Arbeitsblatt: Rekursion

Eben haben Sie gesehen, wie man die Fakultät rekursiv implementiert. Diese Implementation geht direkt aus der (ebenfalls rekursiven) mathematischen Definition hervor (aus Wikipedia):

> Die Fakultät lässt sich rekursiv definieren:
>
> $$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n > 0 \end{cases}$$
>
> Fakultäten für negative oder nicht ganze Zahlen sind nicht definiert.

In Haskell:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Bei der Implementation von rekursiven Funktionen ist eine systematische Vorgehensweise empfehlenswert. Hier die wichtigsten Regeln der Rekursion:

1. Basisfall: Es braucht immer einen einfachen Basisfall, der ohne Rekursion lösbar ist. Der Basisfall garantiert, dass die Rekursion nicht „unendlich“ wird.
2. Fortschritt: Für die anderen Fälle die rekursiv gelöst werden, muss der rekursive Aufruf immer einen Fortschritt hin zum Basisfall machen. Nur so wird der Basisfall tatsächlich auch erreicht. D.h. die Problemgrösse muss bei jedem rekursiven Aufruf verkleinert werden!
3. Rekursionsschritt: Gegeben sei die Lösung des Problems der Grösse n-1. Löse nun mit Hilfe dieser Lösung das Problem der Grösse n.

Schauen wir diese drei Regeln einmal im Zusammenhang mit der Fakultät an:

1. Der Basisfall ist hier sogar schon gegeben. Wenn n = 0 dann kann das Resultat angegeben werden ohne dass factorial nochmals aufgerufen werden muss.

   ```
   factorial 0 = 1
   ```

   Der „Boden“ der Rekursion ist erreicht, und zum ersten Mal in der Aufrufreihenfolge wird ein Resultat zurückgegeben.

2. Für den anderen Fall (der Parameter ist ungleich 0) muss sichergestellt werden, dass bei jedem rekursiven Aufruf ein Fortschritt gemacht wird hin zum Basisfall. Wir erreichen das, indem der Wert des Parameters den wir im rekursiven Aufruf verwenden um 1 kleiner ist als den, den wir erhalten haben.

   ```
   factorial n = n * factorial (n-1)
   ```

   So wird sichergestellt, dass der übergebene Parameter irgendwann einmal 0 ist und damit dem Basisfall entspricht.

3. Bisher haben wir nur Trivialitäten gelöst (Basisfall) oder das Problem weiterdelegiert (rekursiver Aufruf). Aber irgendwann einmal muss auch mal etwas gearbeitet werden ☺. Die Idee ist ja, das Problem nicht aufs Mal zu lösen, sondern Schritt für Schritt. Indem man nämlich ein etwas kleineres Problem zuerst löst und dann nur noch einen kleinen Rest lösen muss.

   ```
   factorial n = n * factorial (n-1)
   ```

   Das Weiterdelegieren löst uns das Problem der Fakultät von n-1. Was bringt uns das? Sehr viel, jetzt müssen wir nämlich nur noch dieses Resultat mit n multiplizieren und schon haben wir das Problem der Fakultät von n gelöst!

Aufgabe 1
Erklären Sie was passiert, wenn man `factorial (-1)` aufruft?
Warum ist dieses Verhalten OK?

Hinweis: mit Ctrl-C kommen Sie da wieder raus!

Versuchen Sie es nun selber einmal. Lösen Sie folgende Aufgaben. Wenn Sie unsicher sind, dann führen wir sie in der ersten Aufgabe Schritt für Schritt zur Lösung, wenn Sie sich sicher sind, können Sie die Ausführungen überspringen und gleich mit der Lösung beginnen.

Aufgabe 2
Implementieren Sie eine Funktion countDown, welche von einem gegebenen Startwert ≥ 0 auf 0 herunter „zählt" und diese Zahlen in einer Liste zurück gibt.
```
countDown :: Int -> [Int]
```

In GHCi sollte ein Aufruf dann so aussehen:
```
*Main> countDown 10
[10,9,8,7,6,5,4,3,2,1,0]
```

Heranführung (kann übersprungen werden)

1. **Überlegen Sie sich einen Basisfall:**
   Gibt es einen Fall der so einfach ist, dass sie die Lösung einfach hinschreiben können? Manchmal ist es auch einfacher mit 2. fortzufahren und danach nochmals zu 1. Zurückzukommen.
2. **Überlegen Sie sich den Rekursionsschritt**:
   a. Nehmen sie willkürlich eine Problemgrösse an und notieren Sie sich die erwartete Lösung.
   b. Verkleinern Sie das Problem und notieren Sie sich auch das erwartete Resultat des verkleinerten Problems. Verkleinern bedeutet immer einen Parameter hin zum Basisfall zu verändern. Also z.B. einen numerischen Wert zu verkleinern, einen String oder eine Liste zu verkürzen, ein Element aus einer Datenstruktur entfernen etc.
   c. Wenn Sie nun die beiden Resultate von a. und b. betrachten: Wie können Sie das Resultat von a. mit Hilfe des Resultates von b. berechnen/konstruieren?

   Gehen Sie davon aus, dass Sie das Problem z.B. für 9 lösen können indem Sie countDown 9 aufrufen. Das Ergebnis ist:
   `[9,8,7,6,5,4,3,2,1,0]`
   Wie können Sie dieses nutzen um das Resultat von countDown 10 (siehe oben) zu berechnen?
3. Wie sieht die **Typsignatur** aus? Denken Sie daran, dass der Typ der Funktion sowohl auf den Basisfall als auch auf den Rekursionsschritt passen muss.
4. **Implementieren Sie eine Fallunterscheidung:**
   Der Basisfall und der Rekursionsschritt müssen unterschieden werden. In Haskell können Sie dies z.B. auch mit Pattern Matching lösen (oder mit einer if-then-else-Expression oder mit Guards…)
5. **Überprüfen der 3 Regeln:**
   a. Ist die Basisregel implementiert und in einer Fallunterscheidung vom Rekursionsschritt getrennt?
   b. Wird im Rekursionsschritt die Funktion selbst wieder aufgerufen? Und tun sie dies auch so, dass die Problemgrösse verkleinert wird hin zum Basisfall?
   c. Tun Sie im Rekursionsschritt auch etwas Sinnvolles um das Problem zu lösen?

Aufgabe 3
Das geht auch umgekehrt! Schreiben Sie eine Funktion
```
countUp :: Int -> [Int]
```
welche die Zahlen von 0 bis n in eine Liste füllt.
Hinweis: Sie können die Struktur der Lösung aus Aufgabe 2 übernehmen und müssen nur ganz wenig ändern!
In Aufgabe 2 haben Sie die Liste in dieser Reihenfolge erzeugt:
[0], [1, 0], [2, 1, 0]
nun soll die Liste in dieser Folge erstellt werden:
[0], [0, 1], [0, 1, 2].

Aufgabe 4
Dann ist jetzt diese Aufgabe ein Kinderspiel: Implementieren Sie eine Funktion
```
countDownUp :: Int -> [Int]
```
welche eine Liste mit Zahlen erzeugt, die zuerst von n absteigend bis 0 verläuft und dann wieder bis n hoch geht. Verwenden Sie weder countUp noch countDown:
```
*Main> countDownUp 10
[10,9,8,7,6,5,4,3,2,1,0,1,2,3,4,5,6,7,8,9,10]
```