Homework 2

Functional Programming (ITI0212)

Deadline: Wednesday 9th April 2025

Task 1
A queue is a data structure with two “ends”. An empty queue contains no data. We can define a typeclass `Queue` with operations for pushing an element (onto the back) and for popping an element (from the front) in a first-in–first-out manner, and also a field for the empty queue.

```
class Queue (t : Type -> Type) where
  emp  : t α
  push : α → t α → t α
  pop  : t α → Option (α × t α)
```

A list can be used as a simple (if inefficient) form of queue. Write an implementation of `Queue` for the `List` type constructor. The behaviour should be as follows, e.g.

```
#eval (((Queue.push 3 ∘ Queue.push 2 ∘ Queue.push 1)
   Queue.emp) : List Nat)
----> [1, 2, 3]
#eval Queue.pop [1, 2, 3]
----> some (1 , [2, 3])
#eval Queue.pop ([] : List Nat)
----> none
```

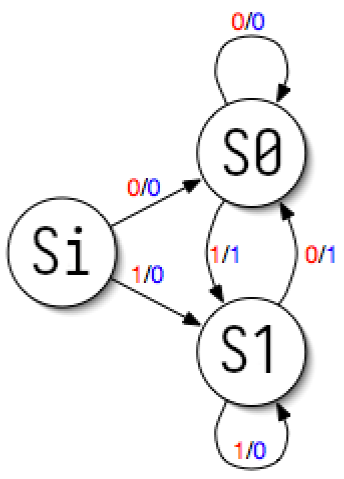
Task 2
Using either `do` notation or applicative style (`pure` and `<*>`), write a function

```
add3 (m n o : Option Nat) : Option Nat
```

that returns the sum of the three inputs when they all exist and `none` otherwise.

Task 3
A *Mealy machine* is a kind of state machine that, in a given state, can read an input, which causes it to transition to a new state and omit an output. For example, a cipher machine could be modelled as a Mealy machine.

Here is an example of a Mealy machine with three states:



The labellings (in/out) of each arrow say what input must be received for the transition to occur (left, red), and what is outputted when the transition occurs (right, blue). For example, if the machine is in state `Si` and it receives `1` as an input, it will output `0` and move to state `S1`.

A Mealy machine with inputs of type α and outputs of type β is thus like a function from $\alpha \to \beta$ that has some internal state of type σ. We can therefore represent it as a function α `→ StateM` σ β.

Use the inductive type,

```
inductive MealyState
| Si : MealyState
| S0 : MealyState
| S1 : MealyState
deriving Repr
```

for the type of states of the machine above, and `Bool` for the type of the inputs and outputs.

Write a function

`mealy (input : Bool) : StateM MealyState Bool`

which given an input and a state, returns the new state and the output, according to the diagram above.

Recall that `StateM MealyState Bool` is defined to be the type

`MealyState → Bool × MealyState`,

so the function above really is a function of two arguments.

Task 4
Since Lean provides a (generic) instance of `Monad` for `StateM` σ, we can leverage the generic function `List.mapM` to lift the function `mealy` from the previous task, to a function taking a *list* of inputs and returning the corresponding *list* of outputs, obtained by running the machine on the list of inputs, assuming that some state is designated as the initial one that the machine starts in.

You should take `Si` as the initial state of the machine. Use `List.mapM` to write this (one-line) function

`run_mealy (inputs : List Bool) : List Bool`

For example,

```
#eval run_mealy [true]
----> [false]
#eval run_mealy [true, false, false, true]
----> [false, true, false, true]
```

Bonus: can you tell what function this Mealy machine is computing?

Hint: the output is a function of the previous two inputs.

Task 5
Write a function that takes either ($\oplus$) a computation that when run (possibly does some IO) and returns an α or a computation that when run (possibly does some IO) and returns a β, and returns a computation that when run, runs whichever computation was given and produces the corresponding result:

```
sumIO : (IO α) ⊕ (IO β) → IO (α⊕β)
```

Now write a function that takes both a computation that when run (possibly does some IO) and returns an α and a computation that when run (possibly does some IO) and returns a β, and returns a computation that when run, runs the two computations in order and produces the product of their results:

```
bothIO : IO α × IO β → IO (α × β)
```

This illustrates how values of type `IO` α can be passed to functions and manipulated like values of any other type.