3. Aufgabenblatt zu Funktionale Programmierung vom Mi, 02.11.2016. Fällig: Mi, 09.11.2016 (15:00 Uhr)

Themen: *Funktionen über algebraischen Datentypen, Zahldarstellungen und Funktionale*

Für dieses Aufgabenblatt sollen Sie die zur Lösung der unten angegebenen Aufgabenstellungen zu entwickelnden Haskell-Rechenvorschriften in einer Datei namens `Aufgabe3.lhs` im home-Verzeichnis Ihres Accounts auf der Maschine g0 ablegen. **Anders** als bei der Lösung zu den ersten beiden Aufgabenblättern sollen Sie dieses Mal also ein "**literate Script**" schreiben. Versehen Sie wie auf den bisherigen Aufgabenblättern alle Funktionen, die Sie zur Lösung benötigen, mit ihren Typdeklarationen und kommentieren Sie Ihre Programme aussagekräftig. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten.

Zur Frist der Zweitabgabe: Siehe allgemeine Hinweise auf Aufgabenblatt 1.

1. Wir betrachten folgende Typen zur Darstellung ganzer Zahlen:

```
data Digit      = Zero | One | Two deriving (Eq,Enum,Show)
type Digits     = [Digit]
data Sign       = Pos | Neg  deriving (Eq,Show) -- Pos fuer Positive,
                                                -- Neg fuer Negative
newtype Numeral = Num (Sign,Digits) deriving (Eq,Show)
```

   Numerale erlauben die Darstellung ganzer Zahlen zur Basis 3. Als kanonische Darstellung der Dezimalzahl 0 wählen wir die Darstellung `Num (Pos,[Zero])`, als kanonische Darstellung von 0 verschiedener Dezimalzahlen z die Darstellung von z als Numeral zur Basis 3 ohne führende Nullen.

   Die Dezimalzahl 10 hat mit diesen Festlegungen als Wert vom Typ `Numeral` die (eindeutig bestimmte) Darstellung `Num (Pos,[One,Zero,One])`, die Dezimalzahl -21 die (eindeutig bestimmte) Darstellung `Num (Neg,[Two,One,Zero])`

   Die Repräsentationen `Num (Pos,[])` und `Num (Neg,[])` sind keine gültigen Zahldarstellungen.

   Schreiben Sie drei Haskell-Rechenvorschriften

```
canonize :: Numeral -> Numeral
int2num :: Integer -> Numeral
num2int :: Numeral -> Integer
```

   die Numerale in ihre kanonische Darstellung gleichen Werts sowie Dezimalzahlen in ihre kanonische Numeraldarstellungen und umgekehrt überführen. Angewendet auf ein Numeral d liefert die Funktion `canonize` die kanonische Darstellung des von d repräsentierten Wertes. Angewendet auf eine Dezimalzahl n liefert die Funktion `int2Num` die kanonische Numeraldarstellung von n; angewendet auf eine gültige Numeraldarstellung d (gleich, ob kanonisch oder nicht), liefert die Funktion `num2int` den Dezimalwert der wertgleichen kanonischen Darstellung von d; ist d eine nichtgültige Zahldarstellung, bricht die Auswertung von `num2int` mit dem Aufruf von `error "Invalid Argument"` ab.

   Anwendungsbeispiele:

```
canonize (Num (Neg,[Zero])) ->> Num (Pos,Zero)
canonize (Num (Pos,[Zero,Zero,Zero])) ->> Num (Pos,[Zero])
canonize (Num (Neg,[Zero,Zero,Two,One,Zero])) --> Num (Neg,[Two,One,Zero])
int2num 10 ->> Num (Pos,[One,Zero,One])
num2int (Num (Neg,[One,Zero,One])) ->> -10
num2int (Num (Neg,[Zero,Zero,Zero,One,Zero,One])) ->> -10
num2Int (Num (Neg,[Zero])) ->> 0
```

2. Schreiben Sie Haskell-Rechenvorschriften

```
inc :: Numeral -> Numeral
dec :: Numeral -> Numeral
```

zum inkrementieren und dekrementieren (ohne mit Argumenten oder Zwischenergebnissen den Datentyp `Numeral` zu verlassen). Werden `inc` und `dec` auf nichtgültige Zahldarstellungen angewendet, bricht ihre Auswertung mit dem Aufruf von `error "Invalid Argument"` ab; werden sie auf gültige Zahldarstellungen angewendet, gleich ob kanonisch oder nicht, liefern sie die kanonische Darstellung des inkrementierten bzw. dekrementierten Werts ihres Arguments.

Anwendungsbeispiele:

```
inc (Num (Neg,[One,Zero,One])) ->> Num (Neg,[One,Zero,Zero])
inc (Num (Pos,[Zero,One,Zero,One])) ->> Num (Pos,[One,Zero,Two])
dec (Num (Neg,[Zero,Zero,One,Zero,One])) ->> Num (Neg,[One,Zero,Two])
dec (Num (Pos,[Zero,Zero,One,Zero,One])) ->> Num (Pos,[One,Zero,Zero])
```

3. Schreiben Sie zwei rekursive Haskell-Rechenvorschriften

```
numAdd  :: Numeral -> Numeral -> Numeral
numMult :: Numeral -> Numeral -> Numeral
```

zur Addition bzw. Multiplikation (kanonischer und nichtkanonischer) gültiger Zahldarstellungen vom Typ `Numeral`. Die Implementierung von `numAdd` soll sich dabei rekursiv auf `inc` und `dec` abstützen, die von `numMult` auf die von `numAdd` und ggf. `inc` und `dec`. In jedem Fall soll das Ergebnis in kanonischer Darstellung ausgegeben werden. Ist mindestens eines der Argumente der Funktionen `numAdd` oder `numMult` keine gültige Zahldarstellung, so bricht die jeweilige Auswertung mit dem Aufruf von `error "Invalid Argument"` ab.

4. Gegeben seien die Funktionale `curry`, `uncurry` und `flip`:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f (x,y) = f x y

flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x
```

Schreiben Sie in Analogie zu diesen Funktionalen entsprechende Funktionale

- `curryFlip   :: (((a,b) -> c) -> (b -> (a -> c)))`
- `uncurryFlip :: (((a -> (b -> c)) -> ((b,a) -> c)))`
- `pairFlip    :: (((a,b) -> c) -> ((b,a) -> c))`

die die Wirkung von `curry` und `flip` und von `uncurry` und `pairFlip` kombinieren.

Verzichten Sie dabei in den Signaturen von `curryFlip`, `uncurryFlip` und `pairFlip` und den auftretenden Funktionstermen in den Definitionen dieser Rechenvorschriften unter Ausnutzung der Klammereinsparungsregeln auf so viele Klammern wie möglich, ohne die Bedeutung der Typsignaturen und Funktionsterme zu verändern.

Überlegen Sie sich geeignete Funktionen, um die korrekte Arbeitsweise der Funktionale `pairFlip`, `curryFlip` und `uncurryFlip` zu überprüfen und testen. Implementieren Sie diese Funktionen (soweit nicht schon vordefiniert) und testen Sie Ihre Implementierungen der Funktionale damit.

Wichtig: Wenn Sie einzelne Rechenvorschriften aus früheren Lösungen für dieses oder spätere Aufgabenblätter wieder verwenden möchten, so kopieren Sie diese in die neue Abgabedatei ein. Ein `import` schlägt für die Auswertung durch das Abgabeskript fehl, weil Ihre alte Lösung, aus der importiert wird, nicht mit abgesammelt wird. Deshalb: Kopieren statt importieren zur Wiederwendung!

Denken Sie bitte daran, dass Sie für die Lösung dieses Aufgabenblatts ein “literate” Haskell-Skript schreiben sollen!

Haskell Live

An einem der kommenden Termine werden wir uns in *Haskell Live* mit den Beispielen der ersten beiden Aufgabenblätter beschäftigen, sowie mit der Aufgabe *City-Maut*.

City-Maut

Viele Städte überlegen die Einführung einer City-Maut, um die Verkehrsströme kontrollieren und besser steuern zu können. Für die Einführung einer City-Maut sind verschiedene Modelle denkbar. In unserem Modell liegt ein besonderer Schwerpunkt auf innerstädtischen Nadelöhren. Unter einem *Nadelöhr* verstehen wir eine Verkehrsstelle, die auf dem Weg von einem Stadtteil A zu einem Stadtteil B in der Stadt passiert werden muss, für den es also keine Umfahrung gibt. Um den Verkehr an diesen Nadelöhren zu beeinflussen, sollen an genau diesen Stellen Mautstationen eingerichtet und Mobilitätsgebühren eingehoben werden.

In einer Stadt mit den Stadtteilen A, B, C, D, E und F und den sieben in beiden Richtungen befahrbaren Routen B–C, A–B, C–A, D–C, D–E, E–F und F–C führt jede Fahrt von Stadtteil A in Stadtteil E durch Stadtteil C. C ist also ein Nadelöhr und muss demnach mit einer Mautstation versehen werden.

Schreiben Sie ein Programm in Haskell oder in einer anderen Programmiersprache Ihrer Wahl, das für eine gegebene Stadt und darin vorgegebene Routen Anzahl und Namen aller Nadelöhre bestimmt.

Der Einfachheit halber gehen wir davon aus, dass anstelle von Stadtteilnamen von 1 beginnende fortlaufende Bezirksnummern verwendet werden. Der Stadt- und Routenplan wird dabei in Form eines Tupels zur Verfügung gestellt, das die Anzahl der Bezirke angibt und die möglichen jeweils in beiden Richtungen befahrbaren direkten Routen von Bezirk zu Bezirk innerhalb der Stadt. In Haskell könnte dies durch einen Wert des Datentyps `CityMap` realisiert werden:

```
type Bezirk     = Integer
type AnzBezirke = Integer
type Route      = (Bezirk,Bezirk)
newtype CityMap = CM (AnzBezirke,[Route])
```

Gültige Stadt- und Routenpläne müssen offenbar bestimmten Wohlgeformtheitsanforderungen genügen. Überlegen Sie sich, welche das sind und wie sie überprüft werden können, so dass Ihr Nadelöhrsuchprogramm nur auf wohlgeformte Stadt- und Routenpläne angewendet wird.