5. Aufgabenblatt zu Funktionale Programmierung vom 18.11.2015.
Fällig: 25.11.2015 / 02.12.2015 (jeweils 15:00 Uhr)

Themen: *Typklassen und Funktionen auf algebraischen Datentypen*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe5.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 brauchen, mit ihren Typdeklarationen und kommentieren Sie Ihre Programme aussagekräftig. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten.

Im einzelnen sollen Sie die im folgenden beschriebenen Problemstellungen bearbeiten.

1. Wir betrachten noch einmal den Datentyp `Nat` mit abgeänderten *deriving*-Klauseln aus Aufgabenteil 1 von Aufgabenblatt 4 zur Modellierung des endlichen Ausschnitts natürlicher Zahlen von 0 bis $maxNat$. Mit Ausnahme der geänderten *deriving*-Klauseln werden alle weiteren Vereinbarungen für den Datentyp `Nat` und die darauf vereinbarten Operationen für diese Aufgabe übernommen. Insbesondere gilt, dass Summe und Produkt zweier Werte aus `Nat` das Minimum aus $maxNat$ und Summe bzw. Produkt dieser Werte ist. Entsprechend ist die Differenz zweier Werte aus `Nat` das Maximum aus 0 und der Differenz dieser Werte, etc.

```
data Zeichen = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P
             | Q | R | S | T | U | V | W | X | Y | Z
                deriving (Eq,Ord,Show,Read,Enum,Bounded)
data Ziffer  = Null | Eins | Zwei | Drei | Vier | Fuenf | Sechs | Sieben
             | Acht | Neun deriving (Eq,Ord,Show,Read,Enum,Bounded)
newtype Nat  = Nat ((Zeichen,Zeichen,Zeichen),(Ziffer,Ziffer,Ziffer))
```

   Im Haskell-Prelude sind u.a. die Typklassen `Eq`, `Ord`, `Show` und `Num` zusammen mit den Signaturen der Rechenvorschriften definiert, die für jede Instanz dieser Typklassen implementiert sein müssen. Da für einige Rechenvorschriften in den Klassendefinitionen bereits Proto-Implementierungen angegeben sind, reicht es bei Instanzbildungen oft aus, für eine Teilmenge der in einer Klasse vorgesehenen Rechenvorschriften typspezifische Implementierungen bei der Instanzbildung anzugeben, wenn sich die so aus den Proto-Implementierungen automatisch ergebenden Vollimplementierungen das gewünschte Verhalten aufweisen. Machen Sie davon bei den folgenden Teilaufgaben Gebrauch und implementieren Sie lediglich die Mininalmenge von Rechenvorschriften und übernehmen überall sonst, wo möglich, die Proto-Implementierung unverändert, d.h. überschreiben Sie diese nicht.

   Machen Sie den Datentyp `Nat` explizit, d.h. ohne Rückgriff auf eine *deriving*-Klausel, zu je einer Instanz der Typklassen

   a) `Eq` (Minimale Vervollständigung: Implementierung von (==) oder (/=): implementieren Sie für diese Aufgabe den Relator (/=).)

   b) `Ord` (Minimale Vervollständigung: Implementierung von (<=) oder `compare`: implementieren Sie für diese Aufgabe den Relator `compare`.)

   c) `Show` (Minimale Vervollständigung: Implementierung von `show` oder `showsPrec`: implementieren Sie für diese Aufgabe die Rechenvorschrift `show`.)

   d) `Num` (Minimale Vervollständigung: Implementierung von (`negate` oder (−)) und aller anderen in der Klasse vorgesehenen Rechenvorschriften: implementieren Sie für diese Aufgabe alle in der Klasse vorgesehenen Rechenvorschriften, also auch beide Rechenvorschriften `negate` und (−).)

   Die Bedeutung aller in den verschiedenen Klassen vorgesehenen Rechenvorschriften soll — soweit nicht anders angegeben, wie etwa für die unter- und überlauffreie Addition, Multiplikation und Subtraktion für Werte des Datentyps `Nat` — in analoger Weise wie für den ebenfalls eingeschränkten Zahlbereich `Int` sein.

   Im einzelnen gelten folgende weitere Abweichungen:

   - Die Rechenvorschrift `negate` liefert für jedes Argument das Resultat 0 als Wert vom Typ `Nat`.

- Die Rechenvorschrift `show` stellt Werte des Typs `Nat` als Zeichenreichen der Form `"XYZ ijk"` dar, wobei `X`, `Y` und `Z` die den Konstruktoren des Typs `Zeichen` entsprechenden Großbuchstaben und `i`, `j` und `k` die den Konstruktoren des Typs `Ziffer` entsprechenden Ziffern von 0 bis 9 sind. Die Buchstaben- und Zifferngruppen sind durch genau ein Leerzeichen voneinander getrennt.

Hinweis: Für die Testung können Sie davon ausgehen, dass die Funktionen nur für Argumente aufgerufen werden, für die bei *vernünftiger* Berechnung keine Gesamt- oder Zwischenergebnisse, auch nicht in Zähler oder Nenner alleine, außerhalb des darstellbaren Zahlbereichs auftreten.

Ohne Abgabe: Vergleichen Sie die Ausgabe Ihrer `show`-Funktion mit derjenigen der automatisch mithilfe der *deriving*-Klausel für `Nat` abgeleiteten `show`-Funktion. Welcher "Darstellungsidee" von Werten des Typs `Nat` folgt die automatisch abgeleitete `show`-Funktion, welcher die von Ihnen selbst implementierte?

2. Mithilfe des Datentyps `Nat` aus Aufgabenteil 1 können wir einen Ausschnitt (positiver) rationaler Zahlen als Paare natürlicher Zahlen darstellen. Wir führen dafür das Typsynonym `PosRat` ein:

```
data Zeichen = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P
             | Q | R | S | T | U | V | W | X | Y | Z
                deriving (Eq,Ord,Show,Read,Enum,Bounded)
data Ziffer  = Null | Eins | Zwei | Drei | Vier | Fuenf | Sechs | Sieben
             | Acht | Neun deriving (Eq,Ord,Show,Read,Enum,Bounded)
newtype Nat  = Nat ((Zeichen,Zeichen,Zeichen),(Ziffer,Ziffer,Ziffer))
type PosRat  = (Nat,Nat)
```

Ein Paar (m, n), $n \neq 0$, aus `PosRat` stellt dabei die (positive) rationale Zahl $\frac{m}{n}$ dar. Die Paare $(m, 0)$, dargestellt als Paare aus `PosRat`, sind für keinen Wert m Darstellung einer (positiven) rationalen Zahl. Diese Paare heißen *ungültig*.

Anders als natürliche Zahlen besitzen (positive) rationale Zahlen keine eindeutige Darstellung in `PosRat`. Eine eindeutige Zahldarstellung in `PosRat` erhalten wir mit folgenden Zusatzvereinbarungen:

- Die Zahl 0 hat die kanonische Darstellung `(Nat (A,A,A),(Null,Null,Null)),Nat (A,A,A),(Null,Null,Eins)))` mit der Bildschirmdarstellung `("AAA 000","AAA 001")`.
- Die Zahl $\frac{m}{n}$, $m, n \neq 0$ hat die kanonische Darstellung $\frac{p}{q}$, $p, q \neq 0$, dargestellt als Paar aus `PosRat`, falls $\frac{m}{n}=\frac{p}{q}$ und p und q teilerfremd sind.

Schreiben Sie Haskell-Rechenvorschriften

1. `isCanPR :: PosRat -> Bool`
2. `mkCanPR :: PosRat -> PosRat`
3. `plusPR :: PosRat -> PosRat -> PosRat`
4. `minusPR :: PosRat -> PosRat -> PosRat`
5. `timesPR :: PosRat -> PosRat -> PosRat`
6. `divPR :: PosRat -> PosRat -> PosRat`

zum Test einer (positiven) rationalen Zahl auf Kanonizität ihrer Darstellung, zur Überführung in ihre eindeutige kanonische Darstellung, sowie zur Addition, Subtraktion, Multiplikation und Division, wobei das Resultat stets in kanonischer Form dargestellt wird. Ist das Argument der Funktion `mkCanPR` bzw. eines der Argumente der Funktionen `plusPR`, `minusPR`, `timesPR` oder `divPR` nicht gültig, oder hat der Divisor, das zweite Argument der Funktion `divPR` den Wert 0 (gleich ob kanonisch oder nicht-kanonisch dargestellt), so liefern die entsprechenden Funktionsaufrufe das den Fehlerfall anzeigende Resultatpaar

```
(Nat (A,A,A),(Null,Null,Null)),Nat (A,A,A),(Null,Null,Null)))
```

mit der Bildschirmdarstellung `("AAA 000","AAA 000")`. Ist das Argument der Funktion `isCanPR` nicht gültig, so liefert sie den Wahrheitswert `False`. Weiters gilt für die Operationen `plusPR` und `timesPR`, dass das Resultat den Wert $maxNat$ hat, wenn das Ergebnis der Operation größer oder gleich dem Wert von $maxNat$ ist. In ähnlicher Weise gilt für die Operation `minusPR`, dass das Resultat den Wert 0 hat, wenn beide Argumente gültig sind und das zweite Argument größer oder gleich dem ersten Argument ist.

Schreiben Sie weiters Haskell-Rechenvorschriften

7. `eqPR :: PosRat -> PosRat -> Bool`
8. `neqPR :: PosRat -> PosRat -> Bool`
9. `grPR :: PosRat -> PosRat -> Bool`
10. `lePR :: PosRat -> PosRat -> Bool`
11. `grEqPR :: PosRat -> PosRat -> Bool`
12. `leEqPR :: PosRat -> PosRat -> Bool`

die angewendet auf zwei Argumente überprüfen, ob die beiden Argumente gleich oder ungleich sind, das erste Argument echt größer bzw. echt kleiner als das zweite Argument ist, das erste Argument größer oder gleich bzw. kleiner oder gleich als das zweite Argument ist. Ist eines der Argumente der Funktionen `eqPR`, `neqPR`, `grPR`, `lePR`, `grEqPR` oder `leEqPR` ungültig, so liefern die Funktionen den Wahrheitswert `False`.

Beachten Sie: Alle Rechenvorschriften zur Behandlung positiver rationaler Zahlen müssen sowohl mit Argumenten in kanonischer als auch in nicht-kanonischer Darstellung umgehen können.

Folgende Beispiele einzelner Funktionen veranschaulichen das gewünschte Ein-/Ausgabeverhalten:

```
isCanPR (Nat ((A,A,A),(Neun,Neun,Acht)),Nat ((A,A,A),(Null,Null,Sechs)))
        ->> False
mkCanPR (Nat ((A,A,A),(Null,Null,Acht)),Nat ((A,A,A),(Null,Null,Sechs)))
        ->> ("AAA 004","AAA 003")
plusPR (Nat ((A,A,A),(Null,Null,Acht)),Nat ((A,A,A),(Null,Null,Vier)))
       (Nat ((A,A,A),(Null,Vier,Zwei)),Nat ((A,A,A),(Null,Null,Eins)))
        ->> ("AAA 044","AAA 001")
minusPR (Nat ((A,A,A),(Null,Null,Acht)),Nat ((A,A,A),(Null,Null,Vier)))
        (Nat ((A,A,A),(Null,Vier,Zwei)),Nat ((A,A,A),(Null,Null,Eins)))
        ->> ("AAA 000","AAA 001")
timesPR (Nat ((A,A,A),(Null,Null,Acht)),Nat ((A,A,A),(Null,Null,Null)))
        (Nat ((A,A,A),(Null,Vier,Zwei)),Nat ((A,A,A),(Null,Null,Eins)))
        ->> ("AAA 000","AAA 000")
eqPR (Nat ((A,A,A),(Null,Null,Acht)),Nat ((A,A,A),(Null,Null,Vier)))
     (Nat ((A,A,A),(Null,Null,Vier)),Nat ((A,A,A),(Null,Null,Zwei)))
        ->> True
leEqPR (Nat ((A,A,A),(Null,Null,Acht)),Nat ((A,A,A),(Null,Null,Vier)))
       (Nat ((A,A,A),(Null,Vier,Zwei)),Nat ((A,A,A),(Null,Null,Eins)))
        ->> True
```

Hinweis: Keine früheren Lösungen als Module importieren!

Wenn Sie zur Lösung einzelne Funktionen früherer Lösungen wiederverwenden möchten, so kopieren Sie diese unbedingt explizit in Ihre neue Programmdatei ein. Importieren schlägt im Rahmen der automatischen Programmauswertung fehl. Es wird nicht nachgebildet. Deshalb: Wiederverwendung ja, aber durch kopieren, nicht durch importieren!

Haskell Live

An einem der kommenden *Haskell Live*-Termine, der nächste ist am Freitag, den 20.11.2015, werden wir uns u.a. mit der Aufgabe *World of Perfect Towers* beschäftigen.

World of Perfect Towers

In diesem Spiel konstruieren wir Welten perfekter Türme. Dazu haben wir n Stäbe, die senkrecht auf einer Bodenplatte befestigt sind und auf die mit einer entsprechenden Bohrung versehene Kugeln gesteckt werden können. Diese Kugeln sind ebenso wie die Stäbe beginnend mit 1 fortlaufend nummeriert.

Die auf einen Stab gesteckten Kugeln bilden einen Turm. Dabei liegt die zuerst aufgesteckte Kugel ganz unten im Turm, die zu zweit aufgesteckte Kugel auf der zuerst aufgesteckten, usw., und die zuletzt aufgesteckte Kugel ganz oben im Turm. Ein solcher Turm heißt *perfekt*, wenn die Summe der Nummern zweier unmittelbar übereinanderliegender Kugeln eine Zweierpotenz ist. Eine Menge von n perfekten Türmen heißt *n-perfekte Welt*.

In diesem Spiel geht es nun darum, n-perfekte Welten mit maximaler Kugelzahl zu konstruieren. Dazu werden die Kugeln in aufsteigender Nummerierung, wobei mit der mit 1 nummerierten Kugel begonnen wird, so auf die n Stäbe gesteckt, dass die Kugeln auf jedem Stab einen perfekten Turm bilden und die Summe der Kugeln aller Türme maximal ist.

Schreiben Sie in Haskell oder einer anderen Programmiersprache ihrer Wahl eine Funktion, die zu einer vorgegebenen Zahl n von Stäben die Maximalzahl von Kugeln einer n-perfekten Welt bestimmt und die Türme dieser n-perfekten Welt in Form einer Liste von Listen ausgibt, wobei jede Liste von links nach rechts die Kugeln des zugehörigen Turms in aufsteigender Reihenfolge angibt.