Fachhochschule Nordwestschweiz

Haskell Lab

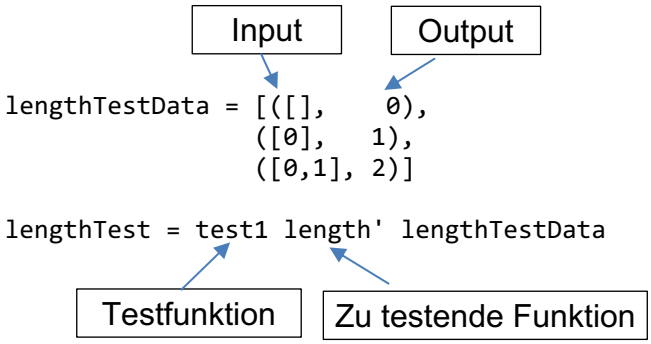
In dieser Woche vertiefen wir nochmals das Gelernte um ideal auf die Prüfung vorbereitet zu sein.

HTest– Unittesting in Haskell

Bis heute haben Sie Ihre Funktionen in einem Texteditor geschrieben, ghci gestartet und die Funktion von Hand mit zwei drei verschiedenen Inputs getestet. Heute implementieren Sie Ihr erstes Mini-Testframework um das Testen der Funktionen zu automatisieren. Sie werden sehen, es macht richtig Spass Tests zu schreiben!

In Haskell existieren diverse unterschiedliche Testframeworks. Heute wollen wir aber ein minimales[1] Testframework selbst implementieren.

Die Idee ist, dass man eine Liste mit Input/Output Paaren definiert. Die zu testende Funktion soll dann mit jedem Input aufgerufen werden und der erzeugte Output wird mit dem erwarteten Output verglichen. Hier ein Beispiel wie es verwendet werden soll:



Aufgabe a):
Implementieren Sie die Funktion `test1`. Als Resultat soll sie mit einem Bool angeben, ob alle Tests erfolgreich waren (`True`) oder ob mindestens ein Test gescheitert ist (`False`). Überlegen Sie sich in einem ersten Schritt die Typsignatur der Funktion `test1`.

Aufgabe b):
Die Funktion `drop'` nimmt aber zwei Parameter. Entsprechend müssen diese beiden Werte in den Inputdaten vorhanden sein. Hier ein Beispiel um die `drop'` Funktion zu testen:

```
dropTestData =
  [((0, []),    []),
   ((1, []),    []),
   ((1, [0]),   []),
   ((1, [0,1]), [1])]

dropTest = test2 drop' dropTestData
```

Implementieren Sie test2.

[1] Es besteht aus zwei Funktionen ☺

1) Recursion

Implementieren Sie die folgenden rekursiven Funktionen über Listen. Lesen Sie in Hoogle nach wie sich die originalen Funktionen (ohne ') verhalten.
Schreiben Sie dazu Tests die mindestens die Abbruchbedingung und mindestens einen Rekursionsschritt überprüfen.

```
drop'   :: Int -> [a] -> [a]
take'   :: Int -> [a] -> [a]
zip'    :: [a] -> [b] -> [(a,b)]
elem'   :: Eq a => a -> [a] -> Bool
eqList  :: Eq a => [a] -> [a] -> Bool -- Vergleicht zwei Listen
```

2) Types

Gesucht ist jeweils der Typ des Ausdrucks. Für numerische Typen können Sie Int annehmen.

```
filter (\x -> True)           ::

map (\a -> 1) []              ::

(\a -> \b -> (a,b,b)) 1       ::

(\(x,y) -> \(x,z) -> (x,z+y)) ::

(\f -> f 2 == "WOW")          ::
```

3) Operatoren

Die Funktionskomposition `(.)` liest sich für viele verkehrt herum. `(f.g)` bedeutet ja, zuerst g auf das Argument anzuwenden und dann f auf das Resultat von g anzuwenden.
In dieser Aufgabe implementieren Sie den Pipe Operator `(|>)`. `f |> g` bedeutet, zuerst f anzuwenden und das Resultat in g reinzugeben.

Folgende Pipeline von Funktionen lässt sich damit ausdrücken:

```
pipe = (fst |> (*2) |> even)

pipe (3,"Three")
~> True
```

a) Überlegen Sie sich zuerst den Typ des |> Operators:

```
(|>) ::
```

b) Geben Sie die Definition: