# Generalvereinbarung für alle Aufgabenblätter im WS 2019/20

**...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.

- 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

**1. Aufgabenblatt zu Funktionale Programmierung von Mo, 07.10.2019.**
**Erstabgabe: Mo, 14.10.2019 (12:00 Uhr)**
(Beurteilt: Teil A; ohne Abgabe und Beurteilung: Teil B)
Themen: *Hugs kennenlernen, erste Schritte in Haskell, erste weiterführende Aufgaben*

## 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.
- **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 gewöhnliches Haskell-Skript und legen Sie es in einer (Abgabe-) Datei namens `Angabe1.hs` in Ihrem Home-Verzeichnis auf der Maschine `g0` ab. Der Name der Abgabedatei ist für Erst- und Zweitabgabe ident.

Machen Sie sich mit den unterschiedlichen Möglichkeiten vertraut, ihre Entwurfsentscheidungen in Haskell-Programmen durch Kommentare zu dokumentieren.

Kommentieren Sie Ihr 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 deren syntaktische Signatur oder kurz, Signatur, explizit an.

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

A.1 Schreiben Sie eine Haskell-Rechenvorschrift `streiche` mit Signatur

```
streiche :: String -> Int -> Char -> String
```

Angewendet auf eine Zeichenreihe $s$, eine ganze Zahl $i$ und ein Zeichen $c$, entfernt `streiche` jedes $i$-te Vorkommen von $c$ in $s$. Ist $i$ kleiner oder gleich 0, ist das Ergebnis von `streiche` die Zeichenreihe $s$ selbst.

*Aufrufbeispiele:*

```
streiche "abcabcabcabcabc" 2 'b' ->> "abcacabcacabc"
streiche "abcabcabcabcabc" 3 'b' ->> "abcabcacabcabc"
streiche "abcabcabcabcabc" 0 'b' ->> "abcabcabcabcabc"
```

```
streiche "abcabcabcabcabc" (-5) 'b' ->> "abcabcabcabcabc"
streiche "abcabcabcabcabc" 2 'd' ->> "abcabcabcabcabc"
```

A.2 Schreiben Sie eine Haskell-Wahrheitswertfunktion `ist_umgekehrt_2er_potenz`, die überprüft, ob die umgekehrte Ziffernfolge einer natürlichen Zahl eine Zweierpotenz ist:

```
type Nat0 = Int
ist_umgekehrt_2er_potenz :: Nat0 -> Bool
```

*Aufrufbeispiele:*

```
ist_umgekehrt_2er_potenz 61 ->> True
ist_umgekehrt_2er_potenz 16 ->> False
ist_umgekehrt_2er_potenz 46 ->> True
ist_umgekehrt_2er_potenz 64 ->> False
ist_umgekehrt_2er_potenz 1 ->> True
ist_umgekehrt_2er_potenz 0 ->> False
```

A.3 Sei eine Liste natürlicher Zahlen gegeben. Welches ist die größte Zahl in dieser Liste, deren Dezimaldarstellung ein Palindrom ist, wenn es eine solche gibt? Gibt es keine solche Zahl in der Liste, soll das Resultat die Zahl $-1$ sein. Als *Palindrom* bezeichnet man dabei eine Zeichenfolge, die vorwärts- und rückwärts gelesen, gleich ist, z.B. die Zeichenreihe "oTTo" oder die Dezimaldarstellung der Zahl zwölftausenddreihunderteinundzwanzig 12321, nicht aber die Zeichenreihe "Otto".

```
type Nat0 = Int
groesstes_palindrom_in :: [Nat0] -> Int
```

*Aufrufbeispiele:*

```
groesstes_palindrom_in [11,1,5,3,7,1221,11,2,1221,5] ->> 1221
groesstes_palindrom_in [112,1,5,3,7,0,4,113,2,5,0] ->> 7
groesstes_palindrom_in [12,13,14] ->> -1
```

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

**Wichtig:** Denken Sie bitte daran, dass Aufgabenlösungen stets auf der Maschine `g0` unter Hugs überprüft werden. Stellen Sie deshalb für Ihre Lösungen zu diesem und auch allen weiteren Aufgabenblättern sicher, dass Ihre Programmierlösungen auf der `g0` unter Hugs die von Ihnen gewünschte Funktionalität aufweisen, und überzeugen Sie sich bei jeder Abgabe davon. Das gilt besonders, wenn Sie für die Entwicklung Ihrer Haskell-Programme mit einem anderen Werkzeug oder einer anderen Maschine arbeiten!

## B Papier- und Bleistiftaufgaben (ohne Abgabe/Beurteilung)

B.1 Gegeben ist die Haskell-Rechenvorschrift:

```
f :: Int -> Int
f n = if n == 0 then 1 else 5 * f (n-1)
```

(a) Was berechnet `f`? (Angewendet auf ein nichtnegatives Argument $n$, liefert `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) Führt diese Funktion für alle Argumentwerte zum gleichen Verhalten wie `f`? Ist es kritisch, wenn das Verhalten für manche Argumentwerte nicht eins-zu-eins übereinstimmt, wenn man das Gesamtverhalten beider Funktionen betrachtet?

(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 1 else (g (n-1)) * 5

h :: Int -> Int
h n = if n == 0 then 1 else h (n-1) * 5
```

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

B.2 Überzeugen Sie sich (durch Ausprobieren in Hugs oder GHC/GHCi), dass in Aufgabe A.1 die Signaturzeile

```
streiche :: String -> Int -> Char -> String
```

durch

```
streiche :: String -> (Int -> (Char -> String))
```

aber nicht durch

```
streiche :: ((String -> Int) -> Char) -> String
```

ersetzt werden darf. Haben Sie eine Vermutung, warum?

## C Hinweise zu Rechnerzugang, Ü-Gruppen, Tutorensprechstunde

1. **Rechnerzugangsinformation** für die `g0` wird spätestens bis Mittwoch, 09.10.2019, an die generische email-Adresse `e<matr-nr>@student.tuwien.ac.at` angemeldeter Teilnehmer versandt; ändern Sie Ihr initiales Passwort möglichst sofort nach Erhalt.

2. Informationen zum Ablauf der **Übungsgruppenanmeldung** wird spätestens bis Freitag, 11.10.2019, über TISS ausgeschickt.

3. Zusätzlich zu den Übungsgruppen wird es (doch) auch eine **Tutorensprechstunde** geben, jeweils freitags von 12 bis 13 Uhr im `complang`-Labor im Innenhof des Institutsgebäudes in der Argentinierstr. 8, erstmals am Freitag, den 11.10.2019.