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

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

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

Ausgabe: in der Woche vom 15.11. bis zum 19.11.10
Abgabe: in der Woche vom 22.11. bis zum 26.11.10
Abnahme: max. zwei Tage nach der Übung

Aufgabe 1 Elementares Haskell (Präsenzaufgabe)

Lösen Sie diese Aufgabe ohne den Haskell Interpreter zur Hilfe zu nehmen.

`0 :: Int  1 :: Int  0 :: Double  True`
`False  "Hallo Welt"  "0"  [1] :: [Int]`
`[1.2] :: [Float]  1 : 2 : [] :: [Float]`
	Repräsentieren alle diese (Haskell-)Ausdrücke unterschiedliche Werte?

`Int, Char, Bool, String, Double, Float`	Sind dies Typen in Haskell?
`Integer -> Integer` `Char -> Bool -> Integer`	Was haben diese beiden Typen gemeinsam?

`2 + 3 == 3 :: Integer`
`"Hal" ++ "lo" :: String`
`['H', 'a', 'l'] ++ ['a', 'l'] :: [Char]`
`True == False :: Bool`
`' ' == " " :: Bool`
`let f = 17 :: Integer`
`let g = (\x -> 2 * x) :: Integer -> Integer`
`let h x = h x :: Integer`
`f == f :: Bool`
`g == g :: Bool`
`g f == g f :: Bool`
`h f == h f :: Bool`
	Sind dies korrekt getypte Eingaben für den Haskell-Interpreter? Geben Sie die Ergebnisse der typkorrekten Ausdrücke an.

`[0, 1, 1, 2, 3, 5]`	Konstruieren Sie diese Liste allein durch Verwendung von `(:)` und `[]`.

`fibs = [0, 1, 1, 2, 3, 5] :: [Integer]`
`null fibs`
`null []`
`length fibs`
`head fibs`
`tail fibs`
`reverse fibs`
`head (tail fibs)`
`head (tail (reverse fibs))`
`null (tail (tail (tail (tail (tail (tail`
`fibs))))))`
`tail (tail [head fibs])`
	Berechnen Sie und geben Sie den Typ des Ergebnisses an.

`"a"  'a'  "aa"  'aa' ()  1 + 1  1.0 + 1.0` `1 `div` 0`	Was sind die Typen dieser Ausdrücke? Wie können Sie feste Typen erzwingen?

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. Versuchen Sie eventuell auftretende Fehler sinnvoll zu korrigieren.

Hinweis: Sie können für diese Aufgabe auch den GHC oder GHCi verwenden.

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

a = x * y
b = x * 2
c = 3.0 * y
d = head (tail [x])
e = x + -3
f = [[]] ++ [[12], [x, 2]]
g = 1 : x : [3] ++ [7]
h = [y + 9] : [y, 4]
```

```
i = [1, y] : [] ++ [[]]
j = y : [2 `div` 0]
k = [] : [] : [] : []
l = ([] : []) : [] : []
m = 1 :: [2, x]
n = [1] + [2, y]
o = [1] +++ [2, y]
p = pi :: Integer
q = (x * x - x) / (x * x - x * 2)
r = tail [undefined]
s = head [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?

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.

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)
```

Hinweis: Ersetzen Sie nach und nach Funktionsaufrufe durch ihre Definition bzw. bei Basis-Funktionen auf Konstanten durch ihr Ergebnis. Denken Sie daran, formale Parameter von Funktionen durch aktuelle zu ersetzen.

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.

Beispiele:

```
summ 4 == 10        summ (-7) == -1        summ 0 == 0
```

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.

Beispiele:

```
verwerfe 4 "Hallo Welt!"  ==  "o Welt!"
verwerfe (-2) "abc"       ==  "abc"
```

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.

Beispiele:

```
oder []                     ==  False
oder [True, False, True]    ==  True
und  [True, False, True]    ==  False
und  [True, True]           ==  True
```

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.

Beispiele:

```
maxi [12, -4, 67, 5]  == 67
maxi []               ==  0
```

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.

Hinweis: Schreiben Sie zuerst eine Haskell-Funktion `istIn :: Integer -> [Integer] -> Bool`, die überprüft, ob ein Element in einer Liste enthalten ist.

Beispiel:

```
schnitt [1, 3, 1, 2, 2, 4] [1, 2, 5]  ==  [1, 1, 2, 2]
```

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.

Beispiele:

```
ziehe [1, 2, 3, 4, 5] "Hallo"  ==  "Haalllllllooooo"
ziehe [1, 5, 1, 2]    "lol"    ==  "loooool"
```

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.

Beispiel:

```
primfaktoren 22230  ==  [2,3,3,5,13,19]
```

Hinweis: Schreiben Sie zunächst eine Funktion, die eine Zahl in zwei Faktoren zerlegt, wenn möglich, und sonst die Zahl selbst als Liste zurück gibt. Überlegen Sie sich dann was Sie anpassen müssen, um das gewünschte Ergebnis zu erhalten.

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

Beispiele:

```
pascal 1  ==       [1]
pascal 2  ==     [1,1]
pascal 3  ==    [1,2,1]
pascal 4  ==   [1,3,3,1]
pascal 5  ==  [1,4,6,4,1]
pascal 6  ==      ...
```

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.

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.

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

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.

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.

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?

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"
```