3. Aufgabenblatt zu Funktionale Programmierung vom Mi, 31.10.2018.
Fällig: Mi, 07.11.2018 (15:00 Uhr)

Themen: *Funktionen über Typsynonymen und neuen Typen*

Zur Frist der Zweitabgabe: Siehe „Hinweise zu Organisation und Ablauf der Übung“ auf der Homepage der LVA.

Aufgabe

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 **'literates' Skript** 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 (Wertvereinbarungen für) Konstanten.

1. Zwischen zwei (möglicherweise gleichen) ganzen Zahlen klafft eine *Schlucht* der Breite n, $n \in IN_0$, wenn zwischen ihnen n ganze Zahlen fehlen.

   Ist l eine Liste ganzer Zahlen, so wollen wir die aufsteigend sortierten Teillisten maximaler Länge aller Zahlen aus l bestimmen, zwischen denen jeweils eine Schlucht der Breite n klafft. Schreiben Sie dafür eine Haskell-Rechenvorschrift `schluchtenfinder` über den Typsynonymen `Nat0`, `Zett`, `Schluchtenbreite`, `Terrain` und `Schluchtenliste`, die diese Aufgabe erfüllt:

   ```
   > type Nat0             = Int
   > type Zett             = Int
   > type Terrain          = [Zett]
   > type Schluchtenbreite = Nat0
   > type Schluchtenliste  = [Terrain]

   > schluchtenfinder :: Terrain -> Schluchtenbreite -> Schluchtenliste
   ```

   Aufrufbeispiele:

   ```
   schluchtenfinder 0 [11,3,6,2,42,7,13,1,42,2,10] ->> [[1,2,2,3],[6,7],[10,11],[13],[42,42]]
   schluchtenfinder 1 [11,3,6,2,42,7,13,1,42,2,10] ->> [[11,13]]
   schluchtenfinder 3 [14,25,2,17,6,16,8,21,12]    ->> [[2,6],[8,12],[17,21,25]]
   ```

2. Unsere Verabredung, dass `Nat0` ausschließlich für positive ganze Zahlen einschließlich der 0 stehen soll, ist einzig durch Einhaltung unserer Programmierdisziplin sichergestellt. Um Typsicherheit zu erreichen, führen wir mithilfe einer `newtype`-Deklaration einen neuen Typ ein, für den wir den Typnamen `IN_0` verwenden, was wir als ASCII-Approximation von IN_0 verstehen; die Ergänzung der `deriving`-Klausel erlaubt, dass `IN_0`-Werte auf dem Bildschirm ausgegeben werden können (s. Kapitel 4.3 der Vorlesung):

   ```
   > newtype IN_0 = IN_0 Integer deriving Show
   ```

   Werte des Typs `IN_0` können wir nicht direkt mithilfe der üblichen arithmetischen Operatoren und Relatoren verknüpfen und vergleichen:

   ```
   > m = IN_0 17
   > n = IN_0 4

   m + n  ->> Typfehler in Anwendung
   m * n  ->> Typfehler in Anwendung
   m == n ->> Typfehler in Anwendung
   m > n  ->> Typfehler in Anwendung
   ```

Schreiben Sie daher typspezifische arithmetische Operationen und Relationen, um mit Werten des Typs `IN_0` wie für natürliche Zahlen erwartet rechnen zu können:

```
> nplus        :: IN_0 -> IN_0 -> IN_0
> nminus       :: IN_0 -> IN_0 -> IN_0
> nmal         :: IN_0 -> IN_0 -> IN_0
> ndurch       :: IN_0 -> IN_0 -> IN_0
> ngleich      :: IN_0 -> IN_0 -> Bool
> nungleich    :: IN_0 -> IN_0 -> Bool
> ngroesser    :: IN_0 -> IN_0 -> Bool
> nkleiner     :: IN_0 -> IN_0 -> Bool
> ngrgleich    :: IN_0 -> IN_0 -> Bool
> nklgleich    :: IN_0 -> IN_0 -> Bool
> ist_ngueltig :: IN_0 -> Bool
```

Dabei soll die Bedeutung der obigen Operatoren und Relatoren auf `IN_0` folgenden Operationen und Relationen auf der Menge der ganzen Zahlen $\mathbb{Z}$ entsprechen, wobei $IB =_{df} \{falsch, wahr\}$ die Menge der Wahrheitswerte $falsch$ und $wahr$ bezeichnet.

Operationen:

$$plus : \mathbb{Z} \times \mathbb{Z} \to IN_0 \text{ mit } plus(z, z') =_{df} \begin{cases} z + z' & \text{falls } z, z' \in IN_0 \\ 0 & \text{sonst} \end{cases}$$

$$minus : \mathbb{Z} \times \mathbb{Z} \to IN_0 \text{ mit } minus(z, z') =_{df} \begin{cases} maximum(0, z - z') & \text{falls } z, z' \in IN_0 \\ 0 & \text{sonst} \end{cases}$$

$$mal : \mathbb{Z} \times \mathbb{Z} \to IN_0 \text{ mit } mal(z, z') =_{df} \begin{cases} z * z' & \text{falls } z, z' \in IN_0 \\ 0 & \text{sonst} \end{cases}$$

$$durch : \mathbb{Z} \times \mathbb{Z} \to IN_0 \text{ mit } (z, z') =_{df} \begin{cases} z \text{ div } z' & \text{falls } z \in IN_0, z' \in IN_1 \\ 0 & \text{sonst} \end{cases}$$

Relationen:

$$gleich : \mathbb{Z} \times \mathbb{Z} \to IB \text{ mit } gleich(z, z') =_{df} \begin{cases} z = z' & \text{falls } z, z' \in IN_0 \\ falsch & \text{sonst} \end{cases}$$

$$ungleich : \mathbb{Z} \times \mathbb{Z} \to IB \text{ mit } ungleich(z, z') =_{df} \begin{cases} z \neq z' & \text{falls } z, z' \in IN_0 \\ falsch & \text{sonst} \end{cases}$$

$$groesser : \mathbb{Z} \times \mathbb{Z} \to IB \text{ mit } groesser(z, z') =_{df} \begin{cases} z > z' & \text{falls } z, z' \in IN_0 \\ falsch & \text{sonst} \end{cases}$$

$$kleiner : \mathbb{Z} \times \mathbb{Z} \to IB \text{ mit } kleiner(z, z') =_{df} \begin{cases} z < z' & \text{falls } z, z' \in IN_0 \\ falsch & \text{sonst} \end{cases}$$

$$grgleich : \mathbb{Z} \times \mathbb{Z} \to IB \text{ mit } grgleich(z, z') =_{df} \begin{cases} z \geq z' & \text{falls } z, z' \in IN_0 \\ falsch & \text{sonst} \end{cases}$$

$$klgleich : \mathbb{Z} \times \mathbb{Z} \to IB \text{ mit } klgleich(z, z') =_{df} \begin{cases} z \leq z' & \text{falls } z, z' \in IN_0 \\ falsch & \text{sonst} \end{cases}$$

$$ist\_gueltig : \mathbb{Z} \to IB \text{ mit } ist\_gueltig(z) =_{df} \begin{cases} wahr & \text{falls } z \in IN_0 \\ falsch & \text{sonst} \end{cases}$$

Anmerkung: Typname und Datenkonstruktorname sind in der Deklaration `newtype IN_0 = IN_0 Integer` ident gewählt. Diese Wahl ist in Übereinstimmung mit üblichen Namens(verwendungs)konventionen in Haskell getroffen und erspart das (manchmal schwierige) Ersinnen eines zweiten 'guten' Namens. Probleme entstehen daraus nicht, da für jedes Vorkommen von `IN_0` im Programmtext aus dem Kontext hervorgeht, ob der Typname oder der Datenkonstruktorname `IN_0` gemeint ist.

3. Das Komprimierungsverfahren von Aufgabenblatt 2 erzielt keine Ersparnis bei der Komprimierung von Läufen der Längen 2 und 1. Für Läufe der Länge 1 ist das Komprimierungsvorgehen sogar kontraproduktiv, da zwei Zeichen zur Kodierung eines einzigen Zeichens benötigt werden.

   Aus dieser Beobachtung resultiert folgende Optimierungsidee: Nur Läufe der Länge 3 und größer werden nach dem Verfahren von Aufgabenblatt 2 komprimiert, kürzere Läufe jedoch unverändert und unmittelbar in die Komprimierung aufgenommen.

Angewandt auf das illustrierende Beispiel von Aufgabenblatt 2 erhalten wir mit dieser Optimierungsidee folgende komprimierte Form:

```
"aaaaBBBaaBBBBBBccccccccccccAbCbDDDDDDDDDDDDDDDDDD" ->> "4a3Baa6B12cAbCb18D"
```

Schreiben Sie zwei Haskell-Rechenvorschriften `opt_komp` und `opt_expnd`

```
> type Xp_Zf = String   -- fuer eXPandierte ZeichenFolge
> type Kp_Zf = String   -- fuer KomPrimierte ZeichenFolge

> opt_komp  :: Xp_Zf -> Kp_Zf
> opt_expnd :: Kp_Zf -> Xp_Zf
```

die diese Optimierungsidee von Komprimierung und Expansion umsetzen und sich angesetzt auf 'ungeeignete' Zeichenfolgen im Sinn von Aufgabenblatt 2 wie ihre nichtoptimierten Vorbilder `komp` und `expnd` von Aufgabenblatt 2 verhalten.

4. Schreiben Sie zwei weitere Haskell-Rechenvorschriften `komp'` und `expnd'`, die nach derselben Idee und unter den gleichen Randbedingungen, von Ziffern freie Zeichenreihen komprimieren und wieder expandieren, jedoch auf folgenden Datentypen operieren:

   ```
   > newtype Xp_Zf'   = Xp String deriving Show
   > type Nat1        = Int
   > type Lauflaenge = Nat1
   > type Zeichen     = Char
   > newtype Kp_Zf'   = Kp [(Lauflaenge,Zeichen)] deriving Show

   > komp'  :: Xp_Zf' -> Kp_Zf'
   > expnd' :: Kp_Zf' -> Xp_Zf'
   ```

5. Schreiben Sie zwei Konversionsfunktionen `einpacken` und `auspacken`, die eine Zeichenfolge als `Xp_Zf`-Wert in den entsprechenden `Xp_Zf'`-Wert umwandeln und umgekehrt und implementieren Sie mit deren Hilfe und den Funktionen `komp'` und `expnd'` zwei weitere Komprimierungs- und Expansionsfunktionen `komp''` und `expnd''`, die Zeichenreihen in die optimierte Komprimierungsform überführen und diese wieder zur Zeichenreihe expandieren.

   ```
   > einpacken :: Xp_Zf -> Xp_Zf'
   > auspacken :: Xp_Zf' -> Xp_Zf
   > komp''     :: Xp_Zf -> Kp_Zf'
   > expnd''    :: Kp_Zf' -> Xp_Zf
   ```

6. Schreiben Sie eine Haskell-Rechenvorschrift

   ```
   > aufteilen3 :: String -> [(String,String,String)]
   ```

   die angewendet auf eine Zeichenreihe z eine Liste von Tripeln von Zeichenreihen liefert, die alle Tripelaufspaltungen von z darstellt. Folgendes Beispiel veranschaulicht die Bedeutung der Funktion:

   ```
   aufteilen3 "Fun" ->> [("","","Fun"),("","F","un"),("","Fu","n"),("","Fun",""),
                         ("F","","un"),("F","u","n"),("F","un",""),("Fu","","n"),
                         ("Fu","n",""),("Fun","","")]
   ```

Hinweis: Wenn Sie einzelne Rechenvorschriften aus früheren Lösungen 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 'literates' Haskell-Skript schreiben sollen!

Haskell Live

An einem der kommenden *Haskell Live*-Termine (der nächste ist aufgrund des übermorgigen Feiertags am Freitag, den 09.11.2018) werden wir uns in *Haskell Live* mit den Beispielen der ersten beiden Aufgabenblätter beschäftigen, sowie mit der Aufgabe *City-Maut*.

City-Maut

Viele Städte, auch Wien, überlegen die Einführung einer City-Maut, um die Verkehrsströme besser kontrollieren und 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     = Int
type AnzBezirke = Int
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.