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

Wintersemester 2012/13
Abschlussklausur
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]`

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.

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']
```

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)
```

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`

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

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

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.

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
```

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
```
