Name: _____

0.1 Funktionale Programmierung

Bäume gehören zu den wichtigsten in der Informatik auftretenden Datenstrukturen. In dieser Aufgabe verwenden wir den Datentyp Tree zur Darstellung von Binärbäumen, die an den Blättern mit Werten markiert sind und an den Knoten mit Operationen:

```
data Tree =
        Value Integer
      | Mark Op Tree Tree
      deriving (Eq, Show)

data Op = Plus | Mult | Minus
      derivin (Eq, Show)
```

Schreiben Sie eine Haskell-Funktion *balanced*, die einen Baum als Eingabe entgegennimmt und entscheidet, ob in diesem Baum genausoviele positive Werte als Blätter vorkommen wie negative. Die Null soll dabei weder als positiv noch negativ zählen. *(6 Punkte)*

Beispiel:

```
balanced (Mark Plus (Value 3) (Mark Minus (Value 0) (Value (-4)))) == True
balanced (Mark Plus (Value 3) (Mark Minus (Value 2) (Value (-4)))) == False
```

0.1.1 Beispiellösung

Für die Aufgabenstellung ist nur der Wert der einzelnen Knoten interessant. Die Tatsache, dass bei den Knoten ein Operator dabei ist, bzw. dass die Datenstruktur Terme ausdrücken kann, ist nicht weiter wichtig.

```
-- Lösung von David
balanced :: Tree -> Bool
balanced x
       | count (x) == 0  = True
       | otherwise       = False

count :: Tree -> Integer
count (Value x)
      | x < 0 = -1
      | x > 0 = 1
      | otherwise = 0
count (Mark x t1 t2) = (count t1) + (count t2)
```

0.2 Funktionale Programmierung

Schreiben Sie die Funktionen *laenge*, die die Länge einer Liste berechnet, und *summe*, die die Summe der ersten n Zahlen berechnet. Beachten Sie, dass alle Eingaben behandelt werden müssen und geben Sie die Signatur der Funktionen an. *(4 Punkte)*
Beispiel:

```
laenge [[], [1,2], [2]]     -- => 3
summe 4                     -- => 10
```

0.2.1 Beispiellösung

Die „normale“ rekursive Lösung wurde ja früher schon besprochen:

```
laenge :: [a] -> Integer
laenge [] = 0
laenge (x:xs) = 1 + laenge xs

summe :: Integer -> Integer
summe x | x < 0 = 0
        | otherwise x + summe (x - 1)
```

In der Aufgabenstellung steht hier deutlich dabei, dass die Signaturen anzugeben sind! In der Funktion *laenge* interessiert uns der Typ nicht weiter, wir rechnen nur mit dem Ergebnis des rekursiven Funktionsaufrufs.

Markus hat auch die etwas kürzere, ebenfalls richtige Version abgegeben, die Higher-Order-Funktionen nutzt:

```
laenge :: [a] -> Integer
laenge = foldl (\i _ -> i+1) 0

summe :: Integer -> Integer
summe i = foldl (+) 0 [0..i]
```

Für die Interpretation der Aufgabenstellung als „Die Summe der ersten n Zahlen in einer Liste an Zahlen“ gab es höchstens einen Punkt. Man kann den Text zwar mit viel gutem Willen so verstehen, jedoch ist ein klärendes Beispiel angegeben und man hätte auch eine Verständnisfrage stellen können.

Für diese Interpretation wäre folgende Lösung korrekt:

```
summe' :: [Integer] -> Integer -> Integer
summe' [] _ = 0
summe' _ 0 = 0
summe' (x:xs) n = x + summe' xs (n-1)
```

0.3 Funktionale Programmierung

Gegeben sei folgender Datentyp *Zeit*, der die Menge von Zeitdauer-Angaben im Format *Wochen : Tage : Stunden* modelliert.

```
data Zeit = WTS Integer Integer Integer
```

Hierbei gilt, dass die Angaben normiert vorliegen, das heißt alle Komponenten Werte größer gleich 0 beinhalten und die Komponenten Tage und Stunden nur Werte kleiner 7 bzw. 24 enthalten.

(a) Schreiben Sie eine Funktion *inStunden*, die eine Zeit als Parameter entgegennimmt und diese in die Anzahl von Stunden umwandelt. *(2 Punkte)*

(b) Schreiben Sie eine Funktion *alsZeit*, die eine ganze Zahl, die eine Anzahl von Stunden darstellt, als Parameter entgegennimmt und diese in eine Zeit umwandelt. *(2 Punkte)*

0.3.1 Beispiellösung

In dieser Aufgabe geht es zur Abwechslung nicht um Rekursion.

```
inStunden :: Zeit -> Integer
inStunden (WTS w t s) = w * 168 + t * 24 + s

alsZeit :: Integer -> Zeit
alsZeit x = WTS (x `div` 168) ((x `mod` 168) `div` 24) ((x `mod` 168) `mod` 24)
```

Mit $168 = 7 \cdot 24$; hier macht es sich Sinn, die Bedeutung von *div* und *mod* zu veranschaulichen: x `div` 168 berechnet die Anzahl der vollen Tage, die in die gegebene Stundenzahl passt, x `mod` 168 berechnet den Rest, der danach weiter aufgeteilt werden muss.

0.3.2 Beispiellösung 2

Wer es unbedingt doch rekursiv machen will, benötigt dann eine Hilfsfunktion, mit der div und mod von Hand nachgebaut werden, d.h. solange etwas hinein passt, wird abgezogen.

```
inStunden' :: Zeit -> Integer
inStunden' (WTS 0 0 s) = s
inStunden' (WTS 0 t s) = inStunden' (WTS 0 0 (s + 24 * t))
inStunden' (WTS w t s) = inStunden' (WTS 0 (t + 7 * w) s)

alsZeit' :: Integer -> Zeit
alsZeit' x = alsZeitR x (WTS 0 0 0)

-- Hilfsfunktion
alsZeitR :: Integer -> Zeit -> Zeit
alsZeitR 0 z = z
```

```
alsZeitR x (WTS w t s)
       | x >= 168  = alsZeitR (x - 168) (WTS (w + 1) t s) -- div 168
       | x >= 24   = alsZeitR (x - 24) (WTS w (t + 1) s)  -- div 24
       | otherwise = WTS w t x
```