Generalvereinbarung für alle Aufgabenblätter

...für den Zahlbereich *IN* natürlicher Zahlen:

- IN_0, IN_1 bezeichnen die Menge der natürlichen Zahlen beginnend mit 0 bzw. 1.
- In Haskell (wie in anderen Programmiersprachen) sind natürliche Zahlen nicht als elementarer Datentyp vorgesehen.
- Bevor wir sukzessive bessere sprachliche Mittel zur Modellierung natürlicher Zahlen in Haskell kennenlernen, vereinbaren wir deshalb in Aufgaben für die Zahlräume IN_0 und IN_1 die Namen `Nat0` (für IN_0) und `Nat1` (für IN_1) in Form sog. *Typsynonyme* eines der beiden Haskell-Typen `Int` bzw. `Integer` für ganze Zahlen (zum Unterschied zwischen `Int` und `Integer` siehe z.B. Kap. 2.1.2 der Vorlesung):

```
type Nat0 = Int          type Nat0 = Integer
type Nat1 = Int          type Nat1 = Integer
```

- Die Typen `Nat0` und `Nat1` sind ident (d.h. wertgleich) mit `Int` (bzw. `Integer`), enthalten deshalb wie `Int` (bzw. `Integer`) positive wie negative ganze Zahlen einschließlich der 0 und können sich ohne Bedeutungsunterschied wechselweise vertreten.
- Unsere Benutzung von `Nat0` und `Nat1` als Realisierung natürlicher Zahlen beginnend ab 0 bzw. ab 1 ist deshalb rein konzeptuell und erfordert die Einhaltung einer Programmierdisziplin.
- In Aufgaben verwenden wir die Typen `Nat0` und `Nat1` diszipliniert in dem Sinn, dass ausschließlich positive ganze Zahlen ab 0 bzw. ab 1 als Werte von `Nat0` und `Nat1` gewählt werden.
- Entsprechend dieser Disziplin verstehen wir die Rechenvorschrift

```
fac :: Nat0 -> Nat1
fac n
 | n == 0 = 1
 | n > 0  = n * fac (n-1)
```

  als unmittelbare und "typgetreue" Haskell-Implementierung der Fakultätsfunktion im mathematischen Sinn:

$$!\,: IN_0 \to IN_1$$

$$\forall\, n \in IN_0.\; n\,! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)\,! & \text{falls } n > 0 \end{cases}$$

  Wir verstehen `fac` also als total definierte Rechenvorschrift auf dem Zahlbereich natürlicher Zahlen beginnend ab 0, nicht als partiell definierte Rechenvorschrift auf dem Zahlbereich ganzer Zahlen.

  Dieser Disziplin folgend stellt sich deshalb die Frage einer Anwendung von `fac` auf negative Zahlen (und ein mögliches Verhalten) nicht; ebensowenig wie die Frage einer Anwendung von `fac` auf Wahrheitswerte oder Zeichenreihen und ein mögliches Verhalten.
- Verallgemeinernd werden deshalb auf `Nat0`, `Nat1` definierte Rechenvorschriften im Rahmen von Testfällen nicht mit Werten außerhalb der Zahlräume IN_0, IN_1 aufgerufen. Entsprechend entfallen in den Aufgaben Hinweise und Spezifikationen, wie sich eine solche Rechenvorschrift verhalten solle, wenn sie (im Widerspruch zur Programmierdisziplin) mit negativen bzw. nichtpositiven Werten aufgerufen würde.

1. Aufgabenblatt zu Funktionale Programmierung vom Mi, 17.10.2018.
Fällig: Mi, 24.10.2018 (15:00 Uhr)

Themen: *Hugs kennenlernen, erste Schritte in Haskell, erste weiterführende Aufgaben*

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

Aufgaben

Für dieses Aufgabenblatt sollen Sie die unten angegebenen Aufgabenstellungen in Form eines gewöhnlichen Haskell-Skripts lösen und in einer Datei mit Namen `Aufgabe1.hs` im Homeverzeichnis Ihres Accounts auf der Maschine `g0` ablegen.

Kommentieren Sie Ihre Programme aussagekräftig und machen Sie sich so auch mit den unterschiedlichen Möglichkeiten vertraut, ihre Entwurfsentscheidungen in Haskell-Programmen durch zweckmäßige und problemangemessene Kommentare zu dokumentieren. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten.

Versehen Sie insbesondere alle Funktionen, die Sie zur Lösung der Aufgaben brauchen, auch mit ihren Typdeklarationen, d.h. geben Sie deren syntaktische Signatur oder kurz, Signatur, explizit an.

Laden Sie anschließend Ihre Datei mittels „`:load Aufgabe1`“ (oder kurz „`:l Aufgabe1`“) in das Hugs-System und prüfen Sie, ob die Funktionen sich wie von Ihnen erwartet verhalten. Nach dem ersten erfolgreichen Laden können Sie Änderungen der Datei mithilfe der Hugs-Kommandos `:reload` oder kurz `:r` aktualisieren.

1. Gegeben ist die Funktion $h : IN_0 \to IN_0$ definiert durch:

$$\forall\, n \in IN_0.\ h(n) =_{df} \prod_{i=0}^{n} \sum_{j=0}^{i} i^j$$

   Schreiben Sie eine Haskell-Rechenvorschrift

```
h :: Integer -> Integer
```

   die angewendet auf ein Argument `n` den Wert h(`n`) liefert, wenn `n` ≥ 0 ist, und den Wert `n` sonst.

   Aufrufbeispiele:

   `h(-42) ->>` -42
   `h(-1) ->>` -1
   `h(0) ->>` 0^0 `->>` 1
   `h(1) ->>` $0^0 * (1^0 + 1^1)$ `->>` 1 * (1+1) `->>` 1 * 2 `->>` 2
   `h(2) ->>` $0^0 * (1^0 + 1^1) * (2^0 + 2^1 + 2^2)$ `->>` 1 * (1+1) * (1+2+4) `->>` 1 * 2 * 7 `->>` 14
   `h(3) ->>` $0^0 * (1^0 + 1^1) * (2^0 + 2^1 + 2^2) * (3^0 + 3^1 + 3^2 + 3^3)$
   `->>` 1 * (1+1) * (1+2+4) * (1+3+9+27) `->>` 1 * 2 * 7 * 40 `->>` 560

2. Die *Quersumme* einer natürlichen Zahl ist die Summe ihrer Ziffern. Einige natürliche Zahlen besitzen die Eigenschaft, dass ihre Quersumme mit der Summe der Quersummen ihrer Primfaktoren übereinstimmt. Zahlen mit dieser Eigenschaft nennen wir *quer*. Ein Beispiel für eine Querzahl ist die Zahl 6.036. Es gilt:

$$6.036 = 2 * 2 * 3 * 503$$

$$6 + 0 + 3 + 6 = 15 = 2 + 2 + 3 + 5 + 0 + 3$$

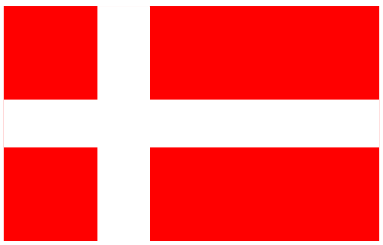
Primzahlen (1 ist keine Primzahl) sind trivialerweise Querzahlen. Wir nennen eine Querzahl eine *echte Querzahl*, wenn sie eine Querzahl, aber nicht prim ist; eine prime Querzahl nennen wir *unechte Querzahl*. 6.036 ist eine echte Querzahl, 11 ist eine unechte Querzahl.

Schreiben Sie eine Haskell-Rechenvorschrift `klassifiziere` zur Klassifikation natürlicher Zahlen in echte Querzahlen, unechte Querzahlen und nicht quere Zahlen:

```
type Nat1 = Integer
klassifiziere :: Nat1 -> String
```

Angewendet auf eine natürliche Zahl n, $n \geq 1$, als Argument, liefert die Funktion `klassifiziere` die Zeichenreihe `"echt quer"`, wenn n eine echte Querzahl ist, `"unecht quer"`, wenn n eine Querzahl, aber keine echte Querzahl ist und `"nicht quer"` sonst.

3. Die dänische Flagge zeigt ein weißes Kreuz auf rotem Grund, wobei die Summen der weißen und roten Flächenanteile gleich groß und der senkrechte und der waagrechte Kreuzbalken gleich breit sind.



Flaggenhersteller müssen daher immer wieder für gewünschte Länge und Breite einer Flagge die Breite der weißen Kreuzbalken bestimmen.

Helfen Sie den Flaggenherstellern, indem Sie zwei Haskell-Rechenvorschriften `kbb` und `kbb'` (für 'Kreuzbalkenbreite') schreiben, die diese Aufgabe für vorgegebene Länge und Breite einer Flagge lösen und sich in ihrer Typsignatur wie folgt unterscheiden:

```
type Nat1 = Int
type Nat0 = Int
type Zentimeter0 = Nat0
type Zentimeter1 = Nat1
type Fl_Laenge = Zentimeter1                  – Flaggenlaenge in Zentimeter
type Fl_Breite = Zentimeter1                  – Flaggenbreite in Zentimeter
type Kb_Breite = Zentimeter0                  – Kreuzbalkenbreite in Zentimeter

kbb  :: Fl_Laenge -> Fl_Breite -> Kb_Breite   – Curryfizierte Deklaration
kbb' :: (Fl_Laenge,Fl_Breite) -> Kb_Breite    – Uncurryfizierte Deklaration
```

Angewendet auf Argumentwerte, die die Länge und Breite einer Flagge in Zentimetern angeben, liefern die Funktionen `kbb` und `kbb'` die gesuchte Breite der Kreuzbalkenstreifen, wenn diese ganzzahlig ist.

Gibt es keine ganzzahlige Lösung für die Streifenbreite, so soll die Summe aus Länge und Breite erhöht um 1 geliefert werden, um die Fehlersituation anzuzeigen.

(Anm.: Es soll uns für diese Aufgabe nicht stören, wenn Flaggen gleich breit wie lang oder breiter als lang gewünscht werden oder Breite und Länge sonstwie in flaggenuntypischen Verhältnissen stehen).

4. Von Zeit zu Zeit kommen nachlässig hergestellte Flaggen auf den Markt, die die hälftige Rot/Weiß-Verteilung und Gleichbreitigkeit der Kreuzbalken nicht einhalten. Um dem Marktamt zu helfen, solche Flaggen schnell zu identifizieren und vom Markt zu nehmen, schreiben Sie zwei Wahrheitswertfunktionen `ist_ok` und `ist_ok'`, die zu gegebenen Argumentwerten feststellen, ob die Rot/Weiß- und Gleichbreitigkeitsbedingungen eingehalten sind.

```
type Ok = Bool
type Kb_senkr = Zentimeter0  – Breite des senk- bzw. waagrechten
type Kb_waagr = Zentimeter0  – Kreuzbalkens jeweils in Zentimeter

ist_ok  :: Fl_Laenge -> Fl_Breite -> Kb_senkr -> Kb_waagr -> Ok  – Curryfiziert
ist_ok' :: (Fl_Laenge,Fl_Breite,Kb_senkr,Kb_waagr) -> Ok         – Uncurryfiziert
```

Machen Sie bei der Implementierung des Funktionenpaars `kbb`, `kbb'` für eine der Funktionen von `curry` bzw. `uncurry` Gebrauch, d.h. implementieren Sie eine der beiden Funktionen `kbb`, `kbb'` von Grund auf und stützen Sie die Implementierung der anderen auf diese und dazu passend `curry` oder `uncurry` ab. Schreiben Sie für das Funktionenpaar `ist_ok`, `ist_ok'` zu gleichem Zweck entsprechende Erweiterungen `curry4` und `uncurry4` von `curry` und `uncurry` und benutzen Sie eine davon, um die Implementierung von entweder `ist_ok` oder `ist_ok'` darauf abzustützen:

```
curry4   :: ((a,b,c,d) -> e) -> (a -> b -> c -> d -> e)
uncurry4 :: (a -> b -> c -> d -> e) -> ((a,b,c,d) -> e)
```

Wichtig: Denken Sie bitte daran, dass Aufgabenlösungen stets auf der Maschine `g0` unter Hugs überprüft werden. Stellen Sie deshalb für Ihre Lösungen zu diesem und auch allen weiteren Aufgabenblättern sicher, dass Ihre Programmierlösungen auf der `g0` unter Hugs die von Ihnen gewünschte Funktionalität aufweisen, und überzeugen Sie sich bei jeder Abgabe davon. Das gilt besonders, wenn Sie für die Entwicklung Ihrer Haskell-Programme mit einem anderen Werkzeug oder einer anderen Maschine arbeiten!

Haskell Live

Am Freitag, den 19.10.2018, oder einem der späteren Termine, werden wir uns in *Haskell Live* u.a. mit der Aufgabe *"Licht oder nicht Licht - Das ist hier die Frage!"* beschäftigen.

Licht oder nicht Licht - Das ist hier die Frage!

Zu den Aufgaben des Nachtwachdienstes an unserer Universität gehört das regelmäßige Ein- und Ausschalten der Korridorbeleuchtungen. In manchen dieser Korridore hat jede der dort befindlichen Lampen einen eigenen Ein- und Ausschalter und jedes Betätigen eines dieser Schalter schaltet die zugehörige Lampe ein bzw. aus, je nachdem, ob die entsprechende Lampe vorher aus- bzw. eingeschalten war. Einer der Nachtwächter hat es sich in diesen Korridoren zur Angewohnheit gemacht, die Lampen auf eine ganz spezielle Art und Weise ein- und auszuschalten: Einen Korridor mit n Lampen durchquert er dabei n-mal vollständig hin und her. Auf dem Hinweg des i-ten Durchgangs betätigt er jeden Schalter, dessen Position ohne Rest durch i teilbar ist. Auf dem Rückweg zum Ausgangspunkt des i-ten und jeden anderen Durchgangs betätigt er hingegen keinen Schalter. Ein *Durchgang* ist also der Hinweg unter entsprechender Betätigung der Lichtschalter und der Rückweg zum Ausgangspunkt ohne Betätigung irgendwelcher Lichtschalter.

Die Frage ist nun folgende: Wenn beim Eintreffen des Nachtwächters in einem solchen Korridor alle n Lampen aus sind, ist nach der vollständigen Absolvierung aller n Durchgänge die n-te und damit letzte Lampe im Korridor an oder aus?

Schreiben Sie ein Programm in Haskell oder in einer anderen Programmiersprache ihrer Wahl, das diese Frage für eine als Argument vorgegebene positive Zahl von Lampen im Korridor beantwortet.

Für n gleich 3 oder n gleich 8191 sollte Ihr Programm die Antwort "aus" liefern, für n gleich 6241 die Antwort "an".