## 4. Aufgabenblatt zu Funktionale Programmierung vom 12.11.2014. Fällig: 19.11.2014 / 26.11.2014 (jeweils 15:00 Uhr)

Themen: *Funktionen auf algebraischen Datentypen, natürliche und rationale Zahlen*

Für dieses Aufgabenblatt sollen Sie die zur Lösung der unten angegebenen Aufgabenstellungen zu entwickelnden Haskell-Rechenvorschriften in einer Datei namens `Aufgabe4.lhs` im home-Verzeichnis Ihres Accounts auf der Maschine g0 ablegen. Wie bei der Lösung zum dritten Aufgabenblatt sollen Sie auch dieses Mal 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. Im einzelnen sollen Sie die im folgenden beschriebenen Problemstellungen bearbeiten.

**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 "literate" Haskell-Skript schreiben sollen!

Auf diesem Aufgabenblatt beschäftigen wir uns mit Darstellungen natürlicher und positiver rationaler Zahlen und arithmetischen Operationen und Relationen darauf.

In Haskell können wir natürliche Zahlen durch folgenden algebraischen Datentyp realisieren:

```
data Nat = Z | S Nat deriving Show
```

Der Konstruktor `Z` (kurz für "Zero") steht dabei für die Zahl 0, der Konstruktorausdruck `S Z` (kurz für "Successor von Zero") für die Zahl 1, der Konstruktorausdruck `S (S Z)` für die Zahl 2, der Konstruktorausdruck `S (S (S Z))` für die Zahl 3, usw. Auf diese Weise besitzt jede natürliche Zahl eine eindeutige Darstellung als Wert des Datentyps `Nat`.

Schreiben Sie Haskell-Rechenvorschriften

1. `plusN :: Nat -> Nat -> Nat`
2. `minusN :: Nat -> Nat -> Nat`
3. `timesN :: Nat -> Nat -> Nat`
4. `divN :: Nat -> Nat -> Nat`
5. `modN :: Nat -> Nat -> Nat`
6. `powerN :: Nat -> Nat -> Nat`

zur Addition, Subtraktion, Multiplikation, ganzzahligen Division, zur Berechnung des Restes bei ganzzahliger Division und zur Potenzierung. Dabei gilt für die Operation `minusNat`, dass das Resultat den Wert 0 hat, wenn das zweite Argument größer oder gleich dem ersten Argument ist.

Schreiben Sie weiters Haskell-Rechenvorschriften

7. `eqN :: Nat -> Nat -> Bool`
8. `neqN :: Nat -> Nat -> Bool`
9. `grN :: Nat -> Nat -> Bool`
10. `leN :: Nat -> Nat -> Bool`

11. `grEqN :: Nat -> Nat -> Bool`

12. `leEqN :: Nat -> Nat -> Bool`

die angewendet auf zwei Argumente überprüfen, ob die beiden Argumente gleich bzw. 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.

Folgende Beispiele veranschaulichen das gewünschte Ein-/Ausgabeverhalten:

```
plusN (S (S Z)) (S (S (S Z))) ->> S (S (S (S Z))))
minusN (S (S Z)) (S (S (S Z))) ->> Z
timesN (S (S Z)) (S (S (S Z))) ->> S (S (S (S (S Z)))))
divN (S (S Z)) (S (S (S Z))) ->> Z
modN (S (S Z)) (S (S (S Z))) ->> S (S Z)
powerN (S (S Z)) (S (S (S Z))) ->> S (S (S (S (S (S (S Z)))))))
eqN (S (S Z)) (S (S (S Z))) ->> False
neqN (S (S Z)) (S (S (S Z))) ->> True
grN (S (S Z)) (S (S (S Z))) ->> False
leN (S (S Z)) (S (S (S Z))) ->> True
grEqN (S (S Z)) (S (S (S Z))) ->> False
leEqN (S (S Z)) (S (S (S Z))) ->> True
```

Mithilfe des Datentyps `Nat` können wir (positive) rationale Zahlen als Paare natürlicher Zahlen darstellen. Wir führen dafür das Typsynonym `PosRat` ein:

```
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 $(m, 0)$ 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 `(Z,S Z)`.
- 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

13. `isCanPR :: PosRat -> Bool`

14. `mkCanPR :: PosRat -> PosRat`

15. `plusPR :: PosRat -> PosRat -> PosRat`

16. `minusPR :: PosRat -> PosRat -> PosRat`

17. `timesPR :: PosRat -> PosRat -> PosRat`

18. `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` ungültig, 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 `(Z,Z)`. Ist das Argument der Funktion `isCanPR` ungültig, so liefert sie den Wahrheitswert `False`. Weiters 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

19. `eqPR :: PosRat -> PosRat -> Bool`
20. `neqPR :: PosRat -> PosRat -> Bool`
21. `grPR :: PosRat -> PosRat -> Bool`
22. `lePR :: PosRat -> PosRat -> Bool`
23. `grEqPR :: PosRat -> PosRat -> Bool`
24. `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 wie in nicht-kanonischer Darstellung umgehen können.

Folgende Beispiele veranschaulichen das gewünschte Ein-/Ausgabeverhalten:

```
isCanPR (S (S Z),S (S Z)) ->> False
mkCanPR (S (S Z),S (S Z)) ->> (S Z,S Z)
plusPR (S Z,S (S (S Z))) (S Z,S Z) ->> (S (S (S (S Z))),(S (S (S Z))))
minusPR (S Z,S Z) (S Z,S (S (S Z))) ->> (S (S Z),(S (S (S Z))))
timesPR (S Z,S (S Z)) (S Z,S (S Z)) ->> (S Z,S (S (S (S Z))))
divPR (S Z,S (S Z)) (S Z,S (S Z)) ->> (S Z,S Z)
eqPR (S (S Z),S (S Z)) (S Z,S Z) ->> True
neqPR (S (S Z),Z) (S Z,S (S Z)) ->> False
grPR (S Z,S Z) (S Z,S (S (S Z))) ->> True
lePR (S Z,S (S Z)) (S (S Z),S (S (S (S Z)))) ->> False
grEqPR (S Z,S (S Z)) (S (S Z),S (S (S (S Z)))) ->> True
leEqPR (S (S Z),S (S Z)) (S Z,S Z) ->> True
```

## Haskell Live

Am Freitag, den 14.11.2014, werden wir uns in *Haskell Live* mit Lösungsvorschlägen u.a. bereits abgeschlossener Aufgabenblätter beschäftigen, die (gerne auch) von Ihnen eingebracht werden können, sowie mit einigen der schon speziell für *Haskell Live* gestellten Aufgaben.