Prof. Martin Hofmann, PhD
Dr. Steffen Jost

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

7. Übung 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")`

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

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

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
```

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
```

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

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
```

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

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`

ii) `(\a -> b a) (\c -> (d c) e)`

iii) `(\x -> x) (\y -> y) (\z -> z)`

iv) `(\u -> v) (\v -> u) (\w -> w)`

v) `(\f -> (\g -> (\h -> (h f) i))) (g j)`

b) Berechnen Sie jeweils folgende Termsubstitutionen:

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

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

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

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

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

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']
```

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.