7. Aufgabenblatt zu Funktionale Programmierung vom Mi, 29.11.2017.
Fällig: Mi, 06.12.2017 (15:00 Uhr)

Themen: *Funktionen über Ausdrücken und Graphen als algebraischen Datentypen*

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

Aufgabe

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe7.hs` ablegen. Sie sollen für die Lösung dieses Aufgabenblatts also wieder 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.

1. Jeder prädikatenlogische Ausdruck (oder: prädikatenlogische Formel) ist äquivalent zu einem prädikatenlogischen Ausdruck in Negationsnormalform. Dabei heißt ein prädikatenlogischer Ausdruck in *Negationsnormalform*, wenn Negationen höchstens unmittelbar vor Variablen stehen.

   Mithilfe folgender Äquivalenzen ist es möglich, jeden prädikatenlogischen Ausdruck in Negationsnormalform zu überführen, indem Negationen ‘von außen nach innen in Ausdrücke hineingeschoben werden’:

$$
\begin{aligned}
\neg\forall x.\ A &\Leftrightarrow \exists x.\ \neg A \\
\neg\exists x.\ A &\Leftrightarrow \forall x.\ \neg A \\
\neg(A_1 \wedge A_2) &\Leftrightarrow \neg A_1 \vee \neg A_2 \\
\neg(A_1 \vee A_2) &\Leftrightarrow \neg A_1 \wedge \neg A_2 \\
\neg(A_1 \Rightarrow A_2) &\Leftrightarrow \neg(\neg A_1 \vee A_2) \\
\neg\neg A &\Leftrightarrow A \\
\neg Wahr &\Leftrightarrow Falsch \\
\neg Falsch &\Leftrightarrow Wahr
\end{aligned}
$$

   Schreiben Sie eine Haskell-Rechenvorschrift `nnf` (für Negationsnormalform), die einen prädikatenlogischen Ausdruck unter Ausnutzung obiger Äquivalenzen in Negationsnormalform überführt:

```
type Wahrheitswert = Bool
data Name          = N1 | N2 | N3 | N4 | N5 deriving (Eq,Show)
newtype Variable   = Var Name deriving (Eq,Show)

data Ausdruck = K Wahrheitswert                -- Logische Konstante
              | V Variable                     -- Logische Variable
              | Nicht Ausdruck                 -- Logische Negation
              | Und Ausdruck Ausdruck          -- Logische Konjunktion
              | Oder Ausdruck Ausdruck         -- Logische Disjunktion
              | Impl Ausdruck Ausdruck         -- Logische Implikation
              | Esgibt Variable Ausdruck       -- Existentiell quantifizierter Ausdruck
              | Fueralle Variable Ausdruck     -- Allquantifizierter Ausdruck
              deriving (Eq,Show)

nnf :: Ausdruck -> Ausdruck
```

   Nehmen Sie bei Bedarf oder Notwendigkeit weitere Typklasseninstanzbildungen vor.

2. Vladimir ist Vampir. Vladimir reist mit der Bahn. Immer mit seinem Sarg. Nur nachts. Nie vor 18:00 Uhr abends, nie nach 06:00 Uhr morgens. Die Tagesstunden verträumt Vladimir in seinem Sarg, in einer ruhigen Bahnhofsecke, in den ihn die aktuelle Reiseetappe gebracht hat, gern im Keller oder auf dem Boden. Mittags um 12:00 Uhr genießt Vladimir seine tägliche Blutkonserve, 1 Liter, auch im Sarg. Die Blutgruppe erkennt er am Geschmack.

   Hat Vladimir einen auswärtigen Termin oder möchte Verwandte besuchen, plant er seine Reise stets so, dass er möglichst wenige Blutkonserven mitzunehmen hat; der Stauraum im Sarg ist beschränkt.

   Helfen Sie Vladimir bei der Reise- und Proviantplanung und schreiben Sie einen Reise- und Proviantplaner für ihn. Benutzen Sie dafür folgende Typen:

```
type Nat0 = Int
type Nat1 = Int
type Anzahl_Blutkonserven = Nat0
type Reiseproviant = Anzahl_Blutkonserven
type Reisedauer = Nat1   -- In vollen Std., ausschliesslich Werte von 1..12
type Abfahrtszeit = Nat0 -- In vollen Std., ausschliesslich Werte von 0..23
type Ankunftszeit = Nat0 -- In vollen Std., ausschliesslich Werte von 0..23
data Stadt = S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S9 | S10 deriving Show
type Ausgangsort = Stadt
type Zielort = Stadt
type Abfahrtsort = Stadt
type Ankunftsort = Stadt
type Relation = (Abfahrtsort,Abfahrtszeit,Ankunftsort,Ankunftszeit)
type Reiseplan = [Relation]
type Fahrplan = Abfahrtsort -> [(Ankunftsort,Abfahrtszeit,Reisedauer)] -- Total def.

reise_planer    :: Fahrplan -> Ausgangsort -> Zielort -> Maybe Reiseplan
proviant_planer :: Fahrplan -> Ausgangsort -> Zielort -> Maybe Reiseproviant
```

   Gibt es keine Reiseroute vom Ausgangsort zum Zielort, die Vladimirs besonderen Bedürfnissen genügt, liefern die Funktionen `reise_planer` und `proviant_planer` den Wert `Nothing`; anderenfalls liefert die Funktion `proviant_planer` die Zahl an Blutkonserven, die Vladimir mindestens auf seine Reise mitnehmen muss, um stets um 12:00 Uhr mittags seine Erfrischung nehmen zu können, die Funktion `reise_planer` einen zugehörigen Reiseplan, auf der Vladimir tatsächlich mit dieser Mindestzahl an Blutkonserven das Auslangen findet. Beide Resultate werden als `Just`-Werte zurückgeliefert. Gibt es mehrere mögliche Reiserouten, auf denen Vladimir mit der Mindestzahl von Blutkonserven auskommt, ist es egal, welche diese Routen Ihr Reiseplaner Vladimir vorschlägt.

   Nehmen Sie weitere Typklasseninstanzbildungen vor, falls nötig oder hilfreich, um die Berechnung oder die Ausgabe von Ergebnissen zu ermöglichen oder erleichtern. Sie dürfen bei der Implementierung davon ausgehen, dass die Funktionen `reise_planer` und `proviant_planer` nur mit Fahrplanabbildungen aufgerufen werden, die total definiert sind.

Haskell Live

An einem der kommenden *Haskell Live*-Termine, der nächste ist am Freitag, den 01.12.2017, werden wir uns u.a. mit der Aufgabe *Tortenwurt* beschäftigen.

Tortenwurf

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 Tortenwurfopfer 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.

Haskell Private

Anmeldungen zu *Haskell Private* sind noch möglich. Nutzen Sie die Möglichkeit. Nähere Hinweise und die URL zur Anmeldungsseite finden Sie auf der Homepage der Lehrveranstaltung.