Prof. Dr. Thomas Breuel
Dipl.-Inf. Patrick Michel (AG ST)
Dipl.-Ing. Ludwig Schmidt-Hackenberg

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

Lösungshinweise/-vorschläge zum Übungsblatt 4: Software-Entwicklung 1 (WS 2010/11)

Die Hinweise und Vorschläge in diesem Dokument sollen der Lösungsfindung dienen und erheben demnach weder Anspruch auf Vollständigkeit noch Korrektheit. Sollten Sie Fehler finden würden wir uns freuen wenn Sie uns diese mitteilen. (Kontaktdaten finden Sie auf unserer Webpräsenz.)

Bitte beachten Sie auf diesem Blatt auch, dass die Lösungen (bis auf die freiwilligen Teile von Aufgabe 5) nur mit Hilfe der Sprachmittel erstellt wurden, die Ihnen zum Zeitpunkt der Bearbeitung zur Verfügung standen. Es ist eine gute Übung, diese eleganter und/oder kürzer zu formulieren. Insbesondere lassen sich einige Funktionen mit Funktionen höherer Ordnung sehr kurz definieren.

Aufgabe 1 Elementares Haskell (Präsenzaufgabe)

```
0 :: Int   1 :: Int   0 :: Double   True
False   "Hallo Welt"   "0"   [1] :: [Int]
[1.2] :: [Float]   1 : 2 : [] :: [Float]
-- Ja, auch die beiden Nullen.
```

Repräsentieren alle diese (Haskell-)Ausdrücke unterschiedliche Werte?

```
Int, Char, Bool, String, Double, Float
-- Yup, sogar Float.
```

Sind dies Typen in Haskell?

```
Integer -> Integer
Char -> Bool -> Integer
-- Beide sind Funktionstypen und bilden
-- (letztendlich) auf Integer ab.
```

Was haben diese beiden Typen gemeinsam?

```
2 + 3 == 3 :: Integer   -- inkorrekt
"Hal" ++ "lo" :: String -- "Hallo"
['H', 'a', 'l'] ++ ['a', 'l'] :: [Char]
-- ['H', 'a', 'l', 'a', 'l'] oder "Halal"
True == False :: Bool   -- False
' ' == " " :: Bool      -- inkorrekt
-- alle drei lets: korrekt, kein Ergebnis
let f = 17 :: Integer
let g = (\x -> 2 * x) :: Integer -> Integer
let h x = h x :: Integer
f == f :: Bool          -- True
g == g :: Bool          -- inkorrekt, da es
-- keine Eq Instanz fuer Funktionen gibt
g f == g f :: Bool      -- True
h f == h f :: Bool      -- korrekt,
-- terminiert allerdings nicht...
```

Sind dies korrekt getypte Eingaben für den Haskell-Interpreter? Geben Sie die Ergebnisse der typkorrekten Ausdrücke an.

```
[0, 1, 1, 2, 3, 5]   ==
0 : 1 : 1 : 2 : 3 : 5 : []
```

Konstruieren Sie diese Liste allein durch Verwendung von `(:)` und `[]`.

```
"a"       :: String oder [Char]
'a'       :: Char
"aa"      :: String
'aa'      -- Syntaxfehler
()        :: ()
1 + 1     :: Num a => a
1.0 + 1.0 :: Fractional a => a
1 `div` 0 :: Integral a => a
```

Was sind die Typen dieser Ausdrücke? Wie können Sie feste Typen erzwingen?

```
fibs = [0, 1, 1, 2, 3, 5] :: [Integer]
null fibs          == False :: Bool
null []            == True :: Bool
length fibs        == 6 :: Int
head fibs          == 0 :: Integer
tail fibs          == [1, 1, 2, 3, 5] ::
                                [Integer]
reverse fibs       == [5, 3, 2, 1, 1, 0] ::
                                  [Integer]
head (tail fibs)  == 1 :: Integer
head (tail (reverse fibs)) == 3 :: Integer
null (tail (tail (tail (tail (tail (tail
    fibs))))))    == True :: Bool
tail (tail [head fibs]) liefert Exception
```

Berechnen Sie und geben Sie den Typ des Ergebnis an.

Aufgabe 2 Syntax-, Typ- und Laufzeitfehler (Einreichaufgabe)

Überprüfen Sie die Deklarationen `a` bis `s` auf *Syntax-*, *Typ-* und *Laufzeitfehler*. Wenn die Deklaration syntaktisch korrekt ist und einen Typ besitzt, geben Sie den gebundenen Wert und den Typ an.

```
x = 2.0 :: Double
y = 3 :: Integer

a = x * y                  -- Typfehler (Integer <-> Double mismatch)
                           -- Vorschlag: round x * y  ist  6 :: Integer
b = x * 2                  -- 4.0 :: Double
c = 3.0 * y                -- Typfehler (No instance for (Fractional Integer))
                           -- sprich: 3.0 ist kein Literal vom Typ Int
                           -- Vorschlag: 3 * y  ist  9 :: Integer
d = head (tail [x])        -- Typ: Double, Laufzeitfehler (Prelude.head: empty list)
                           -- Vorschlag: head [x]  ist  2.0 :: Double
e = x + -3                 -- Syntaxfehler (Precedence parsing error)
                           -- Vorschlag: x + (-3)  ist  -1.0 :: Double
f = [[]] ++ [[12], [x, 2]] -- [[],[12.0],[2.0,2.0]] :: [[Double]]
g = 1 : x : [3] ++ [7]     -- [1.0,2.0,3.0,7.0] :: [Double]
h = [y + 9] : [y, 4]       -- Typfehler ([Integer] <-> Integer mismatch)
                           -- Vorschlag: [y + 9] ++ [y, 4]  ist  [12,3,4] :: [Integer]
i = [1, y] : [] ++ [[]]    -- [[1,3],[]] :: [[Integer]]
j = y : [2 `div` 0]        -- Typ: [Integer], Laufzeitfehler (divide by zero)
k = [] : [] : [] : []      -- [[],[],[]] :: [[a]]
l = ([] : []) : [] : []    -- [[[]],[]] :: [[[a]]]
m = 1 :: [2, x]            -- Syntaxfehler (parse error on input `,')
n = [1] + [2, y]           -- Typfehler (No instance for (Num [Integer]))
                           -- sprich: (+) ist auf [Integer] nicht definiert
o = [1] +++ [2, y]         -- Syntaxfehler (Not in scope: `+++')
                           -- Namensaufloesung zaehlt zu kontextabhaengiger Syntax
                           -- Vorschlag: [1] ++ [2, y]  ist  [1,2,3] :: [Integer]
p = pi :: Integer          -- Typfehler (No instance for (Floating Integer))
                           -- sprich: pi ist fuer Integer nicht definiert
                           -- Vorschlag: (floor pi) :: Integer  ist  3 :: Integer
q = (x * x - x) / (x * x - x * 2) -- Infinity :: Double
r = tail [undefined]       -- [] :: [a]
s = head [undefined]       -- Typ: a, Laufzeitfehler (Prelude.undefined)
```

Aufgabe 3 Funktionen in Haskell und Rekursion (Präsenzaufgabe)

a) Geben Sie eine Abstraktion für den folgenden Haskell-Ausdruck in Haskell-Notation an:

```
2 * pi * r :: Double
```

Können Sie auch noch andere Abstraktionen bilden?

Bei der Aufgabe (und dann auch für die b) sollen die Studenten mal wieder das Skript aufschlagen.

Die sinnvollste Abstraktion ist wohl diese hier:

```
\r -> 2 * pi * r :: Double          -- Typ: Double -> Double
```

Es spricht aber ja nichts dagegen auch von `pi` zu abstrahieren oder von anderen Variablen, ruhig auch von einem ganz anderen Datentyp oder auch gleich unit:

```
\pi r -> 2 * pi * r :: Double       -- Typ: Double -> Double -> Double
\r pi -> 2 * pi * r :: Double       -- Typ: Double -> Double -> Double
\x    -> 2 * pi * r :: Double       -- Typ: a -> Double
\y    -> 2 * pi * r :: Double       -- Typ: a -> Double
\()   -> 2 * pi * r :: Double       -- Typ: () -> Double
...
```

b) Geben Sie Ihrer Abstraktion aus a) einen Namen und verwenden Sie dazu einmal eine Definition als *Wertvereinbarung* und einmal die spezielle Syntax für Funktionen.

```
einName = \r -> 2 * pi * r :: Double
einName r = 2 * pi * r :: Double
```

c) Werten Sie die Haskell-Funktion `ibo`, gegeben durch folgende Definition, für $n = 3$ schrittweise aus:

```
ibo :: Integer -> Integer
ibo n = if n < 2 then 1 else ibo (n - 1) + ibo (n - 2)
```

Es werden nicht alle Schritte dargestellt, die Nummer am Rand zeigt aber an wieviel Schritte vergangen sind:

```
00:  ibo 3
01:  if 3 < 2 then 1 else ibo (3 - 1) + ibo (3 - 2)
02:  if False then 1 else ibo (3 - 1) + ibo (3 - 2)
03:  ibo (3 - 1) + ibo (3 - 2)
04:  ibo 2 + ibo (3 - 2)
05:  (if 2 < 2 then 1 else ibo (2 - 1) + ibo (2 - 2)) + ibo (3 - 2)
08:  (ibo 1 + ibo (2 - 2)) + ibo (3 - 2)
09:  ((if 1 < 2 then 1 else ibo (1 - 1) + ibo (1 - 2)) + ibo (2 - 2)) + ibo (3 - 2)
10:  ((if True then 1 else ibo (1 - 1) + ibo (1 - 2)) + ibo (2 - 2)) + ibo (3 - 2)
11:  (1 + ibo (2 - 2)) + ibo (3 - 2)
12:  (1 + ibo 0) + ibo (3 - 2)
13:  (1 + (if 0 < 2 then 1 else ibo (0 - 1) + ibo (0 - 2))) + ibo (3 - 2)
16:  (1 + 1) + ibo 1
19:  (1 + 1) + 1
20:  2 + 1
21:  3
```

d) Gesucht ist eine Haskell-Funktion `summ :: Integer -> Integer`, die eine ganze Zahl n als Eingabe erwartet und die Summe der ersten n Zahlen größer 0 berechnet. Bei negativen Eingaben soll die Funktion -1 zurückgeben. Implementieren Sie diese Funktion auf zwei unterschiedliche Arten.

```
-- Rekursiv:
summ n = if n < 0 then -1 else if n == 0 then 0 else n + summ (n-1)

-- Direkt:
summ n = if n < 0 then -1 else n * (n + 1) `div` 2
```

Aufgabe 4 Rekursion in Haskell (Einreichaufgabe)

a) Schreiben Sie eine Haskell-Funktion `verwerfe :: Int -> [Char] -> [Char]`, die eine ganze Zahl n und eine Liste von Zeichen (einen `String`) als Eingabe erwartet und die ersten n Zeichen der Liste entfernt. Im Fall $n < 0$ soll die Liste unverändert bleiben.

```
verwerfe :: Int -> String -> String
verwerfe n w = if n < 1 then w else verwerfe (n-1) (tail w)
```

b) Schreiben Sie die Haskell-Funktionen `und :: [Bool] -> Bool` und `oder :: [Bool] -> Bool`, die jeweils eine Liste von Wahrheitswerten als Eingabe erwarten und deren *Konjunktion* bzw. *Disjunktion* berechnen.

```
und :: [Bool] -> Bool
und l = if null l then True else head l && und (tail l)

oder :: [Bool] -> Bool
oder l = if null l then False else head l || und (tail l)
```

c) Schreiben Sie eine Haskell-Funktion `maxi :: [Integer] -> Integer`, die von einer Liste von ganzen Zahlen das Maximum bestimmt. Das Maximum der leeren Liste soll Null sein.

```
maxi :: [Integer] -> Integer
maxi l = if null l then 0 else maxi' (tail l) (head l) where

  maxi' :: [Integer] -> Integer -> Integer
  maxi' l n = if null l then n else
    if n > head l then maxi' (tail l) n else maxi' (tail l) (head l)
```

d) Schreiben Sie eine Haskell-Funktion `schnitt :: [Integer] -> [Integer] -> [Integer]`, die zwei Listen als Eingabe erwartet und alle Elemente aus der ersten entfernt, die nicht in der zweiten enthalten sind.

```
istIn :: Integer -> [Integer] -> Bool
istIn n l = if null l then False else
  if head l == n then True else istIn n (tail l)

schnitt :: [Integer] -> [Integer] -> [Integer]
schnitt l m = if null l then [] else
  (if istIn (head l) m then [head l] else []) ++ schnitt (tail l) m
```

e) Schreiben Sie eine Haskell-Funktion `ziehe :: [Int] -> String -> String`, die zwei Listen als Eingabe erwartet und jedes Element der zweiten Liste so oft wiederholt, wie im korrespondierenden Element der ersten Liste angegeben ist. Wenn eine der Eingabelisten kürzer als die andere ist, werden Elemente der längeren Liste verworfen. Negative Elemente in der ersten Liste sollen als Null gewertet werden.

```
wiederhole :: Int -> Char -> String
wiederhole n w = if n < 1 then [] else w : wiederhole (n-1) w

ziehe :: [Int] -> String -> String
ziehe n l = if null n || null l then [] else
  wiederhole (head n) (head l) ++ ziehe (tail n) (tail l)
```

f) Schreiben Sie eine Haskell-Funktion `primfaktoren :: Integer -> [Integer]`, die eine ganze Zahl größer 0 als Parameter erwartet und auf die Liste ihrer Primfaktoren abbildet.

Eine Idee ist einfach naiv alle Zahlen von zwei bis zur Zahl selbst durchzuprobieren mit einer Hilfsfunktion (`faktor`). Anstatt bei einem Treffer dann nur die beiden Faktoren zurück zu geben geht man einfach in die Rekursion (der erste Faktor ist immer prim).

```
primfaktoren :: Integer -> [Integer]
primfaktoren x = if x < 2 then [x] else faktor 2 x  where

  faktor :: Integer -> Integer -> [Integer]
  faktor c x = if x == c then [x] else
    if x `mod` c /= 0 then faktor (c + 1) x else c : faktor 2 (x `div` c)
```

g) Schreiben Sie eine Haskell-Funktion `pascal :: Int -> [Integer]`, die für gegebenes `n` die `n`-te Zeile des Pascalschen Dreiecks berechnet.

Wenn man auf die Idee kommt aus einer Zeile die nächste zu berechnen wäre dies eine Lösung:

```
pascal :: Int -> [Integer]
pascal n = if n < 2 then [1] else naechsteZeile (pascal (n - 1))  where

  naechsteZeile :: [Integer] -> [Integer]
  naechsteZeile z = 1 : if length z < 2 then z else
    (head z + head (tail z)) : (tail (naechsteZeile (tail z)))
```

Allerdings ist die Hilfsfunktion da recht umständlich, einfacher ist es mit einer weniger verwirrenden, aber schwerer zu beschreibenden Hilfsfunktion:

```
pascal :: Int -> [Integer]
pascal n = if n < 2 then [1] else 1 : paarSummen (pascal (n - 1))  where

  paarSummen :: [Integer] -> [Integer]
  paarSummen z = if length z < 2 then z else
    (head z + head (tail z)) : paarSummen (tail z)
```

Im Grunde kann man aber auch hingehen und sich die Einträge einer Zeile rekursiv berechnen, also in die Definition des Pascalschen Dreiecks schauen.

Aufgabe 5 Namensketten (Einreichaufgabe)

Namensketten sind ringförmige Schnüre, auf die kleine Würfel aufgereiht sind. Auf jedem Würfel ist ein Großbuchstabe aufgedruckt.

In dieser Aufgabe wollen wir solche Ketten näher betrachten. Um eine Kette in Haskell darzustellen, wird sie an einer Stelle aufgetrennt und dann als Liste von Buchstaben betrachtet. Als Datentyp wird `[Char]` verwendet, welcher gleichbedeutend mit `String` ist.

Beispiel: Eine Listendarstellung einer Kette mit dem Text UEBUNG

`['U', 'E', 'B', 'U', 'N', 'G']  oder einfach  "UEBUNG"`

a) Entwerfen Sie eine Funktion `istKette :: [Char] -> Bool`, die überprüft ob eine gegebene Liste von Buchstaben eine korrekte Darstellung einer Namenskette ist. Beachten Sie dabei, dass es nur Würfel mit Großbuchstaben gibt.

```
istGross :: Char -> Bool
istGross c = c >= 'A' && c <= 'Z'

istKette :: [Char] -> Bool
istKette kette = null kette || istGross (head kette) && istKette (tail kette)
```

b) Ein Problem der Listendarstellung ist, dass sich ein und dieselbe Kette durch verschiedene Listen darstellen lässt. Entwerfen Sie eine Funktion `shift :: [Char] -> [Char]`, die eine Kette in Listendarstellung um eine Position (Würfel) rotiert.

```
shift :: [Char] -> [Char]
shift k = if null k then [] else tail k ++ [head k]
```

c) Geben Sie eine Funktion `shiftN :: Int -> [Char] -> [Char]` an, die eine Listendarstellung einer Kette um n Positionen rotiert.

```
shiftN :: Int -> [Char] -> [Char]
shiftN n k = if n <= 0 then k else shiftN (n-1) (shift k)
```

d) Ab sofort gibt es einen zusätzlichen Würfel mit dem Aufdruck *, der den Anfang des Textes einer Kette markiert. Dieser Würfel kommt auf jeder Kette genau einmal vor. Beachten Sie, dass es jetzt keine

leeren Ketten mehr gibt. Schreiben Sie eine neue Funktion `istKetteStern :: [Char] -> Bool`, die diese Eigenschaft mitprüft.

```
istKetteStern :: [Char] -> Bool
istKetteStern kette = istKetteSternHelper kette False  where

  istKetteSternHelper :: [Char] -> Bool -> Bool
  istKetteSternHelper k found = if null k then found else if head k == '*'
    then if found then False else istKetteSternHelper (tail k) True
    else istGross (head k) && istKetteSternHelper (tail k) found
```

e) Die Standarddarstellung einer Kette ist die Listendarstellung einer Kette, bei der das erste Element der Liste der *-Würfel ist. Schreiben Sie eine Funktion `zuStandard :: [Char] -> [Char]`, die eine beliebige korrekte Kette in die Standarddarstellung transformiert.

```
zuStandard :: [Char] -> [Char]
zuStandard k = if head k == '*' then k else zuStandard (shift k)
```

f) Damit nicht jeder sofort weiß was auf der Kette steht, soll der Text ROT13 verschlüsselt werden. Bei der ROT13-Verschlüsselung wird jeder Buchstabe durch den um 13 Stellen dahinter liegenden Buchstaben des Alphabets ersetzt. Aus `A` wird damit `N` und aus `X` wird `K`. Als Sonderfall haben wir das Zeichen *, das auch verschlüsselt * ist. Schreiben Sie eine Funktion `rot13 :: [Char] -> [Char]`, die den Kettentext ROT13 verschlüsselt. Wie können Sie einen verschlüsselten Text wieder entschlüsseln?

```
rot13 :: [Char] -> [Char]
rot13 k = if null k then [] else rot13' (head k) : rot13 (tail k)  where

  rot13' :: Char -> Char                                    -- you need to import
  rot13' c = if c == '*' then c else                        -- Data.Char
    if c < 'N' then chr (ord c + 13) else chr (ord c - 13)  -- for chr and ord
```

g) *(freiwillige Zusatzaufgabe)* Wir betrachten wieder Ketten ohne den *-Würfel. Schreiben Sie eine Funktion `gleich :: [Char] -> [Char] -> Bool`, die zwei Ketten auf Gleichheit prüft. Beachten Sie, dass eine Kette keinen ausgezeichneten Anfang hat und evtl. spiegelverkehrt vor Ihnen liegen kann. So sind z.B. die folgenden Ketten gleich:

`"AUFGABE"` und `"UAEBAGF"`

Für diese Aufgabe werden zuerst zwei Hilfsfunktionen definiert. Die Funktion `comp` vergleicht zwei Listendarstellungen miteinander und liefert `True` wenn diese identisch sind. Beachten Sie: wenn zwei Listendarstellungen identisch sind, sind die zugehörigen Namensketten es auch. Der Umkehrschluss gilt jedoch nicht, da nicht festgelegt ist, wo die Schnur einer Kette aufgetrennt wird und ob man mit dem oder gegen den Uhrzeigersinn den Text der Kette in die Liste übernimmt.

```
comp :: [Char] -> [Char] -> Bool
comp [] []           = True
comp [] (_:_)        = False
comp (_:_) []        = False
comp (x:xs) (y:ys)   = x == y && xs `comp` ys
```

Eine Anmerkung zur letzten Zeile: In Haskell werden `&&` und `||` mit der sogenannten *Kurzschlussauswertung* ausgewertet. Dabei wird ein Ausdruck von links nach rechts ausgewertet und die Auswertung abgebrochen, sobald der Wert des Ausdrucks feststeht. Hier: wenn `x == y` `False` ergibt, dann kann der Ausdruck `x == y && xs `comp` ys` nur zu `False` ausgewertet werden, egal was der rekursive Aufruf für einen Wert ergeben würde. In diesem Fall wird der Aufruf daher gar nicht mehr ausgewertet und die Rekursion ist damit beendet. Wenn `x == y` zu `True` ausgewertet wird, ist das Ergebnis des Gesamtausdrucks abhängig vom rekursiven Aufruf und daher wird dieser auch ausgewertet. Eine äquivalente Formulierung ohne die Kurzschlussauswertung ist mit **`if-then-else`** leicht zu finden.

Eine zweite Hilfsfunktion vergleicht zwei Listendarstellungen und liefert `True`, wenn die Listen bis auf Rotation gleich sind. Dazu vergleicht sie die Listen mit `comp` und wenn sie nicht gleich sind, wird die zweite Liste um eine Position rotiert und wieder verglichen. Der dritte Parameter der Funktion dient

als Zähler wie viele Rotationen die Liste noch machen darf. Für den Abbruch der Rekursion gibt es jetzt zwei Möglichkeiten, entweder die Listen sind gleich (`comp` liefert `True`) oder die zweite Kette wurde bereits so oft gedreht, dass der Zähler auf 0 steht. Auch hier wird wieder die Kurzschlussauswertung von Haskell verwendet.

```
gleichShift :: [Char] -> [Char] -> Int -> Bool
gleichShift _ _ 0    = False
gleichShift xs ys n  = xs `comp` ys || gleichShift xs (shift ys) (n - 1)
```

Die Funktion `gleich` muss jetzt nur die Hilfsfunktionen mit den richtigen Parametern aufrufen. Wenn die beide Ketten nicht gleich lang sind, kann das Ergebnis direkt zurückgegeben werden. Ansonsten müssen sie entweder durch Rotation gleich werden oder aber mit einer Kette invertiert durch Rotation gleich werden. Da man eine Listendarstellung maximal ihre Länge mal rotieren kann, ohne eine Darstellung zu erreichen, die man schon betrachtet hat, geben wir die Listenlänge als dritten Parameter für `gleichShift` an.

```
gleich :: [Char] -> [Char] -> Bool
gleich xs ys
  | length xs /= length ys = False
  | otherwise = gleichShift xs ys (length ys) ||
                gleichShift xs (reverse ys) (length ys)
```