Prof. Dr. Thomas Breuel
Dipl.-Inf. Patrick Michel (AG ST)
Dipl.-Ing. Ludwig Schmidt-Hackenberg

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

Lösungshinweise/-vorschläge zum Übungsblatt 5: Software-Entwicklung 1 (WS 2010/11)

Die Hinweise und Vorschläge in diesem Dokument sollen der Lösungsfindung dienen und erheben demnach weder Anspruch auf Vollständigkeit noch Korrektheit. Sollten Sie Fehler finden, würden wir uns freuen, wenn Sie uns diese mitteilen. (Kontaktdaten finden Sie auf unserer Webpräsenz.)

Bitte beachten Sie auch auf diesem Blatt, dass nicht alle möglichen Sprachmittel zur Lösung verwendet wurden. Es ist eine gute Übung, diese eleganter und/oder kürzer zu formulieren.

Aufgabe 1 Elementares Haskell (Präsenzaufgabe)

```haskell
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)

fib :: Int -> Integer
fib n
  | n == 0    = 0
  | n == 1    = 1
  | otherwise = fib (n - 1) + fib (n - 2)
```

Schreiben Sie zwei weitere Versionen dieser Funktion, einmal mit Hilfe von Pattern-Matching und einmal mit Hilfe von Guards.

```haskell
summe :: [Integer] -> Integer
summe []     = 0
summe (x:xs) = x + summe xs

summe :: [Integer] -> Integer
summe x
  | null x    = 0
  | otherwise = head x + summe (tail x)
```

Schreiben Sie zwei weitere Versionen dieser Funktion, einmal mit Hilfe von Pattern-Matching und einmal mit Hilfe von Guards.

```haskell
istWerktag :: Wochentag -> Bool
istWerktag Sa = False
istWerktag So = False
istWerktag _  = True

istWerktag :: Wochentag -> Bool
istWerktag t
  | t == Sa   = False
  | t == So   = False
  | otherwise = True

istWerktag :: Wochentag -> Bool
istWerktag t = not (t == Sa || t == So)
```

Schreiben Sie zwei weitere Versionen dieser Funktion, einmal mit Hilfe von Pattern-Matching und einmal mit Hilfe von Guards. Kennen Sie eine noch kürzere Variante die Funktion zu definieren?

`g :: [(Integer, Integer)] -> Integer`
`g []       = -1`
`g [(a, b)] = a * b`
`g _        = -2`
	Definieren Sie diese Funktion mit Hilfe von Pattern-Matching möglichst einfach.

`f :: ((Int, [Int]), Int) -> Int` `f ((_, x : _), y) = x + y`	Definieren Sie diese Funktion mit Hilfe von Pattern-Matching möglichst einfach.

Aufgabe 2 Permutationen (Einreichaufgabe)

a) Geben Sie eine Funktion `augment :: Char -> [String] -> [String]` an, welche zu jedem in einer Liste enthaltenen `String` einen gegebenen Buchstaben vorne hinzufügt und die resultierende Liste von Strings zurückliefert.

```
augment :: Char -> [String] -> [String]
augment _ []      = []
augment e (h : t) = (e : h) : augment e t
```

b) Geben Sie eine Funktion `everywhere :: Char -> String -> [String]` an, die einen Buchstaben und einen String nimmt und den Buchstaben jeweils einmal an jede mögliche Position im String einfügt.

```
everywhere :: Char -> String -> [String]
everywhere n []      = [[n]]
everywhere n (h : t) = (n : h : t) : augment h (everywhere n t)
```

c) Geben Sie eine Funktion `everywhereList :: Char -> [String] -> [String]` an, die einen Buchstaben `c` und einen Liste von Strings nimmt, dann die Operation `everywhere c` auf jeden String der Liste anwendet und die Ergebnislisten konkateniert zurückliefert.

```
everywhereList :: Char -> [String] -> [String]
everywhereList n [] = []
everywhereList n (h : t) = everywhere n h ++ everywhereList n t
```

d) Realisieren Sie nun die Funktion `permutations :: String -> [String]`, welche für einen gegebenen `String` alle möglichen Anordnungen seiner Buchstaben bestimmt.

```
permutations :: String -> [String]
permutations []      = []
permutations [n]     = [[n]]
permutations (h : t) = everywhereList h (permutations t)
```

Aufgabe 3 Datenstrukturen (Präsenzaufgabe)

a) Schreiben Sie drei Selektorfunktionen `tag`, `ausgabe` und `titel` für den Datentyp `Angebot`.

```
tag :: Angebot -> Tag
tag (Angebot t _ _) = t

ausgabe :: Angebot -> Ausgabe
ausgabe (Angebot _ a _) = a

titel :: Angebot -> String
titel (Angebot _ _ t) = t
```

b) Schreiben Sie zwei Diskriminatorfunktionen, die für gegebenen Tag bzw. gegebene Ausgabe testen, ob ein `Angebot` für diesen Tag bzw. diese Ausgabe gültig ist.

```
istTag :: Tag -> Angebot -> Bool
istTag t (Angebot t1 _ _) = t == t1

istAusgabe :: Ausgabe -> Angebot -> Bool
istAusgabe a (Angebot _ a1 _) = a == a1
```

c) Wir betrachten nun einen kompletten `Mensaplan`. Schreiben Sie eine Funktion, die für gegebenen Mensaplan und Tag die Liste aller Angebote ausgibt, sprich Tupel von `Ausgaben` und Titeln.

```
anTag :: Mensaplan -> Tag -> [(Ausgabe, String)]
anTag []                        t = []
anTag (Angebot t1 a b : rest) t =
  if t == t1 then (a, b) : rest `anTag` t else rest `anTag` t
```

d) Mit dieser Modellierung kann man nur den Mensaplan für eine Woche darstellen. Erweitern Sie die Datentypdefinition von `Angebot`, indem Sie vorne ein Feld für die Kalenderwoche (`Int`) einfügen.

```
data Angebot = Angebot Int Tag Ausgabe String
  deriving (Eq, Ord, Show)
```

Wie müssen Sie Ihre bisher definierten Funktionen anpassen?

```
kw :: Angebot -> Int      -- NEU
kw (Angebot k _ _ _) = k

tag :: Angebot -> Tag
tag (Angebot _ t _ _) = t

ausgabe :: Angebot -> Ausgabe
ausgabe (Angebot _ _ a _) = a

titel :: Angebot -> String
titel (Angebot _ _ _ t) = t

istTag :: Tag -> Angebot -> Bool
istTag t (Angebot _ t1 _ _) = t == t1

istAusgabe :: Ausgabe -> Angebot -> Bool
istAusgabe a (Angebot _ _ a1 _) = a == a1

anTag :: Mensaplan -> Tag -> [(Ausgabe, String)]
anTag []                        t = []
anTag (Angebot _ t1 a b : rest) t =
  if t == t1 then (a, b) : rest `anTag` t else rest `anTag` t
```

Aufgabe 4 Datenstrukturen (Einreichaufgabe)

a) Definieren Sie die Datenstrukturen, Datentypen und Funktionen aus Aufgabe 3 in einer Haskell Datei und überprüfen Sie sie im Interpreter. Laden Sie sich dazu auch die Datei "`mensaplan.hs`" von der Vorlesungsseite.

b) Schreiben Sie eine Funktion `suche :: String -> Mensaplan -> Mensaplan`, welche alle Angebote ausfiltert, die nicht das gegebene Suchwort enthalten.

```
import Data.List

suche :: String -> Mensaplan -> Mensaplan
suche _ [] = []
suche t (a:rest) =
  if t `isInfixOf` titel a then a : (suche t rest) else suche t rest
```

c) Schreibe Sie eine Funktion `konsistent :: Mensaplan -> Bool`, die überprüft, ob ein Mensaplan konsistent ist. Wir betrachten einen Plan als konsistent, wenn es zu keiner Zeit zwei Angebote an derselben Ausgabe gibt.

```
nichtEnthalten :: (Int, Tag, Ausgabe) -> Mensaplan -> Bool
nichtEnthalten _ [] = True
nichtEnthalten (k, t, a) (Angebot k1 t1 a1 _ : rest) =
  (k /= k1 || t /= t1 || a /= a1) && nichtEnthalten (k, t, a) rest

konsistent :: Mensaplan -> Bool
konsistent [] = True
konsistent (Angebot k t a _ : rest) =
  nichtEnthalten (k, t, a) rest && konsistent rest
```

Aufgabe 5 Rekursive Datenstrukturen (Einreichaufgabe)

a) Definieren Sie einen Wert `test :: Exp` mit Hilfe der Konstruktoren von `Exp` und `Op`, der folgendem Ausdruck in Haskell-Notation entspricht:

```
(42 `div` 7 - 3) * (7 + 31 `div` 8) + 12
```

```
test :: Exp
test = Node Plus
  ( Node Mult
    ( Node Minus
        ( Node Div (Leaf 42) (Leaf 7) )
        ( Leaf 3 )
    )
    ( Node Plus
        ( Leaf 7 )
        ( Node Div (Leaf 31) (Leaf 8) )
    )
  )
  ( Leaf 12 )
```

b) Schreiben Sie eine Haskell-Funktion `showInfix :: Exp -> String`, welche einen gegebenen Ausdruck in einen `String` in *Infix-Notation* umwandelt.

```
showOp :: Op -> String
showOp Plus   = "+"
showOp Minus  = "-"
showOp Mult   = "*"
showOp Div    = "`div`"

showInfix :: Exp -> String
showInfix (Leaf i) = show i
showInfix (Node o a b) =
  "(" ++ showInfix a ++ " " ++ showOp o ++ " " ++ showInfix b ++ ")"
```

Verwenden Sie diese Funktion um weitere sinnvolle Testeingaben für die vorhergehenden und noch folgenden Aufgabenteile zu erzeugen und testen Sie mit diesen Ihre Implementierungen.

Zum Testen kann man auch folgende Funktion verwenden, die `someExp :: Int -> Exp` benutzt, welche nicht in der Aufgabe erwähnt wird:

```
roundtrip :: Int -> Bool
roundtrip n = let t = someExp n in t == parse (showInfix t)
```

Man kann sie dann auch sehr einfach mal über `[1..]` mappen.

c) Schreiben Sie eine Haskell-Funktion `divConstZero :: Exp -> Bool`, welche testet, ob ein Ausdruck eine Division durch die *Konstante Null* enthält.

```
divConstZero :: Exp -> Bool
divConstZero (Leaf _) = False
divConstZero (Node Div _ (Leaf 0)) = True
divConstZero (Node _ a b) = divConstZero a || divConstZero b
```

d) Schreiben Sie eine Haskell-Funktion `simplify :: Exp -> Exp`, welche Additionen und Subtraktionen auf negativen Konstanten in Ausdrücken vereinfacht:

Es geht hier mehr darum lokale Änderung an einer rekursiven Datenstruktur zu machen, als wirklich alle möglichen Vereinfachungen abzudecken. Zum lokalen Ändern muss man Daten aus Konstruktoren auspacken und danach wieder einpacken. Insbesondere muss man die Änderung über alle nicht tangierten Konstruktoren "drüber mappen", also deren Komponenten auspacken, in die Rekursion gehen und alle Ergebnisse wieder mit einem Konstruktor einpacken.

Was die Vereinfachung angeht macht der Lösungsvorschlag folgendes:

$$
\begin{array}{lrcl}
 & 1+2 & \Rightarrow & 1+2 \\
(\textit{case } 1) & 1+(-2) & \Rightarrow & 1-2 \\
 & 1-2 & \Rightarrow & 1-2 \\
(\textit{case } 4) & 1-(-2) & \Rightarrow & 1+2
\end{array}
\qquad
\begin{array}{lrcl}
(\textit{case } 2) & (-1)+2 & \Rightarrow & 2-1 \\
(\textit{case } 1) & (-1)+(-2) & \Rightarrow & (-2)-1 \\
 & (-1)-2 & \Rightarrow & (-1)-2 \\
(\textit{case } 3) & (-1)-(-2) & \Rightarrow & 2-1
\end{array}
$$

```
simplify :: Exp -> Exp
simplify (Leaf c) = Leaf c
simplify (Node Plus a (Leaf b)) | b < 0 =                    -- case 1
  Node Minus (simplify a) (Leaf $ negate b)
simplify (Node Plus (Leaf a) b) | a < 0 =                    -- case 2
  Node Minus (simplify b) (Leaf $ negate a)
simplify (Node Minus (Leaf a) (Leaf b)) | a < 0 && b < 0 =   -- case 3
  Node Minus (Leaf $ negate b) (Leaf $ negate a)
simplify (Node Minus a (Leaf b)) | b < 0 =                   -- case 4
  Node Plus (simplify a) (Leaf $ negate b)
simplify (Node op a b) = Node op (simplify a) (simplify b)
```

e) Schreiben Sie eine Haskell-Funktion `eval :: Exp -> Integer`, welche einen Ausdruck auswertet.

```
eval :: Exp -> Integer
eval (Leaf i) = i
eval (Node Plus   a b) = eval a + eval b
eval (Node Minus  a b) = eval a - eval b
eval (Node Mult   a b) = eval a * eval b
eval (Node Div    a b) = eval a `div` eval b
```

f) Schreiben Sie eine Haskell-Funktion `divZero :: Exp -> Bool`, welche testet, ob ein Ausdruck eine Division durch Null enthält.

Der Trick hier ist darauf zu achten, dass der Aufruf von `eval` *nach* dem rekursiven Aufruf `divZero b` ausgeführt wird, da ansonsten die Funktion immer noch abrupt terminieren kann, wenn in einem Divisor selbst eine Division durch Null ist. Ist `divZero b == True`, wird `eval b` nie ausgeführt.

```
divZero :: Exp -> Bool
divZero (Leaf _) = False
divZero (Node Div a b) = divZero a || divZero b || eval b == 0
divZero (Node _ a b) = divZero a || divZero b
```

g) Schreiben Sie eine Haskell-Funktion `safeEval :: Exp -> Maybe Integer`, welche einen Ausdruck auswertet und bei einer Division durch Null kein Ergebnis (`Nothing`) zurückliefert.

```
safeEval :: Exp -> Maybe Integer
safeEval (Leaf i) = Just i
safeEval (Node op ea eb) = case (safeEval ea, safeEval eb) of
  (Just a, Just b) -> if op == Div && b == 0 then Nothing else case op of
    Plus  -> Just (a + b)
    Minus -> Just (a - b)
    Mult  -> Just (a * b)
    Div   -> Just (a `div` b)
  _ -> Nothing
```

Testen kann man auch hier sehr schön mit `map (safeEval . someExp) [1..]`.

Wer hier einfach `divZero` aus der Aufgabe davor nimmt und `False` zu `Nothing` macht und bei `True` das Ergebnis der normalen `eval` Funktion in ein `Just` einpackt, der hat wohl die textuell schnellste Lösung gefunden. Allerdings ist diese sehr ineffizient, da man mindestens den doppelten Aufwand hat, schlimmer noch, wenn auch `divZero` wieder `eval` aufruft.

```
safeEval :: Exp -> Maybe Integer
safeEval e = if divZero e then Nothing else Just (eval e)
```