Generalvereinbarung für alle Angaben im WS 2021/22

...für den Zahlbereich *IN* natürlicher Zahlen:

- IN_0, IN_1 bezeichnen die Menge der natürlichen Zahlen beginnend mit 0 bzw. 1.
- In Haskell sind natürliche Zahlen (wie in vielen anderen Programmiersprachen) nicht als elementarer Datentyp vorgesehen.
- Bevor wir sukzessive bessere sprachliche Mittel zur 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. *Typaliase* oder *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:

$$!\,: IN_0 \to IN_1$$

$$\forall\, n \in IN_0.\; n! =_{df} \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)\,! & \text{falls } n > 0 \end{cases}$$

- 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 (was ohnehin von Haskells Typsystem abgefangen würde).
- 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. nichtpositiven Werten aufgerufen würde.

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

Angabe 1 zu Funktionale Programmierung von Fr, 15.10.2021

Erstabgabe: Fr, 22.10.2021, 12:00 Uhr

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

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

Themen: Erste Schritte in Haskell mit *GHCi/Hugs, erste weiterführende Aufgaben*
Stoffumfang: *Kapitel 1 bis einschließlich Kapitel 3.1 sowie Kapitel 4.1*

- **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, Hinweise** zum Übungsrechnerzugang, zur Anmeldung und zum Beginn der Kleinübungsgruppen sowie zu Vorlesungs- und Laborfrage&Antwortforen.

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

   `Angabe1.hs`

   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 Angabe1.hs`.

   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 Angabe1 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 stets 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 `Angabe1.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.

Laden Sie anschließend Ihre Datei mittels „`:load Angabe1`“ (oder kurz „`:l Angabe1`“) in das GHCi- oder 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 des Kommandos `:reload` (kurz `:r`) einspielen.

Eine Zeichenreihe t heißt *Teilzeichenreihe* einer Zeichenreihe s gdw. (genau dann, wenn) es zwei Zeichenreihen z und z' gibt mit der Eigenschaft, dass s mit der Konkatenation der Zeichenreihen z, t und z' übereinstimmt, d.h.: $s{=}ztz'$.

A.1 Schreiben Sie eine Haskell-Wahrheitswertfunktion `ist_tzr` mit syntaktischer Signatur:

```
type Zeichenreihe        = String
type Teilzeichenreihe    = String
type IstTeilzeichenreihe = Bool

ist_tzr :: Zeichenreihe -> Teilzeichenreihe -> IstTeilzeichenreihe
```

die überprüft, ob eine Zeichenreihe Teilzeichenreihe einer anderen Zeichenreihe ist und entsprechend `True` bzw. `False` liefert.

Aufrufbeispiele:

```
ist_tzr "Urknallexplosion" "knall" ->> True
ist_tzr "Urknallexplosion" "Knall" ->> False
ist_tzr "Urknallexplosion" "" ->> True
ist_tzr "" "Urknallexplosion" ->> False
ist_tzr "Urknallexplosionsknall" "knall" ->> True
```

A.2 Schreiben Sie eine Haskell-Rechenvorschrift `tzr_zeuge` mit syntaktischer Signatur:

```
type Zerlegungszeuge = (Zeichenreihe,Zeichenreihe,Zeichenreihe)

tzr_zeuge :: Zeichenreihe -> Teilzeichenreihe -> Zerlegungszeuge
```

Angewendet auf eine Zeichenreihe s und eine Teilzeichenreihe t liefert die Rechenvorschrift `tzr_zeuge` eine Zerlegung von s in drei Zeichenreihen z, z'' und z mit $z'' = t$ und $s = zz''z'$, wenn es eine solche gibt. Gibt es mehr als einen solchen Zerlegungszeugen, ist es egal, welchen dieser Zeugen die Rechenvorschrift liefert. Gibt es keinen Zerlegungszeugen, liefert die Rechenvorschrift das Tripel `("",`tt`,"")`, das das Fehlen eines Zeugen anzeigt.

Aufrufbeispiele:

```
tzr_zeuge "Urknallexplosion" "knall" ->> ("Ur,"knall","explosion")
tzr_zeuge "Urknallexplosion" "Knall" ->> ("","KnallKnall","")
tzr_zeuge "Urknallexplosionsknall" "knall"
 ->> ("Ur,"knall","explosionsknall") oder ("Urknallexplosions","knall","")
tzr_zeuge "Urknall" "" ->> ("","","Urknall") oder ("U","","rknall") oder...
```

A.3 Schreiben Sie eine Haskell-Rechenvorschrift `tzr_zeugen` mit syntaktischer Signatur:

```
type Zerlegungszeugen = [Zerlegungszeuge]

tzr_zeugen :: Zeichenreihe -> Teilzeichenreihe -> Zerlegungszeugen
```

Angewendet auf eine Zeichenreihe s und eine Teilzeichenreihe t liefert die Rechenvorschrift `tzr_zeugen` alle Zerlegungen von s in drei Zeichenreihen z, z'' und z mit $z'' = t$ und $s = zz''z'$, wenn es solche gibt. Gibt es mehrere Zerlegungszeugen, soll ein Zeuge um so weiter links in der Resultatliste stehen, je kürzer die Zeichenreihe z der Zerlegung ist. Gibt es keine solchen Zerlegungszeugen, liefert die Rechenvorschrift die leere Liste als Resultat.

Aufrufbeispiele:

```
tzr_zeugen "Urknallexplosion" "knall" ->> [("Ur,"knall","explosion")]
tzr_zeugen "Urknallexplosion" "Knall" ->> []
tzr_zeugen "Urknallexplosionsknall" "knall"
 ->> [("Ur,"knall","explosionsknall"),("Urknallexplosions","knall","")]
tzr_zeugen "Ur" "" ->> [("","","Ur"),("U","","r"),("Ur","","")]
```

A.4 Schreiben Sie eine Haskell-Rechenvorschrift `wieOft` mit syntaktischer Signatur:

```
type Nat0 = Int

wieOft :: Zeichenreihe -> Teilzeichenreihe -> Nat0
```

die berechnet, wie oft eine Zeichenreihe als Teilzeichenreihe in einer Zeichenreihe vorkommt.

Aufrufbeispiele:

```
wieOft "Urknallexplosion" "knall" ->> 1
wieOft "Urknallexplosion" "Knall" ->> 0
wieOft "Urknallexplosionsknall" "knall" ->> 2
wieOft "Ur" "" ->> 3
```

A.5 **Ohne Beurteilung:** Beschreiben Sie für jede Rechenvorschrift in einem Kommentar knapp, aber gut nachvollziehbar, wie die Rechenvorschrift vorgeht.

A.6 **Ohne Abgabe, ohne Beurteilung:** Testen Sie alle Funktionen umfassend mit aussagekräftigen eigenen Testdaten.

B Papier- & Bleistiftaufgaben (ohne Abgabe, ohne Beurteilung)

B.1 Überzeugen Sie sich (durch Ausprobieren mit GHCi oder Hugs), dass in Aufgabe A.3 bei ansonsten unveränderter Implementierung die Signaturzeile:

```
tzr_zeugen :: Zeichenreihe -> Teilzeichenreihe -> Zerlegungszeugen
```

durch:

```
tzr_zeugen :: Zeichenreihe -> (Teilzeichenreihe -> Zerlegungszeugen)
```

oder auch durch:

```
tzr_zeugen :: [Char] -> [Char] -> [([Char],[Char],[Char])]
```

oder durch:

```
tzr_zeugen :: String -> String -> [(String,String,String)]
```

ersetzt werden darf, nicht aber z.B. durch:

```
tzr_zeugen :: (Zeichenreihe -> Teilzeichenreihe) -> Zerlegungszeugen
```

oder durch:

```
tzr_zeugen :: (Zeichenreihe,Teilzeichenreihe) -> Zerlegungszeugen
```

Warum ist das so?

B.2 Was hat die Abseitsregel (s. Kapitel 3.7) mit der (syntaktischen) Richtigkeit von Haskell-Programmen zu tun? Gibt es andere Programmiersprachen, die eine zu Haskells Abseitsregel ähnliche Regel haben? Wenn ja, was sind einige Beispiele solcher Sprachen?

B.3 Wie müssen Sie Ihre Implementierung aus Aufgabe A.1 ändern, damit Sie die Signaturzeile:

```
ist_tzr :: (Zeichenreihe,Teilzeichenreihe) -> IstTeilzeichenreihe
```

aber nicht länger die Signaturzeile:

```
ist_tzr :: Zeichenreihe -> Teilzeichenreihe -> IstTeilzeichenreihe
```

benutzen dürfen bzw. benutzen müssen?

B.4 Implementieren Sie die Rechenvorschrift

```
ist_tzr' :: (Zeichenreihe,Teilzeichenreihe) -> IstTeilzeichenreihe
```

mit gleicher Bedeutung wie `ist_tzr` aus Aufgabe A.1 ausschließlich mithilfe von `uncurry` (s. Kapitel 3.3) und `ist_tzr` aus Aufgabe A.1.

Iucundi acti labores.
Getane Arbeiten sind angenehm.

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

C Hinweise zum Übungsrechnerzugang, zu Übungsgruppen und zu Frage- und Antwortforen

1. Die **Konto- und Kennwortinformation** für den Übungsrechner `g0` ist am Freitag, 08.10.2021, an die generische email-Adresse `e<matr-nr>@student.tuwien.ac.at` aller angemeldeten Teilnehmer versandt worden. Ändern Sie Ihr initiales Kennwort bitte möglichst umgehend.

2. **Kleinübungsgruppen: Anmeldung, erster Termin**

   (a) Die **Anmeldung zu den Kleinübungsgruppen** (max. Gruppengröße: 20) erfolgt in TISS. Die Anmeldung ist seit Mittwoch, 13.10.2021, 20:00 Uhr, bis spätestens

   Freitag, 22.10.2021, 20:00 Uhr.

   möglich. Nutzen Sie die Möglichkeit zur Mitarbeit in den Kleinübungsgruppen zu Ihrem Vorteil!

   (b) Die **Kleinübungsgruppen finden montags, mittwochs und freitags statt, erstmals in der Woche vom 1. bis zum 5. November 2021**. Aufgrund des Allerheiligenfeiertags entfallen dabei die Kleinübungsgruppentermine am Montag, den 1. November 2021. Teilnehmer an diesen Übungsgruppen schalten sich in der ersten Novemberwoche bitte zu einer der mittwochs am 3. November 2021 bzw. freitags am 5. November 2021 über Zoom stattfindenden Übungsgruppen dazu. Die Teilnahme-URLs sind im TUWEL-Kurs bereitgestellt.

3. Die **Labor- und Vorlesungsfrage- und Antwortforen** finden bis zum Beginn der weihnachtlichen Ferialzeit grundsätzlich jeden Mittwoch bzw. Donnerstag jeweils von 14:15-15:00 Uhr statt. Diese Foren finden auch in Wochen ohne Vorlesungs- oder Übungsgruppentermin statt. Die Teilnahme-URLs für die Frage- und Antwortforen sind im TUWEL-Kurs bereitgestellt.