Eine Sache lernt man, indem man sie macht.
Cesare Pavese (1908-1950)
italien. Schriftsteller

Für das Können gibt es nur einen Beweis, das Tun.
Marie von Ebner-Eschenbach (1830-1916)
österr. Schriftstellerin

3. Angabe zu Funktionale Programmierung von Fr, 28.10.2022.

Erstabgabe: Fr, 04.11.2022, 12:00 Uhr

Zweitabgabe: Siehe „Hinweise zu Org. u. Ablauf der Übung" (TUWEL-Kurs)

(Teil A: beurteilt; Teil B & C: ohne Abgabe, ohne Beurteilung)

Themen: *Literate Haskell-Skripte, Typklassen, Instanzbildung, Rechnen mit Listen*

Stoffumfang: *Kapitel 1 bis Kapitel 9, besonders Kapitel 2 bis 6.*

- **Teil A, programmiertechnische Aufgaben:** Besprechung am ersten Übungsgruppentermin, der auf die *Zweitabgabe* der programmiertechnischen Aufgaben folgt.
- **Teil B, Papier- und Bleistiftaufgaben:** Besprechung am ersten Übungsgruppentermin, der auf die *Erstabgabe* der programmiertechnischen Aufgaben folgt.
- **Teil C, das Sprachduell:** eine Aufgabe besser für Haskell oder für eine andere Sprache?

Wichtig

1. Befolgen Sie die Anweisungen aus den 'Lies-mich'-Dateien (s. TUWEL-Kurs) zu den Angaben sorgfältig, um ein reibungsloses Zusammenspiel mit dem Testsystem sicherzustellen. Bei Fragen dazu, stellen Sie diese bitte im TUWEL-Forum zur LVA.

2. Erweitern Sie für die für diese Angabe zu schreibenden Rechenvorschriften die zur Verfügung gestellte Rahmendatei

   `Angabe3.lhs`

   und legen Sie sie für die Abgabe auf oberstem Niveau in Ihrem *home*-Verzeichnis ab. Achten Sie darauf, dass "Gruppe" Leserechte für diese Datei hat. Wenn nicht, setzen Sie diese Leserechte mittels `chmod g+r Angabe3.lhs`.

   Löschen Sie keinesfalls eine Deklaration aus der Rahmendatei! Auch dann nicht, wenn Sie einige dieser Deklarationen nicht oder nicht vollständig implementieren wollen. Löschen Sie auch nicht die für das Testsystem erforderliche Modul-Anweisung `module Angabe3 where` am Anfang der Rahmendatei.

3. Der Name der Abgabedatei ist für Erst- und Zweitabgabe ident!

4. Benutzen Sie keine selbstdefinierten Module! Wenn Sie (für spätere Angaben) einzelne Rechenvorschriften früherer Lösungen wiederverwenden möchten, kopieren Sie diese bitte in die neue Abgabedatei ein. Eine `import`-Anweisung für selbstdefinierte Module schlägt für die Auswertung durch das Abgabesystem fehl, weil Ihre Modul-Datei, aus der importiert werden soll, vom Testsystem nicht mit abgesammelt wird.

5. Ihre Programmierlösungen werden stets auf der Maschine `g0` mit der dort installierten Version von GHCi überprüft. Stellen Sie deshalb sicher, dass sich Ihre Programme (auch) auf der `g0` unter GHCi so verhalten, wie von Ihnen gewünscht.

6. Überzeugen Sie sich bei jeder Abgabe davon! Das gilt besonders, wenn Sie für die Entwicklung Ihrer Haskell-Programme mit einer anderen Maschine, einer anderen GHCi-Version oder/und einem anderen Werkzeug wie etwa Hugs arbeiten!

A Programmiertechnische Aufgaben (beurteilt, max. 50 Punkte)

Erweitern Sie zur Lösung der programmiertechnischen Aufgaben die Rahmendatei `Angabe3.hs`. Kommentieren Sie die Rechenvorschriften in Ihrem Programm zweckmäßig, aussagekräftig und problemangemessen. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Wertvereinbarungen für konstante Werte (z.B. `> pi = 3.14 :: Float`). Versehen Sie alle Funktionen, die Sie zur Lösung der Aufgaben benötigen, mit ihren Typdeklarationen, d.h. geben Sie stets deren syntaktische Signatur (kurz: Signatur), explizit an.

Wir betrachten Matrizen über ganzen Zahlen:

1. *Matrix:* Eine ganzzahlige Matrix M vom Typ (m, n) ist ein nichtleeres zweidimensionales Schema ganzer Zahlen $a_{ij} \in \mathbb{Z}$ mit m Zeilen und n Spalten, $m, n \in IN_1$:

$$M = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

   Matrizen vom Typ $(1, 1)$ identifizieren wir mit ihrem einzigen Element, ein sog. *Skalar*.

2. *Matrixgleichheit:* Zwei Matrizen M, N sind *gleich* gdw. sie sind typgleich und ihre Elemente stimmen positionsweise überein, *ungleich* sonst.

3. *Matrixaddition:* Die Summe zweier typgleicher Matrizen M, N vom Typ (m, n) ist die typgleiche Matrix S:

$$M + N =_{df} \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix} = S$$

   definiert durch:

$$S =_{df} \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{pmatrix}$$

   mit

$$c_{ij} =_{df} a_{ij} + b_{ij},\ i = 1, \ldots, m,\ j = 1, \ldots, n$$

   Sind M, N typverschieden, sind Addition und Summe von M und N nicht definiert.

4. *Matrixsubtraktion:* Die Differenz zweier typgleicher Matrizen M, N vom Typ (m, n) ist die typgleiche Matrix D:

$$M - N =_{df} \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} - \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix} = D$$

definiert durch:

$$D =_{df} \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{pmatrix}$$

mit

$$c_{ij} =_{df} a_{ij} - b_{ij},\ i = 1, \ldots, m,\ j = 1, \ldots, n$$

Sind M, N typverschieden, sind Subtraktion und Differenz von M und N nicht definiert.

Wir modellieren Matrizen in der Folge zeilenweise als (nichtleere) Listen von (nichtleeren) (Matrix-) Zeilen über ganzen Zahlen:

Als erstes führen wir den Typalias Nat1 ein; seine Werte verwenden wir für die Darstellung des Typs von Matrizen:

```
> type Nat1 = Int
> type Typ  = (Nat1,Nat1)
```

Matrizen modellieren wir als Listen von (Matrix-) Zeilen über entsprechenden selbstdefierten Listentypen:

Zur Namenswahl: LE für ‘Letztes Element’, E für ‘Element’

```
> data Zeile = LE Int
>            | E Int Zeile Liste deriving Show
```

Zur Namenswahl: LZ für ‘Letzte Zeile, Z für ‘Zeile’

```
> data Matrix = LZ Zeile
>             | Z Zeile Matrix deriving Show
```

Um mit Argumenten umzugehen, die keine Matrix darstellen oder im Typ nicht zueinander passen, führen wir den Wert fehler als fehleranzeigenden Wert ein (aufgrund unserer Festlegung von fehler bedeutet das, dass die Rückgabe dieses Werts gleichbedeutend mit dem Aufruf der Funktion error mit dem Argument "Argument(e) typfehlerhaft" ist und die Programmausführung mit Ausgabe der Zeichenreihe "Argument(e) typfehlerhaft" endet).

```
> fehler = error "Argument(e) typfehlerhaft"
```

A.1 Implementieren Sie die Haskell-Rechenvorschrift `matrixtyp`, die überprüft, ob das Argument eine Matrix ist und in diesem Fall ihren Typ bestimmt:

```
> data Matrixtyp = Matrix_vom_Typ Typ
                 | KeineMatrix deriving (Eq,Show)

> matrixtyp :: Matrix -> Matrixtyp
```

Angewendet auf einen `Matrix`-Wert, liefert `matrixtyp` den Wert `Matrix_vom_Typ (m,n)`, wenn das Argument eine Matrix vom Typ `(m,n)` repräsentiert; den Wert `KeineMatrix` sonst.

A.2 Machen Sie den Typ `Matrix` zu einer Instanz der Typklasse `Eq`. Dabei soll gelten: sind beide Argumente des Gleichheits- `(==)` oder Ungleichheitsvergleichs `(==)` Darstellungen typgleicher Matrizen, so liefern die Vergleiche den entsprechenden Wahrheitswert `True` bzw. `False`, sonst den Wert `fehler`.

Ersetzen und vervollständigen Sie den Rumpf der `instance`-Deklaration für `Eq` entsprechend. Überlegen Sie sich, ob aufgrund der Protoimplementierungen in der Klasse `Eq` entweder die Implementierung des Gleichheitsvergleichs oder des Ungleichvergleichs ausreichend ist, oder ob beide implementiert werden müssen.

```
> instance Eq Matrix where
>  m1 == m2 = ...
>  m1 /= m2 = ...
```

A.3 Machen Sie den Typ `Matrix` zu einer (unvollständig implementierten!) Instanz der Typklasse `Num`.

Unvollständig heißt, dass die Funktionen `(*)`, `negate`, `signum` der Klasse `Num` **nicht** implementiert werden sollen, sondern nur die Funktionen `(+)`, `(-)`, `abs`, `fromInteger`!

Dabei soll gelten:

(a) `(+)`, `(-)` liefern das Ergebnis Matrixaddition, -subtraktion, wenn sie auf Matrizen vom gleichen Typ angewendet werden, sonst `fehler`.
Tipp: Ihre Implementierung ist besonders gelungen, wenn Sie `(+)`, `(-)` auf eine Hilfsfunktion abstützen, die neben den Matrixargumenten auch die Operation als Argument übernimmt.

(b) Angewendet auf eine Matrix M, liefert `abs` die Matrix M', die aus M entsteht, indem jedes Element von M durch seinen Absolutbetrag ersetzt wird, den Wert `fehler` sonst.

(c) Angewendet auf eine ganze Zahl z, liefert `fromInteger` die (eindeutig bestimmte) einelementige Matrix vom Typ $(1, 1)$ mit z als (einzigem) Element.

Ersetzen/vervollständigen Sie dazu den Rumpf der `instance`-Deklaration für `Num`.

```
> instance Num Matrix where
>  m1 + m2 = ...
>  m1 - m2 = ...
>  abs m = ...
>  fromInteger z = ...
>  m1 * m2 = error "(*) bleibt unimplementiert!"
>  negate m = error "negate bleibt unimplementiert!"
>  signum m = error "signum bleibt unimplementiert!"
```

A.4 **Ohne Beurteilung:** Beschreiben Sie für jede Rechenvorschrift in einem Kommentar knapp, aber gut nachvollziehbar, wie die Rechenvorschrift vorgeht.

A.5 **Ohne Abgabe, ohne Beurteilung:** Testen Sie alle Funktionen umfassend mit aussagekräftigen eigenen Testdaten.

B Papier- und Bleistiftaufgaben (ohne Abgabe/Beurteilung)

B1. Hätten wir den Typ `Matrix` in den Aufgaben aus Teil A statt als algebraischern Typ:

```
> data Matrix = LZ Zeile
>             | Z Zeile Matrix deriving Show
```

auch als

B.1.1 neuen Typ:

```
newtype Matrix = LZ Zeile
               | Z Zeile Matrix deriving Show
```

B.1.2 Typalias:

```
> type Matrix = LZ Zeile
>             | Z Zeile Matrix deriving Show
```

einführen können? Begründen Sie Ihre Antwort.

B.2 Hätten wir das Ziel der Instanzbildungen für `Eq` in A.2 und `Num` in A.3 auch durch `deriving`-Klauseln für `Eq` bzw. `Num` erreichen können? Begründen Sie Ihre Antwort.

B.3 Konnten Sie bei der Instanzbildung für `Eq` in A.2 die Protoimplementierungen in `Eq` vorteilhaft ausnützen? Sind Sie mit einer Minimalvervollständigung ausgekommen? Wenn nein, warum nicht? (s. Kapitel 4.3 zu Minimalvervollständigungen)

B.4 Warum hätten wir die linken Seiten in den Instanzdeklarationsskeletten in A.2, A.3 auch wie unten angeben können? Warum müssen dabei die Operatoren für Addition, Subtraktion, Multiplikation und die Relatoren für Gleichheit, Ungleichheit anders als in A.2, A.3 in Klammern eingeschlossen werden? Begründen Sie Ihre Antwort.

```
> instance Eq Matrix where
>  (==) m1 m2 = ...
>  (/=) m1 m2 = ...

> instance Num Matrix where
>  (+) m1 m2 = ...
>  (-) m1 m2 = ...
>  abs m = ...
>  fromInteger z = ...
>  (*) m1 m2 = error "(*) bleibt unimplemiert!"
>  negate m = error "negate bleibt unimplemiert!"
>  signum m = error "signum bleibt unimplemiert!"
```

B.5 Wiederholen Sie die Aufgaben aus Teil A&B für folgenden Matrixtyp:

```
> type Zeile   = [Int]
> type Matrix' = [Zeile]
```

Iucundi acti labores.
Getane Arbeiten sind angenehm.

Cicero (106 - 43 v.Chr.)
röm. Staatsmann und Schriftsteller

C Das Sprachduell: eine Aufgabe besser für Haskell oder für eine andere Sprache? (ohne Abgabe/Beurteilung)

City-Maut

Viele Städte, auch Wien, überlegen die Einführung einer City-Maut, um die Verkehrsströme besser kontrollieren und steuern zu können. Für die Einführung einer City-Maut sind verschiedene Modelle denkbar; in unserem Modell liegt der Fokus auf innerstädtischen Nadelöhren.

Unter einem *Nadelöhr* verstehen wir eine Verkehrsstelle, die auf dem Weg von einem Bezirk A zu einem Bezirk B in der Stadt passiert werden muss, für das es also keine Möglichkeit zur Umfahrung gibt.

Um den Verkehr an Nadelöhren zu beeinflussen, sollen genau dort Mautstationen eingerichtet und Verkehrsverursachungsgebühren eingehoben werden.

In der Stadt S mit den sechs Bezirken A, B, C, D, E, F und den sieben in beiden Richtungen befahrbaren Routen B–C, A–B, C–A, D–C, D–E, E–F und F–C führt jede Fahrt von Bezirk A in Bezirk E durch Bezirk C. C ist also ein Nadelöhr und muss demnach mit einer Mautstation versehen werden.

Ihre Aufgabe ist nun, ein Programm zu schreiben, das für eine durch seine Bezirke und Routen beschriebene Stadt die Anzahl und Namen aller Nadelöhre bestimmt.

Der Einfachheit halber gehen wir davon aus, dass anstelle von Bezirksnamen von 1 beginnende fortlaufende Bezirksnummern verwendet werden. Der Stadt- und Routenplan iwird dabei in Form eines Tupels zur Verfügung gestellt, das die Anzahl der Bezirke angibt und die möglichen jeweils in beiden Richtungen befahrbaren direkten Routen von Bezirk zu Bezirk innerhalb der Stadt. In Haskell könnte dies durch einen Wert des Datentyps `CityMap` realisiert werden:

```
type Bezirk     = Int
type AnzBezirke = Int
type Route      = (Bezirk,Bezirk)
newtype CityMap = CM (AnzBezirke,[Route])
```

Gültige Stadt- und Routenpläne müssen offenbar bestimmten Wohlgeformtheitsanforderungen genügen. Überlegen Sie sich, welche das sind und wie sie sie überprüft können, so dass Ihr Nadelöhrsuchprogramm nur auf wohlgeformte Stadt- und Routenpläne angewendet wird.

C.1 Treffen Sie Ihre Wahl für das Sprachduell! (Wenn es nicht Haskell ist, kommen Sie am Ende des Semesters noch einmal auf das Duell zurück!)

C.2 Schreiben Sie in der Sprache, die Sie gewählt haben, ein Programm, das die Entschlüsselung vornimmt.

(*Lösungsvorschläge für das Sprachduell werden in einer der Plenumsübungen besprochen, wenn es die Zeit erlaubt.*)