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

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

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

Ausgabe: in der Woche vom 22.11. bis zum 26.11.10
Abgabe: in der Woche vom 29.11. bis zum 03.12.10
Abnahme: max. zwei Tage nach der Übung

Aufgabe 1 Elementares Haskell (Präsenzaufgabe)

```haskell
fib :: Int -> Integer
fib n = if n == 0 then 0 else
        if n == 1 then 1 else
          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 xs = if null xs then 0 else
  head xs + summe (tail xs)
```

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

```haskell
data Wochentag =
  Mo | Di | Mi | Do | Fr | Sa | So
  deriving Eq

istWerktag :: Wochentag -> Bool
istWerktag t =
  if t == Sa then False else
  if t == So then False else True
```

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?

```haskell
g :: [(Integer, Integer)] -> Integer
g xs = if null xs then -1 else
  if null (tail xs)
    then fst (head xs) * snd (head xs)
    else -2
```

Definieren Sie diese Funktion mit Hilfe von Pattern-Matching möglichst einfach.

```haskell
f :: ((Int, [Int]), Int) -> Int
f t = head (snd (fst t)) + snd t
```

Definieren Sie diese Funktion mit Hilfe von Pattern-Matching möglichst einfach.

Aufgabe 2 Permutationen (Einreichaufgabe)

Die Elemente einer Liste der Länge n können auf insgesamt $n!$ verschiedene Weisen angeordnet werden. Die Menge der verschiedenen Anordnungen wird als die Menge der *Permutationen* bezeichnet.

Für die Liste `"abc"` existieren beispielsweise $3! = 1 \cdot 2 \cdot 3 = 6$ verschiedene Anordnungen; konkret sind dies `"abc"`, `"bac"`, `"bca"`, `"acb"`, `"cab"` und `"cba"`.

Im Rahmen dieser Aufgabe soll schrittweise ein Haskell-Programm entwickelt werden, welches die Bestimmung der Permutationen einer gegebenen Liste erlaubt, wobei davon ausgegangen werden kann, dass Elemente nicht mehrfach auftreten können.

Tipp: Verwenden Sie in jedem Aufgabenteil die Funktion, die Sie im direkt vorhergehenden Aufgabenteil entwickelt haben.

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 'a' ["bc", "cb"]  ==  ["abc", "acb"]
augment 'a' [""]          ==  ["a"]
augment 'a' []            ==  []
```

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 'X' "bcd"  ==  ["Xbcd", "bXcd", "bcXd", "bcdX"]
everywhere 'a' "bcd"  ==  ["abcd", "bacd", "bcad", "bcda"]
everywhere 'a' ""     ==  ["a"]
```

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 'a' ["bc", "cb"]  ==  ["abc", "bac", "bca",   "acb", "cab", "cba"]
everywhereList 'a' []            ==  []
```

d) Realisieren Sie nun die Funktion `permutations :: String -> [String]`, welche für einen gegebenen `String` alle möglichen Anordnungen seiner Buchstaben bestimmt. Die Permutationen einer Liste lassen sich im Prinzip ganz simpel rekursiv bestimmen:

1. Man berechnet (rekursiv) die Permutationen der Liste ohne das erste Element. Definieren Sie dazu auch die zwei Spezialfälle für eine leere Ausgangsliste und für eine Liste mit nur einem Element.
2. Man fügt das noch fehlende Element an jede mögliche Position in jeder aus 1) folgenden Permutation an (mit Hilfe von `everywhereList`).

Aufgabe 3 Datenstrukturen (Präsenzaufgabe)

Wir betrachten die folgende Modellierung des Mensaplans:

```
data Tag = MO | DI | MI | DO | FR                  deriving (Eq, Ord, Show)
data Ausgabe = Essen1 | Essen2 | Grill | Wok       deriving (Eq, Ord, Show)
data Angebot = Angebot Tag Ausgabe String          deriving (Eq, Ord, Show)

type Mensaplan = [Angebot]

Beispiel: [ Angebot MO Essen1 "SchniPoSa" ]
```

Hinweis: Bearbeiten Sie die folgenden Aufgaben konzeptionell mit Papier und Bleistift.

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

b) Schreiben Sie zwei Diskriminatorfunktionen

```
istTag :: Tag -> Angebot -> Bool
istAusgabe :: Ausgabe -> Angebot -> Bool
```

die für gegebenen Tag bzw. gegebene Ausgabe testen, ob ein `Angebot` für diesen Tag bzw. diese Ausgabe gültig ist.

c) Wir betrachten nun einen kompletten `Mensaplan`. Schreiben Sie eine Funktion

```
anTag :: Mensaplan -> Tag -> [(Ausgabe, String)]
```

die für gegebenen Mensaplan und Tag die Liste aller Angebote ausgibt, sprich Tupel von `Ausgaben` und Titeln.

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.

Wie müssen Sie Ihre bisher definierten Funktionen anpassen?

Aufgabe 4 Datenstrukturen (Einreichaufgabe)

Hinweis: Die Abgabe dieser Aufgabe hat weiterhin handschriftlich zu erfolgen, da die Aufgaben konzeptionell in knapper Form lösbar sind. Natürlich sollen Sie ihre Lösungen dann mit dem GHC(i) überprüfen, ausprobieren und in der Abnahme vorführen.

Wir befassen uns weiter mit Mensaplänen.

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) Wir möchten gerne wissen, wann es unser Lieblingsessen geben wird (oder alternativ: Welche Schlange wir meiden müssen, weil es mal wieder Steak gibt).

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

Hinweis: Um zu testen, ob ein gegebener String `s` in einem anderen String `t` enthalten ist, können Sie die Bibliotheksfunktion "`s `isInfixOf` t`" verwenden, die Sie mit dem Befehl "`import Data.List`" in Ihr Programm einbinden können.

Testen Sie ihre Funktion mit Hilfe der Datei "`mensaplan.hs`" und probieren Sie insbesondere die Anfragen "`suche "steak" testplan`" und "`suche "Pommes" testplan`".

c) Die Modellierung aus Aufgabe 3 hat einige Schwächen, so ist es zum Beispiel sehr einfach inkonsistente Mensapläne zu erstellen.

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.

Tipp: Schreiben Sie zunächst eine Funktion

```
nichtEnthalten :: (Int, Tag, Ausgabe) -> Mensaplan -> Bool
```

die testet, ob eine Kombination aus Kalenderwoche, Tag und Ausgabe nicht in einem Plan enthalten ist.

Aufgabe 5 Rekursive Datenstrukturen (Einreichaufgabe)

*Hinweis: Bei dieser Aufgabe macht es Sinn diese nach abgeschlossener Konzeption direkt am Computer zu entwickeln. Geben Sie deshalb einen **Ausdruck** Ihrer Haskell Datei(en) ab und schicken Sie diese zusätzlich per **Email** an Ihren Betreuer! Achten Sie auch darauf ihre Lösungen gut lesbar zu strukturieren. Es gehört zum Programmieren dazu die Implementierung verständlich zu präsentieren, nicht zuletzt um sie überhaupt jemand anderem erklären zu können.*

Wir betrachten eine kleine Teilmenge der Ausdrücke (engl.: *expressions*), die in der Sprache Hakell gebildet werden können. Wir modellieren diese durch den Haskell-Datentyp `Exp`:

```
data Exp =
    Leaf Integer
  | Node Op Exp Exp
  deriving (Eq, Ord, Show)

data Op = Mult | Div | Plus | Minus
  deriving (Eq, Ord, Show)
```

Der Konstruktor `Leaf` steht dabei für eine Konstante vom Typ `Integer`, der Konstruktor `Node` für die Anwendung eines binären Operators (beschrieben durch den Datentyp `Op`) auf zwei Werte, die ebenfalls durch Ausdrücke gegeben sind. Als Operationen betrachten wir hier Multiplikation, ganzzahlige Division, Addition und Subtraktion, jeweils symbolisiert durch den Konstruktor `Mult`, `Div`, `Plus` bzw. `Minus`.

Benutzen Sie für diese Aufgabe die Datei "`Exp.hs`", welche Sie auf der Webseite der Vorlesung finden und mit dem Befehl **`import`** `Exp` in Ihr Programm einbinden können. In ihr sind die obigen Datentypen definiert, sowie zwei Hilfsfunktionen, die Sie in Aufgabenteilen verwenden sollen.

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
```

*Hinweis: Definieren Sie den Ausdruck **nicht** in einer einzigen Zeile, sondern verwenden Sie Einrückungen und Leerzeichen, um die Struktur des Ausdrucks schon im Quellcode deutlich zu machen!*

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

Beispiel:

```
showInfix (Node Plus (Leaf 3) (Node Div (Leaf 4) (Leaf 5)))  ==
    "(3 + (4 `div` 5))"
```

Das Modul `Exp` definiert die Funktion `parse :: String -> Exp`, welche die inverse Funktionalität zu `showInfix` bereitstellt. Korrekte Ausdrücke in Infix-Notation werden also als `Exp` zurückgegeben, bei Syntaxfehlern terminiert die Funktion abrupt mit einer entsprechenden Fehlermeldung. Verwenden Sie diese Funktion um weitere sinnvolle Testeingaben für die vorhergehenden und noch folgenden Aufgabenteile zu erzeugen und testen Sie mit diesen Ihre Implementierungen.

Testen Sie insbesondere, dass die Ausgabe Ihrer `showInfix` Funktion an `parse` übergeben zu keinem Fehler führt und Sie den ursprünglichen Ausdruck zurück erhalten. Mit Hilfe der vordefinierten Funktion `check` können Sie dies für 100 zufällige Testfälle überprüfen. `check showInfix` testet für jeden Testfall `e` also folgende Gleichung: `e == parse (showInfix e)`

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

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

$$3 + (-4) \Rightarrow 3 - 4 \qquad 6 + (2 - (-7)) \Rightarrow 6 + (2 + 7)$$
$$(-5) + 3 \Rightarrow 3 - 5 \qquad (3 - (-5)) * 9 \Rightarrow (3 + 5) * 9$$

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

```
eval (parse "(17 + 4) * 2")  ==  42
```

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

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