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. Angabe zu Funktionale Programmierung von Fr, 05.11.2021.

Erstabgabe: Fr, 12.11.2021, 12:00 Uhr

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

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

Themen: *Literate Haskell-Skripte, Feldsyntax algebraischer und neuer Datentypen, Rechnen mit Listen und Tupeln, musterbasierte Fallunterscheidungen in Funktionsdefinitionen, Umformen und umrechnen von Werten verschiedener Typen*

Stoffumfang: *Kapitel 1 bis Kapitel 9, besonders Kapitel 2 bis Kapitel 6.*

- **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.

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

   `Angabe4.lhs`

   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 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 für das Testsystem erforderliche 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.
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 `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; geben Sie also stets deren syntaktische Signatur (kurz: Signatur), explizit an.

Wir betrachten eine einfache Buchhaltungs- und Auswertungssoftware:

```
> type Nat1    = Int
> type Name    = String
> newtype Cent = C { cents :: Nat1
>                  } deriving (Eq,Ord,Show)
> type Brutto  = Cent
> type Netto   = Cent
> data Skonto  = KeinSkonto | DreiProzent
>                | FuenfProzent | ZehnProzent deriving (Eq,Ord,Show)
> data Tag     = I | II | III | IV | V | VI | VII | VIII | IX | X
>                | XI | XII | XIII | XIV | XV | XVI | XVII | XVIII
>                | XIX | XX | XXI | XXII | XXIII | XXIV | XXV
>                | XXVI | XXVII | XXVIII | XXIX | XXX
>                | XXXI deriving (Eq,Ord,Show)
> data Monat   = Jan | Feb | Mar | Apr | Mai | Jun | Jul | Aug
>                | Sep | Okt | Nov | Dez deriving (Eq,Ord,Show)
> type Jahr    = Nat1
> data Datum   = D { tag   :: Tag,
>                    monat :: Monat,
>                    jahr  :: Jahr
>                  } deriving (Eq,Show)

> data Geschaeftspartner = GP { partner :: Name,
>                               seit :: Datum
>                             } deriving (Eq,Show)
> data Geschaeftsvorfall = Zahlung { brutto :: Brutto,
>                                    skonto :: Skonto,
>                                    zahlung_vom :: Datum
>                                  }
>                        | Gutschrift { gutschriftsbetrag :: Cent,
>                                       gutschrift_vom :: Datum
>                                     }
>                        deriving (Eq,Show)
> type Kassabucheintrag = (Geschaeftspartner,Geschaeftsvorfall)
> newtype Kassabuch = KB [Kassabucheintrag] deriving (Eq,Show)
```

A.1 Für einen besseren Überblick sollen Kassabucheinträge ausgewertet und plausibilisiert werden. Das Ergebnis sind konsolidierte Einträge, in denen Bruttobeträge und Skontosätze von Zahlungen durch die entsprechenden Nettobeträge ersetzt (Nettobetrag = Bruttobetrag abzüglich des auf volle Cent abgerundeten Skontoabzugs entsprechend des gewährten Skontorabatts) und offensichtlich fehlerhafte Datumsein-

träge zum Beginn von Geschäftsbeziehungen und der Abwicklung von Geschäftsvorfällen durch das jeweils nächstgültige Datum ersetzt sind (z.B. der 31.11. eines Jahres durch den 1.12. desselben Jahres, der 29.02., 30.02. oder 31.02. eines Nichtschaltjahres durch den 1.3. desselben Jahres, etc.; Schaltjahre sind dabei alle durch 4 teilbaren Jahre, es sei denn sie sind durch 100, aber nicht gleichzeitig durch 400 teilbar).

Schreiben Sie eine Haskell-Rechenvorschrift `waup` (für "werte aus und plausibilisiere"), die Kassabucheinträge in eine ausgewertete (A) und plausibilisierte (P) Darstellung überführt:

```
> type P_Geschaeftspartner = Geschaeftspartner
> data AP_Geschaeftsvorfall
>    = AP_Zahlung { netto :: Netto,
>                   zahlungsdatum :: Datum
>                 }
>     | P_Gutschrift { gutschrift :: Cent,
>                      gutschriftsdatum :: Datum
>                    } deriving (Eq,Show)
> type AP_Kassabucheintrag = (P_Geschaeftspartner,AP_Geschaeftsvorfall)

> waup :: Kassabucheintrag -> AP_Kassabucheintrag
```

A.2 Im nächsten Schritt sollen vollständige Kassabücher konsolidiert werden. Zusätzlich zur Auswertung und Plausibilisierung von Kassabucheinträgen wie in A.1, sollen dabei Zahlungs- und Gutschriftsbeträge in üblicher Weise in Euro und Cent ausgewiesen werden; für Centbeträge sind deshalb konsolidiert nur Werte zwischen 0 und 99 möglich.

Schreiben Sie eine Haskell-Rechenvorschrift `konsolidiere`, die diese Konsolidierung (K) von Kassabüchern leistet. Bei der Überführung wird jeder Kassabucheintrag wie beschrieben konsolidiert; die relative Reihenfolge der Einträge bleibt unverändert:

```
> data EuroCent = EC { euro :: Nat1,
>                      cent :: Nat1

                       Nur Werte zwischen 0 und 99 für cent!

>                    } deriving (Eq,Ord,Show)

> data K_Geschaeftsvorfall = K_Zahlung { ec_netto :: EuroCent,
>                                        zahlungsdatum' :: Datum
>                                      }
>                          | K_Gutschrift { ec_gutschrift :: EuroCent,
>                                           gutschriftsdatum' :: Datum
>                                         } deriving (Eq,Show)

> newtype KonsolidiertesKassabuch
>     = KKB [(P_Geschaeftspartner,K_Geschaeftsvorfall)]
>       deriving (Eq,Show)

> konsolidiere :: Kassabuch -> KonsolidiertesKassabuch
```

A.3 Als nächstes soll eine Auswertungsfunktion `saldo` geschrieben werden. Für gegebenen Geschäftspartner und konsolidiertes Kassabuch liefert `saldo` für diesen Geschäftspartner den Forderungssaldo (= Summe der Gutschriften abzüglich Summe der Zahlungen) ausweist, wenn die Summe der Gutschriften die der Zahlungen echt übersteigt; den Zahlungssaldo (= Summe der Zahlungen abzüglich Summe der Gutschriften), wenn die Summe der Zahlungen die der Gutschriften echt übersteigt; den Wert `Ausgeglichen`, wenn beide Summen ident sind; den Wert `Keine_Geschaeftsbeziehung`, wenn keine Geschäftsbeziehung besteht, wenn also weder Zahlungen an noch Gutschriften vom Geschäftspartner im Kassabuch verbucht sind. Geschäftspartner mit gleichem Namen, aber unterschiedlichem Geschäftsbeziehungsbeginn sind verschieden. Für den Aufruf von `saldo` können Sie davon ausgehen, dass alle Datumswerte im Geschäftspartner- und konsolidiertem Kassabuchargument plausibilisiert und gültig sind!

```
> data Saldo = Forderungssaldo { fs :: EuroCent }
>            | Zahlungssaldo { zs :: EuroCent }
>            | Ausgeglichen
>            | Keine_Geschaeftsbeziehung deriving (Eq,Show)

> saldo :: P_Geschaeftspartner -> KonsolidiertesKassabuch -> Saldo
```

A.4 Zum Abschluss soll eine Haskell-Rechenvorschrift `saldiere` geschrieben werden, die die in einem Kassabuch verbuchten Geschäftsvorfälle (unabhängig von der Gültigkeit eines Geschäftsdatums) geschäftspartnerweise zusammenfasst. Dabei sollen die Einträge in saldierten Kassabüchern lexikographisch aufsteigend (gegeben durch die Ordnung `(<)` auf Zeichenreihenwerten) nach Geschäftspartnernamen angeordnet sein, Geschäftspartner gleichen Namens mit längerer Geschäftsbeziehung vor denen mit kürzerer. Als Beginn der Geschäftsbeziehung gilt dabei stets das plausibilisierte Datum wie in A.1 festgelegt.

```
> newtype SaldiertesKassabuch = SKB [(Geschaeftspartner,Saldo)]
>                               deriving (Eq,Show)

> saldiere :: Kassabuch -> SaldiertesKassabuch
```

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- und Bleistiftaufgaben (ohne Abgabe/Beurteilung)

B.1 Von welchem Rekursionstyp sind die Funktionen, die sie für Angabe 4 rekursiv implementiert haben? Woran haben Sie die Rekursionstypen jeweils erkannt? Welche Rekursionstypen aus Kapitel 7.2 sind nicht vorgekommen (falls es solche gibt)?

B.2 Geben Sie die Aufrufgraphen der Funktionen an, die Sie für die Lösung von Angabe 4 geschrieben haben (s. Kapitel 7.3).

B.3 Haben Sie einige Aufgaben von diesem Aufgabenblatt mit oder mit Mitverwendung von Listenkomprehensionen gelöst? Wenn ja, wie hätten Sie diese Aufgaben auch ohne Mitverwendung von Listenkomprehensionen lösen können?

B.4 Schreiben Sie eine Haskell-Rechenvorschrift `komprimiere'`, die wie `komprimiere` ein komprimiertes Kassenbuch liefert, aber von konsolidierten statt nichtkonsolidierten Kassabüchern ausgeht:

```
> saldiere' :: KonsolidiertesKassabuch -> SaldiertesKassabuch
```

B.5 Überprüfen Sie, ob die Funktionen `saldiere` und `(saldiere' . konsolidiere)` angewendet auf ein Kassabuchargument stets dasselbe Ergebnis liefern (der . in `(saldiere' . konsolidiere)` steht dabei für die sequentielle Komposition von Funktionen) (siehe Kapitel 10.4.2, Methode 6).

Iucundi acti labores.
Getane Arbeiten sind angenehm.

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