

# Programmieraufgaben

insgesamt 20 Punkte

Lösen Sie die folgenden Programmieraufgaben. In dem Ordner **Aufgaben** finden Sie neben der Datei **Tests.hs** vier Haskell-Dateien, welche jeweils den Rumpf mit Typsignatur enthalten. Ergänzen Sie diese mit einer Implementation, und beachten Sie bitte:

- Verändern Sie die vorgegeben Typsignaturen nicht.
- Benennen Sie die Dateien nicht um, und legen Sie keine weiteren Dateien an.
- Achten Sie darauf, dass sich Ihre Abgabe übersetzen läßt; nicht übersetzbare Lösungen werden mit jeweils 0 Punkten bewertet.

## Weitere nützliche Hinweise

- Sie können Visual Studio Code zum Bearbeiten der Programmieraufgabe nutzen; es ist mit Erweiterungen für Haskell vorkonfiguriert, und wird durch das Icon auf dem Desktop gestartet.
- Zum Übersetzen benutzen Sie den `ghci`. Auf dem Desktop finden Sie ein Icon, welches den `ghci` im richtigen Verzeichnis startet, so dass Sie mit `:l <Aufgabe>.hs` direkt die Quelldatei laden können.
- Die Datei **Tests.hs** enthält die Beispiele aus der Aufgabenstellung als Unit-Tests. So können Sie schnell prüfen, ob Ihre Lösung plausibel ist. Natürlich sind diese Tests nur notwendig, aber nicht hinreichend für eine vollständige Lösung der Aufgaben.  
Zum Ausführen der Tests laden Sie bitte **Tests.hs** in den `ghci`, und rufen die Funktion `run` auf.
- Sie finden auf dem Desktop Icons zur Dokumentation des `ghc` (insbesondere der Standard-Bücherei).

*Viel Glück!*

## 1 Diagonalen

5 Punkte

Implementieren Sie eine Funktion, welche aus einer Liste von Listen die *Diagonale* extrahiert, d.h. aus der  $i$ -ten Liste das jeweils  $i$ -te Element selektiert (wobei  $i$  angefangen mit dem Index 0 die Liste durchläuft):

$$\text{diagonal} :: [[a]] \rightarrow [a]$$

Die Funktion soll terminieren, wenn die  $i$ -te Liste weniger als  $i$  Elemente hat.

Die Funktion soll sowohl eine unendliche Argumentliste als auch darin enthaltene unendliche Listen korrekt behandeln (1 Punkt).

*Beispiel:*

```
diagonal [[1], [10, 20, 30], [100, 200, 300]] ~> [1, 20, 300]
diagonal ["abc", "def", "ghi", "jklmn", "o", "pqrstu"] ~> "aeim"
diagonal (repeat [1, 2, 3,...]) ~> [1, 2, 3,...]
```

## 2 Rauten

5 Punkte

Schreiben Sie eine Funktion `diamond c n`

```
diamond :: Char → Int → String
```

welche einen String zurückgibt, der auf der Kommandozeile ausgegeben eine Raute aus dem gegebenen Zeichen `c` mit der Kantenlänge `n` darstellt:

*Beispiel:*

```
diamond 'x' 1 ~> "x\n",  
diamond '*' 5 ~> "uuuu*\nuuuu***\nuuu*****\nu*****\nn*****\nu*****\nuuu*****\nuuuu**\nuuuu*\n"
```

Wenn wir den String mit `putStr` ausgeben, bewirkt `\n` einen Zeilenumbruch:

```
putStr $ diamond 'x' 1
x
```

```
putStr $ diamond '*' 5
```

```

      *
    ***
  *****
*****
*****
*****
*****
  *****
    ***
      *

```

*Hinweise:*

- Gehen Sie zeilenweise vor:
  - (i) Die Ausgabe enthält  $2n - 1$  Zeilen.
  - (ii) Die  $i$ -te Zeile (für  $i = 1, \dots, 2n - 1$ ) besteht aus  $n - j$  Leerzeichen, gefolgt von  $2j - 1$  Kopien von `c`, mit  $j = i$  für  $i \leq n$  und  $j = 2n - i$  für  $i > n$ .

- Die folgende vordefinierte Funktion ist sicherlich hilfreich:

```
replicate :: Int -> a -> [a]
```

### 3 Variadische Bäume

4 Punkte

In einem *variadischen Baum* hat jeder Knoten eine beliebige Anzahl von Kinderbäumen:

```
data NTree a = NT a [NTree a]
```

Die Höhe eines variadischen Baumes ist dabei die um eins erhöhte maximale Höhe der Kinderbäume oder 1, falls der Knoten keine Kinder hat. Ein variadischer Baum ist *ausgeglichen*, falls die Höhe der Kinderbäume sich nicht um mehr als 1 unterscheidet. Schreiben Sie zwei Funktionen

```
height :: NTree a → Int
balanced :: NTree a → Bool
```

welche die Höhe eines variadischen Baumes berechnen bzw. prüfen, ob ein variadischer Baum balanciert ist.

*Beispiel:*

```
tree1 = NT 4 []
tree2 = NT 3 [tree1, tree1]
tree3 = NT 2 [tree1, tree2, tree1]
tree4 = NT 1 [tree1, tree3, tree2]
```

```
map height [tree1, tree2, tree3, tree4] ~== [1,2,3,4]
map balanced [tree1, tree2, tree3, tree4] ~== [True, True, True, False]
```

*Hinweis:*

- Folgende vordefinierte Funktionen könnten hilfreich sein:

```
maximum :: Ord a ⇒ [a] → a
minimum :: Ord a ⇒ [a] → a
```

### 4 Frequenzen

6 Punkte

Implementieren Sie eine Funktion

```
frequency :: String → [(Char, Int)]
```

welche die Vorkommen von Buchstaben in der Zeichenkette zählt, und in absteigender Häufigkeit ausgibt. (Die Häufigkeit 0 wird nicht mit ausgegeben.) Zeichen, die keine Buchstaben sind, werden ignoriert; Groß- und Kleinschreibung werden nicht unterschieden. Die Reihenfolge, in der gleich oft auftretende Buchstaben auftreten, wird nicht spezifiziert.

*Beispiel:*

```
frequency "" ~== []
```

```
freq "abc123AB!ax." ~== [('a',3),('b',2),('x',1),('c',1)]
```

```
frequency "Straßenbahnlinie_6?" ~==
[('n',3),('i',2),('e',2),('a',2),('ß',1),('t',1),('s',1),('r',1),('l',1),('h',1),('b',1)]
```

*Hinweis:*

- Folgende Funktionen (aus `Data.Char` und `Data.List`) sind sicherlich hilfreich:

```
isAlpha :: Char → Bool
toLower :: Char → Char
sortOn :: Ord b ⇒ (a → b) → [a] → [a]
```

Hierbei sortiert `sortOn` eine Liste, indem die Elemente in einen anderen Datentypen abgebildet und dort verglichen werden.