Fachhochschule Nordwestschweiz

Haskell Lab

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

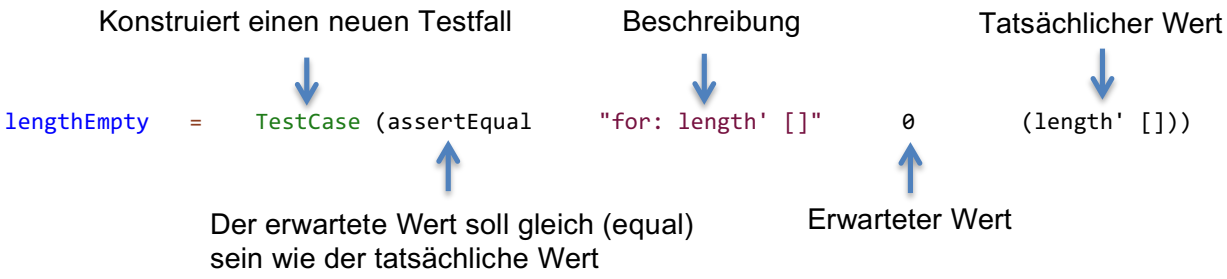
HUnit – Unittesting in Haskell

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

Wir betrachten nun gleich an einem Beispiel wie das funktioniert:

```
-- Hier importieren Sie die verwendete Bibliothek
import Test.HUnit

-- Hier ist ihre Definition der Funktion die Sie testen wollen
length' []     = 0
length' (_:xs) = 1 + length' xs

-- Und hier kommen die Tests (wird weiter unten detailiert erklärt):
lengthEmpty     = TestCase (assertEqual "for: length' []"    0 (length' []))
lengthSingleton = TestCase (assertEqual "for: length' [1]"   1 (length' [1]))
lengthTwoElems  = TestCase (assertEqual "for: length' [1,2]" 3 (length' [1,2]))

-- Eine Liste aller Tests die Sie ausführen wollen
lengthTests = TestList [lengthEmpty, lengthSingleton, lengthTwoElems]

-- Funktion zum Ausführen der Tests
main = runTestTT lengthTests
```

Das einzige was eine Erklärung bedarf ist die Definition der Tests:



Wenn Sie die Tests nun laufen lassen, wird folgender Report ausgegeben:

```
### Failure in: 2
for: length' [1,2]
expected: 3
but got: 2
Cases: 3  Tried: 3  Errors: 0  Failures: 1
Counts {cases = 3, tried = 3, errors = 0, failures = 1}
```

Ein Test ist fehlgeschlagen. Ist ja klar, wir haben einen Fehler im Test. Die Länge der Liste [1,2] ist zwei und nicht drei.

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
words'  :: String -> [String]
```

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: