Generalvereinbarung für alle Angaben im WS 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, 14.10.2022

Erstabgabe: Fr, 21.10.2022, 12:00 Uhr

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

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

Themen: Erste Schritte in Haskell mit *GHCi/Hugs, erste weiterführende Aufgaben*
Stoffumfang: *Kapitel 1 bis einschließlich Kapitel 3.3, Kapitel 4.1 und Kapitel 5.2.1*

- **Teil A, programmiertechnische Aufgaben:** Besprechung am ersten Plenumsübungstermin, der auf die *Zweitabgabe* der programmiertechnischen Aufgaben folgt.
- **Teil B, Papier- und Bleistiftaufgaben:** Besprechung am ersten Plenumsübungstermin, 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

   `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 aussagekräftig, zweckmäßig 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.

A.1 Schreiben Sie eine Rechenvorschrift `haeufigkeit`, die für jedes Zeichen in einer Zeichenreihe bestimmt, wie oft es darin vorkommt. Dabei steht ein Zeichen und seine Häufigkeit um so weiter links in der Ergebnisliste, je weiter links das Zeichen zum ersten Mal in der Argumentzeichenreihe vorkommt:

```
type Nat0         = Int
type Zeichen      = Char
type Zeichenreihe = [Zeichen]
type Haeufigkeit  = Nat0
type Histogramm   = [(Zeichen,Haeufigkeit)]

haeufigkeit :: Zeichenreihe -> Histogramm
```

Aufrufbeispiele:

```
haeufigkeit "" ->> []
haeufigkeit "Haskell"
  ->> [('H',1),('a',1),('s',1),('k',1),('e',1),('l',2)]
haeufigkeit "Programmierung mit Haskell"
  ->> [('P',1),('r',3),('o',1),('g',2),('a',2),('m',3,),('i',2),('e',2),
       ('u',1),('n',1),(' ',2),('t',1),('H',1),('s',1),('k',1),('l',2)]
```

A.2 Schreiben Sie eine Rechenvorschrift `gewicht`, die das Gewicht einer Zeichenreihe als Summe der Gewichte ihrer Zeichen berechnet. Das Gewicht eines Zeichens ist dabei in einem Gewichtsverzeichnis angegeben; kommt ein Zeichen in diesem Verzeichnis nicht vor, so hat es Gewicht 0. Kommt ein Zeichen mehr als einmal darin vor, ist das Verzeichnis fehlerhaft; in solch einem Fall liefert *gewicht* den fehleranzeigenden Wert -1.

```
type Gewicht             = Nat0
type Gewichtsverzeichnis = [(Zeichen,Gewicht)]
fehlerwert = -1

gewicht :: Zeichenreihe -> Gewichtsverzeichnis -> Gewicht
```

Aufrufbeispiele:

```
gewicht "Haskell"  [(‘a‘,2),(‘k‘,14),(‘H‘,10),(‘l‘,8)] ->> 42
gewicht "Haskell" [(‘a‘,6),(‘b‘,4),(‘H‘,0)] ->> 6
gewicht "Haskell" [] ->> 0
gewicht "Haskell" [(‘a‘,2),(‘k‘,14),(‘H‘,10),(‘k‘,3),(‘l‘,8),(‘a‘,9)] ->> -1
```

A.3 Fehlerhafte Gewichtsverzeichnisse sind ärgerlich. Schreiben Sie eine Rechenvorschrift `korrigiere`, die ein fehlerhaftes Gewichtsverzeichnis g in folgender Weise berichtigt: kommt ein Zeichen in mehr als einem Zeichen/Gewicht-Paar in g vor, so wird nur das am weitesten links stehende Paar behalten, die anderen gelöscht:

```
korrigiere :: Gewichtsverzeichnis -> Gewichtsverzeichnis
```

Aufrufbeispiele:

```
korrigiere [(‘a‘,2),(‘k‘,14),(‘H‘,10),(‘l‘,8)]
  ->> [(‘a‘,2),(‘k‘,14),(‘H‘,10),(‘l‘,8)]
korrigiere [(‘a‘,2),(‘k‘,14),(‘H‘,10),(‘k‘,3),(‘l‘,8),(‘a‘,9)]
  ->> [(‘a‘,2),(‘k‘,14),(‘H‘,10),(‘l‘,8)]
korrigiere [] ->> []
```

A.4 Schreiben Sie eine zweite Rechenvorschrift `korrigiere'` für die Berichtigung fehlerhafter Gewichtsverzeichnisse. Anders als `korrigiere`, berichtigt `korrigiere'` ein fehlerhaftes Verzeichnis, in dem die Gewichte für ein Zeichen aufaddiert werden. Das in der Argumentliste am weitesten links stehende Zeichen/Gewicht-Paar enthält dann das berichtigte Gewicht für dieses Zeichen, weitere Zeichen/Gewicht-Paare mit diesem Zeichen gibt es in der berichtigten Gewichtsliste nicht.

```
korrigiere' :: Gewichtsverzeichnis -> Gewichtsverzeichnis
```

Aufrufbeispiele:

```
korrigiere' [(‘a‘,2),(‘k‘,14),(‘H‘,10),(‘l‘,8)]
  ->> [(‘a‘,2),(‘k‘,14),(‘H‘,10),(‘l‘,8)]
korrigiere' [(‘a‘,2),(‘k‘,14),(‘H‘,10),(‘k‘,3),(‘l‘,8),(‘a‘,9)]
  ->> [(‘a‘,11),(‘k‘,17),(‘H‘,10),(‘l‘,8)]
korrigiere' [] ->> []
```

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 Implementieren Sie die Rechenvorschrift `gewicht'` mit gleicher Bedeutung wie `gewicht`, aber folgender Signatur:

```
gewicht' :: (Zeichenreihe,Gewichtsverzeichnis) -> Gewicht
```

Überlegen Sie, ob Ihnen eine der Funktionen `curry` oder `uncurry` zusammen mit der Funktion `gewicht` helfen, `gewicht'` „in einer Zeile“ besonders einfach zu implementieren.

B.2 Erklären Sie anhand der Funktionen `gewicht` und `gewicht'` den Unterschied zwischen curryfizierten und uncurryfizierten Funktionen.

B.3 Implementieren Sie die Rechenvorschrift `gewicht''` mit gleicher Bedeutung wie `gewicht`, aber anders als `gewicht` berichtigt `gewicht''` vor der Gewichtsberechnung eine fehlerhafte Argumentsgewichtsliste in Abhängigkeit des Korrekturmodusargumentwerts mit `korrigiere` bzw. `korrigiere'`:

```
data Korrekturmodus = Korr | Korr' deriving Eq
type Modus          = Korrekturmodus

gewicht'' :: Zeichenreihe -> Gewichtsverzeichnis -> Modus -> Gewicht
```

B.4 Ersetzen Sie die `deriving`-Klausel in B.3 durch eine bedeutungsgleiche `instance`-Deklaration.

B.5 Haben Sie in B.4 von den Protoimplementierungen in der Typklasse `Eq` Gebrauch gemacht? Auf welche Weise?

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?

Licht oder nicht Licht - das ist hier die Frage!

Um in der aktuellen Energiekrise Strom zu sparen, gehört nun zu den regelmäßigen Aufgaben des Nachtwachdienstes der TU Wien das Löschen nicht benötigter Korridorlampen.

In manchen dieser Korridore haben die Lampen alle einen eigenen Ein- und Ausschalter; jedes Betätigen eines Schalters schaltet die zugehörige Lampe ein bzw. aus, je nachdem, ob sie vorher aus bzw. an war.

Einer der Nachtwächter hat es sich als Zeitvertreib angewöhnt, in diesen Korridoren die Lampen noch einmal auf eine besondere Art und Weise ein- und auszuschalten, bevor er die nichtbenötigten endgültig löscht: hat der Korridor n Lampen, so geht er n-mal vollständig auf und ab; jedes vollständige Auf- und Abgehen ist ein Durchgang. Nur auf dem Hinweg eines Durchgangs betätigt er Lichtschalter, nie auf dem Rückweg: ist er auf dem Hinweg des i-ten Durchgangs, so betätigt er jeden Schalter, dessen Position ohne Rest durch i teilbar ist.

Jedes Mal versucht sich der Nachtwächter im vorhinein die Frage zu beantworten: wenn die n Lampen im Korridor alle aus sind, wenn ich komme, wird die n-te und damit letzte Lampe im Korridor an oder aus sein, wenn ich mit meinen n Durchgängen fertig bin?

Meist liegt der Nachtwächter mit seiner Antwort falsch. Helfen Sie ihm! Schreiben Sie ein Programm, das die Antwort berechnet! In einer Sprache Ihrer Wahl!

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 Frage des Nachtwächters für positive Zahlen n von Lampen in einem Korridor beantwortet.

(Für $n = 3$ oder für $n = 8191$ sollte Ihr Programm die Antwort “aus” liefern, für $n = 6241$ die Antwort “an”.)

(*Lösungsvorschläge für das Sprachduell werden im Rahmen ausreichender Zeit in einer der Plenumsübungen besprochen.*)