Übungsblatt zu Vorlesung 09— Algebraic View On Monads

In dieser Übung beschäftigen wir uns mit dem State-Monad und wiederholen kurz applikative Funktoren. Die Signaturen der in den Aufgaben geforderten Methoden sowie die vorgegebenen Implementierungen finden Sie online unter `https://gitlab2.informatik.uni-wuerzburg.de/intro-to-fp/tasksheets` als Git-Repository.

1 Stack-Based Calculator

In dieser Aufgabe implementieren wir einen stackbasierten Taschenrechner, der einfache ganze Zahlen addieren und multiplizieren kann.

Den Zustand unseres Taschenrechners modellieren wir über einen einfachen Stack vom Typ `List[Int]`. Das Ergebnis dieser Berechnungen ist immer ein `Int`. Um den Zustand über mehrere Berechnungen hinweg zu verwalten, benutzen wir den aus der Vorlesung bekannten `State`-Monad.

```
type Calc = State[List[Int],Int]
```

Cats bietet mit `cats.data.State` bereits eine `State` Implementierung, die der aus der Vorlesung sehr ähnlich ist und die wir hier verwenden.

Implementieren Sie:

`push(nr: Int): Calc` Legt eine Zahl oben auf den Stack. In unserer Implementierung entspricht das dem Anhängen vorne an die Liste.

`pop: Calc` Holt die oberste Zahl vom Stack. Ist der Stack leer, soll `0` ausgegeben werden.

`add: Calc` Holt die oberen beiden Zahlen vom Stack und schreibt deren Summe zurück. Gibt außerdem die Summe aus.

`mul: Calc` Holt die oberen beiden Zahlen vom Stack und schreibt deren Produkt zurück. Gibt außerdem das Produkt aus.

Hinweis: `State` hat zwei Parameter, der erste für den Zustand (unser Stack) den zweiten für die Ausgabe (in unserem Fall `Int`). Die meisten oben geforderten Methoden nutzen beide Werte.

Im Git liegt außerdem eine `main`-Methode, mit der Sie Ihre Implementierung beliebig testen können. Die Eingabe erwartet Zahlen und Operatoren in Postfix-Notation. Die Eingabe `3 4 * 5 +` entspricht also der Berechnung `3 * 4 + 5`. Ist ihr Rechner korrekt implementiert, kommt `17` heraus.

2 CandyMachine

In dieser Aufgabe implementieren Sie einen Endlichen Automaten (*finite state machine*), der einen einfachen Bonbonautomat modelliert. Der Automat hat zwei Arten von input: Man kann eine Münze einwerfen oder den Drehknopf betätigen um Bonbons zu bekommen. Der Automat kann entweder *locked* oder *unlocked* sein und merkt sich, wie viele Bonbons übrig sind und wie viele Münzen er beinhaltet.

```
enum Input:
  case Coin
  case Turn

case class Machine(locked: Boolean, candies: Int, coins:Int)
```

Die Regeln der `Machine` sind folgende:

- Das Einwerfen einer Münze sorgt dafür, dass der Automat *unlocked* wird, falls noch *candies* übrig sind.
- Den Drehknopf bedienen, wenn der Automat *unlocked* ist, sorgt dafür, dass ein *candy* ausgegeben wird und der Automat wieder *locked* wird.
- Wenn der Automat *locked* ist und der Drehknopf bedient wird geschieht nichts.
- Sind keine *candies* mehr übrig ignoriert der Automat jegliche inputs.

Die Funktion `simulateMachine` steuert den Automat anhand einer Liste von Inputs und gibt die Anzahl an Münzen und Bonbons zurück, die am Ende im Automat verbleiben. Falls der Automat am Anfang 10 Münzen und 5 Bonbons besitzt und 4 Bonbons erfolgreich gekauft wurden, sollte der Output `(14,1)` sein.

Hinweis: Für `simulateMachine` ist `sequence` wieder hilfreich.

```
def simulateMachine(inputs: List[Input]): State[Machine, (Int, Int)] = ???
```

3 Applicatives Filtern

Für Listen haben wir bereits die Funktion `filter` kennen gelernt, die Listen anhand eines Prädikats filtert. Mit `Applicative`s können wir eine Verallgemeinerung dieser Funktion definieren, die die Auswahl der Elemente abhängig von einem bestimmten Kontext durchführt.

```
def filterA[F[_],A](l: List[A])(f: A => F[Boolean])(using Applicative[F]): F[List[A]]
```

Hinweis: Cats definiert bereits eine Methode `filterA`, die auch das gleiche macht, wie die, die wir hier definieren. Die Implementierung weicht aber etwas von der hier vorgestellten ab.

Die an `filterA` übergebene Funktion gibt für jedes Listenelement einen Wahrheitswert mit einem Effekt aus. Diese werden kombiniert und auf die Liste angewandt. Der Wahrheitswert entscheidet ob, das entsprechende Element in das Ergebnis aufgenommen wird. Wie genau das passiert entscheidet sich durch die Wahl der Funktion und des Effekttyps.

Im Git finden Sie ein Beispiel für `filterA` in Kombination mit `Validated`. Cats bietet eine Variante von `Validated`, die wie in der Vorlesung gezeigt dazu genutzt werden kann, Fehler zu akkumulieren. Welcher Typ für die Fehlerakkumulation genutzt wird, ist nicht vorgegeben. Im Beispiel nutzen wir `List[String]`. Für eine Liste von ganzen Zahlen wird per `Validated` entschieden, ob ungerade Zahlen in der Liste sind und entweder eine Reihe von Fehlermeldungen für jede gefundene ungerade Zahl oder, falls alle Zahlen gerade sind, eine Liste derjenigen Zahlen, die auch durch vier teilbar sind.

- Schreiben Sie eine eigene Implementierung von `filterA`. Sie soll die gleichen Ergebnisse liefern, wie die aus Cats. *Tipp:* Verwenden Sie `foldRight` und `map2`, um die `Applicative`s zu verbinden.

- `Validated` bietet den Effekt, korrekte von Fehlerfällen zu unterscheiden. Den Effekt von `List` kann man hingegen als Mehrdeutigkeit auffassen. Eine Liste fasst mehrere Optionen zusammen. Schreiben Sie eine Funktion, die die Potenzmenge einer Liste konstruiert. Die Potenzmenge ist die Menge aller möglichen Teilmengen einer Menge. Verwenden Sie ausschließlich `filterA`.

```scala
def powerset[A](l: List[A]): List[List[A]] = ???
```

- Beschreiben Sie, wie sich `filterA` verhält, wenn man als Effekttyp die Binärbäume aus der letzten Übung verwendet. Gehen Sie insbesondere auf die Struktur des resultierenden Baums ein und was in dem/n Blatt/Blättern steht.