Exercise sheet for lecture 09— Algebraic View On Monads

In this exercise we take a look at the state monad and do a short recap of applicative functors. You can find some of the signatures for the exercises as well as the given implementations online at `https://gitlab2.informatik.uni-wuerzburg.de/intro-to-fp/tasksheets` as a Git repository.

1 Stack-Based Calculator

In this task we implement a stack based calculator. It will be able to add and multiply integers.

The State of the calculator is modelled as a simple stack (`List[Int]`). The result of every calculation is always an integer. To hold the state for multiple calculations we use the `State`-Monad.

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

The Cats library already provides `cats.data.State`, which is a state monad implemention very similar to the one from our lecture. We will use it here.

Implement:

`push(nr: Int): Calc` Pushes an int onto the top of the stack. In our case this means prepending.

`pop: Calc` Pops the top integer from the stack. If the stack is empty yield `0`.

`add: Calc` Takes the top 2 integers from stack, sums them and store the sum back. Also yields the sum as the result.

`mul: Calc` Same, but with multiplication.

Hinweis: `State` has two parameters, the first for the state (Stack) and the second for the result (`Int`). Most of the methods above use both.

Within the Git repository there is a `main`-Method which you can use to test your implemention. The input is expected to be in reverse Polish notation. If you want to calculate `3 * 4 + 4` you have to input `3 4 * 5 +`. If your calculator is implemented correctly the result is `17`.

2 CandyMachine

In this task you will be implementing a finite state machine modelling a candy machine. The machine has two inputs: insert a coin or turn the nob. The machine will either be *locked* or *unlocked* and remember how much candy is left as well as how many coins it has.

```
enum Input:
  case Coin
  case Turn

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

The rules are as follows:

- Inserting a coin *unlocks* the machine as long as there are any *candies* left.
- Turning the nob on an *unlocked* machine disposes a *candy* and *locks* the machine.
- Nothing happens when turning the nob on a *locked* machine.
- If there are no *candies* left the machine ignores any input.

The method `simulateMachine` controls the machine according to a list of inputs and returns the number of coins and candy at the end. If the machine starts with 10 coins and 5 candy and 4 candy have been bought successfully the output should be `(14, 1)`.

Hinweis: To implement `simulateMachine` `sequence` is very useful.

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

3 Applicative Filtering

We already learned about `filter` on lists. It takes a predicate to filter a list. Using `Applicative`s we can write a more general version of that function which filters the elements based on a specific context.

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

Hinweis: Cats already provides a `filterA` method doing the same thing we want to do. It differs slightly from the one proposed here though.

The predicate passed to `filterA` provides a boolean value inside a context/effect for each element of a list. Those are evaluated and then combined. As expected, the truth value inside the context/effect dicates whether the element should be in the resulting list. How exactly this works depends on the predicate and effect type.

Inside the Git repository you can find an example for `filterA` in combination with `Validated`. Cats provides a version of `Validated` which can be used to accumulate errors the same way as in the lecture. You can decide which type you want to use for error accumulation. The example uses `List[String]`. The example iteraties over a list of integers and for each checks whether it is even or odd. It returns either a list of errors if there are odd integers or a list of integers divisible by four.

- Write your own implemention of `filterA`. It should yield the same results as the version from Cats. *Hint:* Use `foldRight` and `map2` to combie the `Applicatives`.
- `Validated` provides the capability/effect to decide between valid and invalid cases. The Effect of `List` could be said to be multitude. A list contains a multitude of options. Implement a method which constructs the power set of a list. The power set of a set is the set of all possible subsets of said set. Only use `filterA` to achieve it.

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

- Describe what `filterA` does if you use the binary trees from the last exercise as a effect type Take a close look at the structure of the resulting Tree. What is in the leaves?