universität innsbruck

Funktionale Programmierung	WS 2026/2027	LVA 703206/703025
Übungsblatt 5, 10 Punkte	Abgabefrist: Dienstag, 24. November 2026, 20:00 Uhr	

- Kreuzen Sie gelöste Aufgaben im OLAT Kurs des Proseminars an.
- Lösen Sie Programmieraufgaben in Template_05.hs und laden Sie diese Datei in OLAT hoch.
- Ihre Template-Datei sollte mit `ghci` ohne Fehlermeldung kompilieren.

Aufgabe 1 *Fakultät und Eulersche Zahl* **5 P.**

In dieser Aufgabe betrachten wir die Fakultätsfunktion wie in der Vorlesung definiert. Mithilfe von Fakultäten ist es möglich, die Eulersche Zahl $e \approx 2.718$ durch die folgende unendliche Summe, die sogenannte Taylorreihe, zu berechnen:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

Bei Verwendung einer endlichen Summe können wir die Annäherung von e rekursiv definieren, wobei e_n mit $n \in \mathbb{N}$ die Approximation mit der oberen Schranke n darstellt:

$$e_n = \begin{cases} 1 & \text{falls } n = 0 \\ \frac{1}{n!} + e_{n-1} & \text{sonst} \end{cases}$$

1. Schreiben Sie eine Funktion `euler :: Integer -> Double`, die eine Zahl `n` als Eingabe nimmt und e_n berechnet. Werfen Sie bei negativen Eingaben einen Fehler mit einer geeigneten Fehlermeldung.

   Tipp: Die Funktion `fromIntegral` wandelt einen `Integer` in einen `Double` um. (1 Punkt)

2. In der Vorlage finden Sie eine Funktion `factorialListNaive :: Integer -> [Integer]`, die bei Übergabe von `n` als Eingabe die Liste `[factorial n, ..., factorial 0]` zurückgibt. Diese ist nicht sehr effizient, da keine Zwischenwerte wiederverwendet werden. Implementieren Sie eine effizientere Funktion `factorialList :: Integer -> [Integer]`, die dasselbe Ergebnis wie `factorialListNaive` liefert, aber keine Zwischenwerte doppelt berechnet. Werfen Sie bei negativen Eingaben einen Fehler mit einer geeigneten Fehlermeldung. (1.5 Punkte)

3. Verwenden Sie die Funktion `factorialList` (oder `factorialListNaive`, falls Sie diese nicht implementiert haben) und vervollständigen Sie die Funktion `eulerListAux :: [Integer] -> Double`, die als Eingabe die Liste der Fakultäten erhält und die Eulersche Zahl unter ausschließlicher Verwendung der Fakultäten aus der Liste berechnet.

   Tipp: Berücksichtigen Sie, in welcher Reihenfolge die Fakultäten in der Liste vorkommen. (1 Punkt)

4. Mathematisch konvergiert die Taylorreihe gegen e, erreicht diesen Wert jedoch erst für $\lim_{n\to\infty}$. Aufgrund der endlichen Präzision des `Double`-Typs werden Sie bei dieser Berechnung jedoch feststellen, dass $e_{n+1} == e_n$ ab einem bestimmten Punkt gilt. Schreiben Sie eine Funktion `eulerConvergence :: [Double]`, die die Zahlenfolge $[e_0, ..., e_n]$ ausgibt, wobei n die kleinste Zahl ist, für die $e_n == e_{n+1}$ gilt. Ermitteln Sie n **nicht** manuell und hardcodieren Sie diesen Wert nicht in der Funktion. (1.5 Punkte)

Aufgabe 2 *Exaktes Rechnen mit Brüchen* **5 P.**

In Aufgabe 1 wurde e_n mithilfe von `Double` berechnet. In dieser Aufgabe arbeiten wir mit exakteren Brüchen. Verwenden Sie in dieser Aufgabe **nicht** die vordefinierte Funktion `gcd` und **nicht** den Typ `Rational` (bzw. den Operator `%` aus `Data.Ratio`).

1. Einen Bruch $\frac{a}{b}$ stellen wir als Paar `(a, b)` ganzer Zahlen dar. (2 Punkte)

   Definieren Sie ein Typsynonym `Fraction` für solche Paare von `Integer`-Werten.

   Implementieren Sie außerdem mithilfe von Guarded Equations eine Funktion
   `ggT :: Integer -> Integer -> Integer`,
   die den größten gemeinsamen Teiler zweier Zahlen mit dem euklidischen Algorithmus berechnet:

   $$\mathrm{ggT}(a,b) = \begin{cases} |a| & \text{falls } b = 0 \\ \mathrm{ggT}(b,\ a \bmod b) & \text{sonst} \end{cases}$$

   Hinweis: Bei der ganzzahligen Division von `a` durch `b`:
   `a `div` b` berechnet das ganzzahlige Resultat, und `a `mod` b` berechnet den Rest.

2. Ein Bruch heißt hier normalisiert, wenn $b > 0$ gilt und der Zähler a und Nenner b keinen gemeinsamen Teiler (> 1) haben. Implementieren Sie eine Funktion `normalize :: Fraction -> Maybe Fraction`, die einen Bruch normalisiert. Ist der Nenner 0, soll `Nothing` zurückgegeben werden. Verwenden Sie für Zwischenergebnisse lokale Definitionen mit `where`. (1 Punkt)

   Beispiele:

   - `normalize (6, -4) = Just (-3, 2)`
   - `normalize (0, 5) = Just (0, 1)`
   - `normalize (3, 0) = Nothing`

3. Implementieren Sie die Funktion `addFrac :: Fraction -> Fraction -> Maybe Fraction`, die zwei Brüche addiert. Die Eingaben müssen nicht normalisiert sein, aber das Ergebnis soll normalisiert sein. Ist eine der Eingaben ungültig (d.h. Nenner 0), soll `Nothing` zurückgegeben werden. Verwenden Sie dafür einen `case`-Ausdruck. (1 Punkt)

   Beispiele:

   - `addFrac (1, 6) (1, 3) = Just (1, 2)`
   - `addFrac (1, 0) (1, 2) = Nothing`
   - `addFrac (1, 2) (-1, 2) = Just (0, 1)`

4. Implementieren Sie analog eine Funktion `divFrac :: Fraction -> Fraction -> Maybe Fraction`, die den ersten Bruch durch den zweiten dividiert. Auch hier soll das Ergebnis normalisiert sein. Ist eine der Eingaben ungültig oder wird durch 0 dividiert, soll `Nothing` zurückgegeben werden.

   Verwenden Sie wieder einen `case`-Ausdruck. (1 Punkt)

   Beispiele:

   - `divFrac (1, 2) (3, 4) = Just (2, 3)`
   - `divFrac (2, 1) (2, -4) = Just (-4, 1)`
   - `divFrac (1, 2) (0, 7) = Nothing`

5. **Bonus:** Mit `addFrac` und `divFrac` lässt sich die Approximation e_n aus Aufgabe 1 exakt berechnen, ohne Fakultäten zu benötigen. Bezeichnet $t_i = \frac{1}{i!}$ den i-ten Summanden, so gilt:

   $$t_i = \begin{cases} 1 & \text{falls } i = 0 \\ \frac{t_{i-1}}{i} & \text{sonst} \end{cases} \qquad e_i = \begin{cases} 1 & \text{falls } i = 0 \\ e_{i-1} + t_i & \text{sonst} \end{cases}$$

```haskell
eulerFrac :: Integer -> Maybe Fraction
eulerFrac n
  | n < 0 = Nothing
  | otherwise = eulerFracAux 1 (1, 1) (1, 1)
  where
    -- i: aktueller Index, t: 1/(i-1)! als Bruch, acc: e_(i-1) als Bruch
    eulerFracAux i t acc
      | i > n = Just acc
      | otherwise = case divFrac t (i, 1) of
          Nothing -> Nothing
          Just t' -> case addFrac acc t' of
            Nothing -> Nothing
            Just s -> eulerFracAux (i + 1) t' s
```

(a) Geben Sie der Reihe nach alle Aufrufe von `eulerFracAux` an, die bei der Auswertung von `eulerFrac 3` entstehen, einschließlich des letzten Aufrufs, der das Ergebnis liefert. Geben Sie dabei jeweils die konkreten Werte aller drei Argumente an.

Der erste Aufruf lautet:

```haskell
eulerFracAux 1 (1, 1) (1, 1)
```

Erklären Sie anschließend, welche der oben definierten Werte t_j und e_j die Argumente `t` and `acc` in einem Aufruf mit Index `i` enthalten.

(b) Oben finden Sie die Funktion `fracToDouble :: Fraction -> Double`, die einen Bruch in einen `Double` umwandelt. Für das Ergebnis von `eulerFrac 20` liefert sie `2.718281828459045`.[1]

Berechnet man e_{20} hingegen direkt mit `Double` gemäß der rekursiven Definition aus Aufgabe 1, also

```haskell
euler n = 1 / fromIntegral (factorial n) + euler (n-1)
```

erhält man `2.7182818284590455`.

Erklären Sie, warum zwei Berechnungen derselben Summe unterschiedliche Ergebnisse liefern können.

Hinweis: Je nach Reihenfolge der Additionen in Ihrer eigenen Funktion `euler` kann Ihr Ergebnis auch mit dem ersten Wert übereinstimmen.

[1] Die letzten Stellen können bei Ihrer eigenen Funktion `euler` abweichen, je nachdem, in welcher Reihenfolge Sie die Summanden addieren (in seltenen Fällen auch je nach Rechnerarchitektur).