4. Aufgabenblatt zu Funktionale Programmierung vom 11.11.2015. Fällig: 18.11.2015 / 25.11.2015 (jeweils 15:00 Uhr)

Themen: *Funktionen auf algebraischen Datentypen*

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 in der ersten Teilaufgabe mit einer Darstellung eines endlichen Ausschnitts natürlicher Zahlen und arithmetischen Operationen und Relationen darauf, in der zweiten Teilaufgabe werden wir die Wochentagsaufgabe der früheren Aufgabenblätter weiter verallgemeinern.

1. Um einen Ausschnitt der natürlichen Zahlen in Haskell zu modellieren, greifen wir noch einmal die Darstellung der Autokennzeichen aus Teilaufgabe 4 von Aufgabenblatt 3 auf. Das Kennzeichen `AAA 000` verstehen wir als Darstellung der Zahl 0, das Kennzeichen `AAA 001` als Darstellung der Zahl 1, das Kennzeichen `AAA 999` als Darstellung der Zahl 999, das Kennzeichen `AAB 000` als Darstellung der Zahl 1000, usw. Das Kennzeichen `ZZZ 999` ist dann die Darstellung der größten auf diese Weise darstellbaren natürlichen Zahl. Wir bezeichnen den Wert der Zahldarstellung `ZZZ 999` für diese Aufgabe als $maxNat$.

   Zur konkreten Modellierung dieses Ausschnitts der natürlichen Zahlen von 0 bis $maxNat$ verwenden wir den algebraischen Datentyp `Nat`, der sich auf die Aufzählungstypen `Zeichen` und `Ziffer` abstützt:

```
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)
data Ziffer  = Null | Eins | Zwei | Drei | Vier | Fuenf | Sechs | Sieben
             | Acht | Neun deriving (Eq,Ord,Show)
newtype Nat  = Nat ((Zeichen,Zeichen,Zeichen),(Ziffer, Ziffer, Ziffer))
             deriving (Eq,Show)
```

   Die Angabe der `deriving`-Klauseln in den Typdeklarationen von `Zeichen`, `Ziffer` und `Nat`, deren genaue Bedeutung wir in einem späteren Kapitel der Vorlesung besprechen werden, erlaubt uns Werte aller drei Typen auf Gleichheit und Ungleichheit zu überprüfen, sowie auf dem Bildschirm auszugeben. Zusätzlich können wir Werte der Aufzählungstypen `Zeichen` und `Ziffer` bezüglich ihrer (relativen) Größe vergleichen.

   Schreiben Sie Haskell-Rechenvorschriften

   (a) `plusN :: Nat -> Nat -> Nat`

   (b) `minusN :: Nat -> Nat -> Nat`

   (c) `timesN :: Nat -> Nat -> Nat`

(d) `divN :: Nat -> Nat -> Nat`

(e) `modN :: Nat -> Nat -> Nat`

(f) `powerN :: Nat -> Nat -> Nat`

zur Addition, Subtraktion, Multiplikation, ganzzahligen Division, zur Berechnung des Restes bei ganzzahliger Division und zur Potenzierung. Dabei gilt für die Operationen `plusN`, `timesN` und `powerN`, dass das Resultat den Wert $maxNat$ dargestellt als `Nat ((Z,Z,Z),(Neun,Neun,Neun))` hat, wenn das Ergebnis der Operation größer oder gleich dem Wert $maxNat$ ist. In ähnlicher Weise gilt für die Operation `minusNat`, dass das Resultat den Wert 0 dargestellt als `Nat ((A,A,A),(Null,Null,Null))` hat, wenn das zweite Argument größer oder gleich dem ersten Argument ist. Hat das zweite Argument der Operationen `divN` und `modN` den Wert 0 dargestellt als `Nat ((A,A,A),(Null,Null,Null))`, so wird die Auswertung dieser Funktionen mit dem Aufruf von `error "Division durch 0 nicht moeglich"` abgebrochen.

Schreiben Sie weiters Haskell-Rechenvorschriften

(g) `eqN :: Nat -> Nat -> Bool`

(h) `neqN :: Nat -> Nat -> Bool`

(i) `grN :: Nat -> Nat -> Bool`

(j) `leN :: Nat -> Nat -> Bool`

(k) `grEqN :: Nat -> Nat -> Bool`

(l) `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 für einige Funktionen veranschaulichen das gewünschte Ein-/Ausgabeverhalten:

```
plusN (Nat ((A,A,A),(Neun,Neun,Acht))) (Nat ((A,A,A),(Null,Null,Sechs)))
      ->> Nat ((A,A,B),(Null,Null,Vier))
minusN (Nat ((A,A,B),(Null,Null,Vier))) (Nat ((A,A,A),(Null,Null,Sechs)))
      ->> Nat ((A,A,A),(Neun,Neun,Acht))
minusN (Nat ((A,A,A),(Null,Null,Sechs))) (Nat ((A,A,B),(Null,Null,Vier)))
      ->> Nat ((A,A,A),(Null,Null,Null))
timesN  (Nat ((Z,Z,Z),(Neun,Neun,Neun))) (Nat ((A,A,A),(Null,Null,Zwei)))
      ->> Nat ((Z,Z,Z),(Neun,Neun,Neun))
eqN (Nat ((A,A,B),(Null,Null,Vier))) (Nat ((A,A,A),(Null,Null,Sechs))) ->> False
neqN (Nat ((A,A,B),(Null,Null,Vier))) (Nat ((A,A,A),(Null,Null,Sechs))) ->> True
grN (Nat ((A,A,B),(Null,Null,Vier))) (Nat ((A,A,A),(Null,Null,Sechs))) ->> True
leEqN (Nat ((A,A,B),(Null,Null,Vier))) (Nat ((A,A,A),(Null,Null,Sechs))) ->> False
```

2. Wir verallgemeinern die Wochentagsaufgabe der früheren Aufgabenblätter ein weiteres Mal. Dabei unterstellen wir für diese Aufgabe die Gültigkeit des Gregorianischen Kalenders ab dem Jahr 0. Entsprechend des Gregorianischen Kalenders ist jedes ohne Rest durch 4 teilbare Jahr ein Schaltjahr, es sei denn, es ist auch ohne Rest durch 100 teilbar. Ohne Rest durch 100 teilbare Jahre sind nur dann ein Schaltjahr, wenn sie ohne Rest auch durch 400 teilbar sind. Insbesondere ist das Jahr 0 für diese Aufgabe ein Schaltjahr.

   Ist nun bekannt, auf welchen Wochentag irgendein Tag irgendeines Jahres fällt, so steht für alle Tage beginnend mit dem 1. Jänner des Jahres 0 fest, auf welchen Wochentag er fällt. Diesen Wochentag wollen wir für jedes Datum ab dem 1. Jänner des Jahres 0 bis zum 31. Dezember des Jahres $maxNat$ berechnen können. Zur Modellierung dieser Aufgabe in Haskell verwenden wir folgende algebraische Typen:

```
data Wochentag = Montag | Dienstag | Mittwoch | Donnerstag | Freitag
                 | Samstag | Sonntag deriving (Eq,Show)
type Tag       = Nat -- mit Nat aus Teilaufgabe 1
data Monat     = Jaenner | Feber | Maerz | April | Mai | Juni | Juli
                 | August | September | Oktober | November | Dezember
                 deriving (Eq,Show)
type Jahr      = Nat -- mit Nat aus Teilaufgabe 1
type Datum     = (Tag,Monat,Jahr)
```

In Schaltjahren hat der Feber 29 Tage, in den übrigen Jahren 28 Tage. Alle anderen Monate haben 30 bzw. 31 Tage, wie im Gregorianischen Kalender vorgesehen. Zusammen mit dem Gregorianischen Kalender zur Bestimmung von Schaltjahren liegt damit fest, ob ein Tripel (t, m, j) ein gültiges Datum bezeichnet oder nicht.

Schreiben Sie eine Haskell-Rechenvorschrift `wochentag4 :: Datum -> Wochentag -> Datum -> Wochentag`, die für jedes Datum vom 1. Jänner des Jahres 0 bis zum 31. Dezember des Jahres $maxNat$ berechnet, auf welchen Wochentag es fällt: Angewendet auf ein Datum (t, m, j), einen Wochentag wt, wobei (t, m, j) auf wt fällt, und ein Datum (t', m', j'), liefert `wochentag4` denjenigen Wochentag, auf den das durch (t', m', j') bestimmte Datum fällt. Ist eines der Argumente nicht gültig, so wird die Berechnung mit dem Aufruf `error "Falsche Argumente"` abgebrochen.

Haskell Live

Am Freitag, den 13.11.2015, 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.