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. Aufgabenblatt zu Funktionale Programmierung von Mo, 21.10.2019.
Erstabgabe: Mo, 28.10.2019 (12:00 Uhr)
(Beurteilt: Teil A; ohne Abgabe und Beurteilung: Teil B)
Themen: *Funktionen auf algebraischen Datentypen, 'literate' Haskell-Skripte*

Besprechung von Aufgabenlösungen, Zweitabgabefrist

- **Teil A, programmiertechnische Aufgaben:** Am ersten Übungsgruppentermin, der auf die *Zweitabgabe* der programmiertechnischen Aufgaben folgt.
- **Teil B, Papier- und Bleistiftaufgaben:** Am ersten Übungsgruppentermin, der auf die *Erstabgabe* der programmiertechnischen Aufgaben folgt.
- **Teil C, Terminhinweise:** Für Übungsgruppen, Vorlesung, Tutorensprechstunde.
- **Zweitabgabefrist:** Siehe „Hinweise zu Organisation und Ablauf der Übung“ auf der Webseite der Lehrveranstaltung.

A Programmiertechnische Aufgaben (beurteilt, max. 50 Punkte)

Schreiben Sie zur Lösung der folgenden Aufgaben ein **'literates' Haskell-Skript** und legen Sie es in einer (Abgabe-) Datei namens `Angabe3.lhs` in Ihrem Home-Verzeichnis auf der Maschine `g0` ab. Der Name der Abgabedatei ist für Erst- und Zweitabgabe ident.

Kommentieren Sie Ihr Programm wieder zweckmäßig, aussagekräftig und problemangemessen. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten. Versehen Sie alle Funktionen, die Sie zur Lösung der Aufgaben benötigen, mit ihren Typdeklarationen, d.h. geben Sie deren syntaktische Signatur oder kurz, Signatur, explizit an.

Wir führen zwei algebraische Typen `IN_1` und `ZZ` zur Darstellung natürlicher und ganzer Zahlen ein. Die Typnamen `IN_1` und `ZZ` verstehen wir dabei als ASCII-Approximationen von IN_1 und $\mathbb{Z}$. Die `deriving`-Klauseln bei den beiden `data`-Deklarationen ermöglichen, `IN_1`- und `ZZ`-Werte auf dem Bildschirm auszugeben (s. Kapitel 4.3 der Vorlesung):

```
> data IN_1 = Eins | Nf IN_1 deriving Show
> data ZZ   = Null | Plus IN_1 | Minus IN_1 deriving Show
```

Der (nullstellige) Konstruktor `Eins` steht für die natürliche Zahl 1, der Konstruktorausdruck `Nf Eins` (kurz für "Nachfolger von Eins") für die Zahl 2, die Konstruktorausdrücke `Nf (Nf Eins)` und `Nf (Nf (Nf Eins))` für die Zahlen 3 und 4, usw. Entsprechend steht der (nullstellige) Konstruktor `Null` für die ganze Zahl 0, die Konstruktorausdrücke `Plus (Nf (Nf (Nf Eins)))` und `Minus (Nf (Nf (Nf Eins)))` für die ganzen Zahlen 4 bzw. -4, usw.

A.1 Schreiben Sie zwei Konvertierungsfunktionen `von_Zett_nach_ZZ` und `von_ZZ_nach_Zett`, die einen `Zett`-Wert in den entsprechenden `ZZ`-Wert konvertieren und umgekehrt:

```
> type Zett = Integer

> von_Zett_nach_ZZ :: Zett -> ZZ
> von_ZZ_nach_Zett :: ZZ -> Zett
```

Aufrufbeispiele:

```
von_Zett_nach_ZZ 0 ->> Null
von_Zett_nach_ZZ 2 ->> Plus (Nf Eins)
von_ZZ_nach_Zett (Minus (Nf (Nf Eins))) ->> -3
von_ZZ_nach_Zett Null ->> 0
```

A.2 Werte des Typs `ZZ` können wir nicht unmittelbar mithilfe der üblichen arithmetischen Operatoren und Relatoren verknüpfen und vergleichen:

```
> m = Plus Eins  :: ZZ
> n = Minus Eins :: ZZ

-- Aufrufbeispiele
m + n  ->> Typfehler in Anwendung
m * n  ->> Typfehler in Anwendung
m == n ->> Typfehler in Anwendung
m > n  ->> Typfehler in Anwendung
```

Schreiben Sie deshalb – ohne die Konvertierungsfunktionen aus Aufgabe A.1 auszunutzen – typspezifische arithmetische Operationen und Relationen, um mit Werten des Typs `ZZ` rechnen und sie vergleichen zu können:

```
> plus     :: ZZ -> ZZ -> ZZ
> minus    :: ZZ -> ZZ -> ZZ
> mal      :: ZZ -> ZZ -> ZZ
> durch    :: ZZ -> ZZ -> ZZ
> gleich   :: ZZ -> ZZ -> Bool
> ungleich :: ZZ -> ZZ -> Bool
> groesser :: ZZ -> ZZ -> Bool
> kleiner  :: ZZ -> ZZ -> Bool
> ggleich  :: ZZ -> ZZ -> Bool
> kgleich  :: ZZ -> ZZ -> Bool
```

Dabei soll die Bedeutung der Operatoren und Relatoren auf `ZZ` den nachstehenden Operationen und Relationen auf der Menge $\mathbb{Z}$ ganzer Zahlen entsprechen, wobei $IB =_{df} \{falsch, wahr\}$ die Menge der Wahrheitswerte $falsch$ und $wahr$ bezeichnet:

Operationen:

$$plus : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \text{ mit } plus(z, z') =_{df} z + z'$$

$$minus : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \text{ mit } minus(z, z') =_{df} z - z'$$

$$mal : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \text{ mit } mal(z, z') =_{df} z * z'$$

$durch : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ mit $durch(z, z') =_{df} \begin{cases} q & \text{falls } z' \neq 0 \wedge z = q * z' + r \wedge 0 \leq r < |z'| \\ 0 & \text{falls } z'{=}0 \text{ (Festlegung für diese Aufgabe)} \end{cases}$

Relationen:

$$gleich : \mathbb{Z} \times \mathbb{Z} \to IB \text{ mit } gleich(z, z') =_{df} z = z'$$

$$ungleich : \mathbb{Z} \times \mathbb{Z} \to IB \text{ mit } ungleich(z, z') =_{df} z \neq z'$$

$$groesser : \mathbb{Z} \times \mathbb{Z} \to IB \text{ mit } groesser(z, z') =_{df} z > z'$$

$$kleiner : \mathbb{Z} \times \mathbb{Z} \to IB \text{ mit } kleiner(z, z') =_{df} z < z'$$

$$ggleich : \mathbb{Z} \times \mathbb{Z} \to IB \text{ mit } ggleich(z, z') =_{df} z \geq z'$$

$$kgleich : \mathbb{Z} \times \mathbb{Z} \to IB \text{ mit } kgleich(z, z') =_{df} z \leq z'$$

Aufrufbeispiele (mit `m`, `n` *wie oben vereinbart):*

```
plus m n ->> Null
minus m n ->> Plus (Nf Eins)
(Plus (Nf (Nf Eins))) `mal` (Minus (Nf Eins))
  ->> Minus (Nf (Nf (Nf (Nf (Nf Eins)))))
(Plus (Nf (Nf (Nf (Nf (Nf (Nf Eins)))))))  `durch` (Plus (Nf (Nf Eins)))
  ->> Plus (Nf Eins)
(Plus (Nf (Nf (Nf (Nf (Nf (Nf Eins)))))))  `durch` (Minus (Nf (Nf Eins)))
  ->> Minus (Nf Eins)))
durch (Plus Eins) Null ->> Null
```

Testen Sie alle Funktionen auch mit weiteren eigenen Testdaten (ohne Abgabe).

Wichtig:

1. **'Literates' Haskell-Skript:** Denken Sie bitte daran, dass Sie für die Lösung dieses Aufgabenblatts ein **'literates'** Haskell-Skript schreiben sollen, kein gewöhnliches.
2. **Wiederverwendung früherer Lösungsteile:** Wenn Sie einzelne Rechenvorschriften aus früheren Lösungen wieder verwenden möchten, kopieren Sie diese bitte in die neue Abgabedatei ein. Ein `import` (eigener Module) schlägt für die Auswertung durch das Abgabeskript fehl, weil Ihre alte Lösung, aus der importiert wird, nicht mit abgesammelt wird. Deshalb: Kopieren statt importieren zur Wiederwendung!
3. **Abgaben auf `g0` auf gewünschtes Verhalten prüfen:** Aufgabenlösungen werden stets auf der Maschine `g0` unter Hugs überprüft. Überzeugen Sie sich deshalb bitte, dass Ihre Lösungen sich auf der `g0` unter Hugs wie von Ihnen erwartet und gewünscht verhalten. Das gilt besonders, wenn Sie für die Entwicklung Ihrer Haskell-Programme mit einem anderen Werkzeug oder einer anderen Maschine als der `g0` arbeiten!

B Papier- und Bleistiftaufgaben (ohne Abgabe/Beurteilung)

B.1 Um Speicherplatz zu sparen, werden Zeichenfolgen oft in komprimierter Form abgelegt, aus der bei Bedarf wieder die originale Zeichenfolge gewonnen werden kann.

Eine einfache Komprimierungsmethode macht sich Folgen wiederholender Zeichen in einer Zeichenfolge zunutze, sog. *Läufe*. Das folgende Beispiel verdeutlicht die Komprimierungsidee. Die Zeichenfolge

```
"aaaaBBBaaBBBBBBccccccccccccAbCbDDDDDDDDDDDDDDDDDD"
```

wird kodiert als Folge von Paaren aus einer Ziffernfolge und einem von einer Ziffer verschiedenen Zeichen; der Dezimalzahlwert jeder Ziffernfolge gibt dabei die Länge des Laufs des auf diese Ziffernfolge folgenden Zeichens an. Nach dieser Methode hat die obige Zeichenfolge die Komprimierung:

```
"4a3B2a6B12c1A1b1C1b18D"
```

B.1.1 Geben Sie die syntaktischen Signaturen zweier Haskell-Rechenvorschriften `komprimiere`, `dekomprimiere` an, die – vollständig implementiert – die obige Komprimierungsidee umsetzten. Benutzen Sie, wenn möglich, Typsynonyme, um die Funktionssignaturen möglichst 'sprechend' zu machen.

B.1.2 Intuitiv sollen die Rechenvorschriften `komprimiere`, `dekomprimiere` invers zueinander sein, d.h. folgende Gleichungen erfüllen:

```
dekomprimiere (komprimiere s) == s    (*)
komprimiere (dekomprimiere t) == t    (**)
```

i. Können `komprimiere` und `dekomprimiere` so implementiert werden, dass die Gleichungen $(*)$ und $(**)$ stets erfüllt sind?

ii. Welche Rand- oder Sonderfälle sind ggf. zu beachten? Wie sollen sie behandelt werden?

iii. Beschreiben Sie die Bedeutung von `komprimiere` und `dekomprimiere` so genau und präzise, dass jeweils für möglichst viele Argumentwerte die Gleichungen $(*)$ und $(**)$ erfüllt sind, und für etwaig verbleibende für beide Funktionen in eindeutiger Weise ein Funktionsergebnis festgelegt ist.

iv. Überlegen Sie sich aussagekräftige Testfälle, die das gewünschte Verhalten von `komprimiere` und `dekomprimiere` zeigen (und nach Implementierung zu überprüfen erlaubten).

v. Wenn Sie möchten, vervollständigen Sie die Implementierung von `komprimiere` und `dekomprimiere` aus B1.1 und testen Sie (z.B mit Hugs), ob Ihre Implementierung der beiden Funktionen Ihre Anforderungen aus (ii), (iii) und (iv) erfüllen oder/und ob Sie noch weiteres möglicherweise überraschendes Verhalten aufdecken.

B.2 Gegeben ist folgende Wertvereinbarung:

```
wert = [length ((++) [c] [d]) | c <- ['a'..'k'], d <- ['l'..'z']] !! 42
```

Welchen Typ hat (der Ausdruck) `wert`? Ist dieser Typ eindeutig bestimmt? Wenn ja, überzeugen Sie sich davon. Wenn nein, zeigen Sie das. Gehen Sie in beiden Fällen wie in Beispiel 1 und Beispiel 2 aus Kapitel 3.2 vor.

C Terminhinweise

1. **Übungsgruppen**: Alle 14 Übungsgruppen finden in dieser Woche erstmals statt; und zwar von heute, Montag, 21.10.2019, bis Donnerstag, 24.10.2019.

   Alle weiteren geplanten Übungsgruppentermine finden Sie auf der Webseite zur Lehrveranstaltung. Mögliche Änderungen werden dort oder/und unmittelbar in den (von möglichen Änderungen betroffenen) Übungsgruppen bekanntgegeben.

2. **Vorlesung:** Der nächste Vorlesungstermin ist am Dienstag, den 29.10.2019, von 08:15 Uhr bis 09:45 Uhr im Informatik-Hörsaal in der Treitlstraße.

   Alle weiteren geplanten Vorlesungstermine finden Sie auf der Webseite zur Lehrveranstaltung. Mögliche Änderungen werden ebenfalls dort oder/und in der Vorlesung bekanntgegeben.

3. **Tutorensprechstunde:** Die Tutorensprechstunde findet immer freitags von 12 bis 13 Uhr im `complang`-Labor statt; das nächste Mal am Freitag, den 25.10.2019. Sie finden das Labor im Innenhof des Institutsgebäudes in der Argentinierstraße 8.