

8. Aufgabenblatt zu Funktionale Programmierung vom 04.12.2013.

Fällig: 11.12.2013 / 18.12.2013 (jeweils 15:00 Uhr)

Themen: *Funktionen auf Zahlen und anderen Datenstrukturen*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe8.hs` ablegen. Sie sollen für die Lösung dieses Aufgabenblatts also 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. In einem Brief vom 7. Juni 1742 an Leonhard Euler äußerte Christian Goldbach die Vermutung, dass sich "jede gerade positive Zahl größer als 2 als Summe zweier Primzahlen darstellen lasse" und legte ihm nahe, sich einen Beweis für diese Behauptung zu überlegen. Diese Vermutung ist seitdem als Goldbachsche Vermutung bekannt. Der Beweis oder die Widerlegung der Goldbachschen Vermutung hat sich als außerordentlich schwierig herausgestellt und auch in jüngster Vergangenheit vorgelegte Beweisversuche scheinen noch weiterer Prüfung zu bedürfen.

Aus diesem Grund wollen wir Haskell-Rechenvorschriften schreiben, mit denen sich die Goldbachsche Vermutung für einzelne Argumente bestätigen oder möglicherweise widerlegen lässt. Schreiben Sie dazu folgende Rechenvorschriften über dem Datentyp

```
data Goldbach = RefutedFor Integer
               | MightHoldBecause [Integer]
               | AllWitnesses [[Integer]]
               | Irrelevant
               deriving (Eq,Ord,Show)
```

- (a) `gb :: Integer -> Goldbach`: Angewendet auf eine ganze Zahl z liefert `gb` den Wert `Irrelevant`, falls z kleiner oder gleich 2 oder ungerade ist. `gb` liefert den Wert `RefutedFor z`, falls z eine positive gerade Zahl größer als 2 ist, die nicht als Summe zweier Primzahlen darstellbar ist und ansonsten einen Zeugen für diese Summendarstellung, d.h. zwei Primzahlen, deren Summe z ist. Gibt es mehrere Paare von Primzahlen mit Summe z (z.B. $30 = 7 + 23 = 11 + 19 = 13 + 17$) ist es egal, welches Paar als Zeuge geliefert wird und in welcher Reihenfolge dieses Paar geliefert wird. Angewendet auf 30 sind also (u.a.) `MightHoldBecause [11, 19]`, `MightHoldBecause [19, 11]`, `MightHoldBecause [23, 7]` zulässige Ergebnisse.
 - (b) `gbAll :: Integer -> Goldbach`: Analog zu `gb` mit dem Unterschied, dass `gbAll` für positive gerade Zahlen größer als 2 alle Paare von Primzahlen (ohne Vertauschungen) als Zeugen liefert, falls es solche gibt. Die Zeugen sollen dabei in der Resultatliste aufsteigend geordnet sein, d.h. innerhalb jedes Zeugen steht die kleinere Primzahl zuerst; in der Liste steht ein Zeuge weiter links als ein anderer, wenn er mit der kleineren Primzahl beginnt. Angewendet auf 30 liefert `gbAll` also das Resultat `AllWitnesses [[7, 23], [11, 19], [13, 17]]`.
2. Weihnachten steht vor der Tür. Brave Kinder bekommen die meisten Geschenke heißt es. Hierfür wollen wir auf empirischer Grundlage Hinweise finden, die diese These stützen oder fraglich erscheinen lassen. Dazu wollen wir aus einer Beobachtungsreihe vom vergangenen Weihnachtsfest, die uns in Form einer Liste von Beobachtungspaaren vorliegt, entsprechende Hinweise gewinnen. Jeder Eintrag der Beobachtungsreihe gibt dabei den als Bravität bezeichneten Bravheitsgrads eines Kindes und die Zahl seiner Geschenke an. Aus der Beobachtungsreihe wollen wir nun mithilfe der Rechenvorschriften `pro` und `con` zwei Teillisten konstruieren, die die Ausgangsthese stützen oder schwächen.
 - Die Funktion `pro` konstruiert dazu aus der der Beobachtungsreihe eine längst mögliche Teilliste, so dass von links nach rechts die Bravität und die Zahl der Geschenke stets (echt) zunehmen.
 - Die Funktion `con` konstruiert dazu aus der der Beobachtungsreihe eine längst mögliche Teilliste, so dass von links nach rechts die Bravität stets echt abnimmt, die Zahl der Geschenke aber stets (echt) zunimmt.

Bravität und Geschenkezahl sind dabei als dimensionslose echt positive ganze Zahlen gegeben. Paare in einer Beobachtungsreihe dürfen in Bravität oder/und Geschenkezahl übereinstimmen. Sind Bravität oder/und Geschenkezahl in einem Beobachtungspaar negativ, so wird ein solches Beobachtungspaar als Mess- bzw. Beobachtungsfehler von der Betrachtung ausgeschieden.

Zur Modellierung führen wir folgende Typen ein:

```
type Bravitaet      = Integer
type Geschenkezahl = Integer
data Beobachtungspaar = Bp Bravitaet Geschenkezahl deriving (Eq,Ord,Show)
type Beobachtungsreihe = [Beobachtungspaar]
```

Schreiben Sie nun zwei Haskell-Rechenvorschriften `pro :: Beobachtungsreihe -> Beobachtungsreihe` und `con :: Beobachtungsreihe -> Beobachtungsreihe`, die das Entsprechende leisten:

- Angewendet auf eine Beobachtungsreihe r liefert `pro` eine Beobachtungsreihe s mit Beobachtungspaaren aus r mit der Eigenschaft, dass von links nach rechts Bravität und Geschenkezahl der Beobachtungspaare in s beide jeweils echt zunehmen und dass s mindestens so lang ist wie jede andere Beobachtungsreihe t mit diesen Eigenschaften.
- Angewendet auf eine Beobachtungsreihe r liefert `con` eine Beobachtungsreihe s mit Beobachtungspaaren aus r mit der Eigenschaft, dass von links nach rechts die Bravität der Beobachtungspaare in s echt abnimmt, die Geschenkezahl aber echt zunimmt und dass s mindestens so lang ist wie jede andere Beobachtungsreihe t mit diesen Eigenschaften.

Welche der Beobachtungsreihen `pro` und `con` zurückliefern, wenn es mehrere gibt, spielt keine Rolle.

Hinweis:

- Verwenden Sie *keine* Module. Wenn Sie Funktionen wiederverwenden möchten, kopieren Sie diese in die Abgabedatei `Aufgabe8.hs`. Andere Dateien als diese werden vom Abgabeskript ignoriert.

Haskell Live

An einem der kommenden *Haskell Live*-Termine, der nächste ist am Freitag, den 06.12.2013, werden wir uns, wenn es die Zeit erlaubt, u.a. mit der Aufgabe *Tortenvurf* beschäftigen.

Tortenvurf

Wir betrachten eine Reihe von $n + 2$ nebeneinanderstehenden Leuten, die von paarweise verschiedener Größe sind. Eine größere Person kann stets über eine kleinere Person hinwegblicken. Demnach kann eine Person in der Reihe so weit nach links bzw. nach rechts in der Reihe sehen bis dort jemand größeres steht und den weitergehenden Blick verdeckt.

In dieser Reihe ist etwas Ungeheuerliches geschehen. Die ganz links stehende 1-te Person hat die ganz rechts stehende $n + 2$ -te Person mit einer Torte beworfen. Genau p der n Leute in der Mitte der Reihe hatten während des Wurfs freien Blick auf den Tortenwerfer ganz links; genau r der n Leute in der Mitte der Reihe hatten freien Blick auf das Opfer des Tortenwerfers ganz rechts.

Wieviele Permutationen der n in der Mitte der Reihe stehenden Leute gibt es, so dass gerade p von ihnen freie Sicht auf den Werfer und r von ihnen auf das Tortenvurfopfer hatten?

Schreiben Sie in Haskell oder einer anderen Programmiersprache ihrer Wahl eine Funktion, die zu einer vorgegebenen Zahl $n \leq 10$ von Leuten in der Mitte der Reihe, davon p mit $1 \leq p \leq n$ mit freier Sicht auf den Werfer und r mit $1 \leq r \leq n$ mit freier Sicht auf das Opfer, diese Anzahl von Permutationen berechnet.