# Generalvereinbarung für alle Angaben im WS 2020/21

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

- $IN_0$, $IN_1$ bezeichnen die Menge der natürlichen Zahlen beginnend mit 0 bzw. 1.
- In Haskell (wie in anderen Programmiersprachen) sind natürliche Zahlen 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. *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, 16.10.2020.**

**Erstabgabe: Fr, 23.10.2020 (12:00 Uhr)**

**Zweitabgabe: Siehe „Hinweise zu Org. u. Ablauf der Übung" (LVA-Webseite)**

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

Themen: *GHCi/Hugs kennenlernen, erste Schritte in Haskell, erste weiterführende Aufgaben (Kap. 1 bis einschl. Kap. 3.1)*

- **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 zu:** Rechnerzugang, Bildung von Klein- und Großübungsgruppen.

## Wichtig

1. Befolgen Sie die Anweisungen aus den Begleitdateien zu Angabe 1 sorgfältig, um ein reibungsloses Zusammenspiel mit dem Testsystem sicherzustellen. Wenn Sie Fragen dazu haben, stellen Sie diese bitte im TISS-Forum zur Lehrveranstaltung.
2. Erweitern Sie für die für diese Angabe zu schreibenden Rechenvorschriften die zur Verfügung gestellte Rahmendatei

   `Angabe1.hs`

   und legen Sie diese 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 Modul-Anweisung `module Angabe1.hs 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 Konstanten. 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 Haskell-Rechenvorschrift `filtere` mit syntaktischer Signatur:

```
type Nat1 = Int
filtere :: Nat1 -> [Int] -> [Int]
```

Angewendet auf eine natürliche Zahl $n$ größer oder gleich 1 und eine Liste ganzer Zahlen liefert `filtere` eine absteigend sortierte Liste aller Zahlen der Argumentliste, die genau $n$-mal in der Argumentliste vorkommen.

*Aufrufbeispiele:*

```
filtere 4 [] ->> []
filtere 1 [42] ->> [42]
filtere 2 [42] ->> []
filtere 1 [4,-2,5,4,3,-2,5,4,-12] ->> [3,-12]
filtere 2 [4,-2,5,4,3,-2,5,4,-12] ->> [5,-2]
filtere 3 [4,-2,5,4,3,-2,5,4,-12] ->> [4]
```

A.2 Schreiben Sie eine Haskell-Wahrheitswertfunktion `kommt_vor` mit syntaktischer Signatur:

```
kommt_vor :: Int -> [(Int,Int)] -> Bool
```

Angewendet auf eine ganze Zahl $n$ und eine Liste von Paaren ganzer Zahlen überprüft `kommt_vor`, ob $n$ in einem dieser Paare als Komponente vorkommt.

```
kommt_vor 3 [] ->> False
kommt_vor 3 [(2,3),(3,4),(5,6)] ->> True
kommt_vor 5 [(2,3),(3,4),(5,6)] ->> True
kommt_vor 6 [(2,3),(3,4),(5,6)] ->> True
kommt_vor 9 [(2,3),(3,4),(5,6)] ->> False
```

A.3 Schreiben Sie eine Haskell-Rechenvorschrift `aus` (‘auffüllen und sortieren’) mit syntaktischer Signatur:

```
aus :: [Int] -> [Int]
```

Angewendet auf eine Liste ganzer Zahlen liefert `aus` als Ergebnis eine aufsteigend sortierte Liste ganzer Zahlen, in der genau die Elemente der Argumentliste so oft

vorkommen, wie das oder die in der Argumentliste am häufigsten vorkommende(n) Element(e).

*Aufrufbeispiele:*

```
aus [] ->> []
aus [42] ->> [42]
aus [4,-2,5,4,3,-2,5,4,-12] ->> [-12,-12,-12,-2,-2,-2,3,3,3,4,4,4,5,5,5]
```

A.4 Seien $s = s_1 s_2 \ldots s_n$ und $t = t_1 t_2 \ldots t_n$ zwei Zeichenreihen der Länge $n$ über einem Zeichenvorrat $Z$. Die Zahl der Positionen, an denen die jeweiligen Zeichen in $s$ und $t$ voneinander verschieden sind, heißt *Hamming-Abstand* von $s$ und $t$, in Zeichen (lies das Symbol '$=_{df}$' als *'ist definitionsgemäß gleich'*):

$$H(s,t) =_{df} |\{i \in \{1,...n\} \mid s_i \neq t_i\}|$$

Für Zeichenreihen unterschiedlicher Länge ist der Hamming-Abstand nicht definiert.

Schreiben Sie eine Haskell-Rechenvorschrift `h` mit syntaktischer Signatur:

```
h :: String -> String -> Int
```

Angewendet auf zwei Zeichenreichen $s$ und $t$ berechnet `h` den Hamming-Abstand von $s$ und $t$ berechnet, wenn dieser definiert ist; ansonsten liefert `h` den Wert $-1$.

*Aufrufbeispiele:*

```
h "Fahrrad" "Autobus" ->> 7
h "Funken" "funken" ->> 1
h "1001" "1111" ->> 2
h "Fakt" "Fake" ->> 1
h "Funktional" "Objektorientiert" ->> -1
```

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, auch mit eigenen Testdaten.

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

B.1 Gegeben ist die Haskell-Rechenvorschrift:

```
f :: Int -> Int
f n = if n == 0 then 0 else (f (n-1)) + 3
```

(a) Was berechnet `f`? (Angewendet auf ein nichtnegatives Argument $n$, berechnet `f` als Resultat den Wert...; angewendet auf ein echt negatives Argument...).

(b) Was wäre ein besserer, ein sprechenderer Name für die Funktion `f`?

(c) Welche vordefinierte Funktion in Haskell kann zur Berechnung von `f` ausgenutzt werden? (Stichwort: Operatorabschnitt, Kap. 3.5).

(d) Besitzt die vordefininierte Funktion für alle Argumentwerte das gleiche Verhalten wie `f`? Wenn nicht, fir welche Argumentwerte unterscheidet sich das Verhalten und in welcher Weise?

(e) Schreiben Sie die Funktion `f`

i. mithilfe bewachter Ausdrücke.

ii. mithilfe der in Haskell vordefinierten Funktion.

iii. argumentfrei mithilfe einer anonymen $\lambda$-Abstraktion.

(Siehe Kapitel 3.1).

(f) Haben die Haskell-Rechenvorschriften

```
g :: Int -> Int
g n = if n == 0 then 0 else g (n-1) + 3

h :: Int -> Int
h n = if n == 0 then 0 else 3 + h (n-1)
```

dieselbe Bedeutung wie `f`? Was schließen Sie daraus für Klammereinsparungsregeln in Haskell?

B.2 Überzeugen Sie sich (durch Ausprobieren mit GHCi oder Hugs), dass in Aufgabe A.1 bei ansonsten gleich bleibender Implementierung die Signaturzeile:

```
filtere ::  Nat1 -> [Int] -> [Int]
```

durch:

```
filtere ::  Nat1 -> ([Int] -> [Int])
```

aber nicht durch:

```
filtere ::  (Nat1 -> [Int]) -> [Int]
```

und auch nicht durch:

```
filtere ::  (Nat1,[Int]) -> [Int]
```

ersetzt werden darf. Warum? Haben Sie eine Vermutung?

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

```
filtere ::  (Nat1,[Int]) -> [Int]
```

aber nicht länger die Signaturzeile:

```
filtere ::  Nat1 -> [Int] -> [Int]
```

benutzen dürfen bzw. benutzen müssen?

*Iucundi acti labores.*
*Getane Arbeiten sind angenehm.*

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

## C Hinweise zu Rechnerzugang, Klein-/Plenumsübungsgruppenanmeldung

1. **Konto- und Kennwortinformation** für die `g0` ist am Montag, 12.10.2020, an die generische email-Adresse `e<matr-nr>@student.tuwien.ac.at` angemeldeter Teilnehmer versandt worden. Ändern Sie Ihr initiales Kennwort möglichst umgehend.
2. **Information zur Anmeldung zu je einer**
   - Kleinübungsgruppe (max. Gruppengröße: 25)
   - Plenumsübungsgruppe (max. Gruppengröße: 175)

   wird vorauss. in der kommenden Woche über TISS ausgeschickt.