State Monad
State vis-a-vis Pure Functions

by definition, a pure function is stateless

this means:

① it has no side-effects (e.g. no global/static variable mutations)

② its result for given input(s) will always be the same
   — provides us with referential transparency
but sometimes state is a very useful concept!

e.g. say we’re implementing a game of a character that can jump and fire projectiles:
- we can model the game world as a “state” value:
- and “jump” and “fire” as functions that compute new states from input states:
and now we can “chain” together multiple stateful functions by just feeding the result of one into the input of another
sometimes, however, a stateful function is interested in computing or extracting a value from the input state.

e.g., maybe we want to get the current position of the character ...

\[ \text{ourPosition} \rightarrow (x, y) \]
but this breaks our stateful function chaining!

\[ \text{currPosition} \rightarrow (x, y) \]

\[ \text{fire} \]
So we can update our stateful functions to take an input state and return both an output state and a computed value!

\[
\text{originPosition} \rightarrow \left( \begin{array}{c} \text{new state} \\ (x, y) \end{array} \right)
\]
So, our notion of a "stateful function" has evolved to be ... a function that:

1. takes an input state
2. returns a tuple of:
   - an output state
   - some arbitrary value, possibly computed from the input state.
In Haskell, we can define the polymorphic type:

```
data States a = State (s → (s, a))
```

- **Value constructor**: `State (s → (s, a))`
- **Value type**: `s` and `a`
- **Function**: `s → (s, a)`
- **Input stack**: `s`
- **Output stack**: `(s, a)`
- **Type constructor**: `States a`
to make this an instance of Functor/Applicative/Monad, we need to be able to view a "State" value as a context for a value.

```haskell
data State s a = State (s → (s, a))
```

we pick this as the type of the value "contained in" a stateful function

practically, it is the value returned by the stateful function after applying it to an input state.
So, we make "States" an instance of these classes ...

instance **Functor** (States) where

\[ \mu \text{f-map} f \cdot (\text{stalest}) = \text{State} \ldots \]

instance **Applicative** (States) where

\[ (\text{State stf}) \circ (\text{State stx}) = \text{State} \ldots \]

instance **Monad** (States) where

\[ (\text{State stx}) \gg f = \text{State} \ldots \]
data State s a = State (s → (s, a))

instance Functor (State s) where
  fmap f (State st) = State...

- st is a stateful function that, when applied to some input state, gives us a value X
- st' is a stateful function that, when applied to some input state, gives us the value fX
data State s a = State (s → (s, a))

instance Functor (State s) where

\( \lambda s \rightarrow \text{let } (s', x) = \text{st } s \text{ in } (s', f x) \)

fmap f (State st) = State ...
data State s a = State (s → (s, a))

instance Applicative (State s) where

(State stf) <*> (State stx) = State ...


\[
\begin{align*}
S & \xrightarrow{sf} S' \\
S' & \xrightarrow{stf} (S, a) \\
S & \xrightarrow{fx} f_x \\
S' & \xrightarrow{x} \end{align*}
\]
data State s a = State (s → (s, a))

instance Monad (State s) where

(State st) >>= f = state

s :