Dr. Steffen Jost
Leah Neukirchen

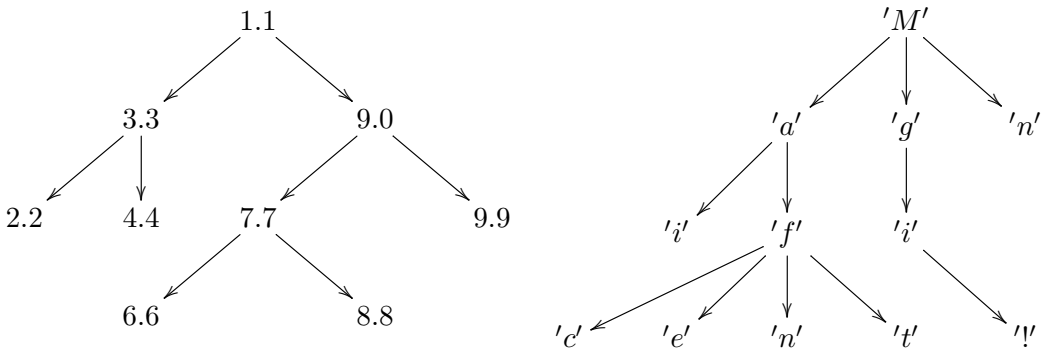
Ludwig-Maximilians-Universität München
Institut für Informatik
5. Juni 2018

Lösungsvorschlag zur 07. Übung zur Vorlesung

Programmierung und Modellierung

A7-1 *Verallgemeinerte Bäume*

a) Schreiben Sie einen Datentyp, mit dem man folgende Bäume repräsentieren kann:



Um Ihre Lösung zu überprüfen ist es hilfreich, anschließend Werte des kreierten Datentypen zu definieren, welche die gezeigten Bäume implementieren.

Hinweis: Knoten können beliebig viele Kinder besitzen, wie im rechten Beispiel gezeigt.

LÖSUNGSVORSCHLAG:

```
data RoseTree a = RoseTree a [RoseTree a]
  deriving (Show) -- um den Baum anzeigen zu können

rleaf x = RoseTree x []

baum1 :: RoseTree Double
baum1 = RoseTree 1.1
          [ RoseTree 3.3 [rleaf 2.2, rleaf 4.4]
          , RoseTree 9.0 [RoseTree 7.7 [rleaf 6.6, rleaf 8.8], rleaf 9.9]
          ]

baum2 :: RoseTree Char
baum2 = RoseTree 'M'
          [ RoseTree 'a' [rleaf 'i',
            RoseTree 'f' [rleaf 'c', rleaf 'i', rleaf 'e', rleaf 'n']]
          , RoseTree 'g' [RoseTree 'i' [rleaf 't']]
          , rleaf 'n'
          ]
```

b) Machen Sie Ihren Baumtyp zu einer Instanz der Klassen `Functor` und `Foldable`. Beachten Sie den geforderten Kind dieser Typklassen!

LÖSUNGSVORSCHLAG:

```
instance Functor RoseTree where
  fmap f (RoseTree r rs) = RoseTree (f r) (fmap (fmap f) rs)

instance Foldable RoseTree where
  foldMap f (RoseTree r rs) = (f r) `mappend` (foldMap (foldMap f) rs)
```

Achtung, der jeweils äußere Aufruf ist keine Rekursion, sondern ruft `fmap`/`foldMap` für die Instanz des Listentyps `[]` auf.

A7-2 ***Lexer*** Implementieren Sie eine Funktion `lexer :: String -> [Token]`, wie auf Folie 7.48 beschrieben. Dies ist eine leichte Übungen zum Aufwärmen! Einfaches Pattern-Matching und Rekursion reichen zur Lösung aus. Diesem Übungsblatt sollte die Dateivorlage `A7-2.hs` beiliegen, welche den Code aus der Vorlesung und Beispiele zum Testen enthält. 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.

A7-3 *Parser* Vervollständigen Sie den auf Folie 7.50 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 die beiliegende Vorlage `A7-2.hs`. 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"
```

Ende der Lösungsvorschläge für die Präsenzaufgaben.
Lösungsvorschläge für die Hausaufgaben folgen nach Ende der Abgabezeit.

H7-1 ***Monoid der Paare*** (2 Punkte; Abgabe: `H7-1.txt` oder `H7-1.pdf`)
Beweisen Sie, dass das kartesische Produkt zweier Monoide wieder eine Monoid ist!

Hinweis: Ruhe bewahren! Überlegen Sie erst einmal, was zu tun ist! Verwenden Sie die Instanzdeklaration von Folie 7.26 als Grundlage. Wie viele Gesetze sind hier nachzurechnen? Versuchen Sie den Beweis zunächst einmal ohne Induktion.

Hinweis: Suchen Sie sich eine der beiden nachfolgenden Aufgaben aus!

Die volle Punktzahl 4 für dieses Blatt kann also **entweder** durch korrekte Bearbeitung von H7-1 und H7-2 **oder** durch H7-1 und H7-3 erreicht werden.

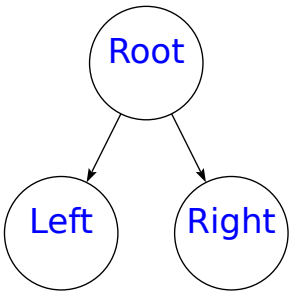
H7-2 ***Bäume darstellen*** (2 Punkte; Datei `H7-2.hs` als Lösung abgeben)

Stellen Sie beliebige binäre Bäume als Vektorgrafik dar!

Diesem Blatt sollte eine Dateivorlage `H7-2.hs` beiliegen, welche Sie verwenden sollten. Die Vorlage zeigt, wie Sie eine Vektorgrafik eines Baumes mit 3 Knoten erzeugen. Die Bibliothek `diagrams` nimmt uns einen Großteil der Arbeit ab. Zur Lösung dieser Aufgabe müssen Sie nicht die Dokumentation von `diagrams` lesen; die Vorlage enthält bereits alles, was Sie zur Lösung dieser Aufgabe benötigen. Hier ein Beispiel:

```
diagNode :: String -> Diag
diagNode s = text s `atop` circle 1

diagTriangle :: Diag
diagTriangle = connectOutside "X" "L" $
               connectOutside "X" "R" $
                    nx
                    ===
                (nl ||| nr) # center
  where
    nx = diagNode "Root"  # named "X"
    nl = diagNode "Left"  # named "L"
    nr = diagNode "Right" # named "R"
```



Dabei ist die besondere Einrückung ohne Bedeutung. Werte des Typs `Diag` sind Diagramme, welche wir wie folgt miteinander zu größeren Diagrammen kombinieren können:

- `diagNode s` zeichnet einen Text `s` innerhalb eines Kreises.
- `x `atop` y` kombiniert Diagramme, zeichnet `x` über `y` drüber.
- `x === y` kombiniert zwei Diagramme, wobei `x` oberhalb von `y` platziert wird.
- `x ||| y` kombiniert zwei Diagramme, wobei `y` rechts von `x` platziert wird.
- `center :: Diag -> Diag` zentriert ein Diagramm
- `named :: String -> Diag -> Diag` gibt einem Diagramm einen Namen.
  Achtung: Alle Namen müssen eindeutig sein d.h. ein Name darf nicht doppelt im Diagramm vorkommen! Dies ist explizit eine der Schwierigkeiten dieser Aufgabe.

- `connectOutside :: String -> String -> Diag -> Diag` zeichnet einen Pfeil zwischen benannten Teildiagrammen. Diese Teildiagramme müssen in dem übergeben Diagramm bereits mit den angegebenen eindeutigen Namen enthalten sein.
- `(#) :: a -> (a -> b) -> b` füttert ein Argument an eine gegebene Funktion. Dabei ist `x # f` identisch zu `f $ x`. Diese Infix-Funktion `(#)` aus `diagrams` dient wie `($)` auch lediglich zur hübschen Formatierung unseres Codes und spart Klammern.

Die Vektorgrafik wird dann durch Kompilieren mit ghc und anschließendem Ausführen des Codes erzeugt, wobei die `main` Funktion der Vorlage nicht mehr verändert werden muss. Leider ist `diagrams` nicht in der Standardbibliothek enthalten, weshalb wir den Aufruf von `ghc` mit Hilfe von `stack` durchführen:

```
>  stack ghc H7-2.hs --resolver lts-10 --package diagrams
Selected resolver: lts-10.10
[1 of 1] Compiling Main             ( H7-2.hs, H7-2.o )
Linking H7-2 ...
> ./H7-2 -o Demo.svg -h 640 -S Triangle
> firefox Demo.svg
```

Kompilieren des Codes mit `ghc` erzeugt eine ausführbare Datei. Diese führen wir dann einmal aus, wobei wir als Parameter den Namen der Ausgabedatei, die Höhe der Grafik, und die Auswahl des zu berechnenden Diagramms angeben (ansatt `-S Triangle` später also z.B. `-S Tree1` eintippen). Danach können wir uns die Grafik in einem Webbrowser anschauen.[1] Es ist Ihnen überlassen, ob Sie leere Blattknoten anzeigen möchten oder nicht.

Hinweise:

- Der erste Aufruf kann eine Stunde dauern, da ggf. diagrams und ghc in passender Version von stack heruntergeladen und kompiliert werden müssen.
- Wer `stack` nicht benutzt, kann stattdessen `cabal install diagrams` versuchen, weitere Hinweise finden Sie in der Vorlage.
- Die Bibliothek `diagrams` ist nicht prüfungsrelevant. Sie ist aber auch gar nicht der Inhalte dieser Aufgabe! Zur Lösung dieser Aufgabe müssen wir hier lediglich mit Bäumen und Funktionen umgehen — was natürlich prüfungsrelevant ist.

[1] Mac-User geben zur Ansicht der Datei ein: `open -a firefox Demo.svg`

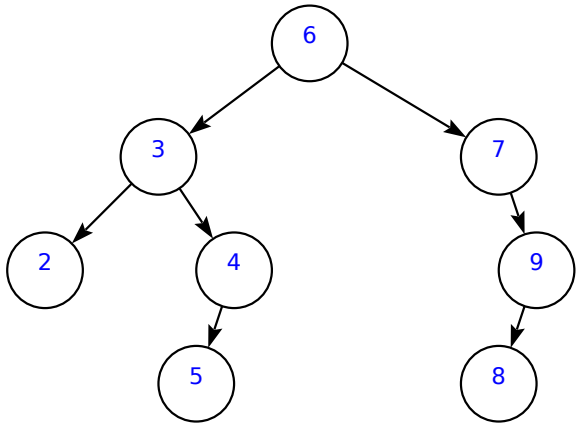
H7-3 *Bäume Drucken* (2 Punkte; Datei `H7-3.hs` als Lösung abgeben)
In der Vorlesung wurde eine Instanz der Typklasse `Show` für folgenden Typ deklariert

```
data Tree a = Empty | Node { label::a,  left,right::Tree a }
```

welche den unten gegeben Baum wie folgt dargestellt hat: `(6,(3,<2>,(4,<5>,ϵ)),(7,ϵ,(9,<8>,ϵ)))`
Schreiben Sie eine alternative Funktion `printTree :: (Show a) => Tree a -> String`, so dass ein Baum stattdessen hübsch als Ascii-Art dargestellt wird!

Beispiele: Zwei verschiedene Lösungsmöglichkeiten für den Baum in der Mitte:[2]

```
   /-9
  |   \-8
 /-7
6
|   /-4
|  |   \-5
 \-3
   \-2
```




```
6
|-7
| |-9
|   \-8
\-3
  |-4
  | \-5
  \-2
```

Diesem Blatt sollte bereits eine Dateivorlage `H7-3.hs` beiliegen, welche Sie verwenden sollten.

- Unverpflichtende Ratschläge
  - Es ist vermutlich einfacher, den Baum auf der Seite liegen auszugeben.
  - Jede Knotenmarkierung wird in eine eigene Zeile geschrieben, so dass die Länge der Knotenmarkierung unproblematisch ist.
  - Je tiefer ein Knoten im Baum ist, desto weiter rechts wird er gedruckt. Die Funktion zum Drucken sollte also als zusätzliches Argument die aktuelle Tiefe mitführen.
- Bewertungsrelevant für volle Punktzahl
  - Eine Ausgabe ohne durchgezogene Linien (z.B. wie oben, nur ohne `|`) bringt nicht die volle Punktzahl, ist aber für einen Anfang deutlich einfacher.
  - Es sollte klar erkennbar sein, ob es sich jeweils um einen linken oder rechten Teilbaum handelt, also `t2` und `t3` aus der Vorlage sollten klar unterscheidbar sein.

Abgabe: Lösungen zu den Hausaufgaben können bis Samstag, den 16.6.18, mit UniWorX nur als `.zip` abgegeben werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen. Bis zu 3 Studierende können gemeinsam als Gruppe abgeben. Bitte beachten Sie auch die Hinweise zum Übungsbetrieb auf der Vorlesungshomepage (`www.tcs.ifi.lmu.de/lehre/ss-2018/promo/`).

[2] Wir verzichten auf die Ausgabe von `Empty`; Sie können dafür `*` ausgeben, wenn Sie dies einfacher finden.