Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
9. Juni 2015

7. Musterlösung zur Vorlesung

Programmierung und Modellierung

A7-1 ***Lexer*** Implementieren Sie eine Funktion `lexer :: String -> [Token]`, wie auf Folie 07-22 beschrieben.

Dies ist eine leichte Übungen zum Aufwärmen! Einfaches Pattern-Matching und Rekursion reichen zur Lösung aus. Als Vorlage können Sie den Code zur Vorlesung vom 1. Juni verwenden. Dieser Code wurde mit Erscheinen dieses Übungsblattes aktualisiert. Folgende in der Vorlage verfügbaren Hilfsfunktionen könnten dabei nützlich sein:

`isSpace :: Char -> Bool` testet, ob ein Zeichen ein Leerzeichen, Tabulator, etc. ist.

`lexInt :: String -> Maybe (Integer, String)` versucht aus dem Anfang des Strings eine Zahl einzulesen und liefert die Zahl zusammen mit den unverbrauchten Zeichen der Eingabe zurück, falls erfolgreich. *Beispiel:* `lexInt "12x3" = Just (12,"x3")`

LÖSUNGSVORSCHLAG:

Wir müssen einen String, also eine Liste von `Char` in eine Liste des Typs `Token` umwandeln. Da jeweils ein `Char` einem `Token` entspricht, ist das ganz einfach. Ausnahme sind Zahlen, doch hier können wir die vorgegebene Funktion `lexInt` verwenden. Wer mag, kann als Übung auch mal `lexInt` zu Fuss implementieren.

```
lex :: String -> [Token]
lex     ""  = []
lex ('(':s) = LPAREN:lex s
lex (')':s) = RPAREN:lex s
lex ('+':s) = PLUS  :lex s
lex ('*':s) = TIMES :lex s
lex ( c :s) | isSpace c = lex s
lex      s  | Just (i,s2) <- lexInt s = CONST i : lex s2
lex      s  = error $ "Unbekanntes Token: " ++ s
```

Wer mit Pattern-Guards noch Probleme hat, kann die letzen beiden Zeilen auch durch einen Case-Ausdruck erseten:

```
lex      s  = case lexInt s of
                Just (i,s2) -> CONST i : lex s2
                Nothing     -> error "Unbekanntes Token"
```

Wer `isSpace` nicht benutzen möchte, schreibt einfach

```
lex (' ':s) = lex s
```

Das reicht hier auch schon für die Aufgabenstellungen.

A7-2 *Parser* Vervollständigen Sie den auf Folie 07-24 begonnen Parser für die auf der Folie davor vorgestellte Grammatik:

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

Verwenden Sie erneut den Code zur Vorlesung vom 1. Juni als Vorlage. Vervollständigen Sie dort die Definitionen der beiden Funktionen `parseProd` und `parseFactor`. 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:

```
> eval $ read "1 + 2 * 3"
7
> read "3 * (8 + 3)+ 5 * 4 + 32" :: Expr
((3*(8+3))+((5*4)+32))
> eval $ read "3 * (8 + 3)+ 5 * 4 + 32"
85
```

LÖSUNGSVORSCHLAG:
Wir verwenden zur Demonstration für `parseProd` Pattern-Guards. Man kann die Funktionen jedoch genauso gut gleichen Stil von `parseExpr` wie in der Vorlesung implementieren.

```
data Token = CONST Integer | LPAREN | RPAREN | PLUS | TIMES
data Expr  = Const Integer | Plus  Expr Expr | Times Expr Expr

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) <- pfl =  let (factor2 , rest2) = parseProd rest1
                                     in  (Times factor1 factor2, rest2)
  | otherwise = pfl
  where   pfl = 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
```

Hier der komplette Code:

```
module Expr where

import Prelude hiding (lex)
import Data.Char
import Data.Maybe

data Expr = Const Integer
          | Plus  Expr Expr
          | Times Expr Expr
  deriving (Eq)

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 $ lex s in [(e,concatMap show t)]

-- Beispiel:
a2 = Times (Plus (Const 5) (Const 3)) (Const 2)

eval :: Expr -> Integer
eval (Const     n)  = n
eval (Plus  l r)    = eval l + eval r
eval (Times l r)    = eval l * eval r

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   = "*"

-- Beispiel:
s1 = [CONST 3, TIMES, LPAREN, CONST 8, PLUS, CONST 3, RPAREN, PLUS, CONST 5, TIMES, CONST 4]

-- Lexikalische Analyse
lex :: String -> [Token]
lex     ""  = []
lex ('(':s) = LPAREN:lex s
lex (')':s) = RPAREN:lex s
lex ('+':s) = PLUS  :lex s
lex ('*':s) = TIMES :lex s
lex ( c :s) | isSpace c = lex s
lex      s  | Just (i,s2) <- lexInt s = CONST i : lex s2
lex      s  = error $ "Unbekanntes Token: " ++ s

lexInt :: String -> Maybe (Integer, String)
lexInt = listToMaybe . reads

-- Syntaxanalyse
parseExpr   :: [Token] -> (Expr,[Token])
parseProd   :: [Token] -> (Expr,[Token])
parseFactor :: [Token] -> (Expr,[Token])

parseExpr l  = let (summand1,rest1) = parseProd l in
    case rest1 of
      PLUS:rest2 -> let (summand2,rest3) = parseExpr rest2
                    in  (Plus summand1 summand2, rest3)
      _other     -> (summand1,rest1)

parseProd l = let (factor1,rest1) = parseFactor l in
     case rest1 of
        TIMES:rest2 -> let (factor2,rest3) = parseProd rest2
                       in (Times factor1 factor2, rest3)
        _other    -> (factor1,rest1)

parseFactor l = case l of
       CONST n:rest -> (Const n,rest)
       LPAREN:rest ->  let (e,rest2) = parseExpr rest in
                         case rest2 of
                             RPAREN : rest3 -> (e,rest3)
                             _ -> error "Syntaxfehler"
       _other -> error "Syntaxfehler"
```

A7-3 ***Typsignaturen*** Lösen Sie diese Aufgabe mit Papier und Bleistift! Geben Sie zu jeder der folgenden Deklaration eine möglichst allgemeine Signatur an. Begründen Sie Ihre Antwort informell!

Überlegen Sie sich dazu, welche Argumente jeweils auftreten und wie diese verwendet werden, z.B. welche Funktion auf welches Argument angewendet wird. Falls Sie eine der verwendeten Funktion nicht kennen, schlagen Sie diese in den Vorlesungsfolien oder der Dokumentation der Standardbibliothek nach. Verwenden Sie GHCI höchstens im Nachhinein, um Ihre Antwort zu kontrollieren. Numerische Typklassen müssen Sie zur Vereinfachung nicht angeben, verwenden Sie einfach einen konkreten Typ wie `Int` oder `Double`.

a) `gaa xy z vw = if (read z == xy) then minBound else vw`

Lösungsbeispiel: Wir sehen als erstes, dass es sich um eine Funktion mit drei Argumenten handelt, deren Typ wir bestimmen müssen, sowie den Ergebnistyp der Funktion. Nennen wir diesen Typ vorläufig einmal $a \to b \to c \to d$, also `xy` $:: a$, `z` $:: b$ und `vw` $:: c$.

Argument `xy` wird in einem Vergleich verwendet, also muss a in der Typklasse `Eq` sein.

`z` ist ein Argument für die Funktion `read` und damit ist $b =$ `String`. Da das Ergebnis aber mit `xy` verglichen wird, muss es den gleichen Typ haben, und wir wissen also, dass a auch in der Typklasse `Read` sein muss, denn es wurde ja ein Wert dieses Typs geparsed.

`vw` wird nur als Ergebnis verwendet, daraus können wir $c = d$ schließen. Da die beiden Zweige eines Konditionals den gleichen Typ haben müssen und im `then`-Zweig der Wert `minBound` zurückgegeben wird, muss dieser Typ auch noch in der Klasses `Bounded` sein.

Wir haben also `gaa :: (Eq a, Read a, Bounded c) => a -> String -> c -> c`

Hinweis: Namen von Typvariablen und Reihenfolge der Class Constraints ist unbedeutend. `gaa :: (Read y2, Bounded zz, Eq y2) => y2 -> [Char] -> zz -> zz` wäre z.B. eine äquivalente Typsignatur.

b)
```
guu _ (h1:h2:t) = [h2]
guu x _         = show x
```

LÖSUNGSVORSCHLAG:

`guu :: Show a => a -> String -> String`

Die zweite Zeile legt wegen `show :: Show a => a -> String` Rückgabewert auf `String`fest und schränkt den Typ des ersten Argumentes auf die Typklasse `Show` ein. Damit `[h2]` vom Typ `String` ist, muss `h2` vom Typ `Char` sein. `h2` ist aber das zweite Element der Liste, welche wir als zweites Argument erhalten haben, also ist diese eine Liste von Zeichen.

c) `axx u (v,w) = if minBound || w then succ v else u`

LÖSUNGSVORSCHLAG:

`axx :: Enum a => a -> (a, Bool) -> a`

Der boolsche Oder-Operator `(||) :: Bool -> Bool -> Bool` erzwingt, dass `w :: Bool` gilt. `minBound` ist hier uninteressant, da es hier lediglich als erstes Argument für den Oder-Operator dient, also gleich `False` ist. `u` und `v` müssen den gleichen Typ wie das Ergebnis haben, da diese direkt zurückgegeben werden. Die Funktionsanwendung von `succ` auf `v` verändert dessen Typ nicht, erzwingt aber dessen Einschränkung auf die Typklasse `Enum`.

d)
```
foo a b []    e f = show a
foo a b (c:d) e f
  | a==c, read b  = e ++ e
  | a/=c          = show d
  | otherwise     = show e
```

LÖSUNGSVORSCHLAG:

```
foo :: (Eq x, Show x) => x -> String -> [x] -> String -> y -> String
```

Da auf das erste Argument `a` die Funktion `show` angewendet wird, muss es irgendein Typ sein, der in der Typklasse `Show` liegt. Da wir `show a` zurückgegeben, muss der Ergebnistyp der Funktion `foo` schon mal `String` sein.

Das zweite Argument muss wegen der Anwendung von `read` ein String sein. (Das Ergebnis dieses `read`-Aufrufs ist für uns hie unerheblich – wir wissen aber, dass es von dem Typ `Bool` sein muss.)

Das dritte Argument wird mit Listen-Pattern gemacht, es ist also irgendein Listentyp, sagen wir einfach mal `[x]`. Damit ist die Variable `c` vom Typ `x` und die Variable `d` vom Typ `[x]`. Aufgrund des Vergleiches des ersten Argumentes mit der Variable `c` in Guards wissen wir, das beide den gleichen Typ besitzen, welche zusätzlich noch in der Typklasse `Eq` liegen muss. (Zusätzlich zu der Klasse `Show` wie wir bereits für das erste Argument festgestellt hatten.)

Auf das vierte Argument wird die Infix-Funktion `(++)` angewendet. Diese hat den Typ `[a]->[a]->[a]`. Da wir bereits wissen, dass der Ergebnistyp `String`, also `[Char]` ist, so muss auch das vierte Argument vom Typ `String` sein.

Das fünfte Argument wird nirgendwo verwendet. Es kann daher irgendeinen beliebigen Typ haben, der nicht auf eine Typklasse eingeschränkt werden muss.

e)
```
bar a b c d
  | b >= c    = bar b a d c
  | otherwise = a==d
```

LÖSUNGSVORSCHLAG:

```
bar :: (Ord a, Ord b) => a -> b -> b -> a -> Bool
```

Im ersten Zweig wird die Ordnung des zweiten und dritten Argumentes geprüft. Damit müssen diese beide Argumente dem gleichen Typ angehören, welcher der Typklasse `Ord` angehören muss.

Der rekursive Aufruf verrät uns leider gar nichts über den Rückgabetyp der Funktion. Allerdings sehen wir für den rekursiven Aufruf das jeweils das erste und zweite, und das dritte und vierte Argument vertauscht werden. GHC schließt daraus schon, dass alle Typen gleich sein müssen, doch dass stimmt nicht: Der Aufruf kann ja auch polymorph sein - es reicht wenn das erste und vierte Argument einem anderen Typ angehören, der den gleichen Einschränkung unterliegt, in diesem Falle also auch der Typklasse `Ord` angehört.

Der zweite Zweig verrät uns den Rückgabetyp: `Bool`, da dies der Ergebnistyp der verwendeten Funktion `(==)` ist. Die explizite Einschränkung der Argumenttypen auf die Typklasse `Eq` können wir uns hier sparen, da `Ord` eine Unterklasse der Typklasse `Eq` ist - was wir ordnen können, können wir auch vergleichen.

Hinweis: Der rekursive Aufruf in `bar` benötigt eine andere Instantiierung des polymorphen Typs (die Argumente tauschen Ihre Positionen, d.h. Typen a und b vertauschen sich bei jedem rekursiven Aufruf). Die nennt man *polymorphe Rekursion*.

Es ist bewiesen, dass Typinferenz für polymorph rekursive Funktionen unentscheidbar ist. GHC kann den oben angegebenen Typ daher nicht inferieren. Allerdings kann GHC eine Typsignatur für eine polymorph rekursive Funktion sehr wohl überprüfen! D.h. wenn wir die Typsignatur explizit im Quellcode angeben, dann prüft und akzeptiert GHC diese auch.

H7-1 ***Freie Variablen und Substitution*** (4 Punkte; Abgabeformat: PDF)

a) Berechnen Sie jeweils die Menge der freien Variablen folgender Terme:

i) `(p q) r` $\{p, q, r\}$

ii) `(\a -> b a) (\c -> (d c) e)` $\{b, d, e\}$

iii) `(\x -> x) (\y -> y) (\z -> z)` Leere Menge; keine freien Variablen

iv) `(\u -> v) (\v -> u) (\w -> w)` $\{v, u\}$

v) `(\f -> (\g -> (\h -> (h f) i))) (g j)` $\{g, i, j\}$, Variable g kommt auch noch gebunden vor

b) Berechnen Sie jeweils folgende Termsubstitutionen:

i) `(\a -> b a)[x/a, y/b]` Ergebnis: `\`a `->` $y\ a$

ii) `((\x -> (\y -> x (y z))) x)[a/x,b/y,(\c -> d)/z]` `(\`x `->` `(\`y `->` $x\ (y\ (\backslash c$ `->` $d))))\ a)$

iii) `p (q r)[q/r,p/t,s/q]` $p\ (s\ s)$

iv) `(\u -> (\v -> u w))[(u v)/w]` `(\`a `->` `(\`b `->` $a\ (u\ v)$

Tipp: Kontrollieren Sie Ihre Antworten mithilfe Ihrer Lösung zur H7-2.

H7-2 ***Substitution*** (0 Punkte; Datei `H7-2.hs` als Lösung abgeben)
Gegeben ist folgende Datentypdeklaration zur Repräsentation von Termen:

```
data Term = Var Char | Const Int | App Term Term | Abs Char Term
```

a) Schreiben Sie eine Funktion `freeVars :: Term -> [Char]` welche die freien Variablen eines Terms effizient berechnet. *Hinweis:* Die Aufgabe ist sehr einfach, wenn Sie den Unterschied zwischen freien und gebundenen Variablen verstanden haben, siehe dazu auch Folien 08-9

LÖSUNGSVORSCHLAG:

```
freeVars :: Term -> [Char]
freeVars (Var x)     = [x]
freeVars (Const _)   = []
freeVars (App e1 e2) = (freeVars e1) ++ (freeVars e2)
freeVars (Abs x  e1) = filter (/=x) (freeVars e1)
```

b) Implementieren Sie die Substitution von Folie 08-8 effizient als Funktion `subst :: (Char, Term) -> Term -> Term`. `subst ('x', t1) t2` berechnet den Term, den man erhält wenn man in `t2` alle freien Vorkommen von `Var 'x'` durch `t1` ersetzt.

Verwenden Sie zur Vereinfachung folgende partielle Funktion zur Erzeugung frischer Variablen:

```
genFreshV :: [Char] -> Char
genFreshV vs = head $ filter (\c -> not $ c `elem` vs) ['a'..'z']
```

LÖSUNGSVORSCHLAG:

Der einzige schwierige Fall ist die Abstraktion, bei der drei Fälle zu unterscheiden sind:

i) Ist die zu ersetzende Variable identisch mit der durch die Abstraktion gebundene Variable, so ist nichts zu tun, da der Funktionsrumpf ja nur die frisch gebundene Variable referenzieren kann. Substitution ersetzt ja nur frei vorkommende Variablen, den nur freie Variablen referenzieren Dinge außerhalb des betrachteten Bereiches.

ii) Der Term, der in der Substitution eine Variable ersetzt, kann freie Variablen enthalten. Diese Referenzen nach "außerhalb" sollen unverändert erhalten bleiben, d.h. wir müssen aufpassen, dass diese freien Variablen frei bleiben und nicht versehentlich "eingefangen" werden.

Dieser Fall tritt ein, falls die durch die Abstraktion gebundene Variable in dem substituierenden Term frei vorkommt. Um den Konflikt aufzulösen, ändern wir

einfach den Namen der gebundenen Variable durch einen frischen - wiederum mithilfe einer Substitution. Danach führen wir noch die eigentliche Substitution auf dem Funktionsrumpf aus.

Grundlegend ist dabei, dass die Umbenennung gebundener Variablen unbedeutend ist: `\x -> e` ist ja äquivalent zu `\y -> e`$[y/x]$.

iii) Bestehen keine Namenskonflikte, dann setzen wir die Substitution wie gehabt auf dem Funktionsrumpf fort.

```
subst :: (Char, Term) -> Term -> Term
subst (x,e) o@(Var y)
  | x == y    = e
  | otherwise = o
subst (x,e) (Const c) = Const c
subst s o@(App e1 e2) = App (subst s e1) (subst s e2)
subst s@(x,e) o@(Abs y e1)
  | x == y        = o
  | y `elem` fv_e = Abs freshV (subst (x,e) $ subst (y, Var freshV) e1)
  | otherwise     = Abs y (subst s e1)
  where
    fv_e   = freeVars e
    fv_e1  = freeVars e1
    freshV = genFreshV (fv_e ++ fv_e1)
```

Im Fall der Umbenennung der gebundenen Variablen wird zwei Mal mit `subst` durch den Term gegangen, was nicht so effizient ist. Besser wäre es, wenn man mehrere Substitutionen zu einer zusammenfassen und dann auf einmal durchführen könnte. Dazu ist der Typ unserer Substitution `(Char,Term)` nicht gut geeignet, man bräuchte `[(Char,Term)]`, `Char -> Term` oder `Data.Map.Map Char Term`. Die Verwendung von letzterem demonstrieren wir anhand der ähnlichen Typsubstitution, enthalten in der Vorlage zur Aufgabe H8-3.

Abgabe: Lösungen zu den Hausaufgaben können bis Dienstag, den 16.06.2015, 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.