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

Wintersemester 2013/14
Lösungsblatt Endklausur
21. Februar 2014

Einführung in die Informatik 2

Name	Vorname	Studiengang	Matrikelnummer
....	....	☐ Diplom ☐ Inform. ☐ Bachelor ☐ BioInf. ☐ Lehramt ☐ Mathe.	....

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	Σ	Korrektor
Erstkorrektur										
Zweitkorrektur										

Aufgabe 1 (6 Punkte)

Geben Sie den allgemeinsten Typen der folgenden Ausdrücke an:

1. `map reverse`
2. `map (: [])`
3. `\f (x,y) -> f x y`
4. `[(:[])]`
5. `zipWith (+)`

Weiterhin:

6. Warum ist es oft sinnvoller `null` *xs* statt *xs* `== []` zu schreiben?

Lösungsvorschlag

1. `map reverse :: [[a]] -> [[a]]`
2. `map (: []) :: [a] -> [[a]]`
3. `\f (x,y) -> f x y :: (a -> b -> c) -> (a,b) -> c`
4. `[a -> [a]]`
5. `Num a => [a] -> [a] -> [a]`
6. `xs == []` ist nur typkorrekt, wenn der Typ der Elemente von *xs* eine Instanz von `Eq` ist. Das heißt, die Funktion `\xs -> xs == []` hat den Typ `Eq a => [a] -> Bool`. Dieser ist spezieller als der von `null`.

Aufgabe 2 (6 Punkte)

Gegeben sei ein Datentyp zur Darstellung von Knoten in einem Dateisystem:

```
data Node = File String | Dir String [Node]
```

Ein Knoten ist also entweder eine *normale Datei* (`File`) oder ein *Verzeichnis* (`Dir`). Knoten haben einen Namen (von Typ `String`). Zudem enthalten Verzeichnisse eine Liste von Knoten. Ein komplettes Dateisystem lässt sich als Liste von Knoten darstellen:

```
type FileSys = [Node]
```

Beispiel:

```
myFileSys =
  [Dir "Applications"
     [Dir "Aquamacs.app" [File "Info.plist"],
      Dir "Scribus.app" [File "Info.plist"]],
   Dir "Library"
     [Dir "Automator" [],
      Dir "Logs" []],
   Dir "Users"
     [Dir "smith"
        [Dir "bugs" [File "Scratch.hs"],
         File "Scratch.hs"]]]
```

Implementieren Sie eine Funktion `removeFiles :: String -> FileSys -> FileSys`, die alle normalen Dateien mit dem angegebenen Namen – auch in Unterverzeichnissen – löscht (und sonst keine Änderungen vornimmt).

Lösungsvorschlag

```
removeFiles :: String -> FileSys -> FileSys
removeFiles _ [] = []
removeFiles name (File name' : fs) =
  (if name' == name then [] else [File name']) ++ removeFiles name fs
removeFiles name (Dir name' fs' : fs) =
  Dir name' (removeFiles name fs') : removeFiles name fs
```

Aufgabe 3 (5 Punkte)

Eine *Collatz*-Folge ist eine spezielle Folge von natürlichen Zahlen. Das Folgenglied c_{k+1} wird aus dem vorherigen Element c_k wie folgt berechnet:

$$c_{k+1} = \begin{cases} c_k/2 & \text{falls } c_k \text{ gerade} \\ 3 \cdot c_k + 1 & \text{falls } c_k \text{ ungerade} \end{cases}$$

Das erste Folgenglied c_0 kann eine beliebige natürliche Zahl $n > 0$ sein. Die Folge endet, wenn die Zahl 1 erreicht wird.
Beispiel: Sei $c_0 = 6$. Die gesamte Folge ist dann $6, 3, 10, 5, 16, 8, 4, 2, 1$.

1. Definieren Sie eine Funktion `collatz :: Integer -> [Integer]`, die für einen gegebenen Startwert die zugehörige Collatz-Folge als Liste zurückgibt.

2. Implementieren Sie die Funktion `unfold :: (a -> Maybe a) -> a -> [a]`. Für einen Aufruf `unfold` f a soll die Funktion wiederholt f auf a anwenden und bei einem Ergebnis von `Nothing` abbrechen. Die Rückgabe ist die Liste sämtlicher Zwischenergebnisse, einschließich des Startwerts a.

   Beispiel: Für $f\ a_0 =$ `Just` a_1, $f\ a_1 =$ `Just` a_2, ..., $f\ a_n =$ `Nothing` gilt:

   $$\texttt{unfold}\ f\ a_0 = [a_0, a_1, \ldots, a_n]$$

3. Finden Sie eine Funktion `next`, so dass `collatz` $n =$ `unfold next` n für $n > 0$ gilt.

Lösungsvorschlag

```
collatz :: Integer -> [Integer]
collatz 1 = [1]
collatz n | even n = n : collatz (n `div` 2)
          | otherwise = n : collatz (n * 3 + 1)

unfold :: (a -> Maybe a) -> a -> [a]
unfold f a = a : rest (f a)
  where rest Nothing = []
        rest (Just x) = unfold f x

next :: Integer -> Maybe Integer
next 1 = Nothing
next n | even n = Just (n `div` 2)
       | otherwise = Just (n * 3 + 1)
```

Aufgabe 4 (3 Punkte)

Gegeben sei eine Funktion `factorize :: Int -> [Int]`, die eine positive Zahl n als Argument nimmt und eine Faktorisierung von n zurückgibt. Die Faktoren müssen nicht unbedingt Primzahlen sein. Sie müssen aber positiv sein, und 1 darf nicht als Faktor erscheinen. Beispiele:

```
factorize (-666)      -- nicht definiert, beliebig
factorize 1 == []
factorize 2 == [2]
factorize 3 == [3]
factorize 4 == [2, 2] oder [4]
factorize 5 == [5]
factorize 6 == [2, 3] oder [3, 2] oder [6]
```

Wie die Beispiele zeigen darf die Funktion eine triviale, einelementige Liste von Faktoren erzeugen, auch wenn das Argument keine Primzahl ist.

Schreiben Sie eine *korrekte* und *vollständige* QuickCheck-Testsuite für diese Funktion. Begründen Sie kurz Ihre Antwort.

Lösungsvorschlag

```
prop_prod n = n > 0 ==> product (factorize n) == n
prop_geq2 n = n > 0 ==> all (>= 2) (factorize n)
```

Aufgabe 5 (6 Punkte)

Gegeben seien folgende Definitionen:

```
data T a = Tip a | Node (T a) (T a)

flip (Tip a) = Tip a                               -- flip_Tip
flip (Node t1 t2) = Node (flip t2) (flip t1)       -- flip_Node

flat (Tip a) = [a]                                 -- flat_Tip
flat (Node t1 t2) = flat t1 ++ flat t2             -- flat_Node

reverse [] = []                                    -- rev_Nil
reverse (x:xs) = reverse xs ++ [x]                 -- rev_Cons

[] ++ ys = ys                                      -- app_Nil
(x:xs) ++ ys = x : (xs ++ ys)                      -- app_Cons
```

Beweisen Sie:

```
flat (flip t) = reverse (flat t)
```

Sie dürfen dabei das folgende Lemma verwenden:

```
reverse (xs ++ ys) = reverse ys ++ reverse xs -- rev_Dist
```

Lösungsvorschlag

Beweis mit Induktion über `t`.

Basis. Zu zeigen: `flat (flip (Tip a)) = reverse (flat (Tip a))`

```
flat (flip (Tip a))
= flat (Tip a)   -- flip_Tip
= [a]            -- flat_Tip

reverse (flat (Tip a))
= reverse [a]         -- flat_Tip
= reverse [] ++ [a]   -- rev_Cons
= [] ++ [a]           --  app_Nil
= [a]
```

Schritt. Zu zeigen: `flat (flip (Node t1 t2)) = reverse (flat (Node t1 t2))`

IH1: `flat (flip t1) = reverse (flat t1)`

IH2: `flat (flip t2) = reverse (flat t2)`

```
flat (flip (Node t1 t2))
= flat (Node (flip t2) (flip t1))          -- flip_Node
= flat (flip t2) ++ flat (flip t1)         -- flat_Node
= reverse (flat t2) ++ reverse (flat t1)   -- IH1 und IH2

reverse (flat (Node t1 t2))
= reverse (flat t1 ++ flat t2)             -- flat_Node
= reverse (flat t2) ++ reverse (flat t1)   -- rev_Dist
```

Aufgabe 6 (6 Punkte)

Wir betrachten das Streichholzspiel für zwei Spieler. Anfangs liegen 10 Streichhölzer auf dem Tisch. Jetzt nehmen die Spieler abwechselnd Streichhölzer vom Tisch (mindestens 1 und höchstens 5). Gewonnen hat der Spieler, der das letzte Streichholz nimmt.

Definieren Sie eine IO-Aktion `match :: IO ()`, die dieses Spiel implementiert. Vor jedem Zug sollen die Anzahl der verbleibenden Streichhölzer und der Spieler, der am Zug ist, angezeigt werden. Hat ein Spieler gewonnen, so soll das Programm dies anzeigen und sich selbst beenden. Das Program soll sicherstellen, dass der Spieler nur eine gültige Anzahl an Streichhölzern nimmt. Sind nicht mehr ausreichend viele Streichhölzer vorhanden, so werden alle Streichhölzer genommen.

Sie können insbesondere die Funktionen `putStrLn :: String -> IO ()` zum Ausgeben und `readLn :: Read a => IO a` zum Lesen der Eingabe verwenden.

```
Streichhoelzer: 10. Spieler 1?
4
Streichhoelzer: 6. Spieler 2?
6
Eingabe muss zwischen 1 und 5 liegen.
5
Streichhoelzer: 1. Spieler 1?
1
Spieler 1 gewinnt!
```

Lösungsvorschlag

```
match :: IO ()
match = play 10 $ cycle [1,2]

readNum :: IO Int
readNum = do
    x <- readLn
    if 1 <= x && x <= 5 then
        return x
    else do
        putStrLn "Eingabe muss zwischen 1 und 5 liegen."
        readNum

play :: Show a => Int -> [a] -> IO ()
play n (p:ps)= do
    putStrLn $ "Streichhoelzer: "
        ++ show n ++ ". Spieler " ++ show p ++ "?"
    m <- readNum
    if n <= m then
        putStrLn $ "Spieler " ++ show p ++ " gewinnt!"
    else
        play (n - m) ps
```

Aufgabe 7 (3 Punkte)

Geben Sie eine *endrekursive* Implementation der Funktion `sum :: [Integer] -> Integer` an, die die Summe der Elemente der übergebenen Liste berechnet. Außer den arithmetischen Basisoperationen dürfen keine vordefinierte Funktionen verwendet werden.

Lösungsvorschlag

```
sum' :: [Integer] -> Integer
sum' = go 0
  where go acc [] = acc
        go acc (x:xs) = go (acc + x) xs
```

Aufgabe 8 (5 Punkte)

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

1. `null nats && False`
2. `(\b -> False && b) (nats == nats)`
3. `h nats [1]`

wobei

```
nats :: [Int]
nats = 0 : map (1+) nats

null :: [a] -> Bool
null [] = True
null _  = False

(&&) :: Bool -> Bool -> Bool
True  && b = b
False && _ = False

h :: [a] -> [a] -> a
h (x:xs) _ = x
h _ (y:ys) = y
```

Brechen Sie unendliche Reduktionen mit „...“ ab, sobald Nichtterminierung erkennbar ist.

Lösungsvorschlag

```
null nats && False
= null (0 : map (1+) nats) && False
= False && False
= False

(\b -> False && b) (nats == nats)
= False && (nats == nats)
= False

h nats [1]
= h (0 : map (1+) nats) [1]
= 0
```