Technische Universität München
Fakultät für Informatik
Prof. Tobias Nipkow, Ph.D.
Lars Noschinski, Dr. Jasmin Blanchette, Dmitriy Traytel

Wintersemester 2012/13
Lösungsblatt Endklausur
9. Februar 2013

Einführung in die Informatik 2

Name	Vorname	Studiengang	Matrikelnummer
.....	.....	☐ Bachelor ☐ Inform. ☐ Master ☐ W-Inf. ☐ ..... ☐	.....

Hörsaal	Reihe	Sitzplatz	Unterschrift
.....	.....	.....	.....

Allgemeine Hinweise

- Bitte füllen Sie obige Felder in Druckbuchstaben aus und unterschreiben Sie!
- Bitte schreiben Sie nicht mit Bleistift oder in roter/grüner Farbe!
- Die Arbeitszeit beträgt 120 Minuten.
- Alle Antworten sind in die geheftete Angabe auf den jeweiligen Seiten (bzw. Rückseiten) der betreffenden Aufgaben einzutragen. Auf dem Schmierblattbogen können Sie Nebenrechnungen machen. Der Schmierblattbogen muss ebenfalls abgegeben werden, wird aber in der Regel nicht bewertet.
- Es sind keine Hilfsmittel außer einem DIN-A4-Blatt zugelassen.

Hörsaal verlassen von ..... bis ..... / von ..... bis .....
Vorzeitig abgegeben um .....
Besondere Bemerkungen:

	A1	A2	A3	A4	A5	A6	A7	A8	A9	Σ	Korrektor
Erstkorrektur											
Zweitkorrektur											

Aufgabe 1 (5 Punkte)

Geben Sie den allgemeinsten Typ der folgenden Ausdrücke an:

1. `filter not`
2. `[] : []`
3. `\f x y -> f (x,y)`
4. `map (map fst)`

Begründen Sie kurz, warum der folgende Ausdruck nicht typkorrekt ist:

5. `map head [True, False]`

Lösungsvorschlag

1. `filter not :: [Bool] -> [Bool]`
2. `[] : [] :: [[a]]`
3. `\f x y -> f (x,y) :: ((a,b) -> c) -> a -> b -> c`
4. `map (map fst) :: [[(a,b)]] -> [[a]]`
5. Die Funktion `map head` hat den Typ `[[a]] -> [a]`, d.h. sie erwartet eine Liste von Listen als Argument. Hier ist das Argument aber lediglich eine Liste von `Bool`s.

Aufgabe 2 (6 Punkte)

Betrachten Sie die Funktion `f :: [Int] -> [Int]`, die eine Liste *xs* auf die Liste der Absolutbeträge der negativen Zahlen in *xs* abbildet.
Beispiel: `[1,-2,3,-4,-5,6]` soll abgebildet werden auf `[2,4,5]`.
Implementieren Sie *f* auf drei verschiedene Arten:

1. Als rekursive Funktion; ohne die Verwendung von Listenkomprehensionen oder Funktionen höherer Ordnung wie `map`, `filter`, `foldl`, `foldr`.
2. Mit Hilfe einer Listenkomprehension; ohne die Verwendung von Rekursion oder Funktionen höherer Ordnung wie `map`, `filter`, `foldl`, `foldr`.
3. Mit Hilfe von `map` und `filter`; ohne die Verwendung von Listenkomprehensionen oder Rekursion.

Lösungsvorschlag

1. ```
   f [] = []
   f (x:xs) | x < 0 = abs x : f xs
            | otherwise = f xs
   ```

2. ```
   f xs = [abs x | x <- xs, x < 0]
   ```

3. ```
   f = map abs . filter (<0)
   ```

   oder auch

   ```
   f xs = map (\x -> abs x) (filter (<0) xs)
   ```
   ```

Aufgabe 3 (6 Punkte)

Gegeben sei ein Datentyp zur Darstellung von einfachen booleschen Formeln:

```
data Fml = Var Char | Neg Fml | Conj [Fml] | Disj [Fml]
  deriving Eq
```

Eine Formel ist also entweder eine boolesche Variable, die Negation einer Formel, die Konjunktion von null oder mehr Formeln oder die Disjunktion von null oder mehr Formeln.

Implementieren Sie eine Funktion `rename :: (Char -> Char) -> Fml -> Fml`, die die Variablen einer Formel durch die Anwendung des ersten Arguments (der *Umbenennungsfunktion*) systematisch umbenennt.

Beispiel: Die boolesche Formel $\neg\, x \vee y$ wird als `Conj [Neg (Var 'x'), Var 'y']` dargestellt. Für die Umbenennungsfunktion `Data.Char.toUpper` soll `rename` die Formel $\neg\, X \vee Y$ zurückgeben:

```
rename Data.Char.toUpper (Conj [Neg (Var 'x'), Var 'y']) ==
    Conj [Neg (Var 'X'), Var 'Y']
```

Lösungsvorschlag

```
rename :: (Char -> Char) -> Fml -> Fml
rename subst (Var c) = Var (subst c)
rename subst (Neg f) = Neg (rename subst f)
rename subst (Conj fs) = Conj (map (rename subst) fs)
rename subst (Disj fs) = Disj (map (rename subst) fs)
```

Aufgabe 4 (4 Punkte)

Welche der gegebenen Definitionen definieren die gleiche Funktion, welche nicht? Begründen Sie kurz.

```
f1 xs x = filter (> x) xs
f2 xs = \x -> filter (> x)
f3 = \xs x -> filter (> x) xs
f4 x = filter (> x)
```

Lösungsvorschlag

Die Definitionen `f1` und `f3` sind äquivalent; hier wurden nur die Pattern auf der linken Seite durch Lambda-Ausdrücke auf der rechten Seite ersetzt. Die Definitionen `f2` und `f4` haben beide jeweils davon unterschiedliche Typen und können damit auch nicht äquivalent sein.

Aufgabe 5 (7 Punkte)

Beweisen Sie, dass

```
map f (concat xss) = concat (map (map f) xss)
```

wobei

```
map f []      =  []
map f (x:xs)  =  f x : map f xs

concat []        =  []
concat (xs:xss)  =  xs ++ concat xss
```

Sie dürfen das Lemma `map_append` benutzen:
`map f (xs ++ ys) = map f xs ++ map f ys`

Lösungsvorschlag

Beweis mit Induktion ueber `xss`.
Basis: Zu zeigen: `map f (concat []) = concat (map (map f) [])`

```
map f (concat [])
= map f []  -- def concat
= []  -- def map
concat (map (map f) [])
= concat []  -- def map
= []  -- def concat
```

Schritt: Zu zeigen: `map f (concat (xs:xss)) = concat (map (map f) (xs:xss))`

```
map f (concat (xs:xss))
= map f (xs ++ concat xss))  -- def concat
= map f xs ++ map f (concat xss)  -- Lemma map_append
= map f xs ++ concat (map (map f) xss)  -- IH
concat (map (map f) (xs:xss))
= concat (map f xs : map (map f) xss))  -- def map
= map f xs ++ concat(map (map f) xss)  -- def concat
```

Aufgabe 6 (6 Punkte)

Gegeben seien ein Typ `Nat` natürlicher Zahlen ($\{0, 1, 2, \ldots\}$) und eine Funktion `stutt :: [Nat] -> [Nat]`, die $[n_1, \ldots, n_k]$ auf

$$\big[\underbrace{n_1, \ldots, n_1}_{n_1\text{-mal}}, \ldots, \underbrace{n_k, \ldots, n_k}_{n_k\text{-mal}}\big]$$

abbildet. Beispiel: `stutt [2, 0, 3, 1] == [2, 2, 3, 3, 3, 1]`.

Stellen Sie eine *vollständige* Testsuite aus zwei der folgenden QuickCheck-Tests zusammen:

```
prop_stutt_length ns   = length (stutt ns) == sum ns
prop_stutt_contents ns = all (> 0) ns ==> nub (stutt ns) == nub ns
prop_stutt_null        = stutt [] == []
prop_stutt_single n    = stutt [n] == replicate n n
prop_stutt_cons n ns   = stutt (n : ns) == replicate n n ++ stutt ns
prop_stutt_reverse ns  = reverse (stutt ns) == stutt (reverse ns)
prop_stutt_distr ms ns = stutt ms ++ stutt ns == stutt (ms ++ ns)
```

Begründen Sie Ihre Antwort kurz. Für die Funktion `replicate :: Nat -> a -> [a]` gilt

$$\texttt{replicate}\ m\ x = \big[\underbrace{x, \ldots, x}_{m\text{-mal}}\big]$$

Lösungsvorschlag

Die Suite `prop_stutt_single` + `prop_stutt_distr` ist vollständig, weil es für jede Liste $[n_1, \ldots, n_n]$ möglich ist, eine vollständige Spezifikation der Funktion herzuleiten:

$$\begin{aligned}
&\texttt{stutt}\ [n_1, \ldots, n_k] \\
=\ &\texttt{stutt}\ [n_1] \mathbin{+\!\!+} \texttt{stutt}\ [n_2, \ldots, n_k] && \text{durch } \texttt{prop_stutt_distr} \\
&\ \vdots \\
=\ &\texttt{stutt}\ [n_1] \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \texttt{stutt}\ [n_k] && \text{durch } \texttt{prop_stutt_distr} \\
=\ &\texttt{replicate}\ n_1\ n_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \texttt{replicate}\ n_k\ n_k && \text{durch } \texttt{prop_stutt_single}\ (k\text{-mal}) \\
=\ &\big[\underbrace{n_1, \ldots, n_1}_{n_1\text{-mal}}\big] \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \big[\underbrace{n_k, \ldots, n_k}_{n_k\text{-mal}}\big] && \text{durch Definition von } \texttt{replicate}\ (k\text{-mal}) \\
=\ &\big[\underbrace{n_1, \ldots, n_1}_{n_1\text{-mal}}, \ldots, \underbrace{n_k, \ldots, n_k}_{n_k\text{-mal}}\big] && \text{durch Definition von } ++
\end{aligned}$$

Die Suites `prop_stutt_null` + `prop_stutt_cons` und `prop_stutt_distr` + `prop_stutt_cons` sind ebenfalls vollständig.

Aufgabe 7 (5 Punkte)

Werten Sie die folgenden Ausdrücke Schritt für Schritt mit Haskells Reduktionsstrategie vollständig aus:

1. `(\x -> (\y -> (1+2)+x)) 4 5`
2. `head (map (+1) twos)`
3. `f [] xxl`

wobei

```
head :: [a] -> a
head (x:_) = x

twos :: [Int]
twos  =  2 : twos

f :: [a] -> [a] -> Bool
f xs [] = False
f [] xs = True

xxl :: a
xxl = xxl
```

Unendlich lange Reduktionen bitte mit „. . . “ abbrechen, sobald Nichtterminierung erkennbar ist.

Lösungsvorschlag

1. `(\x -> (\y -> (1+2)+x)) 4 5 = (\y -> (1+2)+4) 5 = (1+2)+4 = 3+4 = 7`
2.
```
head (map (+1) twos)
  = head (map (+1) (2 : twos))
  = head ((+1) 2 : map (+1) twos)
  = (+1) 2
  = 2+1
  = 3
```
3. `f [] xxl = f [] xxl = ...`

Aufgabe 8 (4 Punkte)

Geben Sie eine endrekursive Variante der folgenden Funktion an.

```
fac :: Int -> Int
fac n | n > 0 = n * fac (n - 1)
      | otherwise = 1
```

Lösungsvorschlag

```
fac :: Int -> Int
fac n = go 1 n
    where go acc n | n > 0 = go (n * acc) (n - 1)
                   | otherwise = acc
```

Aufgabe 9 (7 Punkte)

Definieren Sie eine IO-Aktion `vokalZaehler :: IO ()`, die Strings zeilenweise vom Benutzer entgegennimmt (mittels `getLine :: IO String`) und die Anzahl der kleingeschriebenen Vokale (`'a','e','i','o','u'`) zählt. Ihr Programm soll dabei nicht terminieren und nach jeder eingegebenen Zeile die **Gesamtzahl** der gezählten Vokale in allen bisherigen Eingaben ausgeben (mit Hilfe der Funktionen `putStrLn :: String -> IO ()` und `show :: Show a => a -> String`). Beispiel (Benutzereingaben sind ***kursiv*** dargestellt):

```
Hallo Welt!
#Vokale: 3
Wie geht's dir?
#Vokale: 7
brb
#Vokale: 7
lol
#Vokale: 8
aAaAaA
#Vokale: 11
aeiou
#Vokale: 16

#Vokale: 16
```

Lösungsvorschlag

```
vokalZaehler :: IO ()
vokalZaehler = count 0
  where
    count :: Int -> IO ()
    count n = do
      s <- getLine
      let nNew = n + length (filter (`elem` "aeiou") s)
      putStrLn ("#Vokale: " ++ show nNew)
      count nNew
```

Die Verwendung von `Integer` statt `Int` und `genericLength` statt `length` ist zwar noch korrekter aber hier nicht gefordert.