Technische Universität München
Fakultät für Informatik
Prof. Tobias Nipkow, Ph.D.
Manuel Eberl, Lars Hupel, Lars Noschinski

Wintersemester 2014/15
Lösungsblatt Endklausur
13. Februar 2015

Einführung in die Informatik 2

Name	Vorname	Studiengang	Matrikelnummer
.....	.....	☐ Bachelor ☐ Inform. ☐ Master ☐ WInfo. ☐ 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 handschriftlich beschrifteten 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. `filter isNothing` (wobei `isNothing :: Maybe a -> Bool`)
2. `[getLine]`
3. `\f (a, b) -> (f a, b)`
4. `concat . map show`
5. Betrachten Sie den Ausdruck `head x y`. Ist es möglich, dass dieser Ausdruck typkorrekt ist? Wenn ja, geben Sie den allgemeinsten Typ von `x` an.

Lösungsvorschlag

1. `[Maybe a] -> [Maybe a]`
2. `[IO String]`
3. `(a -> b) -> (a,c) -> (b,c)`
4. `Show a => [a] -> String`
5. `[a -> b]`

Aufgabe 2 (6 Punkte)

Der Datentyp `Test` modelliert die Ergebnisse von Tests der Hausaufgaben eines Studenten.

```
data Test = Essential String Bool |
            NonEssential String Bool |
            Group String [Test]
```

Ein `Essential` ist ein Test, der für die Bewertung relevant ist; ein `NonEssential` geht nicht in die Bewertung ein. Der `Bool` gibt jeweils an, ob der jeweilige Test bestanden ist. Eine `Group` enthält mehrere Tests. Die `String`s sind die Namen des jeweiligen Tests bzw. der Testgruppe. Die folgende Liste von Tests dient im Rest dieser Aufgabe als Beispiel:

```
testResults =
  [ Essential "A" True,
    Essential "B" False,
    Group "Foo"
      [ NonEssential "C" True,
        Essential "D" True,
        Group "Bar" [Essential "E" True, NonEssential "F" False]
      ]
  ]
```

1. Schreiben Sie eine Funktion `countPassedEssential :: [Test] -> Int`, die die Anzahl der bestandenen essentiellen Tests in der übergebenen Liste zählt.

   Beispiel: `countPassedEssential testResults == 3`

2. Schreiben Sie eine Funktion `showFailedTests :: [Test] -> [String]`, die eine Liste der Namen aller nicht bestandenen Tests (essentiell *und* nichtessentiell) in beliebiger Reihenfolge zurückgibt. Die Namen der übergeordneten Gruppen sollen vorne angehängt werden, abgetrennt durch Schrägstriche (`"/"`).

   Beispiel: `showFailedTests testResults = ["B", "Foo/Bar/F"]`

Lösungsvorschlag

```
countPassedEssential tests = sum (map aux tests)
  where aux (Essential _ b)    = if b then 1 else 0
        aux (NonEssential _ _) = 0
        aux (Group _ tests)    = countPassedEssential tests

showFailedTests tests = concatMap aux tests
  where aux (Essential s False)    = [s]
        aux (NonEssential s False) = [s]
        aux (Group s ts)           =
            [s ++ "/" ++ s' | s' <- showFailedTests ts]
        aux _ = []
```

Aufgabe 3 (6 Punkte)

1. Schreiben Sie Funktionen `safeHead :: [a] -> Maybe a` und `safeLast :: [a] -> Maybe a`, die, falls möglich, das erste bzw. letzte Element einer Liste zurückgeben (ansonsten `Nothing`). Verwenden Sie keine vordefinierten Funktionen.

2. Definieren Sie `safeHead` und `safeLast` erneut. Verwenden Sie `foldr`, aber keine der folgenden Techniken: Listenkomprehensionen, Rekursion, Pattern Matching auf Listen und vordefinierte Funktionen (außer `foldr`).

   Hinweis: `foldr` ist wie folgt definiert:

   ```
   foldr :: (a -> b -> b) -> b -> [a] -> b
   foldr f a [] = a
   foldr f a (x : xs) = f x (foldr f a xs)
   ```

3. Schreiben Sie die Funktion `select :: Eq a => a -> [(a, b)] -> [b]`, die zu einem Wert x und einer Liste von Paaren (y, z) alle Werte z liefert, bei denen y gleich x ist. Verwenden Sie `map` und `filter`, aber keine der folgenden Techniken: Listenkomprehensionen, Rekursion und Pattern Matching auf Listen.

   Beispiele:

   ```
   select 'a' [('b',1), ('a',3), ('c',4), ('a',2)] == [3, 2]
   select 'd' [('b',1), ('a',3), ('c',4), ('a',2)] == []
   ```

Lösungsvorschlag

```
safeHead [] = Nothing
safeHead (x : _) = Just x

safeLast [] = Nothing
safeLast [x] = Just x
safeLast (_ : xs) = safeLast xs

safeHead' = foldr (\h _ -> Just h) Nothing

safeLast' = foldr f Nothing
  where f l Nothing = Just l
        f _ l = l

select x = map snd . filter (\(a, b) -> x == a)
```

Aufgabe 4 (4 Punkte)

Schreiben Sie für die unten spezifizierte Funktion `occs` einen oder mehrere QuickCheck-Tests, die zusammen eine vollständige Testsuite bilden. Eine Testsuite ist vollständig, wenn jede korrekte Implementierung jeden Test besteht und es für jede inkorrekte Implementierung mindestens einen Test gibt, der für geeignete Testparameter fehlschlägt.
Die Funktion `occs :: Eq a => [a] -> [(a, Int)]` zählt, wie häufig jedes Element in einer Liste vorkommt. Die ausgegebene Liste muss dabei absteigend nach Anzahl der Vorkommen sortiert sein und darf keine Duplikate enthalten. Elemente mit gleicher Anzahl von Vorkommen dürfen in beliebiger Reihenfolge ausgegeben werden. Elemente, die in der Eingabe nicht vorkommen, sind in der Ausgabe nicht erlaubt.

Beispiele für korrektes Verhalten:

```
occs "mississippi" ==
  [('i', 4), ('s', 4), ('p', 2), ('m', 1)]
occs "mississippi" ==
  [('s', 4), ('i', 4), ('p', 2), ('m', 1)]
```

Beispiele für inkorrektes Verhalten:

```
occs "mississippi" ==
  [('s', 4), ('i', 4), ('m', 1), ('p', 2)]
occs "mississippi" ==
  [('s', 4), ('i', 4), ('m', 1), ('p', 2), ('x', 0)]
occs "mississippi" ==
  [('s', 4), ('s', 4), ('i', 4), ('p', 2), ('m', 1)]
```

Hinweis: Sie sollen *nicht* die Funktion `occs` implementieren.

Lösungsvorschlag

```
prop_occs1 :: [Int] -> Bool
prop_occs1 xs = all (\(x, n) -> n > 0 && count x xs == n) ys
  where ys = occs xs
        count x xs = length (filter (== x) xs)

prop_occs2 :: [Int] -> Bool
prop_occs2 xs = nub ys == ys
  where ys = map fst (occs xs)

prop_occs3 :: [Int] -> Bool
prop_occs3 xs = sort ys == reverse ys
  where ys = map snd (occs xs)
```

Aufgabe 5 (6 Punkte)

Gegeben seien folgende Definitionen:

```
double [] = []                                -- def double
double (x : xs) = x : x : double xs           -- def double

d x xs = x : x : xs                           -- def d

foldr f a [] = a                              -- def foldr
foldr f a (x : xs) = f x (foldr f a xs)       -- def foldr

[] ++ ys = ys                                 -- def app
(x:xs) ++ ys = x : (xs ++ ys)                 -- def app
```

Beweisen Sie:

```
foldr d xs ys = double ys ++ xs
```

Hinweis: Sie brauchen nicht exakt die Syntax des Werkzeugs `cyp` aus den Übungen verwenden, aber es muss in Ihrem Beweis klar gesagt werden, welches Beweisprinzip verwendet wird, was zu zeigen ist und was angenommen werden darf. Es darf jeweils nur ein Schritt auf einmal gemacht werden, wobei die verwendete Gleichung angegeben werden muss.

Lösungsvorschlag

Beweis per Induktion über `ys`.
Fall `Nil`.
zu zeigen: `foldr d xs [] = double [] ++ xs`

```
                foldr d xs []
(def foldr)  = xs

                double [] ++ xs
(def double) = [] ++ xs
(def app)    = xs
```

Fall `Cons`.
zu zeigen: `foldr d xs (y:ys) = double (y:ys) ++ xs`
IH: `foldr d xs ys = double ys ++ xs`

```
                foldr d xs (y:ys)
(def foldr)  = d y (foldr d xs ys)
(def d)      = y : y : foldr d xs ys
(IH)         = y : y : (double ys ++ xs)

                double (y:ys) ++ xs
(def double) = (y : y : double ys) ++ xs
(def app)    = y : ((y : double ys) ++ xs)
(def app)    = y : y : (double ys ++ xs)
```

Aufgabe 6 (6 Punkte)

Schreiben Sie ein IO-Programm `mathTrainer :: IO ()`, welches dem Benutzer 5 zufällige Rechenaufgaben stellt und das eingegebene Ergebnis auf Korrektheit prüft. Am Ende soll die Anzahl der Fehler ausgegeben werden. Sie dürfen annehmen, dass der Benutzer immer gültige Ganzzahlen eingibt. Rechenaufgaben haben die Form $x + y$, wobei x, y zufällig gewählte Zahlen im Intervall $(0, 1000)$ sind, d.h. $0 \leq x, y \leq 1000$.
Folgende Funktionen könnten nützlich sein:

```
-- Ausgabe von Text
putStr :: String -> IO ()
-- Lesen der Eingabe
readLn :: Read a => IO a
-- Erzeugen einer Zufallszahl im angegebenen Intervall
randomRIO :: (Int, Int) -> IO Int
```

Beispiel (Antworten des Nutzers kursiv angegeben):

```
143 + 121 = 264
794 + 406 = 1200
101 + 103 = 205
505 + 424 = 929
626 + 15 = 614
2 Fehler!
```

Lösungsvorschlag

```
mathTrainer :: IO ()
mathTrainer = go 5 0
  where go 0 n = putStrLn (show n ++ " Fehler!")
        go k n = do
          x <- randomRIO (0, 1000)
          y <- randomRIO (0, 1000)
          putStr $ show x ++ " + " ++ show y ++ " = "
          answer <- readLn
          go (k-1) (if answer == x + y then n else n + 1)
```

Genaugenommen fehlt noch eine Typannotation, die `x` und `y` auf `Int` einschränkt. Da in der Angabe aber vorgegeben ist, dass `randomRIO` auf `Int` eingeschränkt wird, ist das hier unnötig.

Aufgabe 7 (3 Punkte)

1. Geben Sie zu den beiden folgenden Ausdrücken je einen λ-Ausdruck mit der gleichen Semantik an, der keine Section enthält:

   (a) `(++ [1])`

   (b) `(++) [1]`

2. Schreiben Sie einen Ausdruck, der die gleiche Funktion beschreibt wie

   ```
   \xs ys -> reverse xs ++ ys
   ```

   Sie dürfen dabei weder λ-Ausdrücke verwenden noch eigene Hilfsfunktionen definieren. Die Verwendung der Funktionskomposition `(.) :: (a -> b) -> (c -> a) -> (c -> b)` ist erlaubt.

Lösungsvorschlag

1. (a) `\xs -> xs ++ [1]`

   (b) `\xs -> [1] ++ xs`

2. `(++) . reverse`

Aufgabe 8 (4 Punkte)

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

   (a) `(\y -> \x -> y x) (\z -> z) True`

   (b) `f [] fs`

2. Finden Sie definierte Ausdrücke a und b, so dass der Ausdruck `f` a b nicht definiert ist, weil seine Auswertung fehlschlägt.

Gegeben seien die folgenden Definitionen:

```
f :: [a] -> [Bool] -> Bool
f _ (y:ys) = y
f [] _ = True

fs :: [Bool]
fs = False : fs
```

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

Lösungsvorschlag

1.
```
  (\y -> \x -> y x) (\z -> z) True
= (\x -> (\z -> z) x) True
= (\z -> z) True
= True

  f [] fs
= f [] (False : fs)
= False
```

2. Beispiel: `f [True] []`. Allgemein: Wenn a nicht leer und b leer ist.