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 izur 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. nichtposiviten Werten aufgerufen würde.

1. Aufgabenblatt zu Funktionale Programmierung vom Mi, 18.10.2017.
Fällig: Mi, 25.10.2017 (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. Ein Lotterieunternehmen bietet Glücksspiele nach dem Muster des `EuroMillionen`-Spiels an. Bei Spielen dieser Art werden zunächst k Kugeln aus einem Topf mit m Kugeln gezogen, anschließend k' Kugeln aus einem zweiten Topf mit m' Kugeln, jeweils ohne Zurücklegen der gezogenen Kugeln.

   Wieviele Wettkombinationen gibt es bei Gewinnspielen dieses Typs? Schreiben Sie eine Haskell-Rechenvorschrift `anzahlWettkombis`, die diese Frage beantwortet. Dabei sollen folgende Typvereinbarungen verwendet werden:

   ```
   type Nat0              = Integer
   type Nat1              = Integer
   type GesamtKugelZahl   = Nat1
   type GezogeneKugelZahl = Nat1
   type Spiel             = (GesamtKugelZahl,GezogeneKugelZahl)
   type Glueckesspiel     = (Spiel,Spiel)

   anzahlWettKombis :: Glueckesspiel -> Nat0
   ```

2. Die Fibonacci-Funktion Fib ist folgendermaßen definiert:

$$Fib : IN_0 \to IN_0$$

$$Fib(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{sonst} \end{cases}$$

   Die Menge $\{Fib(n) \mid n \in IN_0\}$ ist die Menge der *Bilder* oder kürzer die *Bildmenge* der Funktion Fib.

Sei wieder

```
type Nat0 = Integer
```

Schreiben Sie eine Haskell-Rechenvorschrift `fib'` mit der Typsignatur

```
fib' :: Nat0 -> Nat0
```

Angewendet auf einen Argumentwert `m` aus `Nat0` überprüft `fib'`, ob `m` aus der Bildmenge von *Fib* ist. Falls ja, liefert die Funktion `fib'` diejenige eindeutig bestimmte Zahl `k` mit $Fib(\texttt{k}) = \texttt{m}$ als Resultat. Falls nein, liefert die Funktion `fib'` den Argumentwert `m` als Resultat.

Anwendungsbeispiele:

```
fib' 0 ->> 0
fib' 2 ->> 3
fib' 4 ->> 4
fib' 8 ->> 6
fib' 9 ->> 9
```

3. Wir betrachten noch einmal die Fibonacci-Funktion *Fib* und den Typ `Nat0` aus dem vorigen Aufgabenteil. Schreiben Sie eine Haskell-Rechenvorschrift `fibs` mit der Typsignatur

   ```
   fibs:: Nat0 -> [Nat0].
   ```

   die angewendet auf einen Argumentwert `n` aus `Nat0` die aufsteigend geordnete Liste der Werte der Fibonacci-Funktion für die Werte von `0,1,2,...,n` liefert.

   Anwendungsbeispiele:

   ```
   fibs 0 ->> [0]
   fibs 1 ->> [0,1]
   fibs 5 ->> [0,1,1,2,3,5]
   fibs 8 ->> [0,1,1,2,3,5,8,13,21]
   ```

4. Schreiben Sie eine Haskell-Rechenvorschrift `verflechten` mit der Typsignatur `verflechten :: [Int] -> [Int] -> [Int]`, die die Elemente der beiden Argumentlisten so miteinander verflechtet, dass in der Resultatliste auf ein Element der ersten Argumentliste ein Element der zweiten Argumentliste folgt und umgekehrt. Dabei bleibt die relative Reihenfolge der Elemente der beiden Argumentlisten in der Ergebnisliste erhalten. Sind die beiden Argumentlisten verschieden lang, so werden die Listen solange miteinander verflochten wie möglich; anschließend bildet die verbliebene Restliste der längeren Argumentliste den Rest der Ergebnisliste. Das erste Element der Resulttatliste kommt nur dann nicht aus der ersten Argumentliste, wenn diese leer ist.

   Anwendungsbeispiele:

   ```
   verflechten [1,2,3] [4,5,6] ->> [1,4,2,5,3,6]
   verflechten [1,2,3] [4]     ->> [1,4,2,3]
   verflechten [1,2] [3,4,5,6] ->> [1,3,2,4,5,6]
   verflechten [] [1,2,3]      ->> [1,2,3]
   ```

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 13.10.2017, 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 irgendeiner 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".