Prof. Martin Hofmann, PhD
Dr. Steffen Jost

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

6. Musterlösung 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)
```

LÖSUNGSVORSCHLAG:

```
eval :: Expr -> Integer
eval = foldTree calcNode 0
  where calcNode :: (Integer,Label,Integer) -> Integer
        calcNode (_,Const i ,_) = i
        calcNode (l,Plus    ,r) = l + r
        calcNode (l,Times   ,r) = l * r
```

Wie in der Vorlesung besprochen ist nicht jeder Wert des Typs `Expr` ein gültiger Ausdruck. Dies war ja die Motivation für den spezialisierten Baumtypen auf Folie 8-20. Zum Beispiel macht es keinen Sinn, wenn ein Knoten mit mit Label `Const` kein Blatt ist. Dementsprechend ignorieren wir in diesem Fall die gefalteten Werte des linken und rechten Unterbaums. Wer möchte, könnte auch einen Fehler werfen, falls diese Werte nicht 0 sein sollten.

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.

LÖSUNGSVORSCHLAG:

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

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

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

i) `uncurry3 foldr`

ii) `uncurry $ uncurry foldr`

iii) `(uncurry . uncurry) foldr`

LÖSUNGSVORSCHLAG:

b)i und b)ii unterscheiden sich lediglich in der Gruppierung der Argumente: einmal haben wir die Struktur `(a,b,c)` und einmal `((a,b),c)`. In der Mathematik unterscheidet man üblicherweise nicht zwischen diesen beiden Produkten. In einer Programmiersprache kann man eine Unterscheidung aber nur schwerlich vermeiden, schon allein wegen der Unterschiedlichen Kodierung im Speicher des Computers, welche je nach Verwendung notwendig ist.

Die Funktionen b)ii und b)iii sind dagegen tatsächlich identisch. Es ist ja generell egal, ob man zuerst zwei Funktion komponiert und dann auf ein Argument anwendet, oder ob man die eine Funktion zuerst auf das Argument anwendet und die andere Funktion danach auf das Ergebnis.

Achtung: Die Klammern in der Definition b)iii sind wichtig, sonst kommt eine recht verrückte Funktion dabei heraus.

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

LÖSUNGSVORSCHLAG:

Hier können wir die Funktionen `replicate` aus H1-3 und auch `concatMap` aus A5-1d einsetzen:

```
vervielfache1 :: [(Int,a)] -> [a]
vervielfache1 = concatMap $ uncurry replicate

vervielfache2 :: [(Int,a)] -> [a]
vervielfache2 = concat . (map (uncurry replicate))

vervielfache3 :: [(Int,a)] -> [a]
vervielfache3 = (\xs -> concat [replicate n x | (n,x) <- xs])
```

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` $(\texttt{y} : \texttt{ys})$ ersetzen, wenn $x > y$ angenommen werden darf.

LÖSUNGSVORSCHLAG:

Es seien $n \in \mathbb{N}$ und $x \in ZZ$ feste aber beliebige Zahlen und l eine beliebige Liste von ganzen Zahlen mit $|l| = n$. Wir beweisen $|\texttt{insert x xs}| = 1 + |xs|$ mit Induktion über n.

Für den Induktionsanfang haben wir $n = 0$. Die einzige Liste mit Länge 0 ist die leere Liste. Nach der definierenden Gleichung für `insert` gilt direkt:

$$|\texttt{insert}\ x\ []| = |[x]| = 1 = 1 + 0$$

Es sei nun $n > 0$. Damit ist l nicht leer. Bezeichnen wir das erste Element von l mit y und den Rest der Liste mit ys, also $l = y : ys$. Es gilt $|ys| = n - 1$.

Jetzt müssen wir eine Fallunterscheidung durchführen. Falls $x \leq y$ gilt:

$$|\texttt{insert}\ x\ (y : ys)| = |x : y : ys| = 1 + 1 + |ys| = 2 + (n - 1) = 1 + n$$

Im anderen Fall gilt $x > y$ und wir setzen entsprechend ein:

$$|\texttt{insert}\ x\ (y : ys)| = |y : \texttt{insert}\ x\ ys| = 1 + |y : \texttt{insert}\ x\ ys| = 1 + (1 + |ys|) = 2 + (n - 1) = 1 + n$$

In beiden Fällen haben wir das benötigte Ergebnis $1 + n$ erhalten, was den Beweis vollendet.

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)
```

LÖSUNGSVORSCHLAG:

Es seien $n \in \mathbb{N}$ und $x \in ZZ$ feste aber beliebige Zahlen und l eine beliebige Liste von ganzen Zahlen mit $|l| = n$. Wir beweisen $|l| = \texttt{sort l}$ mit Induktion über n.

Der Induktionsanfang $n = 0$ ist einfach, denn die einzige Liste mit Länge 0 ist die leere Liste. Nach Definition gilt direkt $|\texttt{sort}\ []| = |[]| = 0$.

Es sei nun $n > 0$ Damit ist l nicht leer. Bezeichnen wir das erste Element von l mit x und den Rest mit xs, also $l = x : xs$. Es gilt $|xs| = n - 1$. Wir rechnen durch Einsetzen der definierenden Funktionsgleichungen:

$$|\texttt{sort}\ (x : xs)| = |\texttt{insert}\ x\ (\texttt{sort}\ xs)| = 1 + |\texttt{sort}\ xs| = 1 + (n - 1) = n$$

Die erste Gleichung gilt nach dem Lemma aus Aufgabe A6-3. Da wir dieses Lemma bereits bewiesen haben, können wir dieses für beliebige Listen von beliebiger Länge anwenden. Die zweite Gleichung folgt nach Induktionsannahme, welche wir wegen $|xs| = n - 1$ anwenden dürfen.

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
```

LÖSUNGSVORSCHLAG:

```
reverse1 :: [a] -> [a]
reverse1 = foldl (flip (:)) []

reverse2 :: [a] -> [a]
reverse2 = foldr (:) []

length1 :: [a] -> Int
length1 = foldl (curry fst . succ) 0

length2 :: [a] -> Int
length2 = foldl (const . succ) 0

length3 :: [a] -> Int
length3 = foldl (\acc _ -> succ acc) 0
```

Der Einsatz von `foldl` ist `foldr` hier vorzuziehen, da `foldl` endrekursiv ist und wir in jedem Fall die Eingabeliste bis zum letzten Element durcharbeiten müssen.
`curry fst` ist praktisch identisch zu `const`

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.

LÖSUNGSVORSCHLAG:

Wir führen den Beweis mit Induktion über die Länge der Liste l; die Liste acc ist beliebig. Für $n = 0$ ist die Liste l leer. Wir gehen wie folgt vor: Wir formen die linke Seite der geforderten Gleichung um und danach die rechte Seite, so dass bei beiden Umformungen das Gleiche herauskommt. Dies bitte sich im Gegensatz zu einer Kette von Gleichungen an, wenn wir noch nicht genau wissen, bei welchem Term sich beide Seiten "treffen":

$$\begin{aligned} \texttt{revAcc}\ acc\ [\,] &= acc \\ (\texttt{reverse}\ [\,])\ ++\ acc = [\,]{+}{+}\ acc &= acc \end{aligned}$$

Beide Seiten der geforderten Gleichung sind also gleich acc, und damit gleich.

Es sei nun $n > 0$ und damit $l = (h : t)$ nicht leer.

$$\begin{aligned} \texttt{revAcc}\ acc\ (h:t) &= \texttt{revAcc}\ (h:acc)\ t && (1)\\ &= (\texttt{reverse}\ t)\ ++\ (h:acc) && (2)\\ (\texttt{reverse}\ (h:t))\ ++\ acc &= ((\texttt{reverse}\ t)\ ++\ [h])\ ++\ acc && (3)\\ &= (\texttt{reverse}\ t)\ ++\ ([h]\ ++\ acc) && (4)\\ &= (\texttt{reverse}\ t)\ ++\ (h:acc) && (5) \end{aligned}$$

wobei (1) nach der zweiten definierenden Gleichung von `revAcc` folgt; (2) folgt durch die Anwendung der Induktionshypothese, da t eine Liste der Länge $n-1$ ist; (3) gilt wegen der zweiten definierenden Gleichung von `reverse`; Anwendung der Assoziativität von (++) rechtfertigt (4); und (5) gilt nach der ersten definierenden Gleichung von (++). Damit ist der Beweis vollendet, da wir beide Seiten der geforderten Gleichung zu dem gleichen Ausdruck umformen konnten.

Wer möchte, kann natürlich auch eine Serie von Umformungen vornehmen, dazu ist der zweite Teil der obigen Lösung rückwärts durchzuführen. Dies ist erlaubt, da wir in einer funktionalen Sprache ja wirklich Gleichungen betrachten.

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:

$$
\begin{array}{lcll}
\textit{expr} & ::= & \textit{prod} & \mid \textit{prod} + \textit{expr} \\
\textit{prod} & ::= & \textit{factor} & \mid \textit{factor} * \textit{prod} \\
\textit{factor} & ::= & \textit{const} & \mid \texttt{(}\ \textit{expr}\ \texttt{)}
\end{array}
$$

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))
```

LÖSUNGSVORSCHLAG:
Wir verwenden zur Demonstration für `parseProd` Pattern-Guards. Man kann die Funktionen jedoch genauso gut im Stil von `parseExpr` formulieren, egal ob nach dem Stil der Vorlesung oder der Dateivorlage.

```
data Token = CONST Integer | LPAREN | RPAREN | PLUS | TIMES

instance Show Token where
  show (CONST i) = show i
  show LPAREN    = "("
  show RPAREN    = ")"
  show PLUS      = "+"
  show TIMES     = "*"

lexer :: String -> [Token]
lexer     ""  = []
lexer (' ':s) =        lexer s
lexer ('(':s) = LPAREN:lexer s
lexer (')':s) = RPAREN:lexer s
lexer ('+':s) = PLUS  :lexer s
lexer ('*':s) = TIMES :lexer s
lexer      s  = (CONST $ read num) : lexer rest
  where   (num,rest) = span (`elem` '-':['0'..'9']) s

data Expr = Const Integer | Plus  Expr Expr | Times Expr Expr

instance Show Expr where
  show (Const i)     = show i
  show (Plus  e1 e2) = "("++ show e1 ++ "+" ++ show e2 ++")"
  show (Times e1 e2) = "("++ show e1 ++ "*" ++ show e2 ++")"

instance Read Expr where
  readsPrec _ s = let (e,t) = parseExpr $ lexer s in [(e,concatMap show t)]

parseExpr :: [Token] -> (Expr,[Token]) -- using case
parseExpr l = case parseProd l of
               (summand1, PLUS:rest1) -> let (summand2 , rest2) = parseExpr rest1
                                         in  (Plus summand1 summand2, rest2)
               other -> other

parseProd :: [Token] -> (Expr,[Token]) -- using pattern guards
parseProd l
  | (factor1, TIMES:rest1) <- pl =  let (factor2 , rest2) = parseProd rest1
                                    in  (Times factor1 factor2, rest2)
  | otherwise = pl
  where   pl = parseFactor l -- avoids repeated evaluation

parseFactor :: [Token] -> (Expr,[Token])
parseFactor ((CONST i):s) = ((Const i),s)
parseFactor (LPAREN :s) = let (expr,RPAREN:rest) = parseExpr s in (expr,rest)
parseFactor          s  = error $ "Factor expected, but found " ++ show s
```

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!