Universität Bremen

Praktische Informatik 3 WS 2023/24

Thomas Barkowsky
Tede von Knorre
Yaroslav Purgin
Tim Sperling

7. Übungsblatt

Ausgabe: 04.12.2023
Abgabe: 18.12.2023

Dies ist das erste Gruppenübungsblatt. Bitte bearbeitet es in Übungsgruppen von je drei Studierenden. Das Übungsblatt wird mit insgesamt zwanzig Punkten bewertet und ist innerhalb von zwei Wochen zu bearbeiten. Die Abgabe soll über nur ein Repository getätigt werden und eine Datei "gruppe.txt"mit den Namen der Gruppenmitglieder enthalten. Bei tutorienübergreifenden Gruppen sollten die jeweiligen Tutoren informieren werden.

Für die Visualisierung in 7.2 benötigt ihr Graphviz.

Das Szenario:
Aus dem Nichts bekommst du eine E-Mail von einem dir unbekannten Absender. Es heißt, du seist der Thronerbe einer vergessenen königlichen Familie. Alles, was du machen musst, ist 300 € zu überweisen, damit der zuständige Anwalt die Machtübergabe abschließen kann. Bevor du das aber machst, möchtest du sicherstellen, dass das keine Scam-E-Mail ist. Angenehmerweise ist eine Liste mit den Familienmitgliedern dem Anhang hinzugefügt worden. Du erinnerst dich, dass Haskell eine mächtige und einem Herrscher angemessene Sprache ist und beschließt ein paar Leute zu rekrutieren, die dir für eine kleine Entlohnung damit helfen sollen. Außerdem konntet ihr euren PI3-Dozenten für euer Vorhaben begeistern und er stellt euch ein kleines Framework zusammen. Jetzt müsst ihr nur noch die fehlenden Funktionen implementieren und mehr über die Familie lernen, um dem Reichtum einen Schritt näherzukommen.

7.1 *Personen* — *6 Punkte*

Im ersten Teil möchtet ihr einige Funktionen für den Datentyp `Person` anlegen.

```
type Id = Int
type Year = Int
data Person = Person {
  getId :: Id
  , getName :: String
  , getYearOfBirth :: Year
  , getYearOfDeath :: Maybe Year
  , getParents :: (Maybe Id, Maybe Id)
  , getPartners :: [Id]
  , getChildren :: [Id]
  , getRoyalState :: RoyalState
} deriving (Eq, Show)
```

Partner sind hier alle Personen, mit den die repräsentierte Person in einer offiziellen Ehe ist oder war.

Die Liste mit den Familienmitgliedern ist schon vorgegeben:

```
type Family = [Person]
family = [alex, berta, clemens, dieter, erna, friederike, gustav, hannes, irma, jan, katja, luca]

alex = Person 1 "Alex" 2000 Nothing (Just $ getId clemens, Just $ getId berta)
       [getId irma] [getId luca] RoyalFamily
und so weiter...
```

Um eine Person anhand ihrer Id zu finden, gibt es

```
getPerson :: Family → Id → Person
```

Beispiel:

```
getPerson family 1 ⇝ Person 1 "Alex" 2000 Nothing (Just $ getId clemens, Just $ getId berta)
                     [getId irma] [getId luca] RoyalFamily
```

Die ersten Funktionen, die ihr implementieren sollt, sind `getAge` und `isAlive`. `isAlive` gibt an, ob eine Person im gegebenen Jahr am Leben ist oder war. `getAge` rechnet aus, wie alt eine Person im gegebenen Jahr ist oder war. Bei verstorbenen Personen wird das Alter bis zum angegebenen Jahr, aber maximal bis zum Todesjahr gezählt. In diesem Übungsblatt könnt ihr davon ausgehen, dass alle Menschen am 1. Januar geboren werden und am 31. Dezember sterben.

```
isAlive 2023 clemens ⇝ false
getAge 2023 alex ⇝ 23
getAge 2023 clemens ⇝ 51
```

Mit der Funktion `isOrphan` kann man überprüfen, ob eine Person Waise ist, was bedeutet, dass sie ihre Eltern verloren hat oder diese unbekannt sind. Dabei werden nur Kinder bis zum Alter von 18 Jahren als Waisen bezeichnet; sobald also eine Person 18 oder älter ist, wird sie nicht mehr als Waise bezeichnet.

```
isOrphan family 1952 erna ⇝ True
```

Mit der Funktion `getSiblings` kann man alle Geschwister einer Person herausfinden. `getFullSiblings` gibt nur die Geschwister an, mit den die Person zwei gleiche Eltern hat. Bei `getHalfSiblings` hingegen ist man nur an den Geschwistern interessiert, die nur ein Elternteil mit der Person teilen.

Anm.: Bei den folgenden Beispielen werden die zurückgegebenen Werte verkürzt dargestellt; die tatsächliche Anzeige der Rückgabewerte in der Konsole wird also von den Beispielen abweichen und ausführlicher sein.

```
getSiblings family alex ⇝ [katja,jan]
getFullSiblings family alex ⇝ [katja]
getHalfSiblings family alex ⇝ [jan]
```

`getSiblingsOfParents` soll alle Onkel und Tanten einer Person ausgeben.

```
getSiblingsOfParents family luca ⇝ [katja,jan]
```

`getCousins` hilft dir zu erfahren, wer deine Cousins und Cousinen sind.

```
getCousins family irma ⇝ [alex,jan,katja]
```

7.2 *Stamm**baum*** *9 Punkte*

Nun ist es auch noch interessant herauszufinden, wer alle Nachfahren und alle Vorfahren einer Person sind. Um dies vernünftig darzustellen, nutzen wir einen `VarTree`.

```
getDescendants family dieter ⇝ VarTree 4 [VarTree 2 [VarTree 1 [VarTree 12 []],VarTree 10 [],
                                VarTree 11 []],VarTree 8 [VarTree 9 [VarTree 12 []]]]
```

```
getAncestors family irma ⇝ VarTree 9 [VarTree 8 [VarTree 4 [],VarTree 5 []]]
```

Nachdem wir jetzt einen Baum mit Ids haben, wäre es natürlich auch schön, sich nicht nur die Ids, sondern auch die Namen im Baumformat anzeigen zu lassen. Dies sieht besonders gut aus, wenn man es dann mit `writeDot` und Graphviz anzeigen lässt. Hierfür benötigen wir eine Funktion, um über alle Elemente des `VarTree`s zu mappen. Da wir hier einen Baum und keine Liste haben, implementieren wir `fmap` für den Baum.

```
fmap :: Functor f ⇒ (a → b) → f a → f b
```

Mithilfe von fmap können wir nun die Funktion convertToNames schreiben:

```
convertToNames :: Family → VarTree Id → VarTree String
```

Um das Ganze zu einem .dot-Format zu konvertieren, müssen wir den Baum jetzt nur noch falten können:

```
foldVarTree :: (a → [b] → b) → VarTree a → b
```

Beispiel:

```
foldVarTree (λx ys → x +sum ys)(VarTree 1 [VarTree 2 [], VarTree 3[]]) ⇝ 6
```

Wenn das geht, könnt ihr euch den Graphen in einem Fenster anzeigen lassen:

```
stack ghci
writeDot "family.dot" $ convertToNames family $ getAncestors family alex
:q
dot -Tpng family.dot -o family.png
```

Sollte das nicht klappen, nutzt statt der letzten Zeile

```
dot -Tpng family.dot -O ,
```

um euch eine Bilddatei zu generieren.

Interessant ist weiterhin, mit wem eine Person direkt und indirekt verwandt ist. Es könnte ja sein, dass sich eine Person in die Liste eingeschlichen hat, die in keiner Weise mit dem Rest verbunden ist. Hierbei zählen Eltern, Kinder, und alle Partner als direkt verwandt.

```
getFamily :: [Person] → Person → Family
```

Eine Familie ändert sich auch ab und zu, oder man muss Korrekturen vornehmen. Hierzu brauchen wir eine `update` Funktion, die eine geänderte Person anhand ihrer Id in der Familienliste findet und dort austauscht. Falls diese Person noch nicht in der Liste ist, soll sie hinzugefügt werden. Dadurch kann man zum Beispiel ein Kind einfach einfügen.

```
update alex{getName = "Andreas"} [alex,gustav] ⇝ [alex{getName = "Andreas"},gustav]
```

Bei neuen Kindern muss aber noch mehr gemacht werden. Die Eltern müssen auch jeweils eine aktualisierte Liste an Kindern bekommen. Das wird in `conceiveChild` gemacht.

```
newFamily = conceiveChild family "Marvin" 2023 alex irma
newFamily \\ family ⇝ [Person {getId = 13, getName = "Marvin", getYearOfBirth = 2023,
                       getYearOfDeath = Nothing, getParents = (Just 1,Just 9),
                       getPartners = [], getChildren = [], getRoyalState = RoyalFamily},
                       alex{getChildren = [13,12]}, irma{getChildren = [13,12]}]
```

Man sollte natürlich auch heiraten können. Dabei ist es hier nicht wichtig, ob eine Person schon andere Partner hatte oder hat.

```
marry family2 abel barbara ⇝ [abel{getPartners = [getId barbara]},
                              barbara{getPartners = [getId abel]}, christoph]
```

7.3 *Thronfolge* — *5 Punkte*

Um nun herauszufinden, ob du wirklich der Thronfolger bist, müssen wir nach den Regeln der Thronfolge vorgehen:

1. Falls der aktuelle Herrscher noch am Leben ist, bleibt er Herrscher.
2. Falls nicht, wird eine Tiefensuche durch die Nachfahren nach dem ersten lebenden Nachfahren gemacht.
   Hierbei kommt zuerst das älteste Kind an die Reihe, dann dessen Nachfahren, dann das zweitälteste Kind, dessen Nachfahren, und so weiter.
3. Falls alle Nachfahren nicht mehr leben, wird die Tiefensuche über die Eltern neugestartet. Hierbei wird zuerst das Linke Elternteil betrachtet, und dann das rechte.

Dafür schreiben wir zwei Hilfsfunktionen: Zuerst `getFirstLivingDescendant`, um für Punkt 2 den ersten lebenden Nachfahren herauszufinden. Um die Rekursion zu vereinfachen, wird im Falle, dass die angegeben Person noch lebt, die Person selbst zurückgegeben.

```
getFirstLivingDescendant family 2023 dieter ⇝ hannes
```

Dies muss nun noch mit Punkt 3 zusammengefasst werden, um auch die anderen Nachkommen der Vorfahren zu durchsuchen. Hierbei gilt wieder die gleiche Regelung zur Vereinfachung der Rekursion.

```
getFirstLivingSuccessor family 2023 berta ⇝ berta
getFirstLivingSuccessor family2 2023 christoph ⇝ abel
```

Wenn wir nun den Nachfolger gefunden haben, müssen wir nur noch die Familienliste anpassen. Falls der aktuelle Herrscher noch lebt, ändert sich diese offensichtlich nicht. Wichtig: Auch Personen mit dem Status NonRoyal können Ruler werden.

```
filter ((==Ruler).getRoyalState) (updateRuler family 2023) ⇝ [hannes]
```