PES
Technische Universität Berlin
Software Engineering for Embedded Systems Group – Prof. Dr. Sabine Glesner
www.pes.tu-berlin.de Secr. TEL 12-4 Ernst-Reuter-Platz 7 10587 Berlin

Softwaretechnik und Programmierparadigmen WiSe 2014/2015

Prof. Dr. Sabine Glesner
Joachim Fellmuth joachim.fellmuth@tu-berlin.de
Dr. Thomas Göthel thomas.goethel@tu-berlin.de
Lydia Mattick lydia.mattick@tu-berlin.de
Tutoren

Übungsblatt 14 **Ausgabe: 29.01. (Besprechung: 02.02. und 03.02.)**

1. Programmierparadigmen

Diskutiert den Begriff 'Programmierparadigmen'!

a) Welche Programmierparadigmen sind bekannt?

b) Was ist der Unterschied zwischen funktionaler und imperativer Programmierung?

c) Lässt sich die Metrik der zyklomatischen Komplexität auch auf funktionale Programmierung anwenden? Wo liegen die Schwierigkeiten?

2. Einführung Haskell

Haskell ist eine funktionale Programmiersprache basierend auf dem Lambda-Kalkül. GHC (Glasgow Haskell Compiler) ist ein opensource Compiler für Haskell, welcher auch in Haskell geschrieben ist. Der GHC arbeitet problemlos auf beiden Standards von Haskell: *Haskell 98* und *Haskell 2010*. In der funktionalen Programmierung erfolgt eine Aufteilung zwischen Signatur und Definitionsbereich. Eine Signatur beschreibt den Namen und Typen des Wertes. Beispiel:

```
Signatur:   funktionsName :: Typ
Definition: funktionsName  = Ausdruck

Signatur: myFunction :: Int
Definition: myFunction = 5
```

Dabei wird in der Signatur genau beschrieben wie diese Funktion aufgebaut ist und aufgerufen werden muss.

```
Signatur:    firstFunction :: Int -> Int
Definition:  firstFunction x = x + 5
```

In vielen funktionalen Programmiersprachen wird das Kreuzprodukt in der Signatur verwendet, um die Parameter und den Aufruf der Funktion auszudrücken. Haskell dagegen verwendet das *Currying*-Prinzip. Beim *Currying* werden mehrere Argumente einer Funktion in eine Funktion mit einem Argument umgewandelt.

```
Signatur:    justCurry :: Int -> Int -> Int
Definition:  justCurry x y = x + y

-- Aufruf mit justCurry 5 6
-- Oder:     (justCurry 5) 6
```

a) Wie sieht eine anonyme Funktion aus?

3. Funktional Programmieren mit Haskell

Eure Firma hat den Auftrag von SWTPP erhalten einen Taschenrechner in Haskell zu programmieren. Jedoch soll der Taschenrechner mehr als die üblichen Funktionen beherrschen.

a) Euer Taschenrechner ist in der Lage die Konstante der Erdanziehungskraft G anzugeben (G = 9.81). Implementiert eine Konstante-Funktion *gravity*, die diese Konstante ausgibt.

b) Wie jeder Taschenrechner soll auch dieser Taschenrechner eine simple Addition durchführen können. Schreibt eine Funktion *add* welche zwei ganze Zahlen als Eingabe erhält und die Summe der Zahlen als Ergebnis liefert.

c) Eine Fakultätsberechnung soll der Taschenrechner auch bieten. Schreibt eine Funktion *fac*, welche eine ganze positive Zahl als Eingabe erhält und die Fakultät dieser Zahl berechnet.

d) Nun soll auch die Potenzfunktion in unserem Taschenrechner zur Verfügung stehen. Schreibt eine Funktion *pow*, welche zwei ganze Zahlen als Eingabe erhält und die Potenz berechnet, bsp.: pow 2 3 = 2^3

4. Eigene Datentypen

Euer Taschenrechner beherrscht nun einige Grundfunktionen und Zusatzfeatures. Jetzt hat der Auftraggeber noch einige seltsame Anforderungen. Überlegt euch eine geeigneten Datentypen und implementiert diesen sowie eine Funktion *intOp*, die diese Operationen ausführen kann.

a) Die eingegebene Zahl wird verdoppelt und dann inkrementiert (+1). Name: *DblOne*

b) Die eingegebene Zahl wird quadriert und dann inkrementiert. Name: *SqrOne*

c) Die eingegebene Zahl wird quadriert und dann verdoppelt. Name: *SqrDbl*

d) Die eingegebene Zahl wird ohne eine Änderung wieder ausgegeben. *NothingToDo*

5. Listen in Haskell

Euer Auftraggeber spielt verrückt und will jetzt noch eine Listenfunktion im Taschenrechner haben. Bei Eingabe einer Liste von Zahlen, soll die Liste in umgekehrter Reihenfolge wieder ausgegeben werden. Dafür soll eine Funktion *myreverse* bereitgestellt werden, die die folgende Struktur hat.

```
myreverse :: [Int] -> [Int]
```

6. Polymorphie

Oft ist es der Fall, dass eine Operation auf verschiedene Datentypen angewendet werden kann. So kann eine Addition u.a. mit Integer-Werten oder Gleitkommazahlen durchgeführt werden. Da in der Signatur jedoch festgelegt werden muss, welcher Datentyp verwendet wird, ist die Einführung der Polymorphie praktisch. Durch die Polymorphie muss ein Programmierer nicht die Typen der Funktionen angeben. Nun soll das Prinzip des Polymorphismus angewendet werden, denn die von euch entwickelte *myreverse*-Funktion lässt sich aktuell nur auf *Int*-Werte anwenden. Jedoch wollen wir dies für alle möglichen Datentypen offen halten. Das heißt der Benutzer des Taschenrechners soll entscheiden, welchen Typ Liste dieser erstellt. Erstellt eine polymorphe *mypolyreverse*-Funktion, die jede Liste in umgekehrter Reihenfolge wieder ausgibt.