Aufgabe 3a: Gemischte Daten mit eigenen Klassendefinitionen

Eigene Klassen für Werte

Zwei Repräsentationen von Wochentagen durch Wertklassen

- Repräsentation durch die Klasse DayNum

  DayNum ::= DayNum[:num] :: (1…7)

- Repräsentation durch die Klasse DaySym

  DaySym ::= DaySym[:sym] :: {Mo, :Di, :Mi, :Do, :Fr, :Sa, :So}

- Der **gemischte Typ** Day ist die **disjunkte Vereinigung** (oder strukturelle „**Summe**“) von beiden:

  Day ::= (DayNum | DaySym)

- Aus Day können wir zurzeit **noch keine Klasse machen** (später wird es eine sogenannte abstrakte Klasse)

- Wir haben also nur ein Typprädikat in funktionaler Notation

- Achten Sie darauf, daß die Typprädikate für DaySym und DayNum in **OO-Notation** geschrieben werden, und das Prädikat für Day in **funktionaler Notation**

  o = DaySym[:Mo]

  o.day_sym #=> true

  o.day_num #=> false

  day(o) #=> true

Repräsentation der Bijektion und der Reihenfolge durch Sequenzen aufgefasst als Abbildung

Jede Sequenz mit n Elementen kann als tabellierte Funktion interpretiert werden (dabei steht (0…n) für (0..(n-1))):

(0…n) -> {:Mo, :Di, :Mi, :Do, :Fr, :Sa, :So}

Wenn die Sequenz keine Duplikate enthält, ist es sogar eine Bijektion und es existiert eine Umkehrfunktion

(0…n) <-> {:Mo, :Di, :Mi, :Do, :Fr, :Sa, :So}

In unserem Fall haben wir die zwei Sequenzen

DAY_SYM_SEQ ::= [:Mo, :Di, :Mi, :Do, :Fr, :Sa, :So]

DAY_NUM_SEQ ::= [1, 2, 3, 4, 5, 6, 7]

Diese Sequenzen lassen sich in Ruby mit Arrays direkt umsetzten:

```
DAY_SYM_SEQ = [:Mo, :Di, :Mi, :Do, :Fr, :Sa, :So]
```

Die Abbildung von einem “Index” index aus (0…n) auf ein Symbol geht mit dem Operator [].
[i] bedeutet in diesem Kontext: Projektion auf die i-te Komponente.

```
DAY_SYM_SEQ[index]
```

Die Umkehrabbbildung von einem Element sym auf einen Index geht mit

```
DAY_SYM_SEQ.index(sym)
```

Wenn wir zwei Sequenzen (ohne Duplikate) haben, die auch noch gleich lang sind, haben wir damit indirekt auch eine Bijektion zwischen beiden definiert.

Varianten

Sie können sich auch Sequenzen von den Exemplaren der neuen Klassen bauen, z.B. durch explizite Aufzählung:

DAYSYM_SEQ = [DaySym[:Mo], …, DaySym[:So]]

Das geht in Ruby aber viel kompakter (ein Vorgriff aus später), indem wir die Konstruktionsfunktion DaySym[sym] auf jedes Element der Sequenz DAY_SYM_SEQ anwenden:

DAYSYM_SEQ = DAY_SYM_SEQ.map{|sym| DaySym[sym]}

Damit haben wir jetzt eine neue Funktion definiert:

```
DAYS_IN_WEEK  ::= DAY_SYM_SEQ.size
DayIndex      ::= Nat :: (0…DAYS_IN_WEEK)

DayIndex <-> DaySym
```

Refaktorisierung der Fallunterscheidungen

Sie können viele Fallunterscheidungen durch eine parametrisierte Konversion überflüssig machen.

to_day ::= Day x Day -> Day :: (proto_day,day)

proto_day ist ein Prototyp(irgendein Exemplar von day), dessen Typ angibt, in welche Repräsentation konvertiert werden soll.

Sie finden eine Erläuterung dazu in den Folien über Clocks unter Konversionen.

Erweiterung um Addition und Subtraktion von Tagen

Durch die parametrisierte Konversion wird es sehr einfach neue Operation zu definieren.

Z.B. eine Art Addition und Subtraktion von Tagen

```
day_shift ::= Day x Int -> Day ::
                                    Test{(DaySym[:Mo],-2) => DaySym[:Sa])
```