Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
27. Mai 2014

6. Übung zur Vorlesung
Programmierung und Modellierung

Hinweis: Die Übungen am 10.6., 11.6. und 20.6.(!) entfallen. Übungen finden statt am:

Übung	**Dienstag**	**Mittwoch**	**Freitag**
7.	3.6.	4.6.	6.6.
8.	17.6.	18.6.	**13.6.**
9.	24.6.	25.6.	27.6.

A6-1 ***Bäume*** Implementieren Sie die Funktion `eval :: Expr -> Integer` von Folie 8-17 mit Hilfe der Funktion `foldTree :: ((b,a,b) -> b) -> b -> Tree a -> b` von Folie 8-11.
Zur Erinnerung:

```
data Tree a = Empty | Node { label :: a, left,right :: Tree a }
data Label  = Const Integer | Plus | Times
type Expr   = Tree Label -- erste Version von Folie 8-16 (2. Version: Folie 8-20)
```

A6-2 ***Funktionen höherer Ordnung***

a) Implementieren Sie folgende Funktionen:

```
curry3   :: ((a, b, c) -> d) -> a -> b -> c -> d
uncurry3 :: (a -> b -> c -> d) -> (a, b, c) -> d
```

Hinweis: Die Typsignaturen lassen bei der Implementation einer totalen Funktion hier schon keine Wahl mehr zu, d.h. wenn man auf Schummeleien wie `undefined`, `error` oder endlose Rekursion verzichtet.

b) Diskutieren Sie anhand des Typs den Unterschied zwischen folgenden drei Funktionen:

i) `uncurry3 foldr`

ii) `uncurry $ uncurry foldr`

iii) `(uncurry . uncurry) foldr`

c) Implementieren Sie die Funktion `vervielfache :: [(Int,a)] -> [a]` möglichst im punktfreien Stil.
Beispiel: `vervielfache [(2,'a'),(0,'b'),(4,'c'),(1,'d')] == "aaccccd"`

A6-3 ***Induktion mit Listen I*** Wir bezeichnen mit $|l|$ die Länge einer Liste l. Es gilt $|(x : xs)| = 1 + |xs|$ für eine beliebige Liste $l = (x : xs)$.

a) Beweisen Sie mit Induktion über die Länge der Liste, dass für alle ganzen Zahlen $x \in \mathbb{Z}$ und alle Listen von ganzen Zahlen xs die Gleichung $|\texttt{insert x xs}| = 1 + |xs|$ gilt. Die Funktion `insert` ist definiert wie auf Folie 3-24.

```
insert :: Int -> [Int] -> [Int]
insert x    [] = [x]
insert x (y:ys) | x <= y    = x : y : ys
                | otherwise = y : insert x ys
```

Da Haskell eine rein funktionale Sprache ist, dürfen wir die definierenden Gleichung einer Funktion beliebig von links-nach-rechts oder auch von rechts-nach-links einsetzen. Lediglich bei überlappenden Patternmatching und Guards müssen wir entsprechende Seitenbedingungen beachten. So dürfen wir zum Beispiel in unseren Rechnungen den Ausdruck y : `insert` x ys nur dann durch `insert x` (`y : ys`) ersetzen, wenn $x > y$ angenommen werden darf.

b) Beweisen Sie, dass für eine beliebige Liste l von ganzen Zahlen gilt

$$|l| = |\texttt{sort}\ l|$$

Dabei dürfen Sie die Aussage von Aufgabe A6-3 ohne weiteres direkt verwenden. Die Funktion `sort` ist definiert wie auf Folie 3-24.

```
sort :: [Int] -> [Int]
sort    []  = []
sort (x:xs) = insert x (sort xs)
```

Hinweis: Die Verwendung von Funktionen der Standardbibliothek `Prelude` ist nun prinzipiell zur Lösung der Hausübungen erlaubt, so fern nichts anderes angegeben wurde. Natürlich macht es aber keinen Sinn, wenn Sie zur Lösung von H6-1 die Funktionen `reverse` und `length` verwenden.

H6-1 ***Falten*** **(0 Punkte)** (`.hs`-Datei als Lösung abgeben)
Schnell und ohne dabei in die Folien zu schauen: Definieren Sie die Funktionen `reverse` und `length` ohne direkte Rekursion, sondern unter Verwendung von `foldr` oder `foldl`.
Zur Erinnerung:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b
```

H6-2 ***Induktion mit Listen II*** **(4 Punkte)** (Abgabeformat: Text oder PDF)

```
revAcc :: [a] -> [a] -> [a]
revAcc acc   []  = acc
revAcc acc (h:t) = revAcc (h:acc) t

reverse :: [a] -> [a]
reverse []    = []
reverse (h:t) = (reverse t) ++ [h]

(++) :: [a] -> [a] -> [a]
[]     ++ ys =           ys
[x]    ++ ys = x :       ys -- zur Vereinfachung des Beweises
(x:xs) ++ ys = x : xs ++ ys
```

Beweisen Sie mithilfe von Induktion über die Länge der Liste l:

$$\texttt{revAcc}\ acc\ l = (\texttt{reverse}\ l)\ ++\ acc$$

Hinweis: Sie dürfen die Assoziativität von $(++)$ ohne Beweis verwenden, d.h. für beliebige Listen l_1, l_2, l_3 gilt $(l_1 ++ l_2) ++ l_3 = l_1 ++ (l_2 ++ l_3)$. Dies können Sie als Zusatzaufgabe ebenfalls per Induktion beweisen, wenn Sie möchten.

H6-3 ***Parser*** **(4 Punkte)** (`.hs`-Datei als Lösung abgeben)
Vervollständigen Sie den auf Folie 8-23 begonnen Parser für die auf Folie 8-22 vorgestellte Grammatik:

expr	::=	*prod*	|	*prod* + *expr*
prod	::=	*factor*	|	*factor* * *prod*
factor	::=	*const*	|	(*expr*)

Vervollständigen Sie dazu die Definitionen der beiden Funktionen `parseProd` und `parseFactor` in der Dateivorlage, welche Sie unter Material auf der Vorlesungshomepage finden können. Ihre Funktionsdefinitionen dürfen partiell sein, müssen jedoch alle gültigen Listen von Tokens effizient parsen. Sie dürfen alle Funktionen der `Prelude` verwenden und auch eigene Funktionsdefinitionen hinzufügen, wenn Sie möchten.

Beispiel:

```
> read "3 * (8 + 3)+ 5 * 4 + 32" :: Expr
((3*(8+3))+((5*4)+32))
```

Abgabe: Lösungen zu den Hausaufgaben können bis Dienstag, den 03.06.2014, 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.

FAQ:

Q: "Welche der in der Vorlesung behandelten Funktionen sind 'wichtig'?"

A: Wir werden Sie nicht nach der Definition von `foo` auf Folie *x-y* fragen. Wir verwenden die Funktionsnamen `foo`, `bar`, `baz`, etc. zur Demonstration von speziellen Themen oder um die Bedeutung einer Funktion aus didaktischen Gründen nicht sofort zu verraten.

Wir erwarten, dass Sie aus der Standardbibliothek `Prelude` alle behandelten Funktionen für Basisdatentypen und aus grundlegenden Typklassen zumindest lesen können: z.B. `not`, `(&&)`, `(<=)`, `compare`, `(*)`, `div`, `mod`, `snd`, `(:)`, `(++)`, `length`, `show`, `succ`, ...

Die genaue Hierarchie der numerischen Typklassen müssen Sie auch nicht auswendig lernen. Unter `Num` sollten Sie sich jedoch schon etwas grob vorstellen können.

Von den Funktionen höherer Ordnung sollen Sie zumindest `map`, `filter`, `flip`, `(.)` und `($)` lesen könnnen. Da Sie bei der Lösung von Aufgaben nun generell Funktionen der `Prelude` verwenden dürfen, können Sie natürlich Zeit sparen, wenn Sie weitere Funktion daraus kennen, wie man in H6-1 sieht. Wenn die Verwendung einer Funktion wie `foldl` explizit gefordert wäre, dann würden wir auch die Definition von `foldl` angeben. Es kann also helfen, wenn Sie solche Funktionen kennen und schon einmal vorher benutzt haben, auswendig lernen müssen Sie diese aber nicht. Generell ist eigene Erfahrung im Umgang mit Code natürlich sehr sinnvoll.

Hinweis: Diese Beispiele beziehen sich natürlich nur auf den bisher behandelten Stoff und sind nicht erschöpfend! Auch wenn `(||)` oder `(+)` oben nicht aufgeführt wurden, sind diese als grundlegende Operationen auf Basisdatentypen natürlich enthalten!

Q: "Sind Hausübungen mit 0 Punkten total sinnlos?"

A: Manche Hausübungen werden mit 0 Punkten bewertet, da die Antworten leicht per GHCI, den Folien oder mit Google zu ermitteln sind. Da in der Klausur diese Hilfmittel nicht zur Verfügung stehen, sind Aufgaben in diesem Stil sehr wohl für Klausuren geeignet. Natürlich sollen diese Aufgaben Ihnen helfen, den behandelten Stoff zu verstehen – das ist selbstverständlich das Hauptziel aller Übungen!