

## Programmieraufgaben

insgesamt 70 Punkte

Lösen Sie die folgenden Programmieraufgaben. In diesem Ordner finden Sie neben der Datei `Tests.hs` fünf 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 oben 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 `main` auf.

- Sie finden auf dem Desktop Icons zur Dokumentation des `ghc` (insbesondere der Standard-Bücherei) sowie zu Hoogle.

*Viel Glück!*

## 1. Positionen

10 Punkte

Definieren Sie eine Funktion `positions :: [a] -> [(a, Int)]`, welche zu einer Eingabeliste `xs` eine Liste von Paaren  $(x, i)$  zurückgibt, wobei  $i$  die Position von  $x$  in der Eingabeliste ist (beginnend mit 1).

*Beispiel:*

`positions "Foo" ~> [('F', 1), ('o', 2), ('o', 3)]`

## 2. Listen auffüllen

15 Punkte

Definieren Sie eine Funktion `rpadd :: a -> [[a]] -> [[a]]`, welche alle Elemente einer Liste von Listen auf die Länge der längsten Liste bringt, indem das gegebene einzelne Element hinten entsprechend oft angehängt wird. Sie können annehmen, dass alle Listen endlich sind.

*Beispiel:*

`rpadd '*' ["abcde", "fgh", "uvwxyz"] ~> ["abcde*", "fgh***", "uvwxyz"]`

`rpadd 0 [[1, 3, 5], [1..5], [7, 9]] ~> [[1, 3, 5, 0, 0], [1, 2, 3, 4, 5], [7, 9, 0, 0, 0]]`

## 3. Eigenschaften

10 Punkte

Formulieren Sie folgende Eigenschaft von `rpadd` als Funktion `samePrefix`:

- Die Länge der Eingabeliste und die Länge der Ergebnisliste ist gleich, und
- alle Listen der Eingabeliste sind ein Präfix der entsprechenden Liste der Ergebnisliste.

*Beispiel:*

`samePrefix ["a", "fg", "uvw"] ["abc", "fgh", "uvw"] ~> True`

`samePrefix [[1, 3], [7, 10], [8]] [[1, 3, 5], [7, 9], [8, 12]] ~> False`

`samePrefix [] [] ~> True`

`samePrefix [] ["foo"] ~> False`

*Hinweis:*

- Die Funktion `isPrefixOf :: Eq a => [a] -> [a] -> Bool` (aus `Data.List`) könnte nützlich sein.

## 4. Variadische Bäume

Ein variadischer Baum ist ein Baum mit einer veränderlichen Anzahl von Kindern pro Ebene, in Haskell modelliert als

**data** `NTree a = Node [NTree a] | Leaf a`

In diesem Fall sind die variadischen Bäume an den Blättern markiert.

— wohlgeformt

2 Punkte

Ein variadischer Baum ist *wohlgeformt*, wenn kein Knoten (*Node*) eine *leere* Liste von Kinderknoten enthält. Schreiben Sie ein Prädikat  $wf :: NTree\ a \rightarrow Bool$ , welches wahr ist gdw. der Baum wohlgeformt ist.

*Beispiel:*

```
t1 = Node [Leaf "1", Node [], Leaf "2"]
t2 = Node [Leaf "a", Node [Leaf "b", Leaf "c",
    Node [Leaf "d", Leaf "e"], Leaf "f", Node [Leaf "x" ]]]
```

$wf\ t1 \rightsquigarrow False$

$wf\ t2 \rightsquigarrow True$

— schön ausgedruckt

8 Punkte

Schreiben Sie eine Funktion  $render :: Show\ a \Rightarrow NTree\ a \rightarrow String$ , welche einen variadischen Baum in einen String konvertiert, und dabei die Baumstruktur durch Einrückung darstellt. Sie können davon ausgehen, dass der Baum wohlgeformt ist.

*Beispiel:*

```
putStrLn $ render t2 ~>
  "a"
    "b"
      "c"
        "d"
          "e"
            "f"
              "x"
```

— vermessen

5 Punkte

Implementieren Sie eine Funktion  $height :: NTree\ a \rightarrow Int$ , welche die Höhe eines variadischen Baumes implementiert. Die Höhe eines Blattes ist 1, und die Höhe eines Knoten ist das um eins erhöhte Maximum der Höhen der Kinder, oder 0 falls es keine Kinder gibt.

*Beispiel:*

$height\ t1 \rightsquigarrow 2$

$height\ t2 \rightsquigarrow 4$

— balanciert

10 Punkte

Ein variadischer Baum ist balanciert, wenn sich für jeden Knoten die Höhen der Kinder nur um höchstens 1 unterscheidet (Blätter sind immer balanciert). Implementieren Sie eine Funktion  $balanced :: NTree\ a \rightarrow Bool$  welche prüft, ob ein Baum balanciert ist oder nicht.

*Beispiel:*

$balanced\ t1 \rightsquigarrow True$

$balanced\ t2 \rightsquigarrow False$

*Hinweis:*

- Die Funktion `replicate :: Int -> a -> [a]` könnte für die Funktion `render` nützlich sein.
- Die Funktionen `maximum :: Ord a => [a] -> a` und `minimum :: Ord a => [a] -> a` berechnen das Maximum (Minimum) einer nicht-leeren Liste.

## 5. Funktorinstanz

10 Punkte

Die Konstruktor-Klasse *Functor* ist vordefiniert als

```
class Functor f where fmap :: (a -> b) -> f a -> f b
```

Für die Instanzen von *Functor* muss die Funktion *fmap* folgende Eigenschaften erfüllen:

```
fmap id = id
fmap f . fmap g = fmap (f . g)
```

Definieren Sie eine Instanz von *Functor* für variadische Bäume.

## 6. Unendlich aufgefüllt

5 Bonuspunkte

Implementieren Sie eine Variante *rpadinf* der Funktion *rpad*, so dass die Elemente der Eingabeliste auch unendlich lang sein können.

*Beispiel:*

```
map (take 5) $ rpadinf 0 [[1, 3 ..], [1 ..], [7, 9]] ~>
  [[1, 3, 5, 7, 9], [1, 2, 3, 4, 5], [7, 9, 0, 0, 0]]
```