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

4. Aufgabenblatt zu Funktionale Programmierung von Fr, 06.11.2020.
Erstabgabe: Fr, 13.11.2020 (12:00 Uhr)

(Beurteilt: Teil A; ohne Abgabe und Beurteilung: Teil B)

Themen: *Funktionen auf monomorphen und polymorphen algebraischen Datentypen, Typklassen, Instanzbildungen, 'literate' Haskell-Skripte (Kap. 1 bis Kap. 9, Kap. 11)*

- **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, 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

   `Angabe4.lhs`

   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 Angabe4.lhs`.

   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 Angabe4 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 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. 50 Punkte)

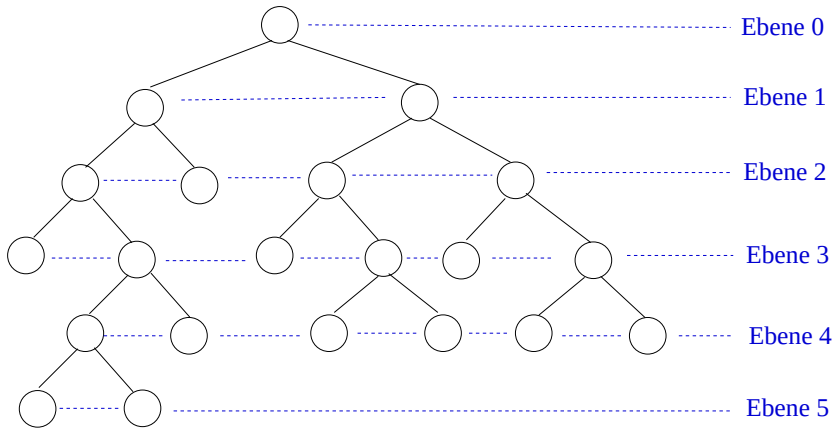
Erweitern Sie zur Lösung der programmiertechnischen Aufgaben die Rahmendatei `Angabe4.lhs`. Kommentieren Sie die Rechenvorschriften in Ihrem Programm zweckmäßig, aussagekräftig und problemangemessen. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten. Versehen Sie alle Funktionen, die Sie zur Lösung der Aufgaben benötigen, mit ihren Typdeklarationen, d.h. geben Sie stets deren syntaktische Signatur (kurz: Signatur), explizit an.

A.1 Die Wurzel eines Baums liegt auf Ebene 0. Knoten oder Blätter eines Baums, die unmittelbar von der Wurzel aus erreicht werden können, liegen auf Ebene 1 usw. Gesucht ist eine Funktion `breitest`, die angewendet auf einen Baum, diejenige oder diejenigen Ebenen bestimmt, auf denen der Baum am breitesten ist, d.h. auf der mindestens so viele Knoten oder Blätter liegen wie auf jeder anderen Ebene des Baums. Dazu betrachten wir folgende Typen, insbesondere den monomorphen Binärbaumtyp `BBaum`:

```
> type Nat0   = Int
> type Ebene  = Nat0
> type Breite = Nat0
> type Text   = String
> data BBaum  = Blatt Text
>             | Knoten Text BBaum BBaum deriving (Eq,Show)
> data Auswertung = Ausw Breite [Ebene] deriving (Eq,Show)

> breitest :: BBaum -> Auswertung
```

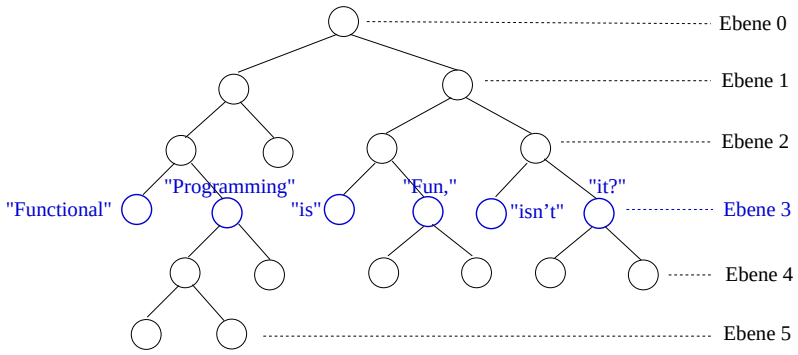
Angewendet auf einen Binärbaumwert liefert `breitest` die größte Breite des Argumentbaums und in aufsteigender Ordnung die Ebenen, in denen der Argumentbaum diese größte Breite annimmt. Im Fall des nachstehend skizzierten Binärbaums liefert `breitest` also den Auswertungswert `Ausw 6 [3,4]`.



A.2 Angewendet auf einen Binärbaumwert b und einen Ebenenwert e liefert die Funktion `tae` (kurz für 'text_auf_ebene') die an den Knoten oder Blättern auf Ebene e stehenden Texte. Dabei steht der Text eines Knotens oder Blatts um so weiter vorne in der Resultatliste, je weiter links ein Knoten oder Blatt im Baum angeordnet ist. Ist die Menge der Knoten und Blätter auf Ebene e leer, so liefert die Funktion den Wert `Nothing`.

```
> tae :: BBaum -> Ebene -> Maybe [Text]
```

Angewendet auf den nachfolgend skizzierten Binärbaumwert und Ebenenwert 3 liefert `tae` das Resultat `Just ["Functional","Programming","is","Fun,","isn't","it?"]`, angewendet auf Ebenenwert 10 den Wert `Nothing`:

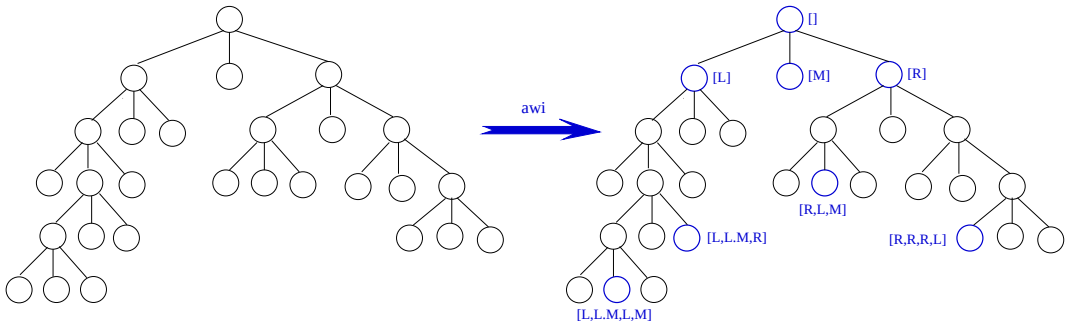


A.3 Wir betrachten zwei monomorphe Trinärbäumvarianten `TBaum` und `TBaum'`. Gesucht ist eine Haskell-Rechenvorschrift `awi` (kurz für 'annotiere_weg_information'), die `TBaum`-Werte in strukturidente `TBaum'`-Werte überführt, wobei jedes Blatt und jeder Knoten im `TBaum'`-Wert mit der Weginformation benannt ist, auf dem ein Blatt oder Knoten von der Wurzel des Baums aus erreicht wird. Die Weginformation ist dabei als Liste der Richtungen gegeben, ob auf dem Weg jeweils dem linken (`L`), mittleren (`M`) oder rechten (`R`) Teilbaum zu folgen ist, um schließlich das entsprechende Blatt oder den Knoten zu erreichen. Für Baumwurzeln ist die Weginformation (deshalb) leer.

```
> data TBaum    = TB
>               | TK TBaum TBaum TBaum deriving (Eq,Show)
> data Richtung = L | M | R deriving (Eq,Show)
> type Weg      = [Richtung]
> data TBaum'   = TB' Weg
>               | TK' Weg TBaum' TBaum' TBaum' deriving (Eq,Show)

> awi :: TBaum -> TBaum'
```

Die folgende Abbildung skizziert die Transformationsidee für ausgewählte blau hervorgehobene Knoten und Blätter:



A.4 Wir führen einen polymorphen Baumtyp über einem polymorphen Typsynonym ein:

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

Machen Sie den Datentyp `Baum a` unter der Annahme, dass `a` Instanz der Typklasse `Eq`, `Ord` bzw. `Show` ist, zu einer Instanz der Typklassen `Eq`, `Ord` und `Show`:

(a) `> instance Show a => Show (Baum a) where...`

(b) `> instance Eq a => Eq (Baum a) where...`

(c) `> instance Ord a => Ord (Baum a) where...`

Die Instanzbildungen sollen dabei so vorgenommen werden, dass gilt:

- Zwei Bäume sind gleich gdw. die Bäume sind struktur- und benennungsident, ungleich sonst.
- Ein Baum t ist echt kleiner als ein Baum t' gdw. t ist ein strukturidenter echter Anfangsbaum von t' und die Information an jedem Knoten oder Blatt von t ist eine echte Anfangsliste derjenigen am entsprechenden Blatt bzw. Knoten von t'.
- Ein Baum t ist echt größer als ein Baum t' gdw. t' ist ein strukturidenter echter Anfangsbaum von t und die Information an jedem Knoten oder Blatt von t' ist eine echte Anfangsliste derjenigen am entsprechenden Blatt bzw. Knoten von t.
- Ein Baum t ist größergleich (kleinergleich) als ein Baum t' gdw. t ist größer oder gleich (kleiner oder gleich) t'.
- Baumwerte werden in spitze Klammern eingeschlossen. Die folgenden Beispiele illustrieren die gewünschte Ausgabedarstellung:

```
show (B []) ->> "<[]>"
show (B [1,2,3]) ->> "<[1,2,3]>"
show (B [True,False,True]) ->> "<[True,False,True]>"
show (B ["Functional","Programming","is","Fun"])
         ->> "<["Functional","Programming","is","Fun"]>"

show (K (B []) [1,2,3] (B [4,5,6]))
  ->> "<Wurzel [1,2,3] <[]> <[4,5,6]>>"
show (K (K (B []) [1,2,3] (B [4,5,6])) [42] (B [7,8,9]))
 ->> "<Wurzel [42] <Wurzel [1,2,3] <[]> <[4,5,6]>> <[7,8,9]>>"
```

- Für nicht angegebene Operatoren/Relatoren aus `Ord` ist kein besonderes Verhalten gefordert; es kann beliebig implementiert sein. Nutzen Sie bei den Instanzbildungen die Protoimplementierugen bestmöglich aus; implementieren Sie nur explizit, was unbedingt nötig ist, um das geforderte Verhalten sicherzustellen.

A.5 Kopieren Sie die Deklaration des Typs `Intervall` von Angabe 3 in Ihre Abgabedatei zusammen mit Ihrer Instanzdeklaration von Intervall für die Typklasse `Show`:

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

> instance Show Intervall where...
```

Testen Sie die Ausgabe von mit Intervalllisten benannten Baumwerten. Wie werden etwa folgende Baumwerte ausgegeben? Überlegen Sie sich zuerst, wie die Ausgabe aussehen sollte, bevor Sie Ihre Vermutung anschließend mit dem Interpretierer überprüfen!

```
show (B [IV (2,5)]) ->> ???
show (B [IV (5,2)]) ->> ???
show (B [Leer]) ->> ???
show (B [IV (2,5),IV (5,2),Leer,Ungueltig]) ->> ???
show (B []) ->> ???

show (K (B [IV (2,2)]) [IV (2,3)] (B [Leer])) ->> ???
show (K (K (B [IV (2,5),IV (5,2)]) [Leer,IV (5,2)] (B [])) (B [Leer])) ->> ???
```

A.6 **Ohne Abgabe, ohne Beurteilung:**

(a) Kopieren Sie auch Ihre `Intervall`-Instanzdeklarationen für die Klassen `Eq` und `Num` in die (Abgabe-) Datei. Welche Ausgabe erzeugt folgender Aufruf?

```
show (B [IV (2,3) + IV (5,7),IV (2,5) * IV (5,5),IV (5,5) - Leer]) ->> ???
```

Überlegen Sie sich weitere Testbeispiele. Überprüfen Sie anschließend, ob Ihre Ausgabevermutung für alle Ihre Beispiele richtig ist.

(b) Wie ändert sich die Ausgabe, wenn Sie Ihre `Intervall`-Instanzdeklaration für `Show` durch eine `deriving`-Klausel ersetzen? Überprüfen Sie Ihre Vermutung!

A.7 **Ohne Beurteilung:** Beschreiben Sie für jede Rechenvorschrift in einem Kommentar knapp, aber gut nachvollziehbar, wie die Rechenvorschrift vorgeht.

A.8 **Ohne Abgabe, ohne Beurteilung:** Testen Sie alle Funktionen umfassend, auch mit eigenen Testdaten.

B Papier- und Bleistiftaufgaben (ohne Abgabe/Beurteilung)

B.1 Um Speicherplatz zu sparen, werden Zeichenfolgen oft in komprimierter Form abgelegt, aus der bei Bedarf wieder die originale Zeichenfolge gewonnen werden kann.

Eine einfache Komprimierungsmethode macht sich Folgen wiederholender Zeichen in einer Zeichenfolge zunutze, sog. *Läufe*. Das folgende Beispiel verdeutlicht die Komprimierungsidee. Die Zeichenfolge

```
"aaaaBBBaaBBBBBBccccccccccccAbCbDDDDDDDDDDDDDDDDDDD"
```

wird kodiert als Folge von Paaren aus einer Ziffernfolge und einem von einer Ziffer verschiedenen Zeichen; der Dezimalzahlwert jeder Ziffernfolge gibt dabei die Länge des Laufs des auf diese Ziffernfolge folgenden Zeichens an. Nach dieser Methode hat die obige Zeichenfolge die Komprimierung:

```
"4a3B2a6B12c1A1b1C1b18D"
```

B.1.1 Geben Sie die syntaktischen Signaturen zweier Haskell-Rechenvorschriften `komprimiere`, `dekomprimiere` an, die – vollständig implementiert – die obige Komprimierungsidee umsetzten. Benutzen Sie, wenn möglich, Typsynonyme, um die Funktionssignaturen möglichst 'sprechend' zu machen.

B.1.2 Intuitiv sollen die Rechenvorschriften `komprimiere`, `dekomprimiere` invers zueinander sein, d.h. folgende Gleichungen erfüllen:

```
dekomprimiere (komprimiere s) == s    (a)
komprimiere (dekomprimiere t) == t    (b)
```

i. Können `komprimiere` und `dekomprimiere` so implementiert werden, dass die Gleichungen (a) und (b) stets erfüllt sind?
ii. Welche Rand- oder Sonderfälle sind ggf. zu beachten? Wie sollen sie behandelt werden?
iii. Beschreiben Sie die Bedeutung von `komprimiere` und `dekomprimiere` so genau und präzise, dass für möglichst viele Argumentwerte von `komprimiere` und `dekomprimiere` die Gleichungen (a) und (b) erfüllt sind und für etwaig verbleibende für beide Funktionen ein Funktionsergebnis in eindeutiger Weise festgelegt ist.
iv. Überlegen Sie sich aussagekräftige Testfälle, die das gewünschte Verhalten von `komprimiere` und `dekomprimiere` zeigen (und nach Implementierung zu überprüfen erlaubten).
v. Wenn Sie möchten, vervollständigen Sie die Implementierung von `komprimiere` und `dekomprimiere` aus B.1.1 und testen Sie (z.B mit Hugs), ob Ihre Implementierung der beiden Funktionen Ihre Anforderungen aus (ii), (iii) und (iv) erfüllen oder/und ob Sie noch weiteres möglicherweise überraschendes Verhalten aufdecken.

Iucundi acti labores.
Getane Arbeiten sind angenehm.

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

C Terminhinweise

1. **Nächster Vorlesungstermin: Mittwoch, 18.11.2020**, 08:15-09:45 Uhr, Echtzeitvideokonferenz unter der bekannten Zoom-URL (siehe TISS):
   – 08:15-09:00 Uhr: Vorlesungsteil V
   – 09:15-09:45 Uhr: Umgek. Klassenzimmer zu Vorlesungsteil IV
2. **Nächste Kleinübungsgruppentermine:** Alle Kleinübungsgruppentreffen finden wie geplant statt, ausschließlich *online* als Echtzeitvideokonferenzen.
   – **Zoom-URLs für alle KÜGs: Siehe Tuwel!**
   – KÜG 1, KÜG 7: Di, 10.11.2020, 16:00-17:00 Uhr, *online* via Zoom
   – KÜG 2, KÜG 8: Do, 12.11.2020, 08:00-9:00 Uhr, *online* via Zoom
   – KÜG 3, KÜG 9: Do, 12.11.2020, 11:00-12:00 Uhr, *online* via Zoom
   – KÜG 4&10: Do, 12.11.2020, 14:00-15:00 Uhr, *online* via Zoom
   – *Fr, 13.11.2020, 08:00-9:00 Uhr: Termin entfällt!*
   – KÜG 5&6, KÜG 11&12: Fr, 13.11.2020, 11:00-12:00 Uhr, *online* via Zoom
   – KÜG 13: Mi, 11.11.2020, 15:00-16:00 Uhr, *online* via Zoom
   – KÜG 14: Fr, 13.11.2020, 15:00-16:00 Uhr, *online* via Zoom
   – KÜG 15: Fr, 13.11.2020, 17:00-18:00 Uhr, *online* via Zoom
3. **Nächste Plenumsübungsgruppentermine:** Beide Plenumsübungsgruppentreffen finden wie geplant statt, ausschließlich *online* als Echtzeitvideokonferenzen.
   – **Zoom-URLs für beide PÜGs: Siehe Tuwel!**
   – PÜG I: Di, 10.11.2020, 09:00-10:00 Uhr, *online* via Zoom
   – PÜG II: Mi, 11.11.2020, 16:00-17:00 Uhr, *online* via Zoom