funktion `safeIndex :: [b] -> Integer -> Maybe b` welche das n-te Element einer Liste zurückliefert, falls dies existiert, und `Nothing` sonst.

```
> [1..10] `safeIndex`  3
Just 4
> [1..10] `safeIndex` 10
Nothing
```

LÖSUNGSVORSCHLAG:

```
safeIndex :: [b] -> Integer -> Maybe b
safeIndex (x: _) 0 = Just x
safeIndex (_:xs) n | n > 0 = safeIndex xs (n-1)
                   | otherwise = Nothing
safeIndex _ _      = Nothing
```

H2-3 *Rekursion II* (3 Punkte) (Geben Sie eine `.hs`-Datei als Lösung ab)
Schreiben Sie eine Funktion `seekMaxMin :: [Double] -> (Double,Double)` welche gleichzeitig das kleinste und das größte Element einer Liste von Fließkommazahlen berechnet. Falls die Eingabeliste leer ist, so soll einfach das Paar `(0,0)` zurückgegeben werden.

```
> seekMaxMin [-2.5,0,3,-7.7,3.001,2.7]
(-7.7,3.001)
```

Um die volle Punktzahl zu erreichen, darf die Eingabeliste nur einmal durchgegangen werden.

LÖSUNGSVORSCHLAG:

```
seekMaxMin :: [Double] -> (Double,Double)
seekMaxMin []    = (0,0)
seekMaxMin (h:t) = seekAux (h,h) t
  where
    seekAux :: (Double,Double) -> [Double] -> (Double,Double)
    seekAux acc            []   = acc
    seekAux acc@(sl,bg) (h:t)         -- @-pattern optional
      | h > bg = seekAux (sl,  h) t
      | h < sl = seekAux ( h, bg) t
      | otherwise = seekAux acc t     -- seekAux (sl,bg) t
```

Der Code hat eine gewisse Ähnlichkeit zur Lösung von Aufgabe A2-1

Abgabe: Lösungen zu den Hausaufgaben können bis Dienstag, den 6.05.2013, 11:00 Uhr mit UniworX abgegeben werden.

Aufgrund des Klausurbonus müssen die Hausaufgaben von Ihnen alleine gelöst werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen.