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

Wintersemester 2013/14
Lösungsblatt Wiederholungsklausur
24. März 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 (5 Punkte)

Geben Sie den allgemeinsten Typen der folgenden Ausdrücke an:

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

Sei `x :: Integer`. Ist der folgende Ausdruck typkorrekt? Wenn nein, geben Sie eine kurze Begründung an. Wenn ja, geben Sie den Typ an.

5. `1 : (if False then x else [2, 3])`

Lösungsvorschlag

1. `[Bool] -> [Bool]`
2. `(a -> b) -> (c -> a) -> (c -> b)`
3. `a -> (a -> b) -> b`
4. `[Bool]`
5. Der Ausdruck ist nicht typkorrekt, weil in den beiden Branches verschiedene Typen stehen.

Aufgabe 2 (5 Punkte)

Gegeben sei die Typklasse `Size` mit folgender Definition:

```
class Size a where
  size :: a -> Float
```

1. Definieren Sie einen Datentyp `Shape`, der die Fälle `Square` und `Circle` enthält. Ein `Square` wird durch eine Seitenlänge, ein `Circle` durch einen Radius (jeweils vom Typ `Float`) charakterisiert. Registrieren Sie `Shape` als Instanz von `Size`, so dass der Flächeninhalt zurückgegeben wird.

   Beispiele:

   ```
   size (Square 3.0) = 9.0
   size (Circle 1.0) = pi
   ```

   Hinweis: Sie dürfen die Konstante `pi :: Float` verwenden. Sie dürfen annehmen, dass sämtliche vorkommenden Zahlen positiv sind.

2. Registrieren Sie den Typ `[a]` als Instanz von `Size`. Die Instanz soll die Summe der Größen aller Elemente ermitteln.

Lösungsvorschlag

1. ```
   data Shape = Square Float | Circle Float

   instance Size Shape where
     size (Square l) = l * l
     size (Circle r) = pi * r * r
   ```

2. ```
   instance Size a => Size [a] where
     size = sum . map size
   ```

Aufgabe 3 (7 Punkte)

Definieren Sie eine Funktion `lups :: Ord a => [a] -> [a]` die eine längste, streng monoton aufsteigende (ununterbrochene) Teilliste der Eingabe zurückgibt.
Beispiele:

```
lups [] = []
lups [2,2,1] = [2]   oder   [1]
lups [2,5,3,6,8] = [3,6,8]
lups [3,7,2,8] = [3,7]   oder   [2,8]
```

Lösungsvorschlag

```
ups :: Ord a => [a] -> [a] -> [[a]]
ups [] ys = [reverse ys]
ups (x:xs) [] = ups xs [x]
ups (x:xs) (y:ys)
  | x > y      = ups xs (x:y:ys)
  | otherwise = reverse (y:ys) : ups xs [x]

longest :: [[a]] -> [a]
longest [] = []
longest (xs:xss) = if length xs > length ys then xs else ys
  where ys = longest xss

lups :: Ord a => [a] -> [a]
lups xs = longest (ups xs [])
```

Aufgabe 4 (4 Punkte)

In dieser Aufgabe stellen wir Mengen durch Listen dar. Wie üblich haben Duplikate und Reihenfolge keinen Einfluss auf die dargestellte Menge. Zum Beispiel entsprechen die Listen `[1, 2]`, `[1, 2, 1]` und `[2, 2, 1]` alle der mathematischen Menge $\{1, 2\}$.

Betrachten Sie die Funktion `takeAny :: [a] -> (a, [a])`. Die Eingabe xs wird als Menge A interpretiert. Die Funktion wählt ein beliebiges Element a aus A aus und gibt ein Paar (a, ys) zurück. Dabei ist ys eine Liste, die die Menge $A \setminus \{a\}$ darstellt.

Ist die Menge A leer, so darf sich die Funktion beliebig verhalten (z.B. nicht terminieren).

Beispiele:

```
takeAny []                  -- nicht definiert, beliebig
takeAny [1]      = (1,[])
takeAny [1,1,1]  = (1,[])
takeAny [1,2]    = (1,[2]) oder (2,[1]) oder (1,[2,2]) oder ...
takeAny [1,2,2]  = (1,[2]) oder (2,[1]) oder (1,[2,2]) oder ...
takeAny [1,2,3]  = (1,[2,3]) oder (3,[2,1,2]) oder ...
```

Wie die Beispiele zeigen, darf die Funktion Duplikate erzeugen oder entfernen oder die Reihenfolge ändern.

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

Lösungsvorschlag

```
prop_takeAny xs =
  xs /= [] ==>
    not (y `elem` ys) &&
    sort (nub xs) == sort (nub (y : ys))
  where (y, ys) = takeAny xs
```

Aufgabe 5 (6 Punkte)

Gegeben seien folgende Definitionen:

```
map f [] = []                        -- map_Nil
map f (x:xs) = f x : map f xs        -- map_Cons

filter p [] = []                     -- filter_Nil
filter p (x:xs)
  | p x       = x : filter p xs      -- filter_ConsT
  | otherwise = filter p xs          -- filter_ConsF
```

Beweisen Sie:

```
map f (filter (p . f) zs) = filter p (map f zs)
```

Falls eine Fallunterscheidung notwendig ist, brauchen Sie für diese nur einen der beiden Fälle zu beweisen. Geben Sie in jedem Schritt die verwendete Gleichung an.

Lösungsvorschlag

Beweis mit Induktion über `zs`.
Basis. Zu zeigen: `map f (filter (p . f) []) = filter p (map f [])`

```
map f (filter (p . f) [])          filter p (map f [])
= map f []      -- filter_Nil      = filter p []   -- map_Nil
= []            -- map_Nil         = []            -- filter_Nil
```

Schritt.
Zu zeigen: `map f (filter (p . f) (z:zs)) = filter p (map f (z:zs))`
IH: `map f (filter (p . f) zs) = filter p (map f zs)`
Beweis mit Fallunterscheidung:
Fall `p (f z) = True`:

```
map f (filter (p . f) (z:zs))
= map f (z : filter (p . f) zs)      -- filter_ConsT
= f z : map f (filter (p . f) zs)    -- map_Cons
= f z : filter p (map f zs)          -- IH

filter p (map f (z:zs))
= filter p (f z : map f zs)          -- map_Cons
= f z : filter p (map f zs)          -- filter_ConsT
```

Fall `p (f z) = False`:

```
map f (filter (p . f) (z:zs))
= map f (filter (p . f) zs)          -- filter_ConsF
= filter p (map f zs)                -- IH

filter p (map f (z:zs))
= filter p (f z : map f zs)          -- map_Cons
= filter p (map f zs)                -- filter_ConsF
```

Aufgabe 6 (5 Punkte)

1. Implementieren Sie eine Funktion `ioSeq :: [IO a] -> IO [a]`, die eine Liste von IO-Aktionen bekommt und daraus eine einzige IO-Aktion macht, die alle diese IO-Aktionen ausführt und die Liste ihrer Ergebnisse zurückgibt.

2. Implementieren Sie eine IO-Aktion `fillForm :: [String] -> IO [String]`. Die Eingabe ist eine Liste von Formularfeldern. Der Benutzer soll nach jedem Feld gefragt werden und der Rückgabewert soll die Liste der Antworten des Benutzers sein. Zum Anzeigen können sie die IO-Aktion `putStrLn :: String -> IO ()`, zum Einlesen die IO-Aktion `getLine :: IO String` verwenden.

   Beispiel. Aufruf von `fillForm ["Name", "Beruf"]`, Benutzereingaben sind kursiv:

   ```
   Name?
   Ford Prefect
   Beruf?
   Journalist
   ```

   Das Ergebnis ist dann `["Ford Prefect", "Journalist"]`.

Lösungsvorschlag

```
ioSeq :: [IO a] -> IO [a]
ioSeq [] = return []
ioSeq (x:xs) = do
    y <- x
    ys <- ioSeq xs
    return (y : ys)

fillForm :: [String] -> IO [String]
fillForm = ioSeq . map ask
  where
    ask s = do
        putStrLn (s ++ "?")
        getLine
```

Aufgabe 7 (4 Punkte)

Beantworten Sie zu jeder Teilaufgabe: Sind die gegebenen Definitionen äquivalent, d.h. haben sie den gleichen Typ und liefern bei gleicher Eingabe die gleichen Ergebnisse? Begründen Sie kurz.

1. ```
   f1 x = x + sqrt x
   f2 = \x -> x + (\x -> sqrt x) 2
   ```

2. ```
   f1 x = x + sqrt x
   f3 x = f
       where f x = x + sqrt x
   ```

3. ```
   g1 x xs = filter (> x) xs
   g2 xs = \x -> filter (> x) xs
   ```

4. ```
   g1 x xs = filter (> x) xs
   g3 x = filter (\y -> y > x)
   ```

Lösungsvorschlag

1. Nicht äquivalent: `f1` berechnet $x \mapsto x + \sqrt{x}$ und `f2` berechnet $x \mapsto x + \sqrt{2}$.
2. Nicht äquivalent: `f3` hat ein zusätzliches, ungenutztes Argument.
3. Nicht äquivalent: die Argumente sind vertauscht.
4. Äquivalent durch Partial Application und Sections.

Aufgabe 8 (4 Punkte)

Gegeben sei folgende Typdefinition für Maps:

```
type Map a b = a -> Maybe b
```

Ein Schlüssel k ist in einer Map m einem Wert v zugeordnet, wenn der Aufruf $m\ k$ den Wert `Just` v liefert.

1. Schreiben Sie die Funktion `update :: Eq a => (a, b) -> Map a b -> Map a b`. Ein Aufruf von `update` $(k, v)\ m$ soll eine neue Map zurückgeben, in der der Schlüssel k dem Wert v zugeordnet ist. Alle anderen Zuordnungen sollen unverändert bleiben.

2. Implementieren Sie die Funktion `mapList :: Map a b -> [a] -> [b]`, die Elemente der Eingabeliste durch die zugeordneten Werte in der übergebenen Map ersetzt werden. Ist ein Element der Liste keinem Wert zugeordnet, soll es entfallen.

3. Sei folgende Map `m` gegeben:

   ```
   m x = if x < 10 then Just 0 else Nothing
   ```

   Terminiert die vollständige Auswertung von `mapList m [0..]`? Begründen Sie Ihre Antwort kurz.

Lösungsvorschlag

1. ```
   update :: Eq a => (a, b) -> Map a b -> Map a b
   update (k, v) m = \x -> if x == k then Just v else m x
   ```

   alternative Schreibweise:

   ```
   update (k, v) m x = if x == k then Just v else m x
   ```

2. ```
   mapList :: Map a b -> [a] -> [b]
   mapList m [] = []
   mapList m (x:xs) =
     case m x of
       Just y -> y : rest
       Nothing -> rest
     where rest = mapList m xs
   ```

3. Nein, die Auswertung terminiert nicht. Obwohl die Ausgabeliste endlich sein müsste (denn es sind nur endlich viele Elemente < 10 in der Eingabeliste), kann das vom Laufzeitsystem nicht erkannt werden.