State Monad

State vis-a-vis Pure Functions

by definition, a pure function is stateless

this means:

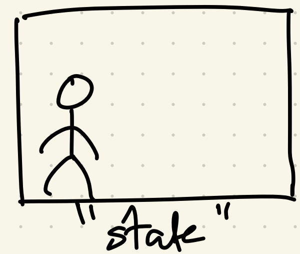
① it has no side-effects (e.g. no global/static variable mutations)

② it's result for given input(s) will always be the same

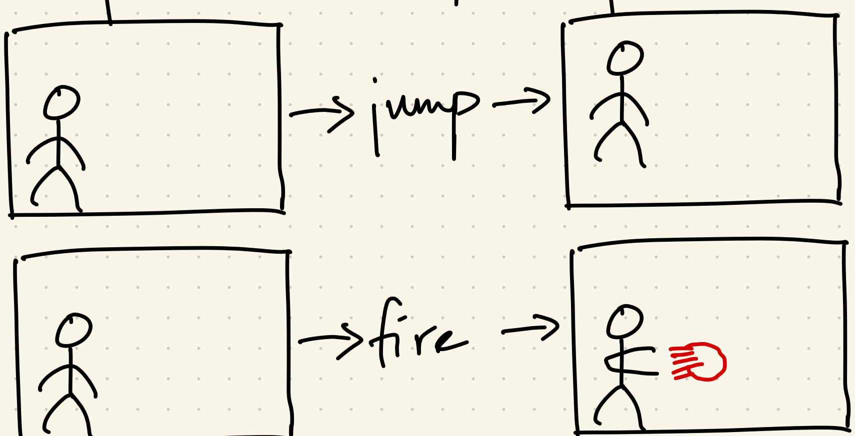
—— provides us with referential transparency but sometimes state is a very useful concept!

e.g. say we're implementing a game w/ a character that can jump and fire projectiles:

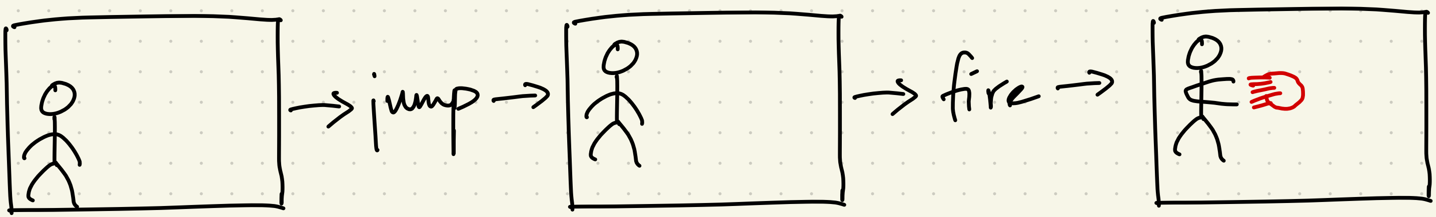
- we can model the game world as a "state" value:



- and "jump" and "fire" as functions that compute new states from input states:

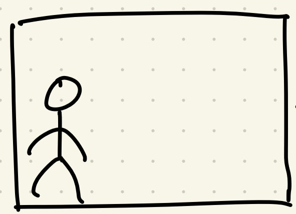


and now we can "chain" together multiple stateful functions by just feeding the result of one into the input of another



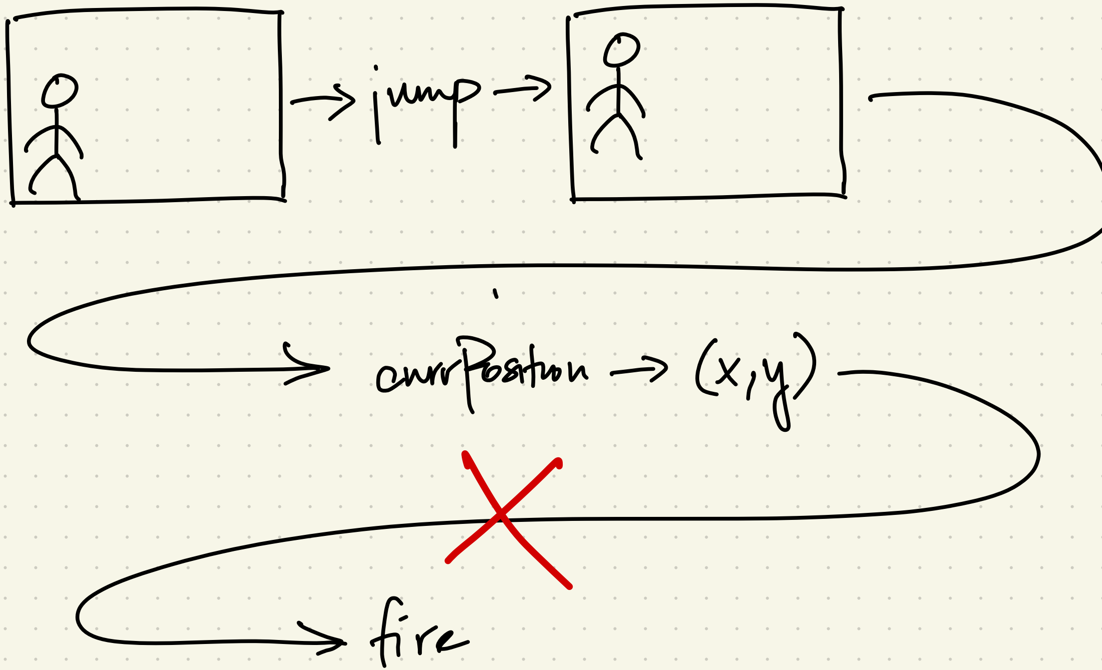
sometimes, however, a stateful function is interested in ==computing or extracting a value== from the input state.

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

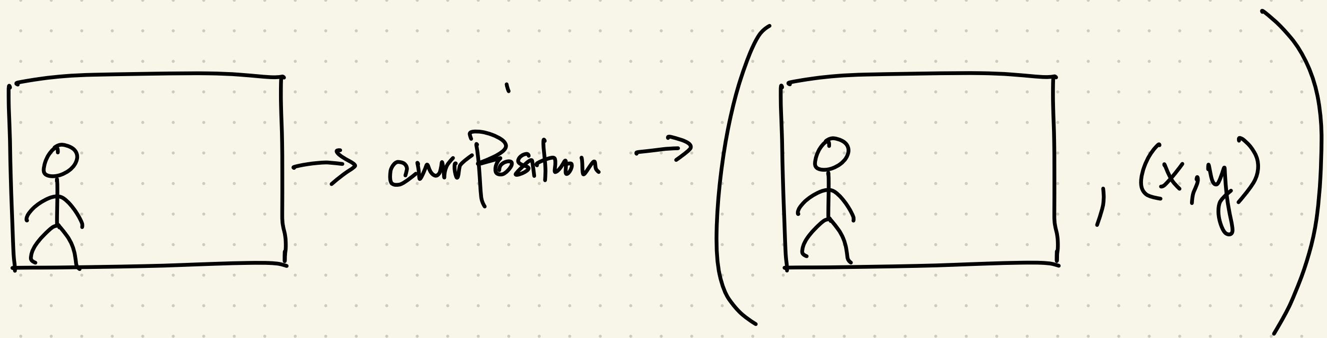


$\rightarrow$ currPosition $\rightarrow$ (x, y)

but this breaks our stateful function chaining!



so we can update our stateful functions to take an input state and return both an output state and a computed value!



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:

value constructor

function

```
data State s a = State (s -> (s, a))
```

type constructor

state type

value type

input state

output state

computed value

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.

```
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 "State s" an instance of these classes . . .

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

instance Applicative (State s) where
    (State stf) <*> (State stx) = State ...

instance Monad (State s) where
    (State stx) >>= f = State ...
```

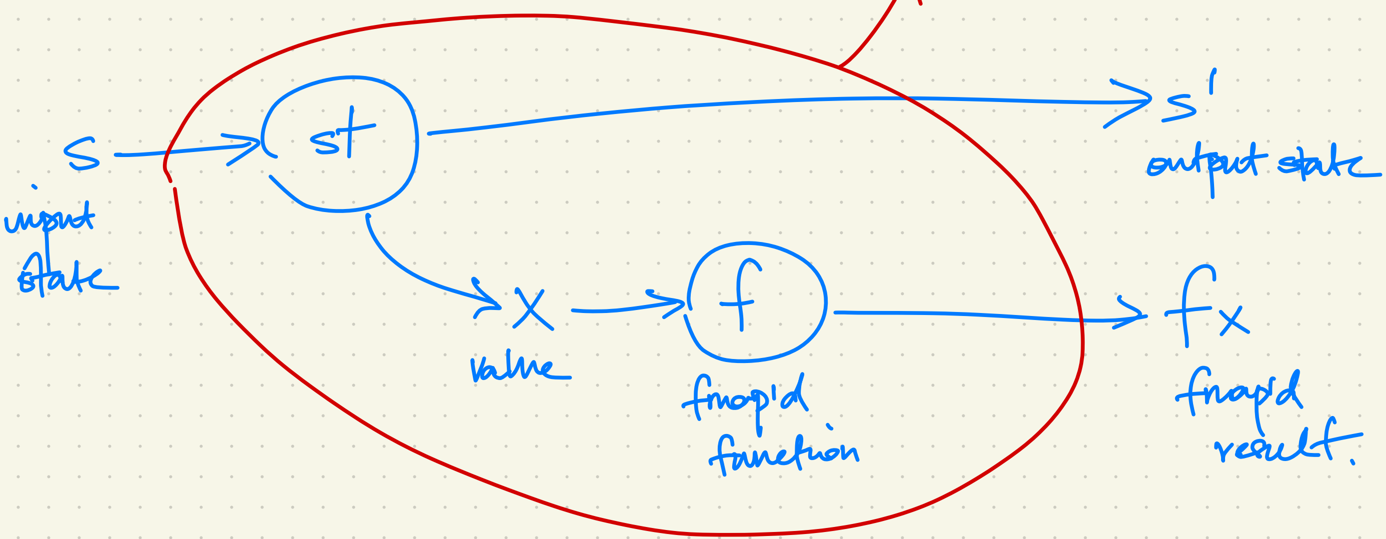
how do we interpret these operations?

```haskell
data State s a = State (s -> (s,a))

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

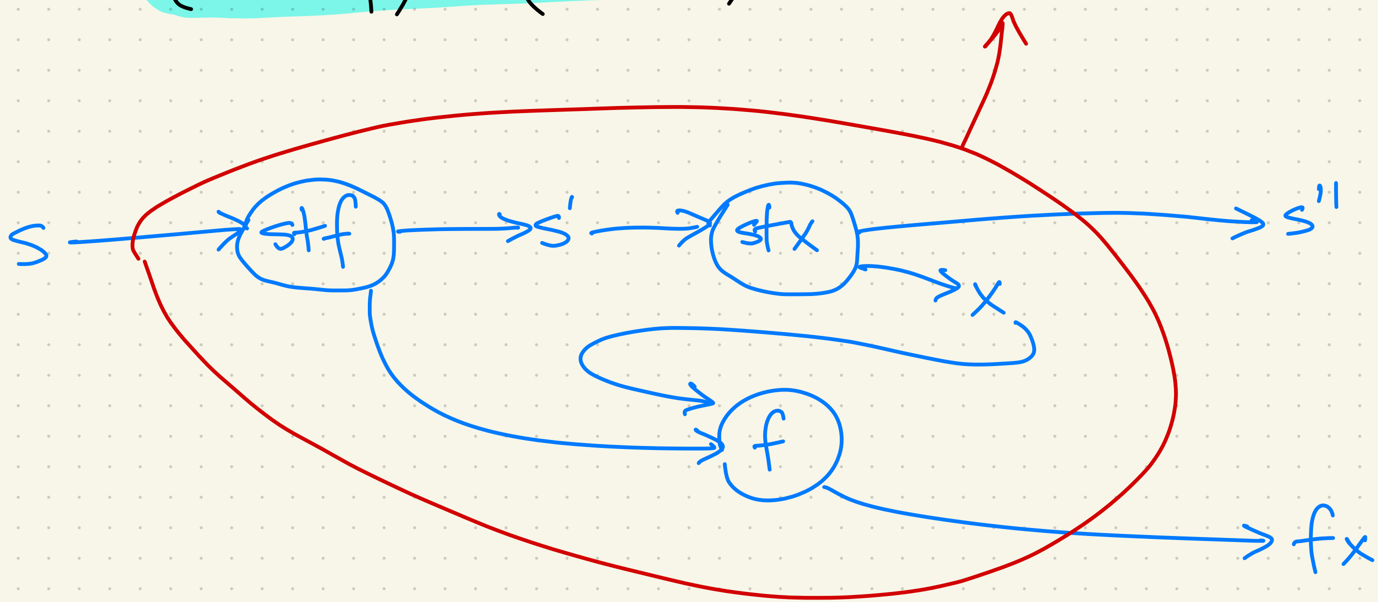
- st is a stateful function that, when applied to some input state, gives us a value x
- st' is a stateful function that, when applied to some input state, gives us the value f x

```haskell
data State s a = State (s -> (s,a))

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

```haskell
\s -> let (s',x) = st s
      in (s', f x)
```



```
data State s a = State (s -> (s,a))

instance Applicative (State s) where
  (State stf) <*> (State stx) = State ...
```



```haskell
data State s a = State (s -> (s,a))

instance Monad (State s) where
    (State st) >>= f = State ...
```