4. Aufgabenblatt zu Funktionale Programmierung vom Mi, 07.11.2018.
Fällig: Mi, 14.11.2018 (15:00 Uhr)

Themen: *Funktionen über neuen und algebraischen Datentypen, Typklassen, Instanzbildungen*

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

Aufgabe

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe4.hs` ablegen. Sie sollen für die Lösung dieses Aufgabenblatts also wieder ein ‘gewöhnliches’ Haskell-Skript schreiben.

Versehen Sie wieder 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. Wir betrachten den Datentyp

   ```
   newtype IN_0 = IN_0 Integer
   ```

   von Aufgabenblatt 3 (ohne `deriving`-Klausel für die Typklasse `Show`).

   Machen Sie den Typ `IN_0` mithilfe einer Instanzdeklaration zu einer Instanz der Typklasse `Show`. Dabei soll gelten: Im Sinn von Aufgabenblatt 3 gültige Werte vom Typ `IN_0` werden als Zeichenreihen über dem Zeichenvorrat von Ziffern {‘0‘,‘1‘,‘2‘,‘3‘,‘4‘,‘5‘,‘6‘,‘7‘} dargestellt, die die Oktaldarstellung des Wertes darstellen (mit Ausnahme der Null ohne führende Nullen); nichtgültige Werte sollen durch die Zeichenfolge `"Nicht gueltig!"` dargestellt werden.

   Beispiele:

IN_0-Wert	Darstellung
IN_0 0	"0"
IN_0 1	"1"
IN_0 2	"2"
IN_0 7	"7"
IN_0 8	"10"
IN_0 9	"11"
IN_0 (-1)	"Nicht gueltig!"
IN_0 (-2)	"Nicht gueltig!"
IN_0 (-9)	"Nicht gueltig!"

2. Wir führen den dreiwertigen Datentyp `Antwort3`, ein Typsynonym `A3` als Abkürzung für `Antwort3` und die Typklasse `Gueltig` ein:

   ```
   data Antwort3 = Ja | Nein | Teilsteils deriving (Eq,Ord,Show)
   type A3       = Antwort3

   class Gueltig a where
    ist_gueltig :: a -> A3
   ```

   Machen Sie den Datentyp `IN_0` zu einer Instanz der Typklasse `Gueltig`, so dass angewendet auf `IN_0`-Werte die Funktion `ist_gueltig` der Typklasse `Gueltig` im Sinn von Aufgabenblatt 3 gültige `IN_0`-Werte auf `Ja` abbildet, nichtgültige auf `Nein`.

3. Machen Sie die Typen `Int`, `Integer`, `Double` und `Float` ebenfalls zu Instanzen der Typklasse `Gueltig`. Dazu legen wir fest:

- Alle Werte der Typen `Int` und `Integer` sind gültig.
- Kein Wert des Typs `Double` ist gültig.
- Sei `z.0` die `Float`-Repräsentation der ganzen Zahl z (z.B. ist `2.0` die `Float`-Repräsentation von 2, `-3.0` die von -3). Ein `Float`-Wert `w` ist ungültig, wenn für alle Zahlen z, $z \in \mathbb{Z}$, der Vergleich der Differenz von `w` und `z.0` mit dem Float-Wert `0.0` falsch ergibt, gültig sonst.

4. In der Praxis ist es lästig, dass wir für das Rechnen mit `IN_0`-Werten (bislang) nicht die üblichen arithmetischen Operator- (`(+)`, `(*)`, `(-)`,...) und Relatorsymbole (`(==)`, `(/=)`,...) verwenden können, sondern unsere auf Aufgabenblatt 3 eingeführten Nichtstandardoperationen und -relationen `nplus`, `nmal`, `nminus`, `ngleich`, `nungleich`,... verwenden müssen. Ein naiver Abhilfeversuch in Form von Deklarationen

```
(+) :: IN_0 -> IN_0 -> IN_0
(+) = nplus
...
(/=) :: IN_0 -> IN_0 -> Bool
(/=) = nungleich
```

schlägt fehl (überlegen Sie sich, warum?), aber wir können unseren Typ `IN_0` zu einer Instanz der Typklassen `Eq, Ord, Num` und `Enum` machen, um die Überladung der Operator- und Relatorsymbole `(+)`, `(*)`, `(-)`, `(==)`, `(/=)`,... und weiterer in diesen Klassen vorgesehener Operatoren und Relatoren wie für Werte der Typen `Int`, `Integer`, `Float`, `Double` vordefiniert auch für Werte des Typs `IN_0` verfügbar zu machen.

Machen Sie dazu den Typ `IN_0` ohne Verwendung von `deriving`-Klauseln zu Instanzen der Typklassen

- `Eq`
- `Ord`
- `Enum`
- `Num`

Dabei soll die Bedeutung der auf Aufgabenblatt 3 für `IN_0`-Werte eingeführten Operationen (`nplus`, `nmal`,...) und Relationen (`ngleich`, `nkleiner`,...) mit einen entsprechenden Operator (`(+)` für `nplus`, `(*)` für `nmal`,...) oder Relator (`(==)` für `ngleich`, `(<)` für `nkleiner`,..) in einer dieser Typklassen auf diese übertragen werden. Die Bedeutung weiterer in den Klassen vorgesehener Operatoren und Relatoren soll in folgender Weise festgelegt werden:

- Typklasse `Ord`:
  - `compare :: a -> a -> Ordering`, `max :: a -> a -> a`, `min :: a -> a -> a`: Festgelegt durch die für `(<)`, `(<=)`, `(>=)`, `(>)` weiter oben getroffenen Vorgaben und die Protoimplementierungen für `compare`, `max` und `min`.
- Typklasse `Enum`:
  - `succ :: a -> a` liefert für gültige `IN_0`-Werte den entsprechenden Nachfolgerwert, für nichtgültige `IN_0`-Werte den Wert Null des Typs `IN_0`.
  - `pred :: a -> a` liefert stets den entsprechenden Vorgängerwert, gleich, ob der Argumentwert gültiger oder nichtgültiger `IN_0`-Wert ist.
  - `toEnum :: Int -> a` bildet nichtnegative Argumente auf den entsprechenden `IN_0`-Wert ab, negative Argumente auf den Wert Null des Typs `IN_0`.
  - Die Funktion `fromEnum :: a -> Int` bildet gültige und nichtgültige `IN_0`-Werte auf den entsprechenden `Int`-Wert ab.
  - Alle übrigen Funktionen in `Enum` identifizieren nichtgültige `IN_0`-Werte als Argument mit dem `IN_0`-Wert Null und verhalten sich ansonsten wie ihre Gegenstücke auf nichtnegativen ganzen Zahlen des Typs `Integer`.
- Typklasse `Num`:
  - `fromInteger :: Integer -> a`: Analog zu `toEnum`.

– `negate :: a -> a`, `abs :: a -> a`, `signum :: a -> a`: Analog zu 'alle übrigen Funktionen in `Enum`'.

Anmerkung: Für Instanzen der Typklasse `Num` wird üblicherweise die Gleichheit `abs x * signum x == x` gefordert. Für die `Num`-Instanz von `IN_0` ist diese Forderung mit obigen Festlegungen für gültige `IN_0`-Werte erfüllt.

Nutzen Sie bei der Instanzbildung für jede der Typklassen (auch für die Typklasse `Show` in Aufgabenteil 1) die in der Klasse gegebenen Protoimplementierungen bestmöglich aus.

Implementieren Sie deshalb bei jeder Instanzbildung nur eine minimal nötige Menge von Funktionen, um das Verhalten aller Funktionen der Typklasse auf Werten vom Typ `IN_0` vollständig festzulegen. Für einige Typklassen haben Sie dabei einen Freiheitsgrad für die Wahl der minimalen Menge, den Sie ausnützen sollen (siehe Kapitel 4.3 der Vorlesung).

5. Wir führen den neuen Typ `Nat_Liste` ein:

```
newtype Nat_Liste = NL [IN_0] deriving (Eq,Show)
```

Ein Wert vom Typ `Nat_Liste` ist gültig gdw. alle Listenelemente gültige `IN_0`-Werte sind.

Machen Sie den Typ `Nat_Liste` zu einer Instanz der Typklasse `Gueltig`. Die Funktion `ist_gueltig` der Klasse `Gueltig` liefert den Wert `Ja`, wenn alle Elemente der Argumentliste gültig sind; den Wert `Nein`, wenn alle Elemente nichtgültig sind und es mindestens ein solches gibt; den Wert `Teilsteils`, wenn es mindestens ein gültiges und ein nichtgültiges Element gibt.

6. Schreiben Sie drei Haskell-Rechenvorschriften

```
summe_integer :: Nat_Liste -> Integer
summe_int     :: Nat_Liste -> Int
tripel_finder :: Nat_Liste -> IN_0 -> [Nat_Liste]
```

mit folgenden Bedeutungen:

- Werden `summe_integer` und `summe_int` auf einen nichtgültigen `Nat_Liste`-Wert angewendet, so liefern sie den Wert -1 als Resultat, anderenfalls die Summe aller Listenelemente, jeweils als `Integer`- bzw. `Int`-Wert.
- Wird `tripel_finder` auf einen nichtgültigen `Nat_Liste`-Wert oder/und nichtgültigen `IN_0`-Wert angewendet, so liefert `tripel_finder` die leere Liste als Resultat, anderenfalls eine Liste paarweise verschiedener Listen mit je 3 Elementen, so dass für jede dieser Listen gilt:
  – Alle drei Elemente sind Elemente der Argumentliste.
  – Kein Element kommt häufiger vor als in der Argumentliste.
  – Die Elemente sind aufsteigend angeordnet.
  – Die Summe der drei Elemente ergibt den Wert des zweiten Arguments von `tripel_finder`.

  Die Reihenfolge der Listen in der Gesamtliste unterliegt keinen Anforderungen.

Aufrufbeispiele:

```
summe_integer (NL [IN_0 1,IN_0 0,IN_0 3,IN_0 2,IN_0 1,IN_0 2]) ->> 9
summe_int (NL [IN_0 1,IN_0 0,IN_0 (-3),IN_0 2,IN_0 1,IN_0 2]) ->> -1
tripel_finder (NL [IN_0 1,IN_0 0,IN_0 3,IN_0 2,IN_0 1,IN_0 2]) (IN_0 5)
 ->> [NL [IN_0 0,IN_0 2,IN_0 3],NL [IN_0 1,IN_0 1,IN_0 3],NL [IN_0 1,IN_0 2,IN_0 2]]
     (oder eine Permutation dieser Liste)
tripel_finder (NL [IN_0 1,IN_0 0,IN_0 (-3),IN_0 2,IN_0 1,IN_0 2]) (IN_0 5) ->> []
```

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!

Haskell Live, Haskell Private

Der nächste *Haskell Live*-Termin findet am Freitag, den 09.11.2018, statt. Die Anmeldung zu *Haskell Private* ist offen. Nutzen Sie die Möglichkeit; auch die Möglichkeit, eigene Terminvorschläge zu machen!