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

6. Aufgabenblatt zu Funktionale Programmierung von Fr, 20.11.2020.
Erstabgabe: Fr, 27.11.2020 (12:00 Uhr)

Themen: *Funktionen höherer Ordnung, Rechnen mit Funktionen, Funktionen als Argument und Resultat, echt und unecht polymorphe Funktionen auf polymorphen Datentypen (Kap. 1 bis Kap. 11)*

- **Teil A, programmiertechnische Aufgaben:** Besprechung am ersten Übungsgruppentermin, der auf die *Zweitabgabe* der programmiertechnischen Aufgaben folgt.
- Teil B, Papier- und Bleistiftaufgaben: Entfällt auf den Angaben 5 bis 7.
- **Teil C, Terminhinweise:** Für Vorlesung, 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

   `Angabe6.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 Angabe6.hs`.

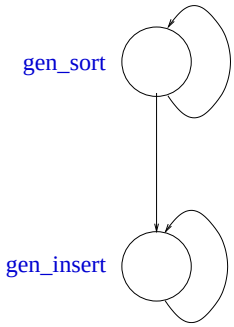
   Löschen Sie keinesfalls eine Deklaration aus dieser *Template*-Datei! Auch dann nicht, wenn Sie einige dieser Deklarationen nicht oder nicht vollständig implementieren wollen. Löschen Sie auch nicht die Modul-Anweisung `module Angabe6 where` am Anfang der *Template*-Datei.
3. Der Name der Abgabedatei 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 sicher, dass sich Ihre Programme (auch) auf der `g0` unter GHCi so verhalten, wie von Ihnen gewünscht, und 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. 100 Punkte)

Erweitern Sie zur Lösung der programmiertechnischen Aufgaben die Rahmendatei `Angabe6.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.

A.1 *Generisches Sortieren und Einfügen:*

Schreiben Sie zwei polymorphe Funktionen höherer Ordnung `gen_sort` und `gen_insert` zum Sortieren und sortierten Einfügen in eine Liste. Die Funktionen sind bezüglich der Sortierrelation parametrisiert (z.B. $\leq$, $\geq$, etc.) und in diesem Sinn generisch; ihre Argumentlisten können sortiert oder unsortiert sein und Elemente mehrfach enthalten kann oder nicht. Zusammen bilden `gen_sort` und `gen_insert` ein hierarchisches System mit folgendem Aufrufgraphen:




```
gen_sort :: (a -> a -> Bool) -> [a] -> [a]

gen_insert :: (a -> a -> Bool) -> a -> [a] -> [a]
```

Im Detail:

- `gen_sort` angewendet auf eine Sortierrelation S (wie z.B. $<$, $\leq$, $>$, $\geq$,...) und eine Liste liefert als Resultat die bezüglich S sortierte Liste. Für die sortierte Liste gilt: Sind x und y zwei wertverschiedene Elemente der sortierten Liste, die in der Relation S stehen (d.h. $x\ S\ y = wahr$), so steht x weiter links als y.
- `gen_insert` angewendet auf eine Sortierrelation S, ein Element x und eine bezüglich S geordnete oder ungeordnete duplikat- oder nicht duplikatfreie Liste ys liefert als Resultat die Argumentliste, in die das Element x bezüglich S sortiert eingefügt ist und die relative Anordung aller anderen Listenelemente unverändert ist. Das Element x wird dabei von `gen_insert` unmittelbar vor dem von links ersten Element y aus ys eingefügt, für das zum ersten Mal $x\ S\ y = wahr$ gilt (für alle y', die weiter links als y in der Argumentliste stehen, gilt also $x\ S\ y' = falsch$).

Aufrufbeispiele:

```
gen_sort (<=) [] ->> []
gen_sort (<=) [3,5,2,4] ->> [2,3,4,5]
gen_sort (<=) [3,5,3,4,2,4] ->> [2,3,3,4,4,5]
gen_sort (<) [] ->> []
gen_sort (<) [3,5,2,4] ->> [2,3,4,5]
gen_sort (<) [3,5,3,4,2,4] ->> [2,3,3,4,4,5]
gen_sort (>=) [] ->> []
```

```
gen_sort (>=) [3,5,2,4] ->> [5,4,3,2]
gen_sort (>=) [3,5,3,4,2,4] ->> [5,4,4,3,3,2]
gen_sort (>) [] ->> []
gen_sort (>) [3,5,2,4] ->> [5,4,3,2]
gen_sort (>) [3,5,3,4,2,4] ->> [5,4,4,3,3,2]

gen_insert (<=) 4 [] ->> [4]
gen_insert (<=) 4 [1,3,5,7,9] ->> [1,3,4,5,7,9]
gen_insert (<=) 4 [1,3,4,4,5,7,9] ->> [1,3,4,4,4,5,7,9]
gen_insert (<=) 4 [1,3,1,3,6,8] ->> [1,3,1,3,4,6,8]
gen_insert (<) 4 [] ->> [4]
gen_insert (<) 4 [1,3,5,7,9] ->> [1,3,4,5,7,9]
gen_insert (<) 4 [1,3,4,4,5,7,9] ->> [1,3,4,4,4,5,7,9]
gen_insert (<) 4 [1,3,1,3,6,8] ->> [1,3,1,3,4,6,8]

gen_insert (>=) 4 [] ->> [4]
gen_insert (>=) 4 [9,7,5,3,1] ->> [9,7,5,4,3,1]
gen_insert (>=) 4 [1,3,4,4,5,7,9] ->> [4,1,3,4,4,5,7,9]
gen_insert (>=) 4 [1,3,1,3,6,8] ->> [4,1,3,1,3,6,8]
gen_insert (>=) 4 [10,13,4,4,5,7,9] ->> [10,13,4,4,4,5,7,9]
gen_insert (>) 4 [] ->> [4]
gen_insert (>) 4 [9,7,5,3,1] ->> [9,7,5,4,3,1]
gen_insert (>) 4 [1,3,4,4,5,7,9] ->> [4,1,3,4,4,5,7,9]
gen_insert (>) 4 [1,3,1,3,6,8] ->> [4,1,3,1,3,6,8]
gen_insert (>) 4 [10,13,4,4,5,7,9] ->> [10,13,4,4,5,7,9,4]
```

A.2 **Ohne Abgabe, ohne Beurteilung:**

(a) Von welchem Polymorphietyp sind `gen_sort` und `gen_insert`? Woran erkennt man das? Welche synonymen Bezeichnungen gibt es für diesen Polymorphietyp?

(b) Wie unterscheiden sich die ergebnisgleichen Aufrufe `gen_insert (>)` und `gen_insert (>=)` bezüglich des Einfügens von `4` in die Liste `[10,13,4,4,3,7,9]`? Welches Vorkommen von `4` ist das jeweils Eingefügte in den Resultatlisten der beiden Aufrufe?

(c) Wie lauten die curryfizierten und nichtcurryfizierten Lesarten von `gen_sort` und `gen_insert`?

Mithilfe von `gen_sort` und `gen_insort`, die zusammen ein hierarchisches System polymorpher Funktionen höherer Ordnung bilden, können mit wenig zusätzlichem Implementierungsaufwand eine Vielzahl von Sortieraufgaben über unterschiedlichsten Typen gelöst werden. Einige davon betrachten wir beispielhaft in den folgenden Aufgaben. Schreiben Sie die gesuchten Sortierfunktionen alle als Spezialisierung von `gen_sort`, d.h.:

```
sortiere_A3_bis_A13 = gen_sort sortierrelation
                        where sortierrelation ... = ...
```

A.3 Schreiben Sie zwei Sortierfunktionen `auf_ord` und `ab_ord`, mit denen sich Listen von Werten eines beliebigen geordneten Typs (d.h. der Typ ist Element von `Ord`) auf- bzw. absteigend sortieren lassen:

```
auf_ord :: Ord a => [a] -> [a]
ab_ord  :: Ord a => [a] -> [a]
```

A.4 **Ohne Abgabe, ohne Beurteilung:**

(a) Von welchem Polymorphietyp sind `auf_ord` und `ab_ord`? Woran erkennt man das? Welche synonymen Bezeichnungen gibt es für diesen Polymorphietyp?

(b) Warum ist für `auf_ord` und `ab_ord` der Typklassenkontext `Ord a =>` nötig?

(c) Ist es möglich, den Kontext in den Signaturen von `auf_ord` und `ab_ord` zu `(Eq a,Ord a) =>` oder zu `(Ord a,Enum a) =>` abzuändern? Welche Auswirkungen hätte das jeweils?

A.5 Kopieren Sie die Deklaration des Typs

(a) `Intervall`

von Angabe 3 zusammen mit den Instanzbildungen für `Eq`, `Ord` und `Show` sowie die Deklarationen der Typen

(b) `BBaum`

(c) `TBaum`

(d) `Baum`

von Angabe 4 mit um die Klasse `Ord` erweiterten `deriving`-Klauseln für `BBaum` und `TBaum` sowie den Instanzbildungen für `Eq`, `Ord` und `Show` für `Baum` in die Abgabedatei:

```
type UntereSchranke = Int
type ObereSchranke  = Int
data Intervall      = IV (UntereSchranke,ObereSchranke)
                      | Leer
                      | Ungueltig

type Text  = String
data BBaum = Blatt Text
             | Knoten Text BBaum BBaum deriving (Eq,Ord,Show)

data TBaum = TB
             | TK TBaum TBaum TBaum deriving (Eq,Ord,Show)

type Info a = [a]
data Baum a = B (Info a)
              | K (Baum a) (Info a) (Baum a)
```

A.6 **Ohne Abgabe, ohne Beurteilung:** Testen Sie die Funktionen `auf_ord` und `ab_ord` auch mit Intervall-, BBaum-, TBaum- und Baumlisten!

A.7 **Ohne Abgabe, ohne Beurteilung:**

(a) Kann in den `deriving`-Klauseln für `BBaum` und `TBaum` in A.6 die Klasse `Eq` weggelassen werden?

(b) Kann entsprechend die `Eq`-Instanzdeklaration für `Baum` entfallen?

Begründen Sie Ihre Antwort jeweils.

A.8 Schreiben Sie zwei Sortierfunktionen `auf_lst` und `ab_lst`, mit denen sich Listen von Listen beliebiger Typen bezüglich der Länge der Elementlisten auf- bzw. absteigend sortieren lassen:

```
auf_lst :: [[a]] -> [[a]]
ab_lst  :: [[a]] -> [[a]]
```

A.9 **Ohne Abgabe, ohne Beurteilung:**

(a) Von welchem Polymorphietyp sind `auf_lst` und `ab_lst`?

(b) Warum ist für `auf_lst` und `ab_lst` kein Typklassenkontext nötig?

(c) Ist es möglich, Kontexte wie `Show a =>` oder `(Ord a,Enum a) =>` in den Signaturen von `auf_lst` und `ab_lst` zu ergänzen? Wie würde sich der Polymorphietyp ändern? Wie wäre der geänderte Polymorphietyp möglichst genau benannt?

(d) Wie verhielten sich diese Kontextergänzungen zum Hauptleitsatz funktionaler Programmierung? Begründen Sie Ihre Antwort.

A.10 Schreiben Sie zwei Sortierfunktionen `auf_fun` und `ab_fun`, mit denen sich Listen ganzzahliger Funktionen auf- bzw. absteigend sortieren lassen. Eine Funktion f heißt dabei kleiner/größer oder gleich einer Funktion g, wenn der Funktionswert von f an der Stelle 0 kleiner/größer oder gleich dem von g an der Stelle 0 ist.

```
auf_fun :: [Int -> Int] -> [Int -> Int]
ab_fun  :: [Int -> Int] -> [Int -> Int]
```

A.11 Verallgemeinern Sie `auf_fun` und `ab_fun` zu zwei Sortierfunktionen `auf_onfun` und `ab_onfun`, die die Sortieraufgabe in gleicher Weise für Funktionen auf geordneten numerischen Typen leisten. Als Null-Wert von `a` gilt dabei das `a`-Bild der ganzen Zahl 0.

```
auf_onfun :: (Ord a,Num a) => [a -> a] -> [a -> a]
ab_onfun  :: (Ord a,Num a) => [a -> a] -> [a -> a]
```

A.12 **Ohne Abgabe, ohne Beurteilung:**

(a) Überlegen Sie, wie Sie mithilfe der Abbildungsidee aus A.11 und der Differenz- und `signum`-Funktion aus `Num` die Funktionen aus A.11 sogar auf numerische Typen mit (lediglich einem) Gleichheitstest verallgemeinern können.

```
auf_nfun :: (Eq a,Num a) => [a -> a] -> [a -> a]
ab_nfun  :: (Eq a,Num a) => [a -> a] -> [a -> a]
```

(b) Überprüfen Sie, dass sich `auf_nfun` und `ab_nfun` für den Typ `Int` für `a` wie die Funktionen aus A.10 verhalten und für die Typen `Integer`, `Float` und `Double` typentsprechend analog.

(c) Nutzen Ihre Implementierungen von `auf_nfun` und `ab_nfun` Eigenschaften von `signum` aus, die für `Int`, `Integer`, `Float`, `Double` gegeben sind, die aber von `Num`-Instanzen nicht ausdrücklich verlangt sind und erwartet werden dürfen? Kennen Sie solche `Num`-Instanzen?

A.13 Wir führen eine einfache Datenbank für Personen über folgenden Datentypen ein:

```
type Nat1                = Int
type Name                = String
type Alter               = Nat1
data Geschlecht          = M | F | D deriving (Eq,Ord,Show)
type Gehalt              = Nat1
type PersNummer          = Int
data Hersteller          = Alcatel | Apple | Huawai | LG | Motorola
                           | Nokia | Samsung deriving (Eq,Ord,Show)
type Hat_Smartphone_von = Maybe Hersteller
data Person              = P PersNummer Name Alter Geschlecht Gehalt
                             Hat_Smartphone_von deriving (Eq,Show)
type Datenbank           = [Person]
type Nutzungssicht       = Datenbank
```

Von Fall zu Fall sind bestimmte Einträge in der Datenbank von größerem Interesse als andere. Deshalb soll es einfach möglich sein, die Datenbank für verschiedene Nutzungssichten zu sortieren. In jeder Nutzungssicht sollen die Einträge nach absteigender Relevanz angeordnet sein. Je wichtiger ein Eintrag für eine Nutzung ist, desto weiter vorne soll er angeordnet sein. Implementieren Sie alle Datenbanknutzungssichten wieder als Spezialisierungen von `gen_sort`:

```
normalsicht                    :: Datenbank -> Nutzungssicht
anlageberatungssicht           :: Datenbank -> Nutzungssicht
personalabteilungssicht        :: Datenbank -> Nutzungssicht
sozialforschungssicht          :: Datenbank -> Nutzungssicht
integritaetssicht              :: Datenbank -> Nutzungssicht
auch_im_chaos_ist_ordnung_sicht :: Datenbank -> Nutzungssicht
```

Im einzelnen soll gelten:

(a) *Normalsicht:* In der Normalsicht ist die Datenbank lexikographisch aufsteigend nach Namen sortiert. Einträge mit gleichem Namen können beliebig angeordnet sein.

(b) *Anlageberatungssicht:* Für die Anlageberatungssicht ist ein Eintrag um so wichtiger, je höher das Gehalt ist. Einträge mit gleichem Gehaltswert können beliebig angeordnet sein.

(c) *Personalabteilungssicht:* Zur Förderung von Diversität und Gleichstellung im Unternehmen vor allem bei jüngeren Mitarbeitern sind für die Personalabteilung Einträge mit Geschlechtswert `D` wichtiger als mit Wert `F` und diese wichtiger als mit Wert `M`. Unter Einträgen mit gleichem Geschlechtswert ist ein Eintrag um so wichtiger, je jünger die Person ist. Einträge mit übereinstimmenden Geschlechts- und Alterswerten können beliebig angeordnet sein.

(d) *Sozialforschungssicht:* Gibt es einen Zusammenhang zwischen Gehalt und bevorzugtem Smartphone-Hersteller? Um diese Frage einfacher beantworten zu können, werden die Einträge in der Sozialforschungssicht in 8 Gruppen angeordnet. Die ersten 7 Gruppen sind die Nutzer der verschiedenen Smartphone-Marken alphabetisch aufsteigend nach Herstellernamen, die 8. Gruppe sind die Personen

ohne Smartphone (Wert `Nothing`). In jeder der 8 Gruppen sind die Einträge absteigend nach Gehalt angeordnet. Einträge mit übereinstimmenden Smartphone- und Gehaltswerten können beliebig angeordnet sein.

(e) *Integritätssicht:* Offenbar ist die Datenbank inkonsistent, wenn es mehrere Einträge mit gleicher Personalnummer gibt. Um solche Inkonsistenzen schnell erkennen zu können, wird die Datenbank so sortiert, dass alle Einträge mit jeweils gleicher Personalnummer unmittelbar aufeinanderfolgen und eine Gruppe von Einträgen mit gleicher Personalnummer um so weiter vorne in der Gesamtliste steht, je mehr Einträge sie umfasst. Gleich lange Gruppen sind dabei aufsteigend nach der Personalnummer angeordnet. Innerhalb der Gruppen ist die Anordnung beliebig.

(f) *Chaossicht:* Auch im Chaos liegt (manchmal) Ordnung! In der Chaossicht gilt, dass ein Eintrag kleiner einem anderen ist (und deshalb weiter vorne steht), wenn nach alphabetisch aufsteigender Sortierung (gemäß vordef. Ordnung 'kleiner' auf `Char`) der Zeichen der Namenswerte zweier Datenbankeinträge das Kopfzeichen des sortierten Namenswerts des ersten Eintrags alphabetisch kleiner als das entsprechende Zeichen des anderen Eintrags ist. Einträge, die in diesem Sinn gleich sind, können beliebig angeordnet sein. Alle Namen in der Datenbank dürfen für die Chaossortierung ungleich des leeren Worts vorausgesetzt werden.

A.14 **Ohne Beurteilung:** Beschreiben Sie für jede Rechenvorschrift in einem Kommentar knapp, aber gut nachvollziehbar, wie die Rechenvorschrift vorgeht.

A.15 **Ohne Abgabe, ohne Beurteilung:** Testen Sie alle Funktionen umfassend, auch mit eigenen Testdaten.

B Papier- und Bleistiftaufgaben

Entfällt auf den Angaben 5 bis 7.

Iucundi acti labores.
Getane Arbeiten sind angenehm.

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

C Terminhinweise

1. **Nächster Vorlesungstermin: Mittwoch, 02.12.2020**, 08:15-09:45 Uhr, Echtzeitvideokonferenz unter der bekannten Zoom-URL (siehe TISS):
   - 08:15-09:00 Uhr: Vorlesungsteil VI
   - 09:15-09:45 Uhr: Umgekehrtes Klassenzimmer zu Vorlesungsteil V
2. **Nächste Kleinübungsgruppentermine:** Die Kleinübungsgruppentreffen finden wie geplant statt, ausschließlich *online* als Echtzeitvideokonferenzen.
   - **Zoom-URLs für alle KÜGs: Siehe Tuwel!**
   - KÜG 1&7: Di, 24.11.2020, 16:00-17:00 Uhr, online via Zoom
   - KÜG 2&8: Do, 26.11.2020, 08:00-9:00 Uhr, online via Zoom
   - KÜG 3&9: Do, 26.11.2020, 11:00-12:00 Uhr, online via Zoom
   - KÜG 4&10: Do, 26.11.2020, 14:00-15:00 Uhr, online via Zoom
   - *Fr, 27.11.2020, 08:00-9:00 Uhr: Termin entfällt!*
   - KÜG 5&6, KÜG 11&12: Fr, 27.11.2020, 11:00-12:00 Uhr, online via Zoom
   - KÜG 13: Mi, 25.11.2020, 15:00-16:00 Uhr, online via Zoom
   - KÜG 14: Fr, 27.11.2020, 15:00-16:00 Uhr, online via Zoom
   - *Fr, 27.11.2020, 17:00-18:00 Uhr: Termin entfällt!*
3. **Nächste Plenumsübungsgruppentermine:** Beide Plenumsübungsgruppentreffen finden wie geplant statt, ausschließlich *online* als Echtzeitvideokonferenzen.
   - **Zoom-URLs für beide PÜGs: Siehe Tuwel!**
   - PÜG I: Di, 24.11.2020, 09:00-10:00 Uhr, online via Zoom
   - PÜG II: Mi, 25.11.2020, 16:00-17:00 Uhr, online via Zoom