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

Wintersemester 2014/15
Lösungsblatt Wiederholungsklausur
30. März 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 Typ der folgenden Ausdrücke an:

1. `map (elem 'a')`
2. `reverse . head`
3. `(\last -> last) last`
4. `\f -> (True : map f [])`
5. Geben Sie eine Funktion mit dem Typ `a -> (a -> (b,b)) -> b` an.

Lösungsvorschlag

1. `[String] -> [Bool]`
2. `[[a]] -> [a]`
3. `[a] -> a`
4. `(a -> Bool) -> [Bool]`
5. `f x g = fst (g x)`

Aufgabe 2 (5 Punkte)

Schreiben Sie einen oder mehrere QuickCheck-Tests für die unten spezifizierte Funktion `sortP`. Zusammen sollen die Tests eine vollständige Testsuite bilden, d.h.: Jede korrekte Implementierung von `sortP` besteht jeden Test und für jede inkorrekte Implementierung gibt es mindestens einen Test, der für geeignete Testparameter fehlschlägt.
Die Funktion `sortP :: Ord a => [(a,b)] -> [(a,b)]` sortiert eine Liste aufsteigend nach dem ersten Element der Tupel. Tupel mit dem gleichen ersten Element dürfen in beliebiger Reihenfolge vorkommen.

Beispiele für korrektes Verhalten:

```
sortP [(3,'a'), (1,'b'), (2,'c')] = [(1,'b'), (2,'c'), (3,'a')]
sortP [(3,'a'), (1,'b'), (3,'c')] = [(1,'b'), (3,'c'), (3,'a')]
sortP [(3,'a'), (1,'b'), (3,'c')] = [(1,'b'), (3,'a'), (3,'c')]
```

Beispiele für inkorrektes Verhalten:

```
sortP [(3,'a'), (1,'b'), (2,'c')] = [(1,'a'), (2,'b'), (3,'c')]
sortP [(3,'a'), (1,'b'), (3,'c')] = [(1,'b'), (3,'a'), (3,'a')]
sortP [(3,'a'), (1,'b'), (2,'c')] = [(3,'a'), (2,'c'), (1,'b')]
```

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

Lösungsvorschlag

```
propIncr xs = isIncr (sortP xs)

isIncr (x:y:xs) = fst x <= fst y && propIncr (y:xs)
isIncr _ = True

propSameSet xs = subsetOf xs ys && subsetOf ys xs
  where ys = sortP xs

subsetOf [] _ = True
subsetOf (x:xs) ys = x `elem` ys && subsetOf xs (delete x ys)
```

Aufgabe 3 (5 Punkte)

Gegeben sei folgender Datentyp für arithmetisch-logische Ausdrücke:

```
data Expr = IntL Integer  | Plus Expr Expr
          | Leq Expr Expr | If Expr Expr Expr
```

Die einzelnen Konstruktoren haben dabei die folgende Bedeutung:

- `IntL i` entspricht dem Integer-Literal `i`
- `Plus` entspricht der Addition auf Integer
- `Leq` entspricht einem Vergleich ($\leq$) auf Integern, gibt also einen Boolean zurück
- `If b e f` entspricht einer Fallunterscheidung, die, wenn `b` zu `True` evaluiert, `e` zurückgibt und sonst `f`.

Für die Darstellung des Typs eines Ausdrucks dient folgender Datentyp:

```
data ExprType = IntTy | BoolTy deriving Eq
```

Ein wohlgetypter Ausdruck darf nur Integer mit Integer addieren, und nur Integer mit Integer vergleichen etc. Für `If b e f` muss `b` `Boolean`-Typ haben und `e` und `f` jeweils den gleichen Typ. Ein wohlgetypter Ausdruck kann also als Ergebnis entweder einen Integer oder einen Boolean haben. Wir sagen dann, der Ausdruck hat den Typ `IntTy` bzw. `BoolTy`. Schreiben Sie eine Funktion `typeOf :: Expr -> Maybe ExprType`, die `Just ty` zurückgibt, wenn der übergebene Ausdruck den Typ `ty` hat und `Nothing`, wenn er nicht wohlgetypt ist.

Beispiel:

```
typeOf (IntL 42) = Just IntTy
typeOf (Plus (IntL 1) (IntL 2)) = Just IntTy
typeOf (Leq (IntL 2) (Leq (IntL 1) (IntL 5)) = Nothing
```

Lösungsvorschlag

```
typeOf (IntL _) = Just IntTy
typeOf (Plus e f)
  | typeOf e == Just IntTy && typeOf f == Just IntTy = Just IntTy
typeOf (Leq e f)
  | typeOf e == Just IntTy && typeOf f == Just IntTy = Just BoolTy
typeOf (If b e f)
  | typeOf b == Just BoolTy && t1 == t2 = t1
  where (t1, t2) = (typeOf e, typeOf f)
typeOf _ = Nothing
```

Aufgabe 4 (5 Punkte)

Betrachten Sie die Funktion `f :: Int -> Int -> [Int]`, die für `f` m n die Liste der Quadrate aller ungeraden Zahlen i mit $m \leq i \leq n$ zurückgibt.
Implementieren Sie `f` auf verschiedene Arten. Die Verwendung beliebiger arithmetischer Funktionen und der Syntax `[`a `..` b`]` ist in allen drei Fällen erlaubt.

1. Mit Hilfe einer Listenkomprehension; ohne die Verwendung von Rekursion oder Funktionen höherer Ordnung wie `map`, `filter`, `foldl`, `foldr`.

2. Mit Hilfe von `map` und `filter`, ohne die Verwendung von Listenkomprehensionen oder Rekursion.

3. Als rekursive Funktion; ohne die Verwendung von Listenkomprehensionen oder Funktionen höherer Ordnung wie `map`, `filter`, `foldl`, `foldr`.

Lösungsvorschlag

1. `f m n = [x^2 | x <- [m .. n], odd x]`

2. `f m n = map (^2) (filter odd [m .. n])`

3.
```
f m n | m > n = []
      | odd m = m^2 : f (m+2) n
      | otherwise = f (m+1) n
```

Aufgabe 5 (6 Punkte)

Gegeben seien folgende Definitionen:

```
data Tree = Tip Integer | Node Tree Tree

sum :: Tree -> Integer
sum (Tip i) = i                                    -- def sum
sum (Node t1 t2) = sum t1 + sum t2                 -- def sum

right :: Tree -> Tree -> Tree
right (Tip i) t = Node (Tip i) t                   -- def right
right (Node t1 t2) t = right t1 (right t2 t)       -- def right
```

Beweisen Sie:

```
sum (right t t') = sum t + sum t'
```

Sie dürfen dabei das folgende Lemma für Typ `Integer` verwenden:

```
(i + j) + k = i + (j + k)                          -- +_assoc
```

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 `t`.

Fall `Tip`.

zu zeigen: `sum (right (Tip i) t') = sum (Tip i) + sum t'`

```
                sum (right (Tip i) t')
(def right) = sum (Node (Tip i) t')
(def sum)   = sum (Tip i) + sum t'
```

Fall `Node`.

zu zeigen: `sum (right (Node t1 t2) t') = sum (Node t1 t2) + sum t'`

IH1: `sum (right t1 u) = sum t1 + sum u`

IH2: `sum (right t2 u) = sum t2 + sum u`

```
                sum (right (Node t1 t2) t')
(def right) = sum (right t1 (right t2 t'))
(IH1)       = sum t1 + sum (right t2 t')
(IH2)       = sum t1 + (sum t2 + sum t')

                sum (Node t1 t2) + sum t'
(def sum)   = (sum t1 + sum t2) + sum t'
(+_assoc)   = sum t1 + (sum t2 + sum t')
```

Aufgabe 6 (5 Punkte)

Unter Unix gibt es ein Tool, mittels dem man eine Datei Zeile für Zeile ausgeben lassen kann, wobei der Nutzer die Ausgabe abbrechen kann. Implementieren Sie eine Version davon in Haskell:

```
more :: FilePath -> IO Integer
```

Ein Aufruf `more` *file* soll die Datei *file* zeilenweise ausgeben und nach jeder Zeile auf die Eingabe des Nutzers warten. Bei Eingabe von `y` soll die nächste Zeile ausgegeben, bei Eingabe von `n` die Ausgabe beendet werden. Die Ausgabe wird auch beendet, wenn nach der letzten Abfrage keine Zeilen mehr übrig sind.
Der Rückgabewert der IO-Aktion soll der Anzahl der ausgegebenen Zeilen entsprechen.

Beispiele (Antworten des Nutzers kursiv angegeben):

- Aufruf von `more "jazz.txt"`, Lesen bis zum Ende der Datei, Rückgabewert: 3

```
Count Basie y
Ella Fitzgerald y
Dizzy Gillespie y
```

- Aufruf von `more "jazz.txt"`, vorzeitiges Ende, Rückgabewert: 1

```
Count Basie n
```

Folgende Hilfsfunktionen sind gegeben:

```
readFile :: FilePath -> IO String
getChar :: IO Char
lines :: String -> [String]
```

Lösungsvorschlag

```
printLines :: [String] -> Integer -> IO Integer
printLines [] n = return n
printLines (x : xs) n = do
  putStr (x ++ " ")
  c <- getChar
  if c == 'y' then do
    putStr "\n"
    printLines xs (n + 1)
  else
    return (n + 1)

more :: FilePath -> IO Integer
more file = do
  contents <- readFile file
  let xs = lines contents
  printLines xs 0
```

Aufgabe 7 (5 Punkte)

Bei dieser Aufgabe sollen Sie Ausdrücke Schritt für Schritt mit Haskells Auswertungsstrategie auswerten. Brechen Sie unendliche Reduktionen mit „...“ ab, sobald Nichtterminierung erkennbar ist.

1. Werten Sie aus: `head (map (\x -> x * x) [1,2,3])`

2. Geben Sie eine Liste `xs` an, sodass `elem 0 (xs ++ [0])` nicht zu `True` auswertet. Werten Sie dann `elem 0 (xs ++ [0])` aus.

Gegeben seien dabei folgende Definitionen:

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


(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

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

elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys)
  | x == y    = True
  | otherwise = elem x ys
```

Lösungsvorschlag

1.
```
  head (map (\x -> x * x) [1,2,3])
= head ((\x -> x * x) 1 : map (\x -> (x * x) [2,3]))
= (\x -> x * x) 1
= 1 * 1
= 1
```

2. Es sei `ones = 1 : ones` .
```
  elem 0 (ones ++ [0])
= elem 0 ((1 : ones) ++ [0])
= elem 0 (1 : (ones ++ [0]))
= elem 0 (ones ++ [0])
= ...
```

Aufgabe 8 (4 Punkte)

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

1. ```
   f1 x xs = map (<x) xs
   f2 xs = \x -> map (<x) xs
   f3 x = map (<x)
   ```

2. ```
   g1 = \x -> x + y 2
        where y x = 3 * x
   g2 x = x + 3 * x
   ```

Lösungsvorschlag

1. Die Definitionen `f1` und `f3` sind äquivalent: Von `f1` zu `f3` kommt man durch Partial Application. Bei `f2` sind die Argumente vertauscht.

2. Die Definitionen sind nicht äquivalent. `g1` berechnet $x \mapsto x + 3 \cdot 2$, `g2` berechnet $x \mapsto x + 3 \cdot x$.