

- Kreuzen Sie gelöste Aufgaben im OLAT Kurs des Proseminars an.
- Lösen Sie Programmieraufgaben in `Template_03.hs` und laden Sie diese Datei in OLAT hoch.
- Ihre Template-Datei sollte mit `ghci` ohne Fehlermeldung kompilieren.

Aufgabe 1 *Pattern Matching***2 P.**

Wir betrachten die folgenden Datentypen:

```
data Ancestry = Human | Dwarf | Elf | Orc | Goblin | Halfling
data SpellTradition = Arcane | Divine | Primal | Occult
data Class =
  Fighter
  | Rogue
  | Wizard SpellTradition
  | Sorcerer SpellTradition
data Character = Character
  String -- name
  Integer -- level
  Ancestry
  Class
```

Entscheiden Sie, welche der Ausdrücke 1.-4. die Patterns in (i) und (ii) matchen. Geben Sie für jeden erfolgreichen Match die entsprechende Substitution an. (2 Punkte)

1. `Character "Bilbo Baggins" 3 Halfling Rogue`
2. `Character "Gideon" 5 Human (Sorcerer Occult)`
3. `Character "Gwen" 12 Elf (Wizard Arcane)`
4. `Character "Borba" 5 Orc (Wizard Primal)`

(i) `Character name _ _ Rogue`

(ii) `Character name 5 ancestry class@(Wizard _)`

Aufgabe 2 *Funktions-Definitionen***3 P.**

Wir betrachten den Datentyp `Date` von Folie 3 der 3. Vorlesung:

```
data Date = DMY Int Int Integer deriving Show
```

1. Definieren Sie eine Funktion `probablyValidDate :: Date -> Bool`, welche überprüft, ob ein Datum die folgenden Bedingungen erfüllt: (1 Punkt)
 - Der Tag ist zwischen 1 und 31
 - Der Monat ist zwischen 1 und 12

In `Date` ist der erste Eintrag der Tag und der zweite Eintrag der Monat. Die folgenden Funktionen können in Haskell Zahlen miteinander vergleichen: `==`, `<`, `>`, `<=`, `>=`.

- Definieren Sie eine Funktion `maxList :: List -> Integer`, welche eine Liste (siehe Folien 5 und 19-21 der 3. Vorlesung) nimmt, und ihr größtes Element zurückgibt. Bei einer leeren Liste soll eine Fehlermeldung mit dem Text „maxList on empty list“ geworfen werden. (1 Punkt)
Hinweis: Die Funktion `max` gibt das größere von zwei Elementen zurück.
- Definieren Sie eine Funktion `addSecond :: Integer -> List -> List` welche eine Zahl und eine List nimmt, und diese Zahl zu jedem zweiten Element der Liste addiert. Wenn die Funktion zum Beispiel mit 2 und der Repräsentation von `[2, 7, 9, 3, 15, 5, 7]` aufgerufen wird, sollte sie die Repräsentation von `[2, 9, 9, 5, 15, 7, 7]` zurückgeben. (1 Punkt)

Aufgabe 3 Rekursive Datentypen und Funktionen

5 P.

Betrachten Sie den folgenden Datentypen:

```
data Expr = Number Integer | Plus Expr Expr | Negate Expr
```

- Implementieren Sie eine rekursive Funktion `convert :: Expr -> Expr`, die alle Vorkommen negativer `Integer` in einem `Expr` beseitigt, indem Sie sie durch entsprechende `Negate` Ausdrücke ersetzen, so dass `eval` für `e` und `convert e` die gleichen `Integer` ergibt. (1 Punkt)

Beispiel: `convert (Plus (Number (-2)) (Number 1)) = Plus (Negate (Number 2)) (Number 1)`

Hinweis: Beachten Sie, dass Sie in Haskell `>`, `>=`, `<`, `<=` und `==` verwenden können, um zwei `Integer` zu vergleichen. Jede dieser Vergleichsfunktionen retourniert ein `Bool`.

Darüber hinaus könnte die folgende Funktion, die zwischen zwei möglichen Ergebnissen je nach booleschem Zustand unterscheidet, nützlich sein.

```
ite True  x y = x
ite False x y = y
```

- Implementieren Sie eine rekursive Funktion `normalize :: Expr -> Expr`, die alle Vorkommen von `Negate` aus einem `Expr` "down the syntax tree"bewegt, so dass `Negate` nur direkt über (`Number n`) Blattknoten auftreten kann. Wie in Unterübung 1 wollen wir keine negativen Ganzzahlen in unserem Ergebnis. Stellen Sie erneut sicher, dass `eval` die gleichen `Integer` für `e` und `normalize e` ergibt. (3 Punkte)

Beispiel:

`normalize (Negate (Plus (Number (-2)) (Number 1))) = Plus (Number 2) (Negate (Number 1))`

- Implementieren Sie eine Funktion `listToExpr :: List -> Expr`, die eine `List` nimmt und eine `Expr` zurückgibt, die die Summe aller `Integer` in der Liste darstellt. Für das Hinzufügen von Zahlen in der leeren Liste können Sie `Number 0` schreiben, aber für nicht-leere Listen fügen Sie keine Zahlen hinzu, die nicht in der Liste sind, d.h. besonders fügen Sie `Number 0` nicht hinzu, wenn die Liste keine `0` enthält. Wie in Unterübung 1 wollen wir keine negativen Ganzzahlen in unserem Ergebnis. (1 Punkt)

Beispiel: `listToExpr Empty = Number 0`

Beispiel: `listToExpr (Cons 1 (Cons (-2) Empty)) = Plus (Number 1) (Negate (Number 2))`