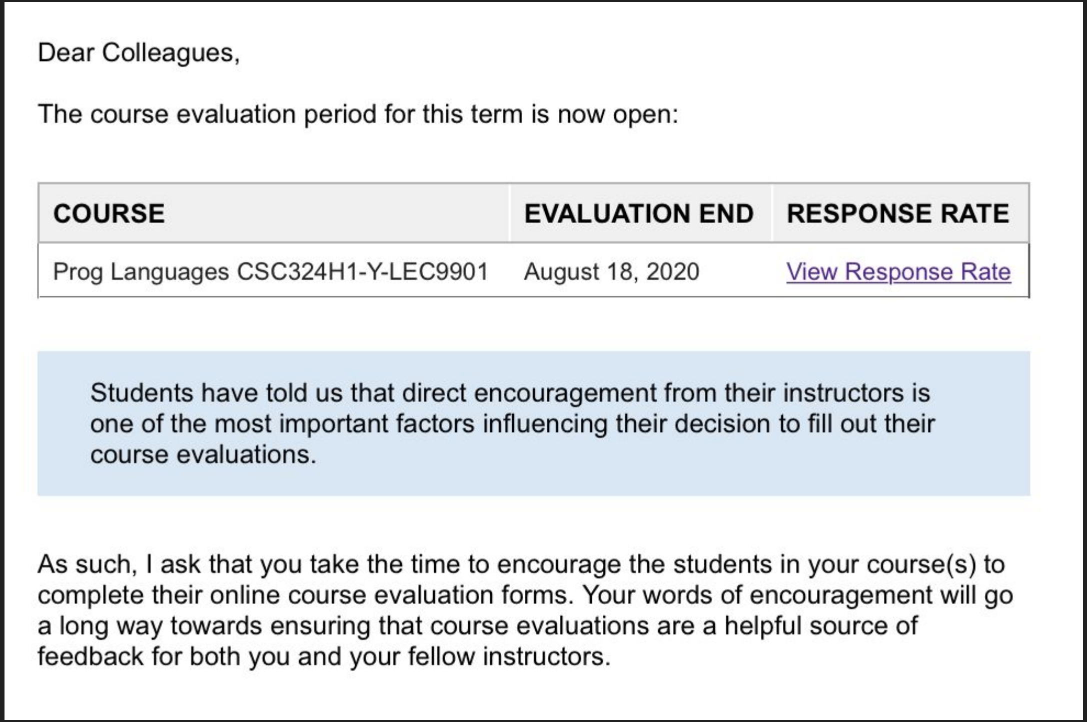
CSC324 Lecture 23

Administrivia

Course feedback is open, if you haven't filled it out yet!



Dear Colleagues,

The course evaluation period for this term is now open:

COURSE	EVALUATION END	RESPONSE RATE
Prog Languages CSC324H1-Y-LEC9901	August 18, 2020	View Response Rate

Students have told us that direct encouragement from their instructors is one of the most important factors influencing their decision to fill out their course evaluations.

As such, I ask that you take the time to encourage the students in your course(s) to complete their online course evaluation forms. Your words of encouragement will go a long way towards ensuring that course evaluations are a helpful source of feedback for both you and your fellow instructors.

Last time...

We were introduced to the **state monad**, which encapsulates operations on a piece of mutable state with a referentially-transparent interface

This is where we got up to on Wednesday.

```haskell
rand :: Integral a => a -> a
rand seed = (7^5 * seed) `mod` (2^32 - 1)
```

```haskell
let tripleRands = State (\s0 ->
        let (_, s1)  = runState (update rand) s0
            (v1, _)  = runState get s1
            (_,  s2) = runState (update rand) s1
            (v2, _)  = runState get s2
            (_, s3)  = runState (update rand) s2
            (v3, _)  = runState get s3
        in ((v1,v2,v3),s3))
```

```haskell
   (evalState tripleRands 31337)
```

```
[MSFT] /tmp runghc Lecture22.hs
(526680959,4294250213,832904711)
[MSFT] /tmp 
```

Last time...

(note: to emphasize the fact that the state element in the tuple is private, I implemented `evalState`, which just returns the first element in the tuple.)

```haskell
rand :: Integral a => a -> a
rand seed = (7^5 * seed) `mod` (2^32 - 1)
```

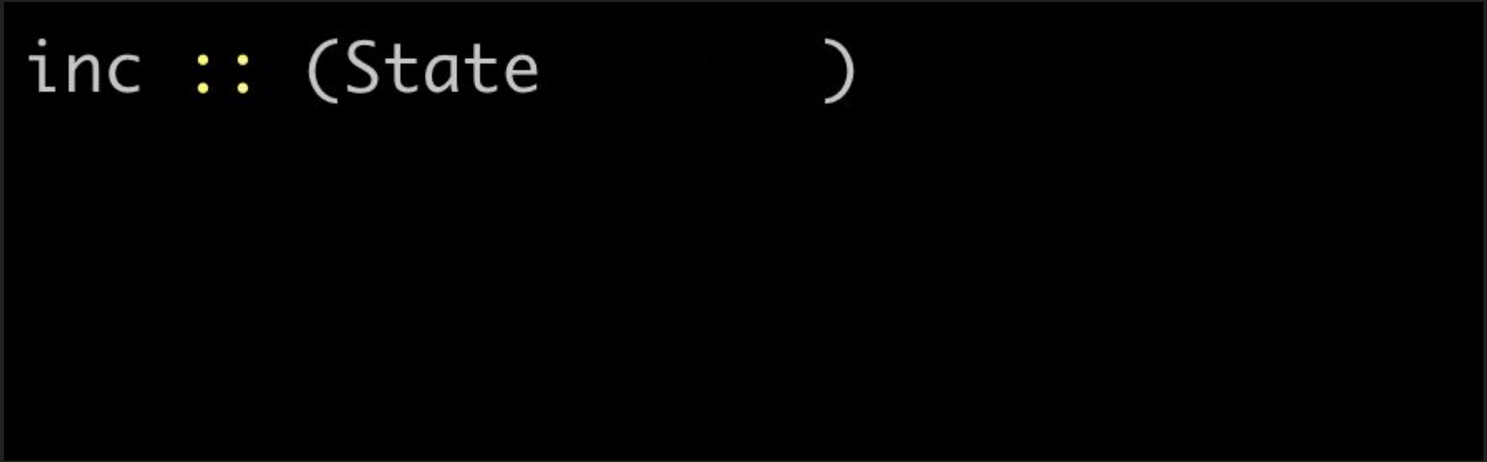
```haskell
let tripleRands = State (\s0 ->
        let (_, s1)  = runState (update rand) s0
            (v1, _)  = runState get s1
            (_,  s2) = runState (update rand) s1
            (v2, _)  = runState get s2
            (_, s3)  = runState (update rand) s2
            (v3, _)  = runState get s3
        in ((v1,v2,v3),s3))
```

```haskell
   (evalState tripleRands 31337)
```

```
[MSFT] /tmp runghc Lecture22.hs
(526680959,4294250213,832904711)
[MSFT] /tmp 
```

A simple stateful computation

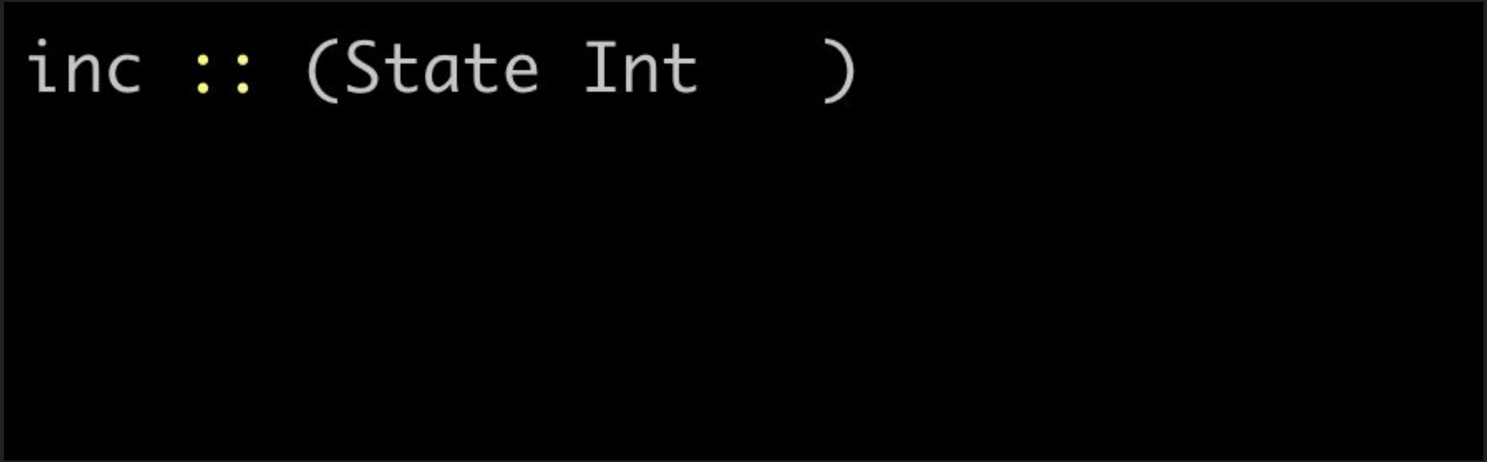
Let's write an expression that consumes a State whose state is some integer, and simply increments the State's state.



```
inc :: (State        )
```

A simple stateful computation

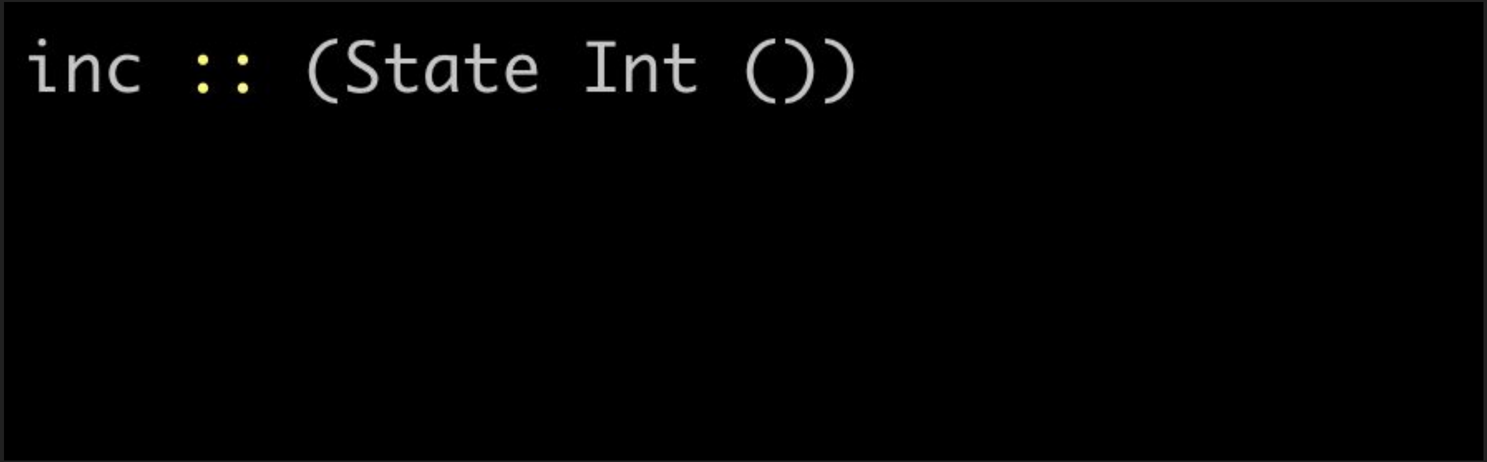
Let's write an expression that consumes a State whose state is some integer, and simply increments the State's state.



```
inc :: (State Int   )
```

A simple stateful computation

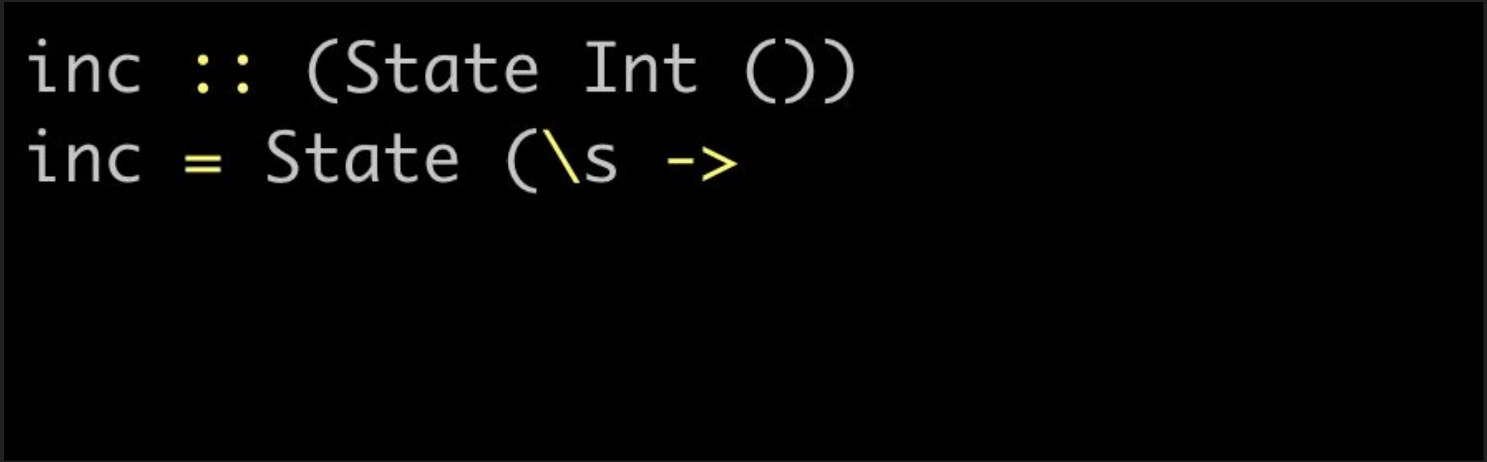
Let's write an expression that consumes a State whose state is some integer, and simply increments the State's state.



```
inc :: (State Int ())
```

A simple stateful computation

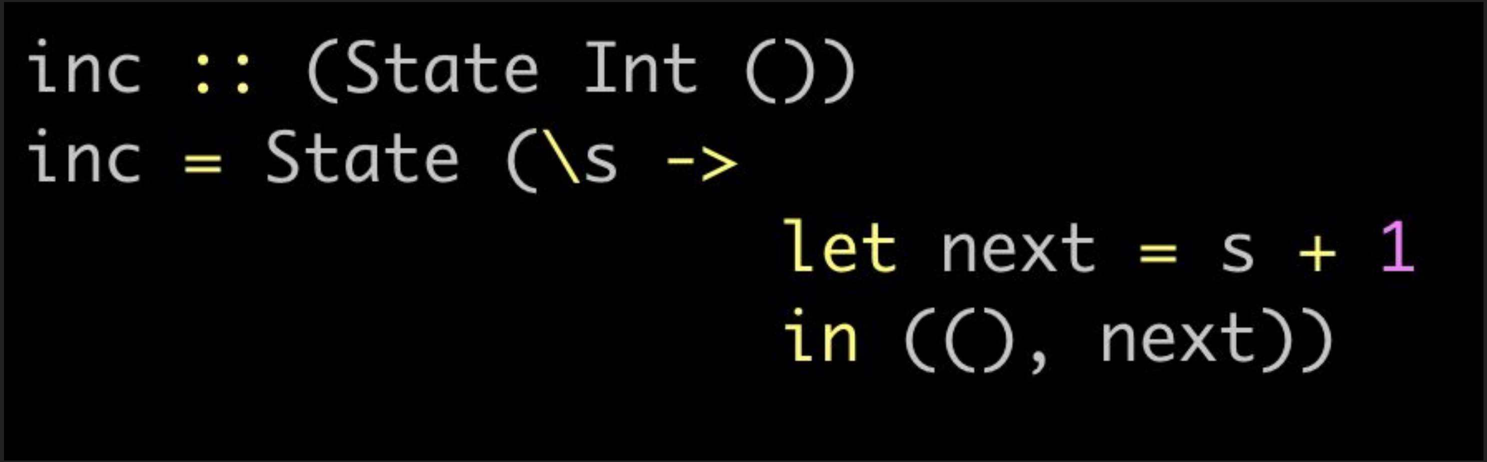
Let's write an expression that consumes a State whose state is some integer, and simply increments the State's state.



```
inc :: (State Int ())
inc = State (\s ->
```

A simple stateful computation

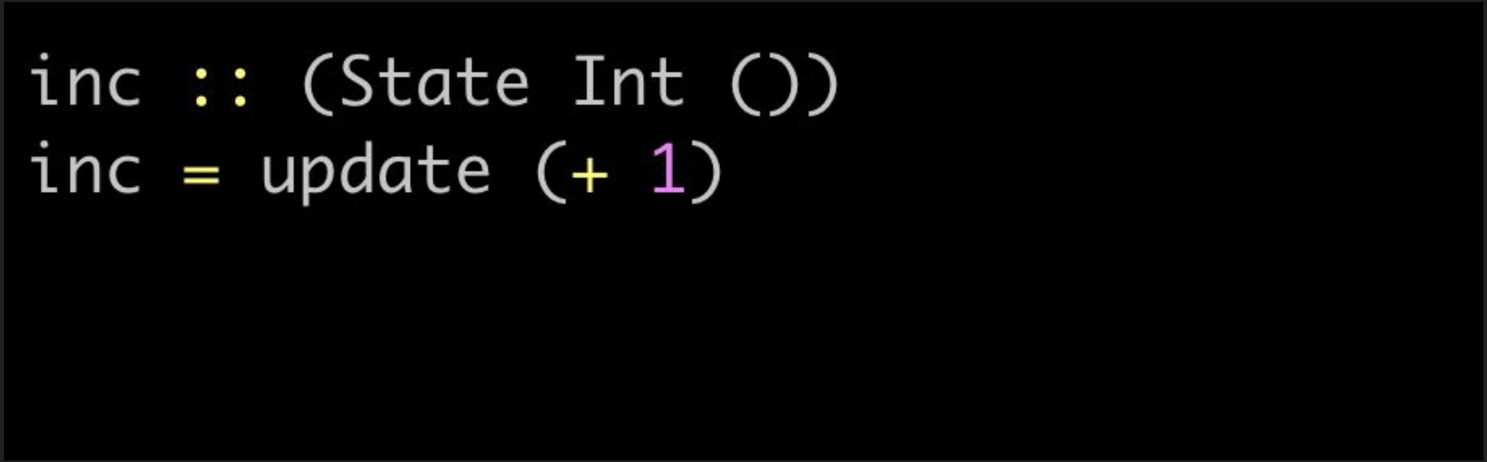
Let's write an expression that consumes a State whose state is some integer, and simply increments the State's state.



```haskell
inc :: (State Int ())
inc = State (\s ->
                   let next = s + 1
                   in ((), next))
```

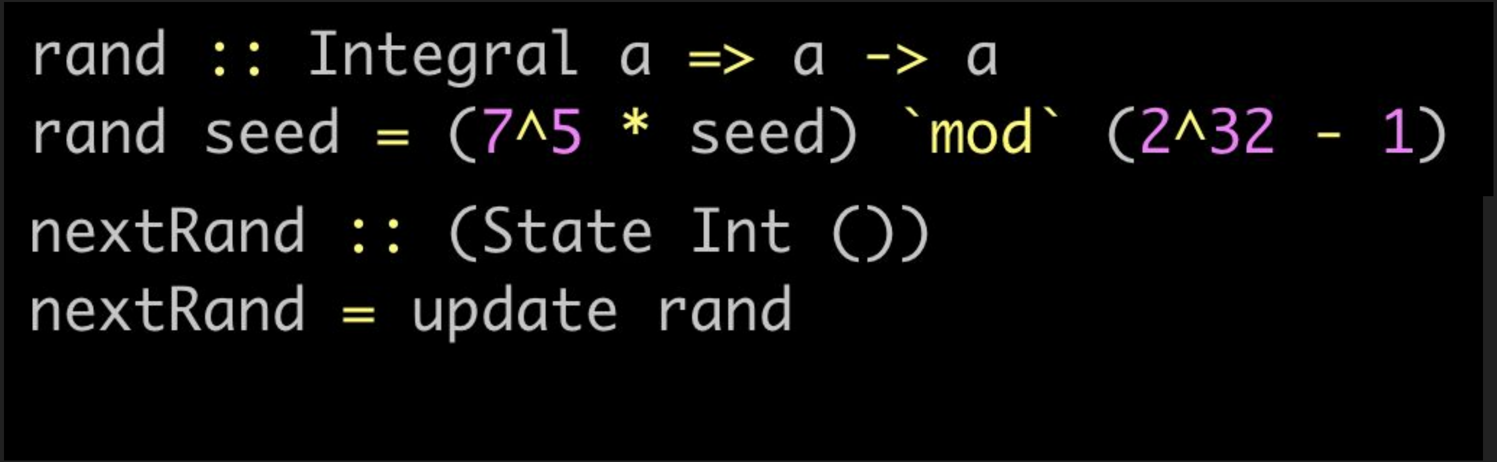
A simple stateful computation

Let's write an expression that consumes a State whose state is some integer, and simply increments the State's state.



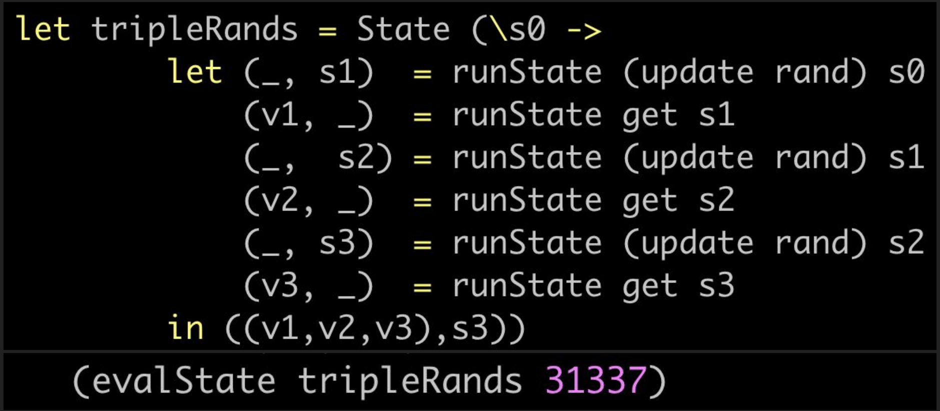
```haskell
inc :: (State Int ())
inc = update (+ 1)
```

A simple stateful computation



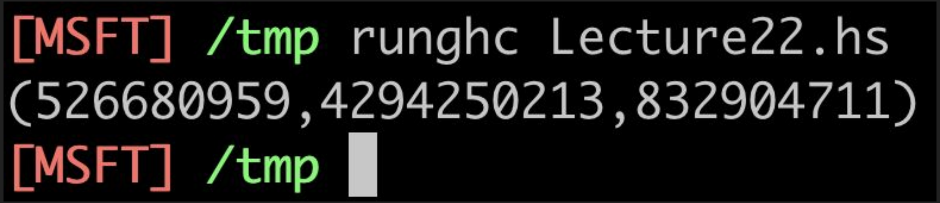
```haskell
rand :: Integral a => a -> a
rand seed = (7^5 * seed) `mod` (2^32 - 1)
nextRand :: (State Int ())
nextRand = update rand
```

We also noted the similarity between this implementation and do-notation from last class.



```haskell
let tripleRands = State (\s0 ->
        let (_, s1)   = runState (update rand) s0
            (v1, _)   = runState get s1
            (_,   s2) = runState (update rand) s1
            (v2, _)   = runState get s2
            (_, s3)   = runState (update rand) s2
            (v3, _)   = runState get s3
        in ((v1,v2,v3),s3))
```

```haskell
   (evalState tripleRands 31337)
```



```
[MSFT] /tmp runghc Lecture22.hs
(526680959,4294250213,832904711)
[MSFT] /tmp
```

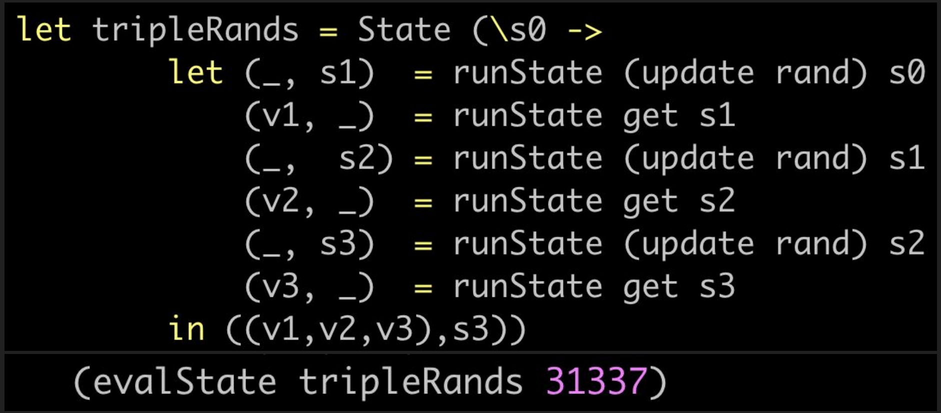
```
(uncons [10,20,50])
  >>= (\ p -> case p of (v1,xs) ->
    (uncons xs)
    >>= (\ p -> case p of (v2,_) ->
      return (v1+v2)))
```

Recall that do-notation offers syntactic sugar for (>>=) calls, so any datatype that implements the Monad typeclass can use do-notation.

```
do (v1,xs) <- (uncons [10,20,50])
   (v2,_)  <- (uncons xs)
   return (v1+v2)
```

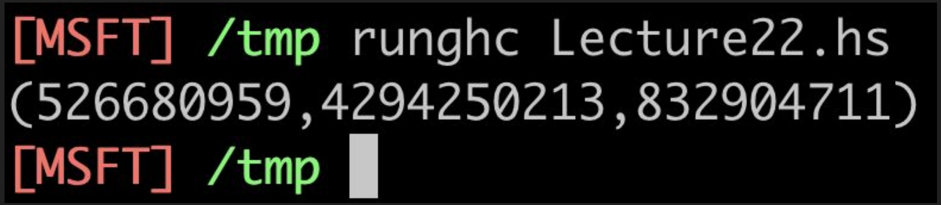
```
do v <- expr1; expr2 ...   ===   expr1 >>= (\ v -> expr2 ...)
```

This suggests that State would improved if we can come up with appropriate implementations of >>= and `return`, in order to monadify it.



```haskell
let tripleRands = State (\s0 ->
        let (_, s1)  = runState (update rand) s0
            (v1, _)  = runState get s1
            (_,  s2) = runState (update rand) s1
            (v2, _)  = runState get s2
            (_, s3)  = runState (update rand) s2
            (v3, _)  = runState get s3
        in ((v1,v2,v3),s3))
```

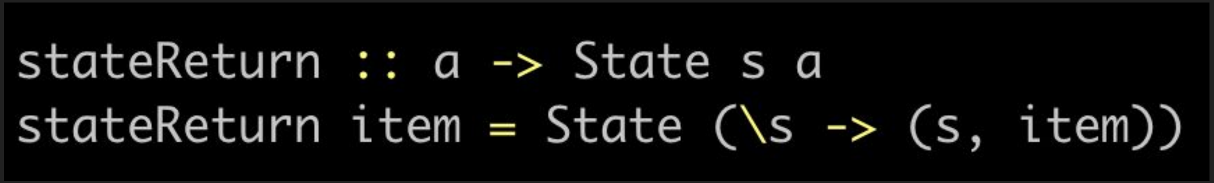
```haskell
   (evalState tripleRands 31337)
```



```
[MSFT] /tmp runghc Lecture22.hs
(526680959,4294250213,832904711)
[MSFT] /tmp
```

State as a monad?

Return is the easy one: It takes a value and produces a state such that the value becomes the State's value and the input state becomes the State's state. (got that?)



```haskell
stateReturn :: a -> State s a
stateReturn item = State (\s -> (s, item))
```

State as a monad?

But what should bind do for a State?

A more logical way of thinking about >>= for State is to treat it as "andThen", which was the original name we gave that function back in the day.

```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
```

This will be our type signature for `andThen`. Notice, critically, that the second argument consumes not the state, but the current value of the computation.

This is **different** from the function contained inside the State datatype, whose argument is the old state. We have to do our best to keep these straight!

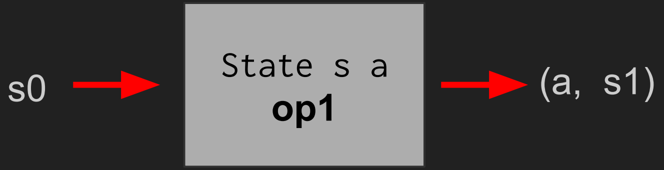
```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
```

Let's give the starting `(State s a)` an name, `op1`, since it's some "op"eration...

State s a
op1

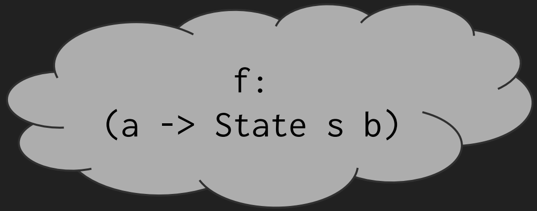
```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
```

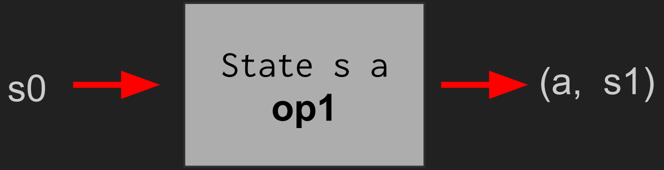
To apply an effect inside a State context, we supply it with the current state, it performs some operation, and produces the output value and the next state.



```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
```

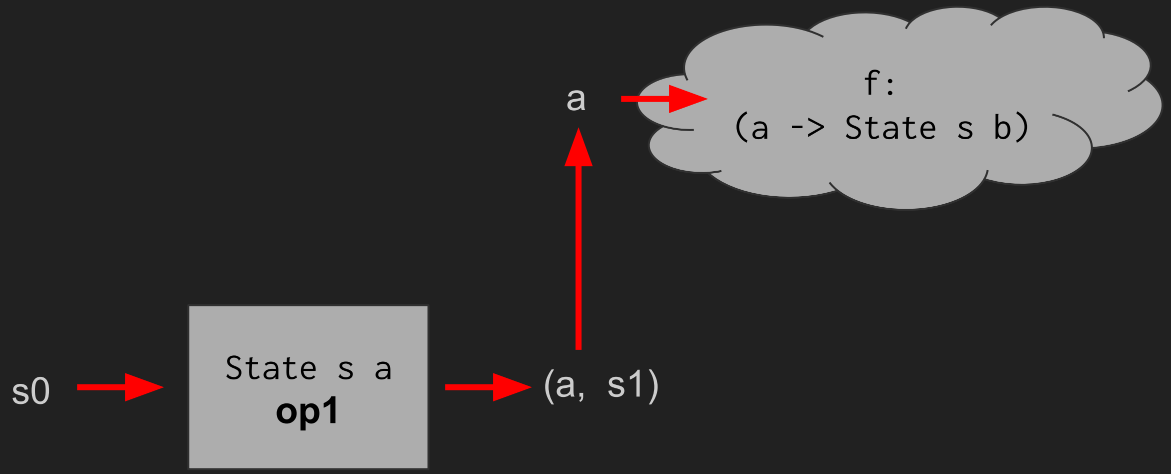
We also have some function that consumes an `a` and produces a new State.





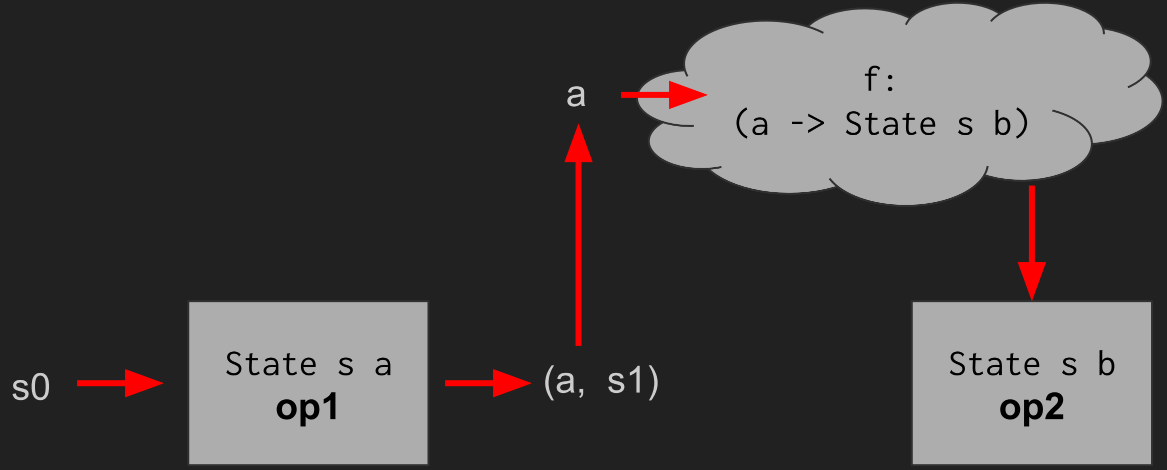
```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
```

We have an a from running `op1`, so if we pass that value into `f`...



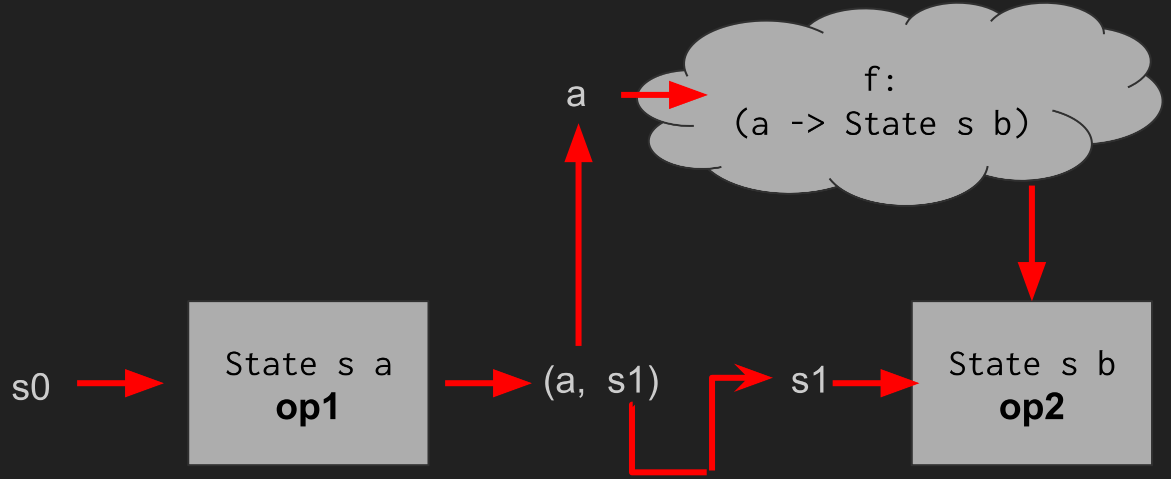
```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
```

...Which produces us another state!



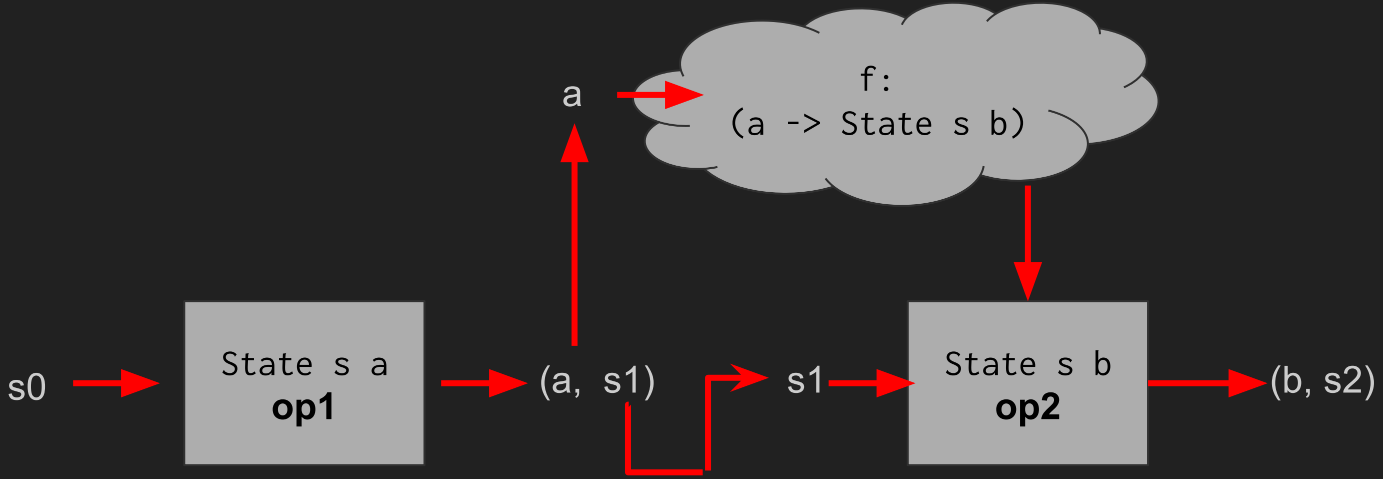
```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
```

Now, if we were to pass in the new updated state we got from `op1` into `op2`...



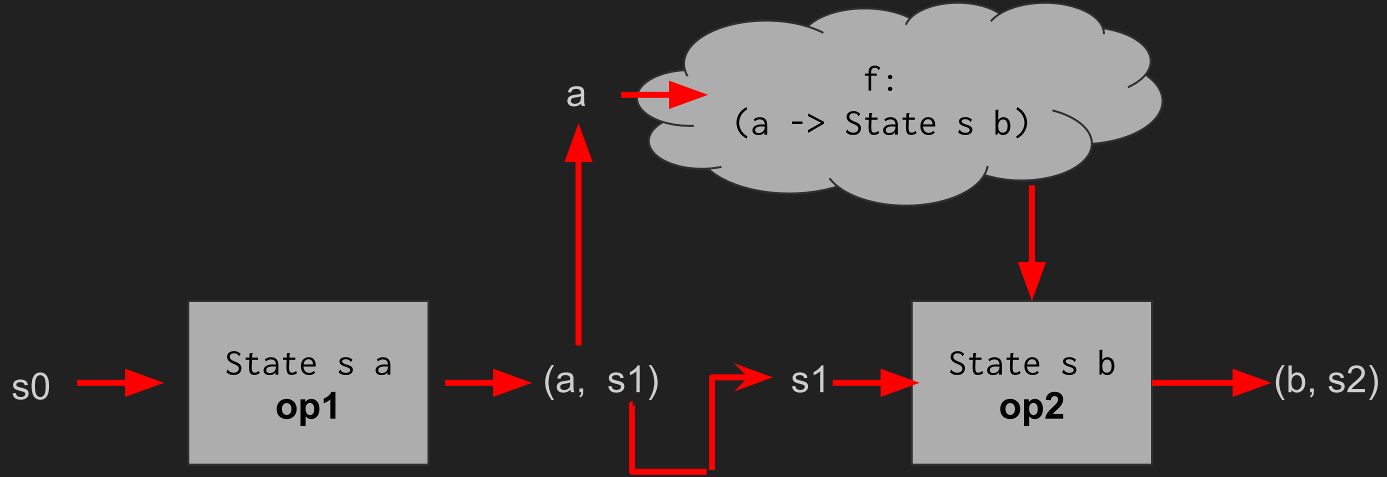
```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
```

We also have some function that consumes an `a` and produces a new State.



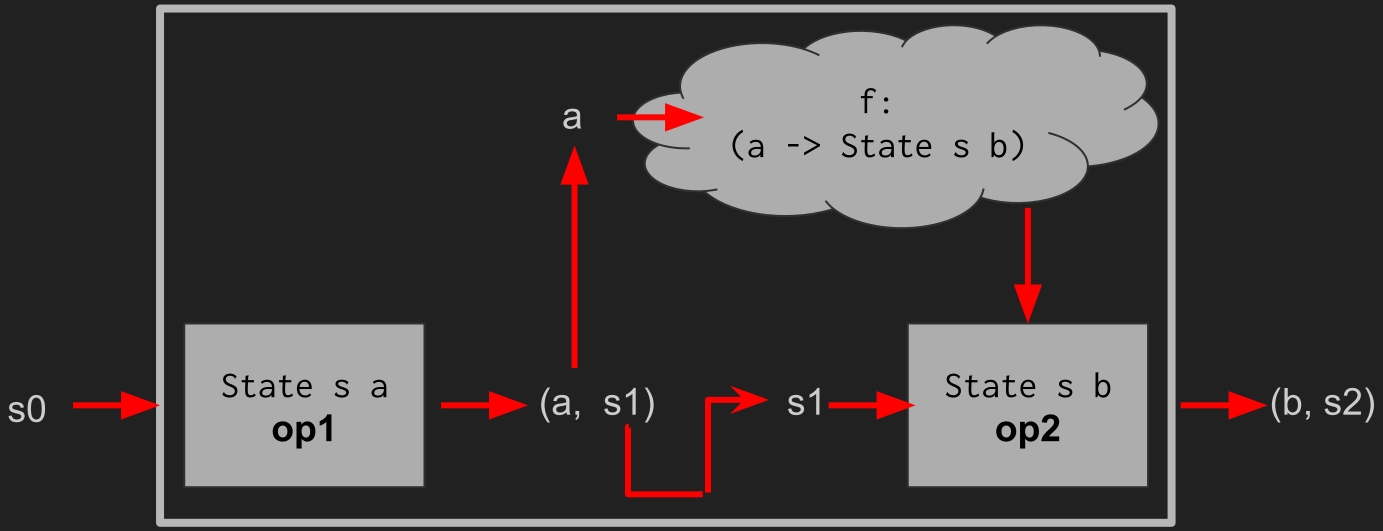
```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
```

So, what can we say about `andThen`?



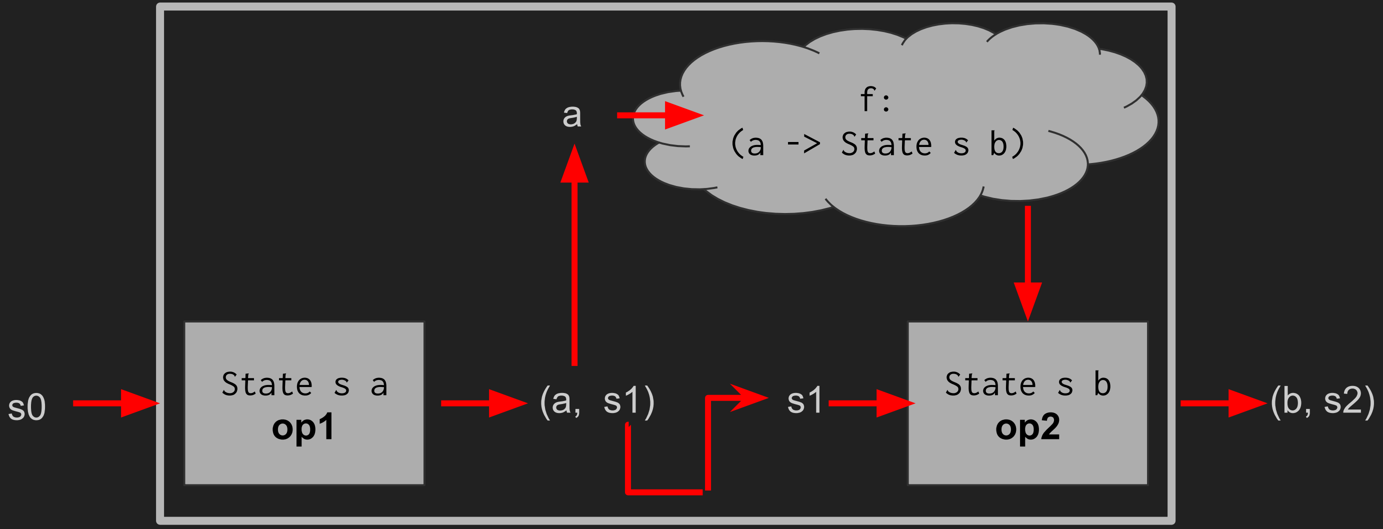
```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
```

`andThen` produces the stateful operation contained in this box.



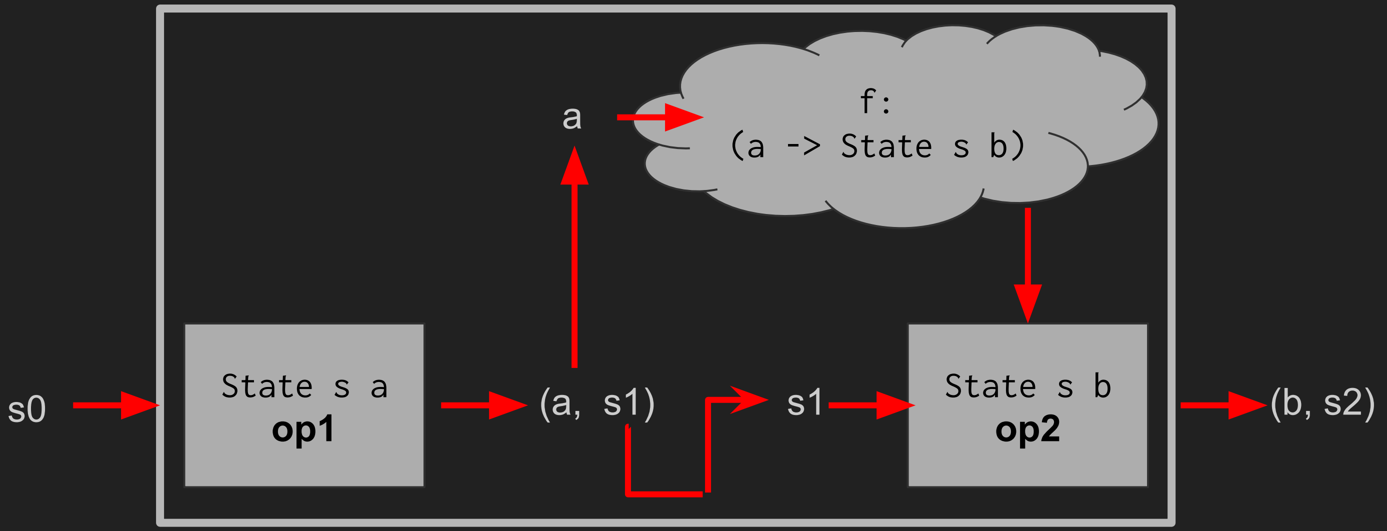
Let's implement bind and return for State, and then everything should fall out automatically from the typeclass system!

```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
```



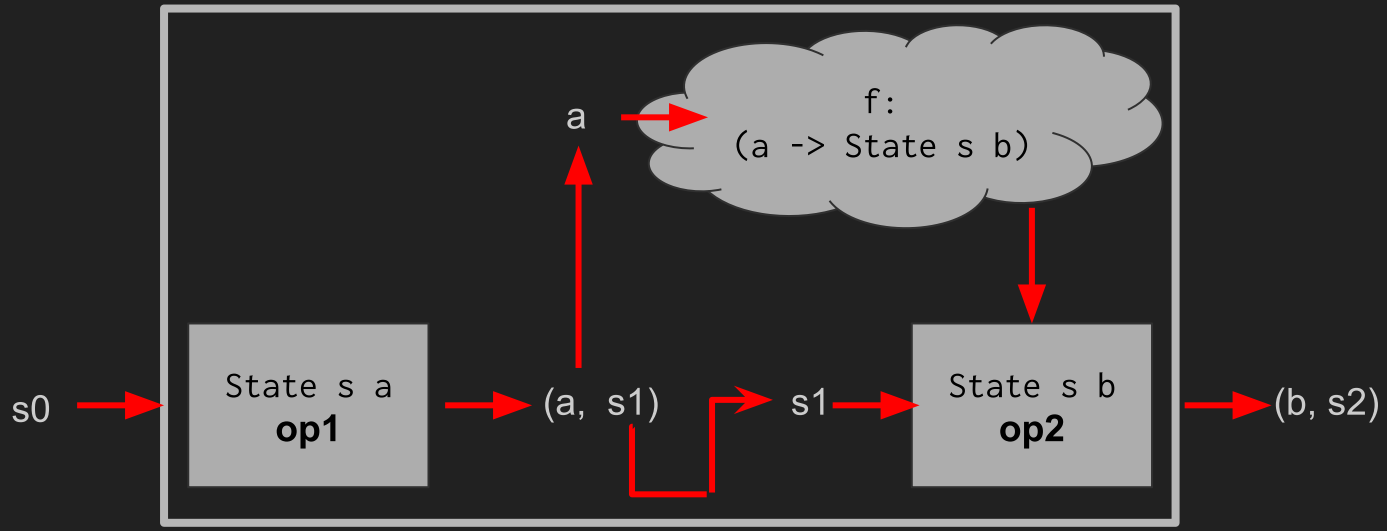
According to the diagram, we'll consume some initial state `s0`

```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
andThen op1 f = State(\s0 ->
```



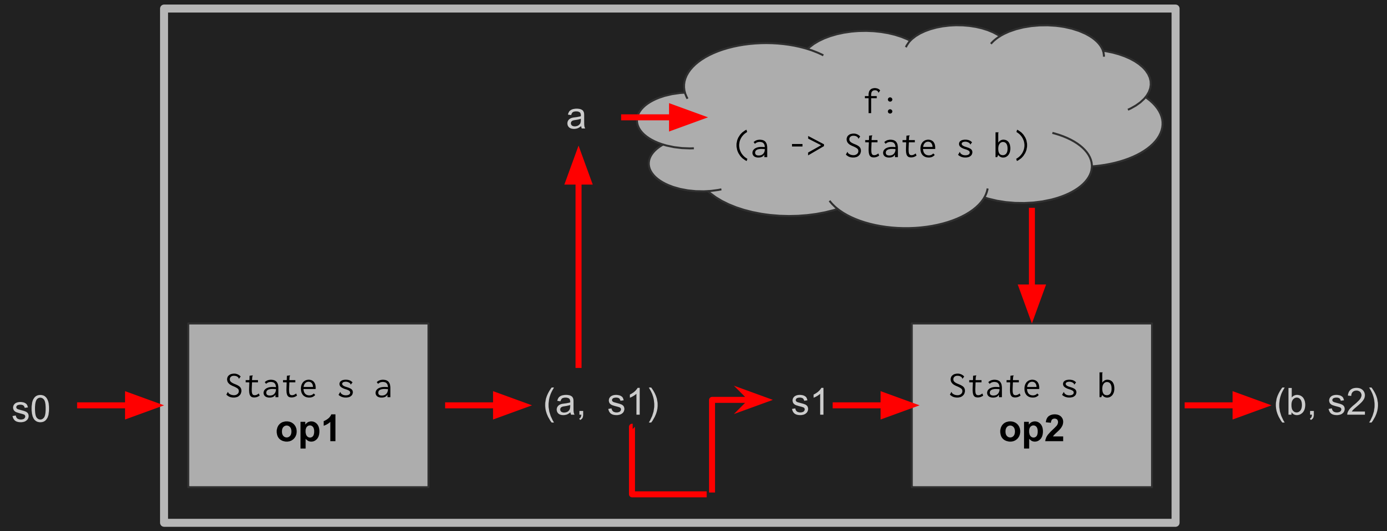
We'll run the supplied operation, producing an output value and the next state...

```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
andThen op1 f = State(\s0 ->
    let (x1, s1) = runState op1 s0
```



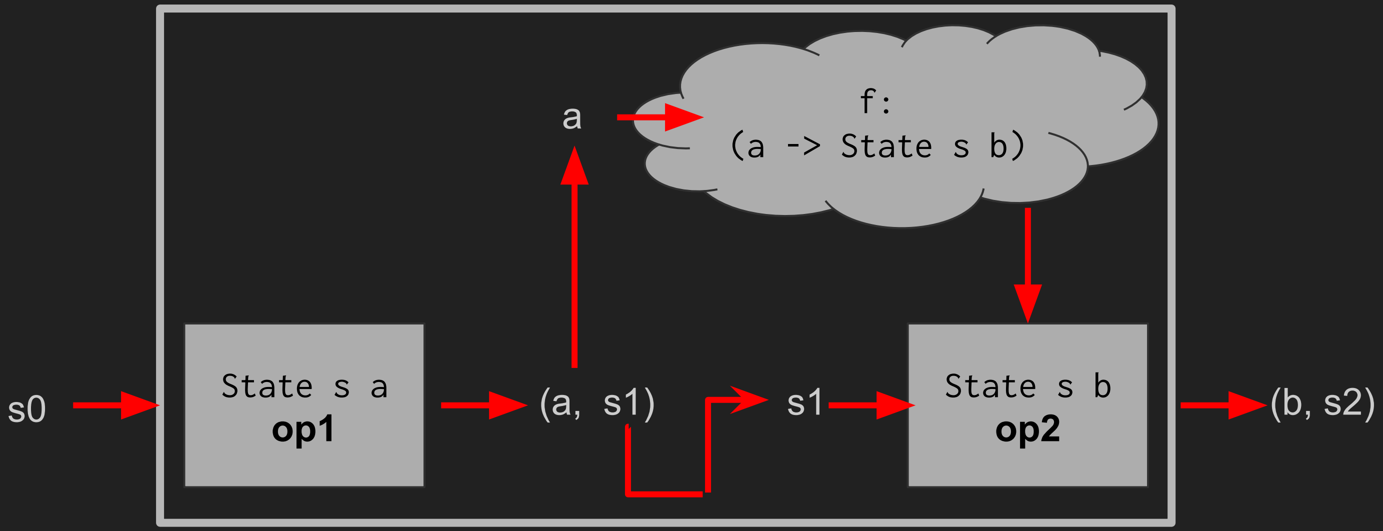
Next, we construct the next operation by applying the output of op1 to the supplied function...

```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
andThen op1 f = State(\s0 ->
    let (x1, s1) = runState op1 s0
        op2 = f x1
```



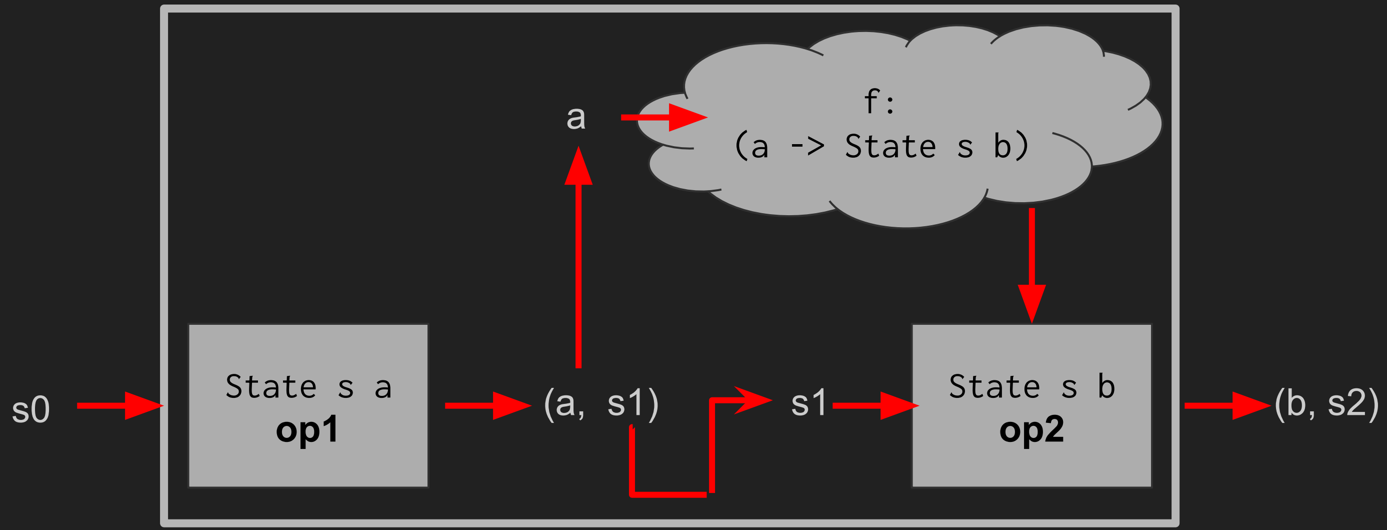
Now that we have a second operation, we'll run it **with the state we generated from** op1.

```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
andThen op1 f = State(\s0 ->
    let (x1, s1) = runState op1 s0
        op2 = f x1
        (x2, s2) = runState op2 s1
```



That generates an output value and state tuple, which is what the whole andThen function produces.

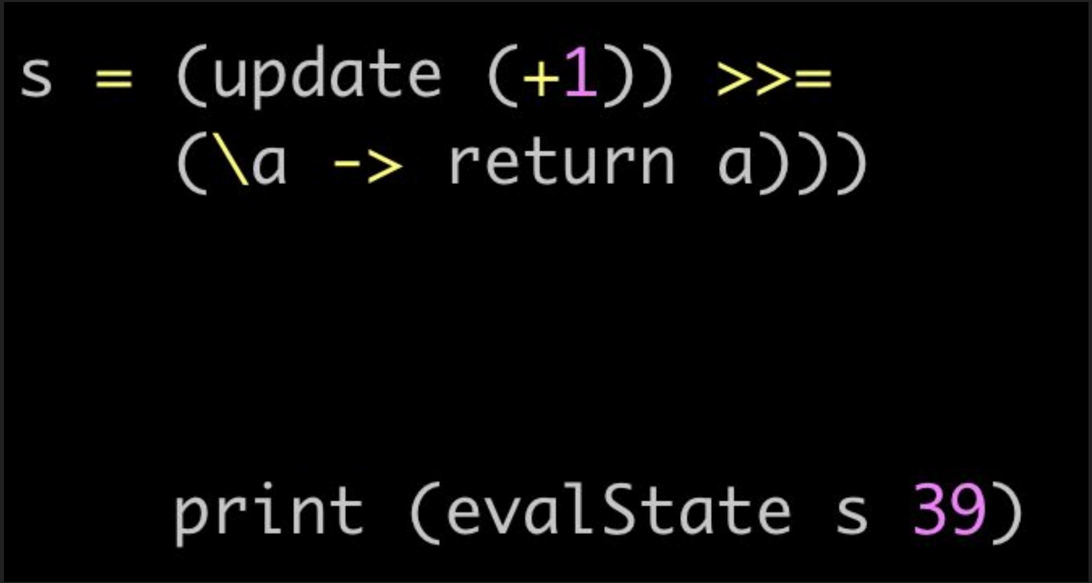
```
andThen :: (State s a) -> (a -> State s b) -> (State s b)
andThen op1 f = State(\s0 ->
    let (x1, s1) = runState op1 s0
        op2 = f x1
        (x2, s2) = runState op2 s1
    in
        (x2, s2))
```



inc in action

Here's an abbreviated program that, given an initial state of 39, increments the state and returns the State's value.

What does this produce?

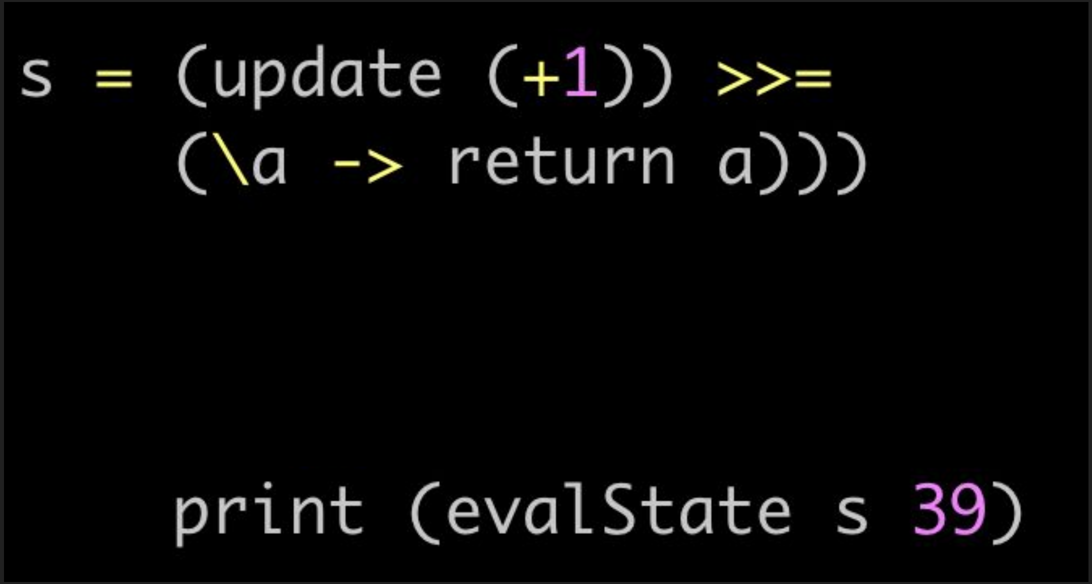


```
s = (update (+1)) >>=
    (\a -> return a)))



    print (evalState s 39)
```

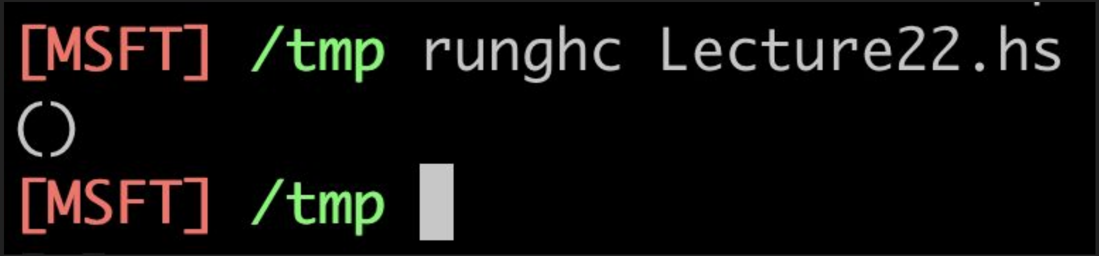
inc … inaction

Whoops! We forgot to actually read the state back!

The update calls incremented the state but set the value to () (also known as `unit`), so `a` is of type `Unit`.



```
s = (update (+1)) >>=
    (\a -> return a)))



    print (evalState s 39)
```

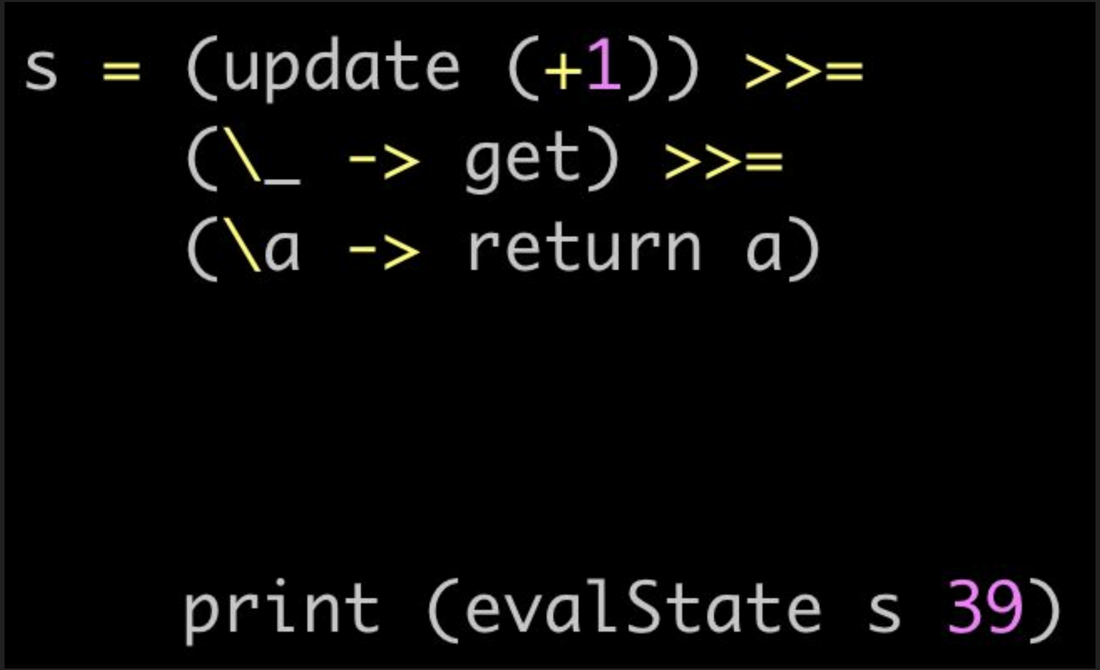


```
[MSFT] /tmp runghc Lecture22.hs
()
[MSFT] /tmp
```

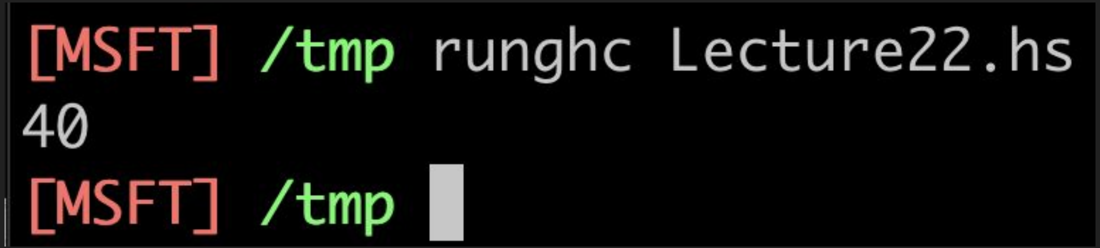
inc in action

This State action is composed of three operations:

- Incrementing the state (from 39 to 40)
- Reading the state into the tuple's "output" slot
- Lifting the output value into a final State



```haskell
s = (update (+1)) >>=
    (\_ -> get) >>=
    (\a -> return a)


    print (evalState s 39)
```



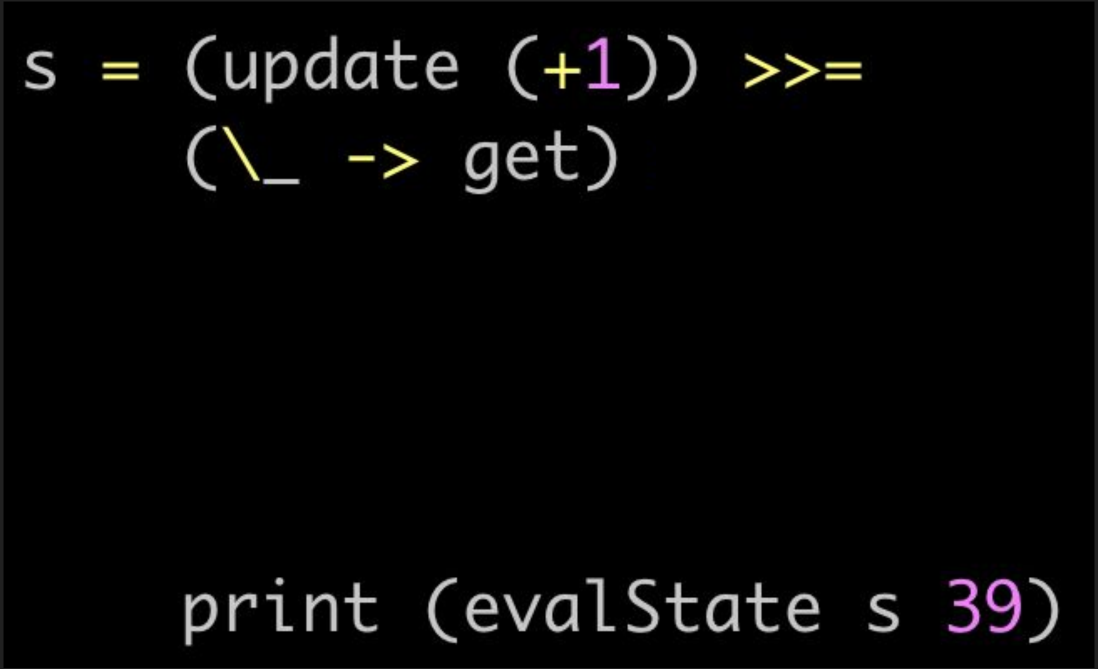
```
[MSFT] /tmp runghc Lecture22.hs
40
[MSFT] /tmp 
```

inc in action

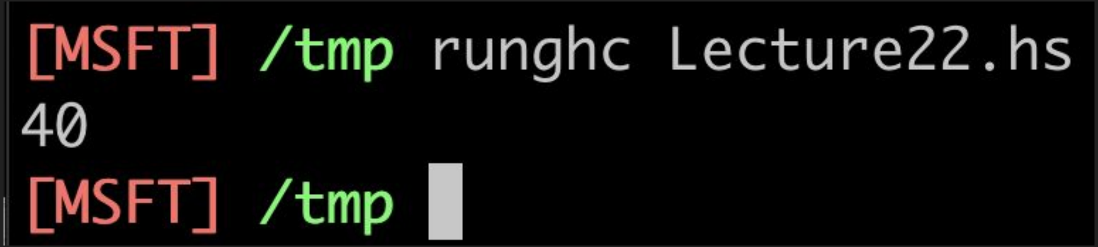
This State action is composed of **two** operations:

- Incrementing the state (from 39 to 40)
- Reading the state into the tuple's "output" slot
- Lifting the output value into a final State

...that third operation is actually redundant!



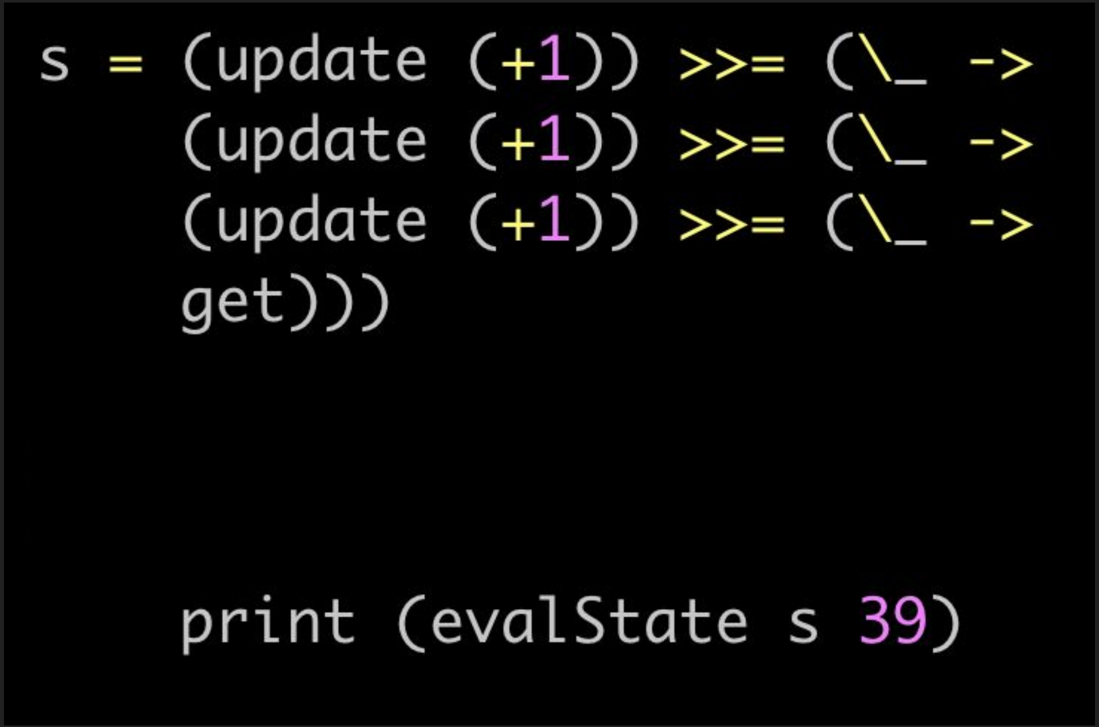
```haskell
s = (update (+1)) >>=
    (\_ -> get)




    print (evalState s 39)
```



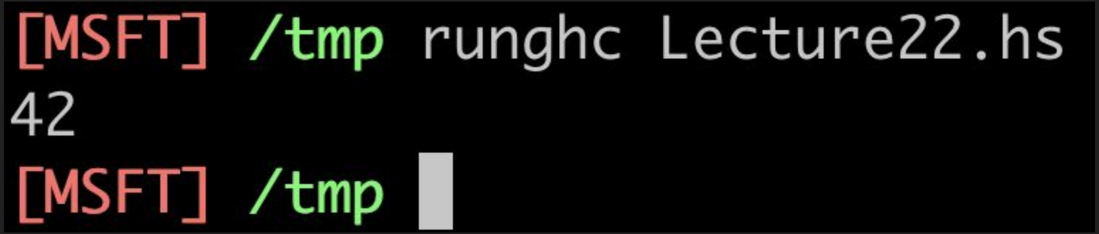
```
[MSFT] /tmp runghc Lecture22.hs
40
[MSFT] /tmp █
```

inc in action

We can perform multiple mutations within >>= calls, of course.



```
s = (update (+1)) >>= (\_ ->
    (update (+1)) >>= (\_ ->
    (update (+1)) >>= (\_ ->
    get)))



    print (evalState s 39)
```

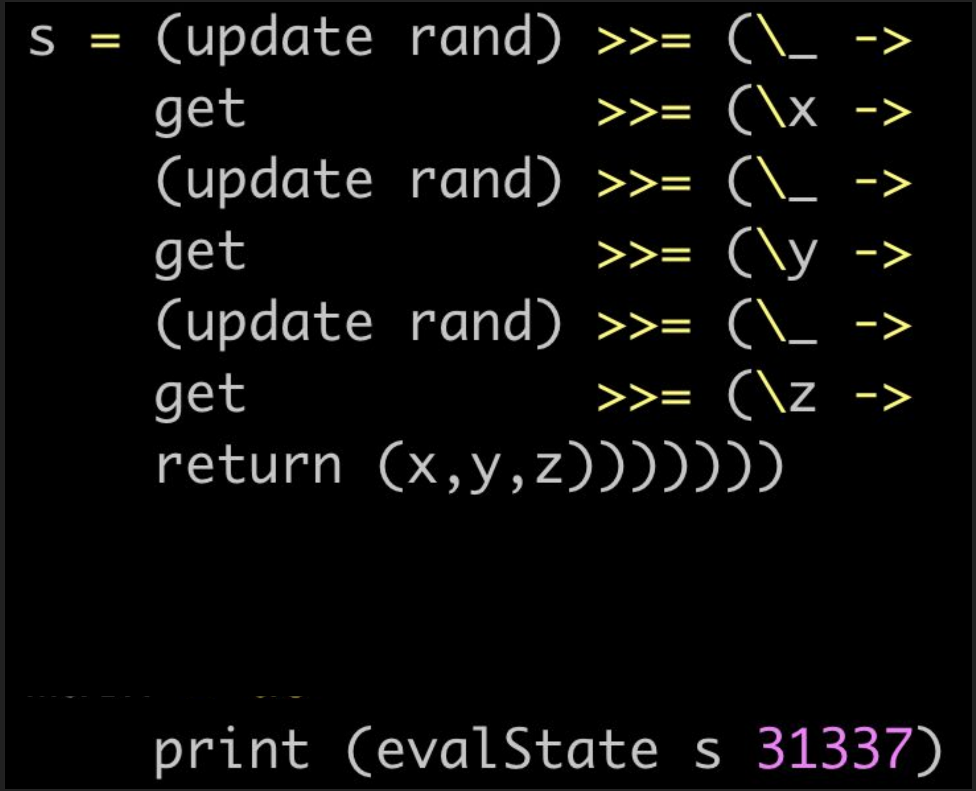


```
[MSFT] /tmp runghc Lecture22.hs
42
[MSFT] /tmp
```

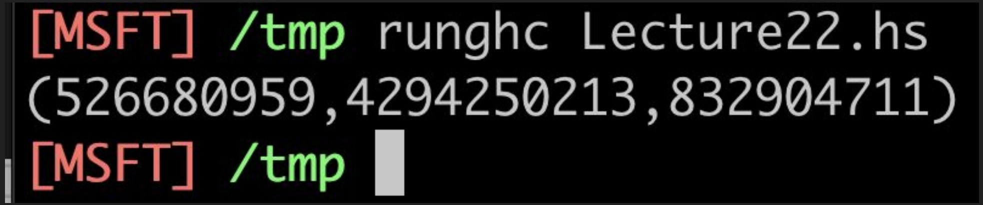
What good is return?

By interleaving calls to get, we're able to bind values by the argument to the lambda function that we pass to >>=

We use return to compose the final value (in this case, forming a tuple from `x,y,z`)



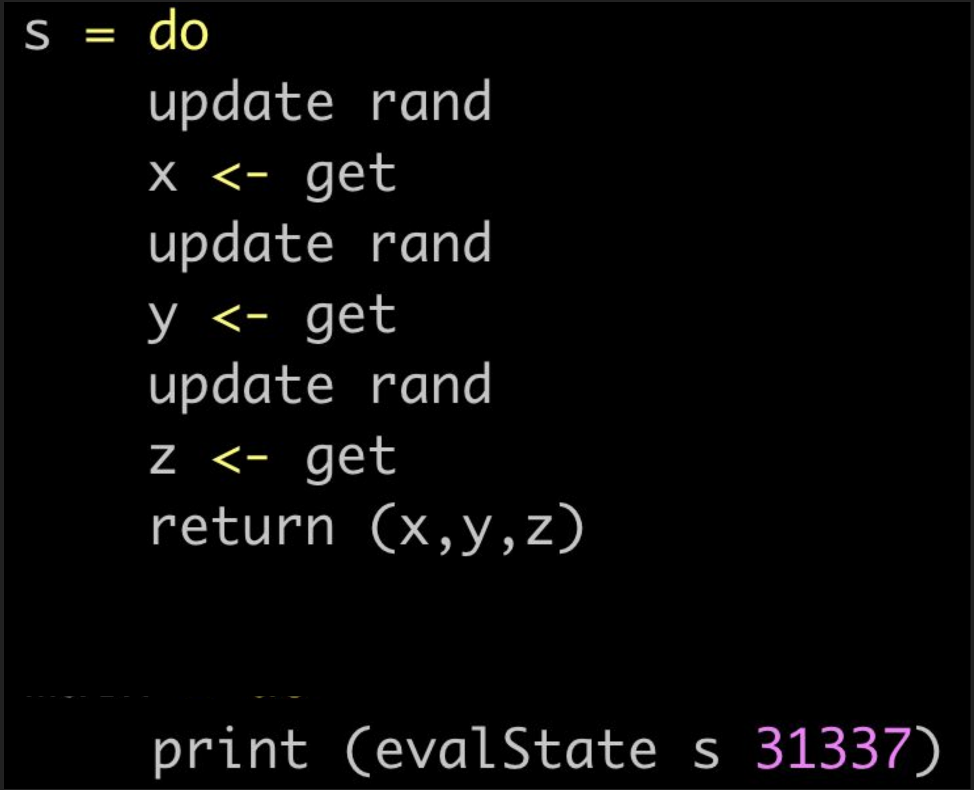
```haskell
s = (update rand) >>= (\_ ->
    get           >>= (\x ->
    (update rand) >>= (\_ ->
    get           >>= (\y ->
    (update rand) >>= (\_ ->
    get           >>= (\z ->
    return (x,y,z))))))

    print (evalState s 31337)
```



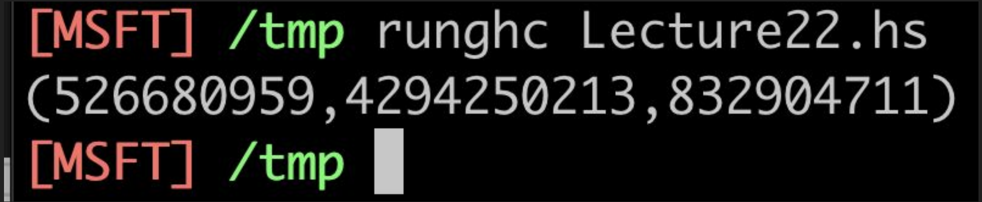
```
[MSFT] /tmp runghc Lecture22.hs
(526680959,4294250213,832904711)
[MSFT] /tmp
```

rand and do

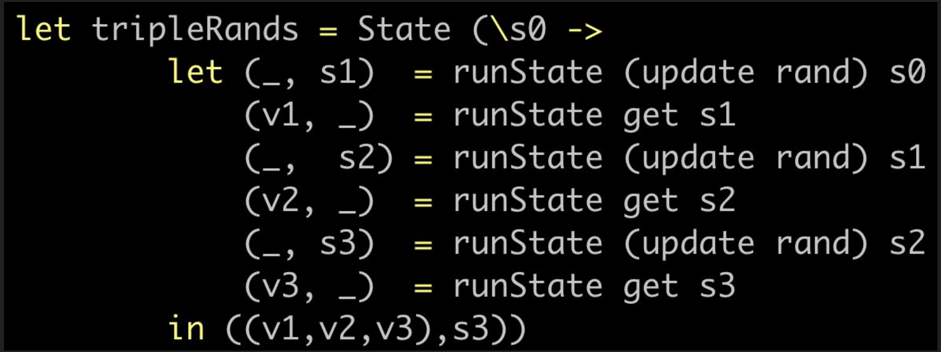
And because we now have implementations for return and >>=, we can write this expression with do-notation for our final transformation.



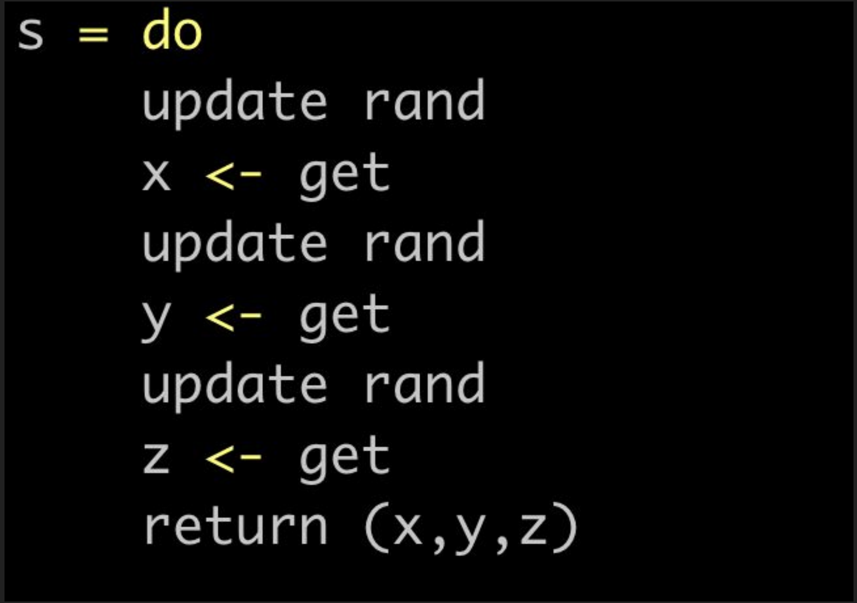
```haskell
s = do
    update rand
    x <- get
    update rand
    y <- get
    update rand
    z <- get
    return (x,y,z)


    print (evalState s 31337)
```



```
[MSFT] /tmp runghc Lecture22.hs
(526680959,4294250213,832904711)
[MSFT] /tmp
```



```haskell
let tripleRands = State (\s0 ->
        let (_, s1)  = runState (update rand) s0
            (v1, _)  = runState get s1
            (_,  s2) = runState (update rand) s1
            (v2, _)  = runState get s2
            (_, s3)  = runState (update rand) s2
            (v3, _)  = runState get s3
        in ((v1,v2,v3),s3))
```



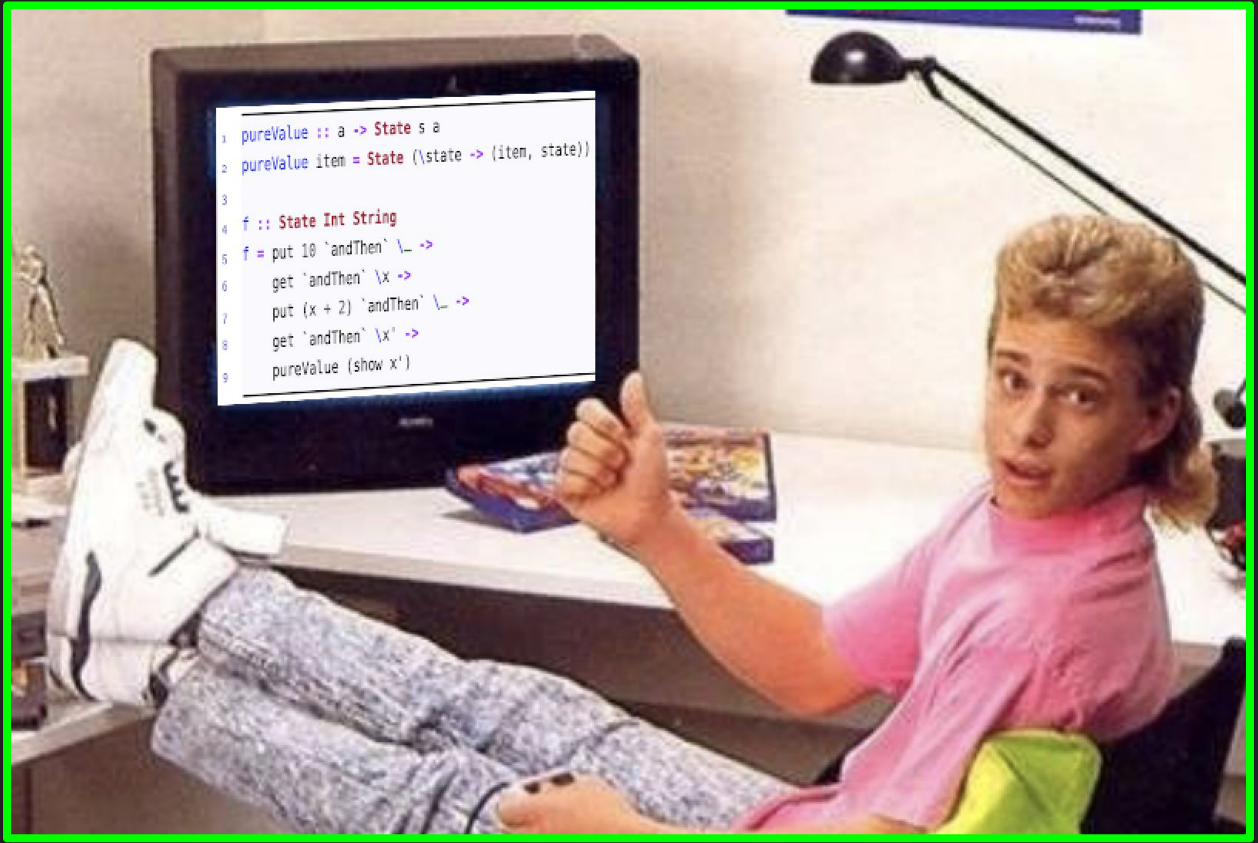
```haskell
s = do
      update rand
      x <- get
      update rand
      y <- get
      update rand
      z <- get
      return (x,y,z)
```

Compare what we began with today vs what we've come up with now. Unlike the expression on the left, the expression on the right has no references to the `State` data constructor, `runState`, or any intermediary random seed values or underbarred lambda parameters.

```
let tripleRands = State (\s0 ->
        let (_, s1)  = runState (update rand) s0
            (v1, _)  = runState get s1
            (_,  s2) = runState (update rand) s1
            (v2, _)  = runState get s2
            (_, s3)  = runState (update rand) s2
            (v3, _)  = runState get s3
        in ((v1,v2,v3),s3))
```

```
s = do
    update rand
    x <- get
    update rand
    y <- get
    update rand
    z <- get
    return (x,y,z)
```

From our perspective, it's as if `rand()` is actually a side-effecting function.

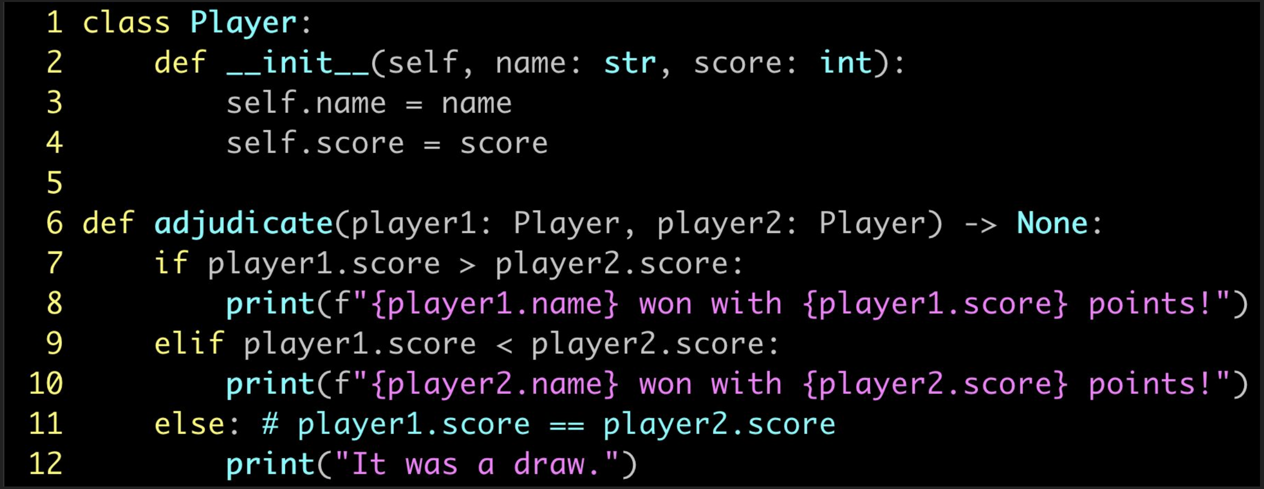


```haskell
pureValue :: a -> State s a
pureValue item = State (\state -> (item, state))

f :: State Int String
f = put 10 `andThen` \_ ->
    get `andThen` \x ->
    put (x + 2) `andThen` \_ ->
    get `andThen` \x' ->
    pureValue (show x')
```

From our perspective, it's as if `rand()` is actually a side-effecting function.

Another form of non-determinism: Input & Output

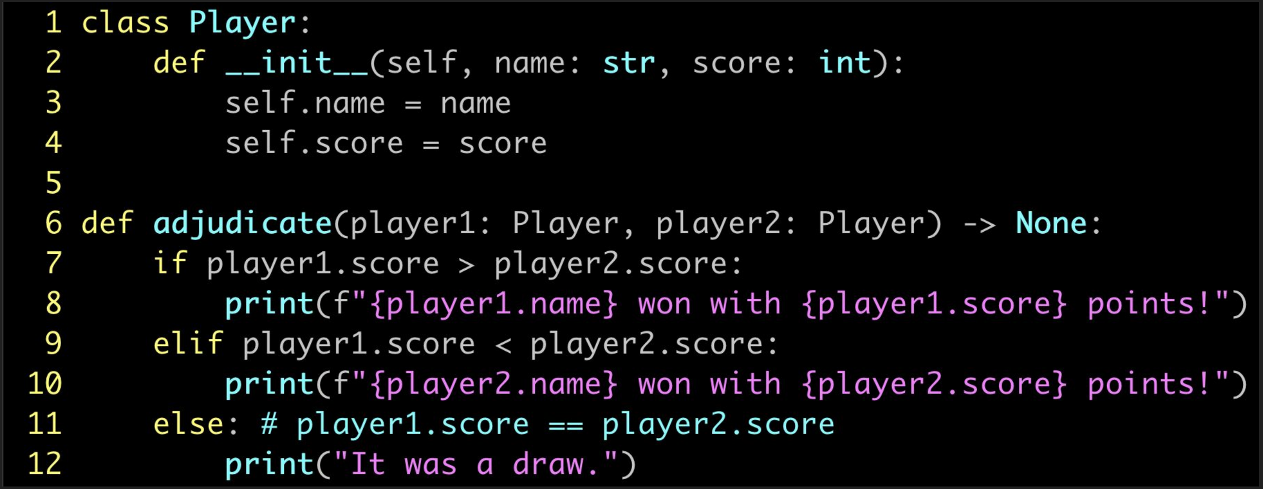
Here's a function that computes which of the two given Players have the greater score, and prints the result to the user.



```python
class Player:
    def __init__(self, name: str, score: int):
        self.name = name
        self.score = score

def adjudicate(player1: Player, player2: Player) -> None:
    if player1.score > player2.score:
        print(f"{player1.name} won with {player1.score} points!")
    elif player1.score < player2.score:
        print(f"{player2.name} won with {player2.score} points!")
    else: # player1.score == player2.score
        print("It was a draw.")
```

Two facts about `adjudicate`:

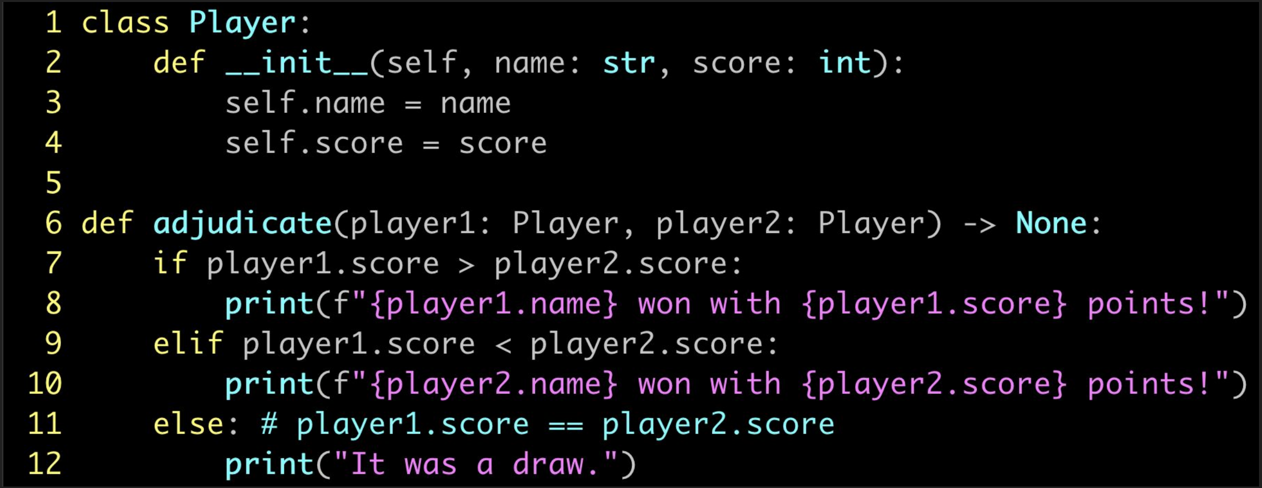
- `adjudicate` is a void function (in Python: it returns the null-like value `None`)
- `adjudicate` is not referentially transparent (it prints output to the screen)



```python
class Player:
    def __init__(self, name: str, score: int):
        self.name = name
        self.score = score

def adjudicate(player1: Player, player2: Player) -> None:
    if player1.score > player2.score:
        print(f"{player1.name} won with {player1.score} points!")
    elif player1.score < player2.score:
        print(f"{player2.name} won with {player2.score} points!")
    else: # player1.score == player2.score
        print("It was a draw.")
```

Another, perhaps less-intuitive fact about adjudicate: **it does two separate things**, and only one of those two things causes the side-effect operation.

Determining which of the two players won can be done in a pure, referentially-transparent way. It's only printing out the result that breaks our rules.



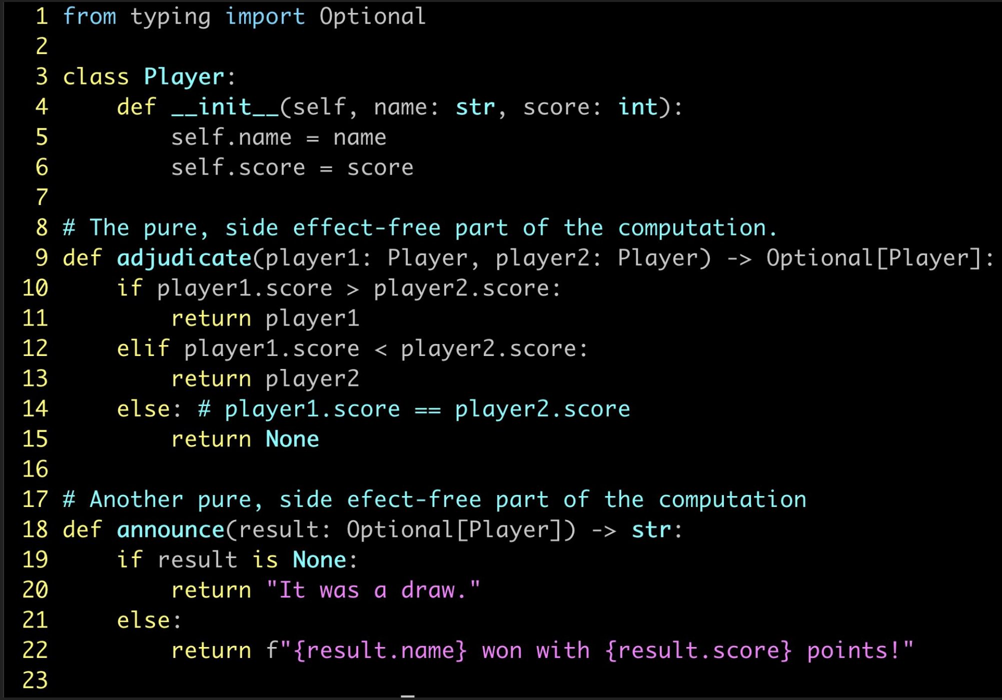
```python
class Player:
    def __init__(self, name: str, score: int):
        self.name = name
        self.score = score

def adjudicate(player1: Player, player2: Player) -> None:
    if player1.score > player2.score:
        print(f"{player1.name} won with {player1.score} points!")
    elif player1.score < player2.score:
        print(f"{player2.name} won with {player2.score} points!")
    else: # player1.score == player2.score
        print("It was a draw.")
```

Here's a program that's slightly longer, but splits up the pure and impure parts of the program.

adjudicate still is responsible for determining which, if any, player won (notice our use of an optional type to report this!) but announce is responsible for printing the string representation of the final result.

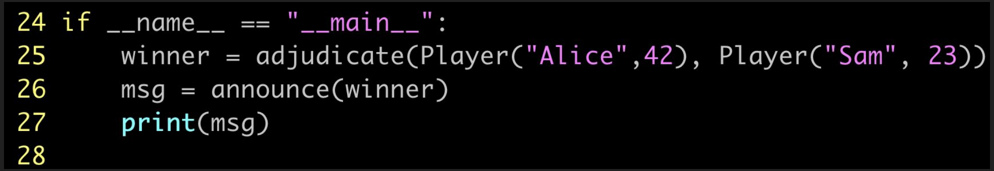
```python
from typing import Optional

class Player:
    def __init__(self, name: str, score: int):
        self.name = name
        self.score = score

# The pure, side effect-free part of the computation.
def adjudicate(player1: Player, player2: Player) -> Optional[Player]:
    if player1.score > player2.score:
        return player1
    elif player1.score < player2.score:
        return player2
    else: # player1.score == player2.score
        return None

# The side-effecting part of the computation.
def announce(result: Optional[Player]) -> None:
    if result is None:
        print("It was a draw.")
    else:
        print(f"{result.name} won with {result.score} points!")
```

We can even take it one step further! The announce function can just return the string form of the announcement, and leave it up to the caller of announce to perform the side-effecting operation of printing it out.

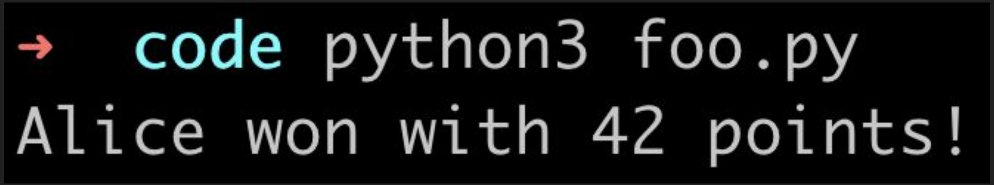


```python
from typing import Optional

class Player:
    def __init__(self, name: str, score: int):
        self.name = name
        self.score = score

# The pure, side effect-free part of the computation.
def adjudicate(player1: Player, player2: Player) -> Optional[Player]:
    if player1.score > player2.score:
        return player1
    elif player1.score < player2.score:
        return player2
    else: # player1.score == player2.score
        return None

# Another pure, side efect-free part of the computation
def announce(result: Optional[Player]) -> str:
    if result is None:
        return "It was a draw."
    else:
        return f"{result.name} won with {result.score} points!"

```

In this example, we factored an impure procedure into one or more pure "core" functions, and deferred doing actual impure operations until we absolutely had to.



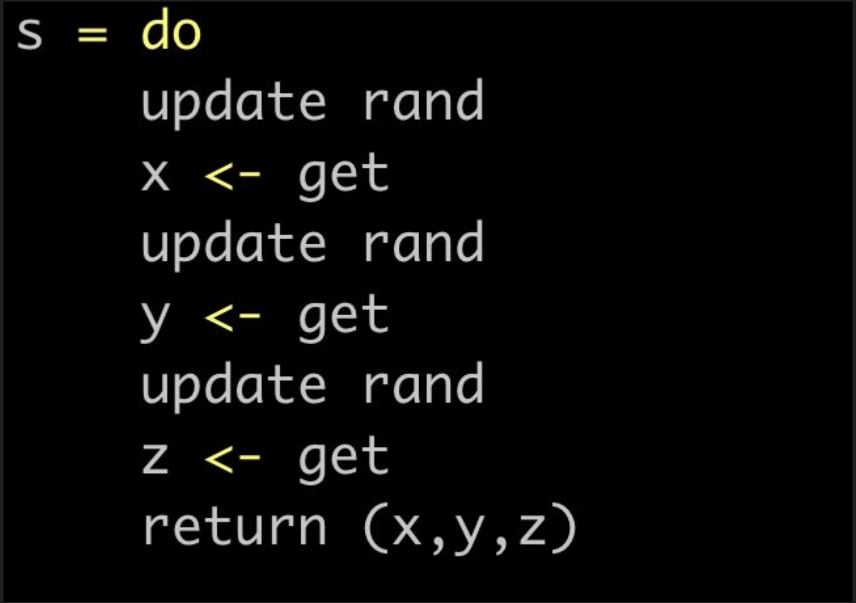
```
if __name__ == "__main__":
    winner = adjudicate(Player("Alice",42), Player("Sam", 23))
    msg = announce(winner)
    print(msg)

```



```
➜  code python3 foo.py
Alice won with 42 points!
```

In a very similar way, we were able to transform rand from being an impure function with a global mutable variable to the pure core of an impure computation.

This suggests that, just like we model mutation with the State monad, we can model IO with another monad



```
s = do
    update rand
    x <- get
    update rand
    y <- get
    update rand
    z <- get
    return (x,y,z)
```

The IO Monad

We are going to take a different approach learning the IO monad to how we built the State monad.

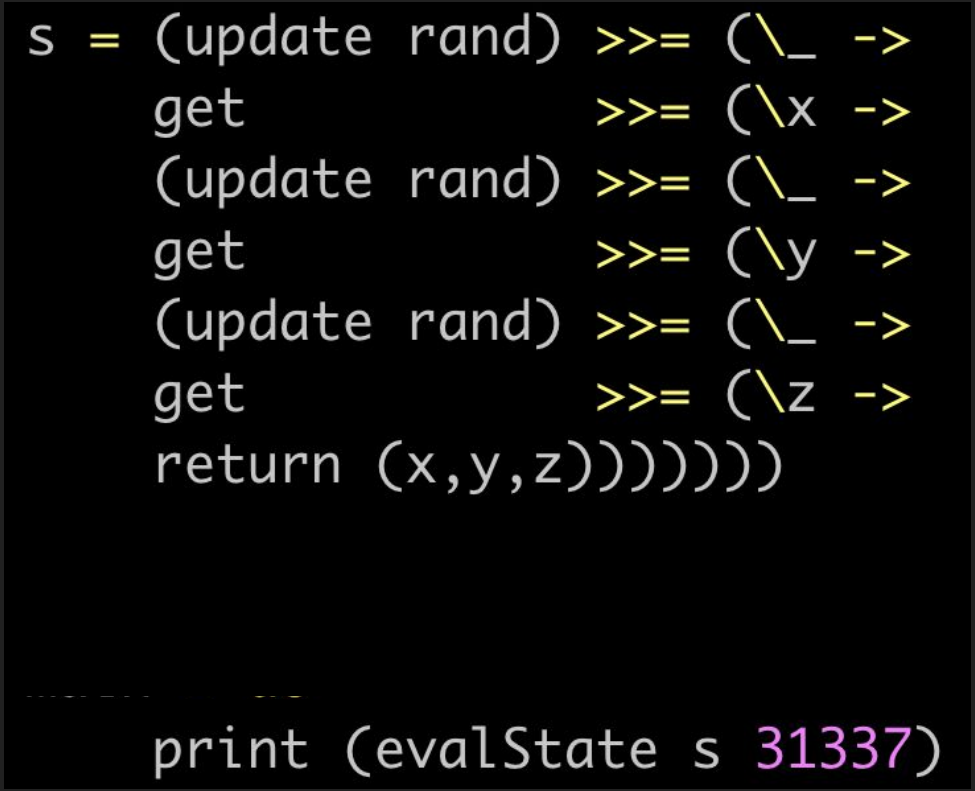
With the State monad, we created a datatype State and associated get/put functions, and then discovered how bind and return had a natural implementation.

Here, we'll start with the knowledge of IO being a Monad, and work backwards.

An IO-producing function

Did any of you point out my sneaky use of `print` in our State example from earlier in class?

Why don't we take a peek at its type signature...



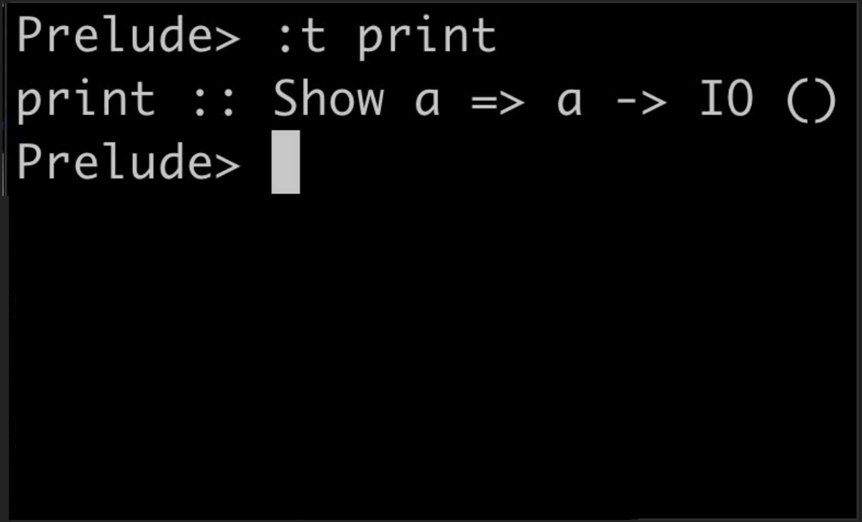
```
s = (update rand) >>= (\_ ->
    get           >>= (\x ->
    (update rand) >>= (\_ ->
    get           >>= (\y ->
    (update rand) >>= (\_ ->
    get           >>= (\z ->
    return (x,y,z)))))))



    print (evalState s 31337)
```

An IO-producing function

Unsurprisingly, print consumes one argument, a polymorphic type that is "toStringable".

It returns not `()` (aka `unit`), but an IO effect of `()`.

As the external effect is writing to the screen, it makes sense that print doesn't hand us back anything interesting, so we get back a `Unit` value.

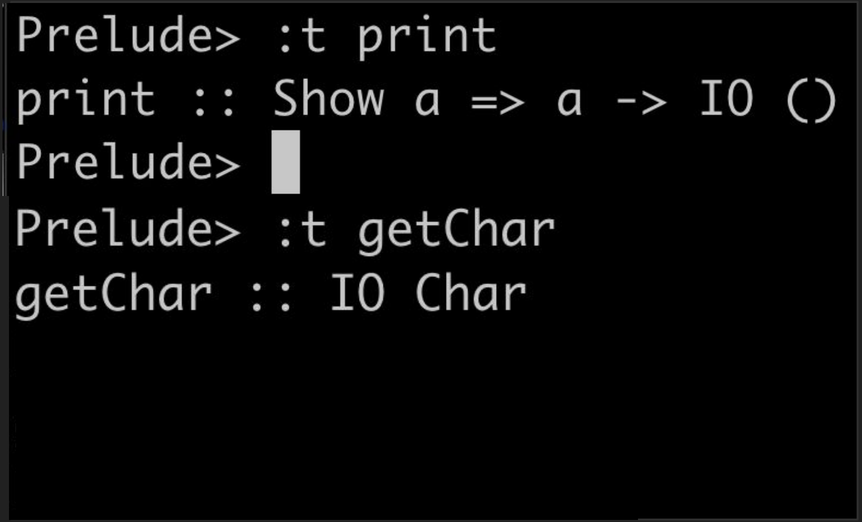


```
Prelude> :t print
print :: Show a => a -> IO ()
Prelude> 
```

An IO-producing function

Here's a more interesting function.

The function `getChar` waits for the user to input a character, and produces the IO action of that character.

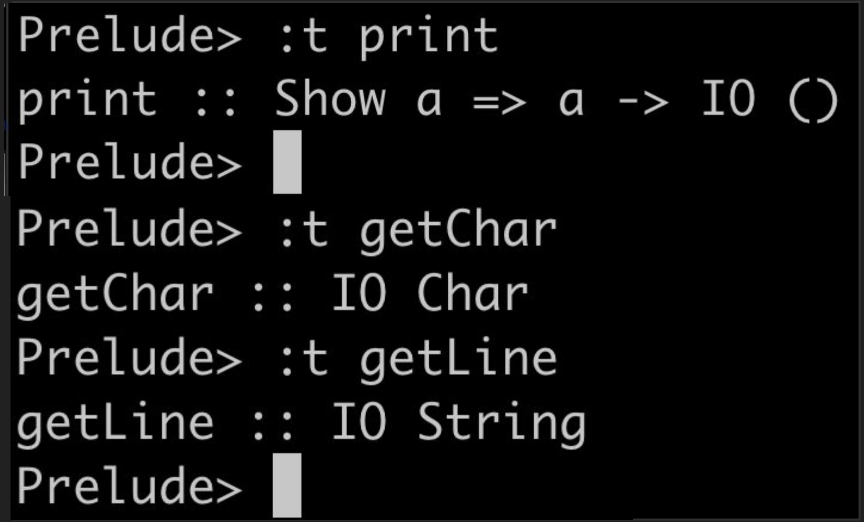


```
Prelude> :t print
print :: Show a => a -> IO ()
Prelude> 
Prelude> :t getChar
getChar :: IO Char
```

An IO-producing function

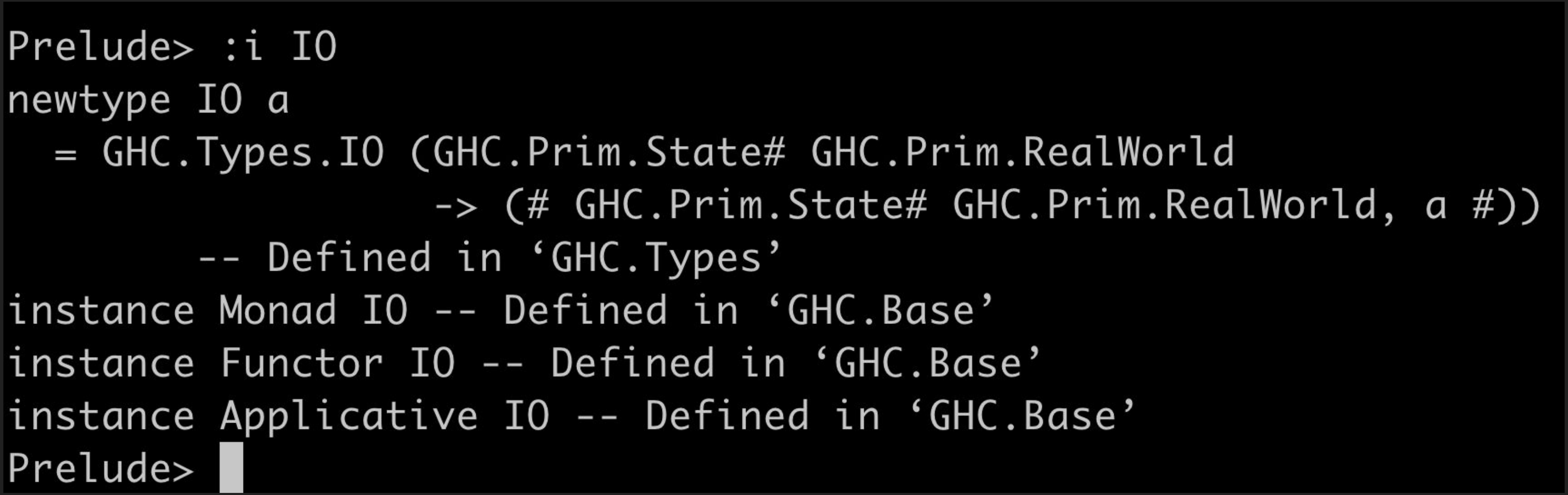
Here's an even more interesting function!

The function `getChar` waits for the user to input a sequence of characters and a newline, and produces the IO action of that inputted string.



```
Prelude> :t print
print :: Show a => a -> IO ()
Prelude> 
Prelude> :t getChar
getChar :: IO Char
Prelude> :t getLine
getLine :: IO String
Prelude> 
```

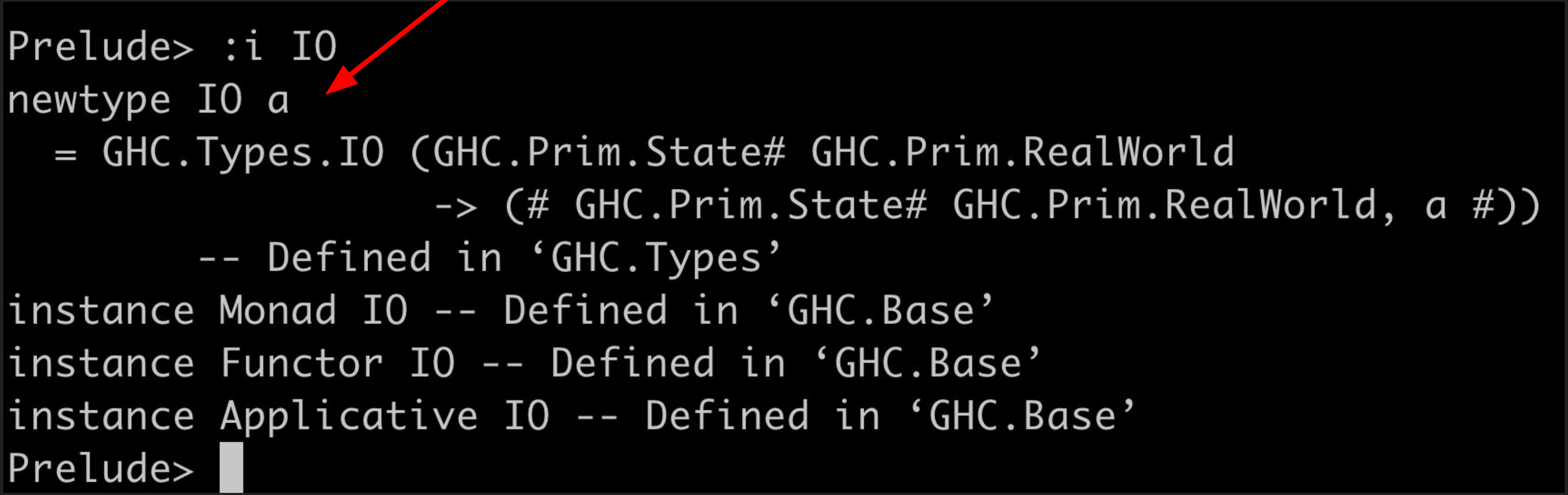
The IO typeclasses



```
Prelude> :i IO
newtype IO a
  = GHC.Types.IO (GHC.Prim.State# GHC.Prim.RealWorld
                  -> (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
        -- Defined in ‘GHC.Types’
instance Monad IO -- Defined in ‘GHC.Base’
instance Functor IO -- Defined in ‘GHC.Base’
instance Applicative IO -- Defined in ‘GHC.Base’
Prelude> 
```

The IO typeclasses

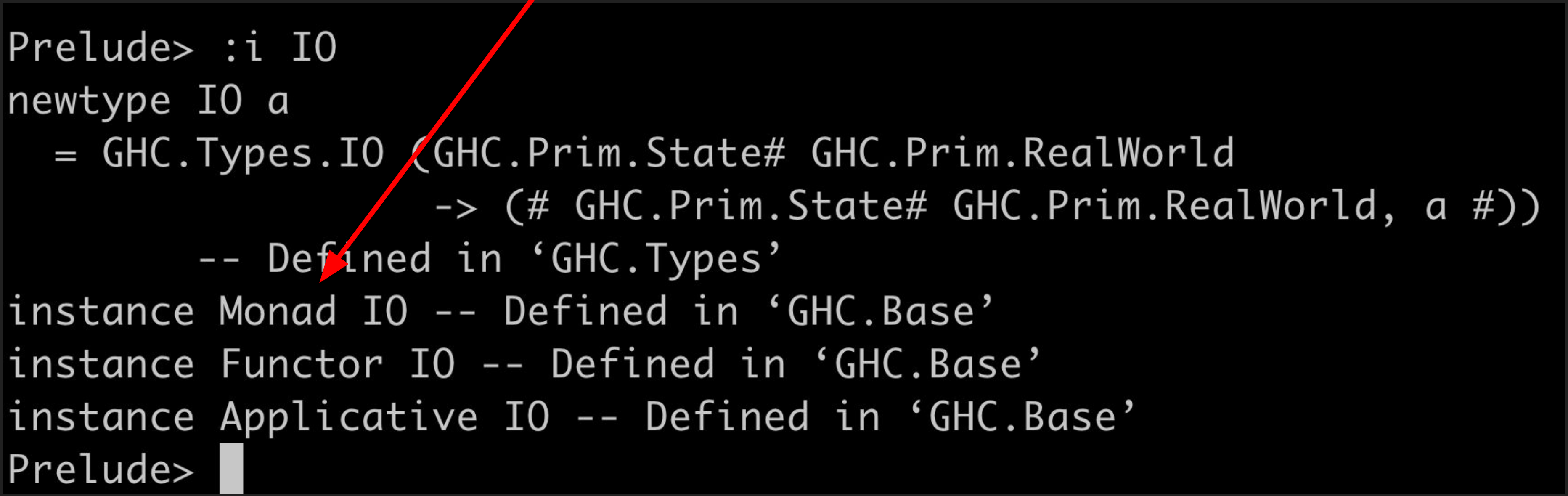
We say `IO a` is **an IO action of type a**, a computation that produces a value of type a **that depends on the state of the external world**



```
Prelude> :i IO
newtype IO a
  = GHC.Types.IO (GHC.Prim.State# GHC.Prim.RealWorld
                  -> (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
        -- Defined in ‘GHC.Types’
instance Monad IO -- Defined in ‘GHC.Base’
instance Functor IO -- Defined in ‘GHC.Base’
instance Applicative IO -- Defined in ‘GHC.Base’
Prelude> 
```

The IO typeclasses

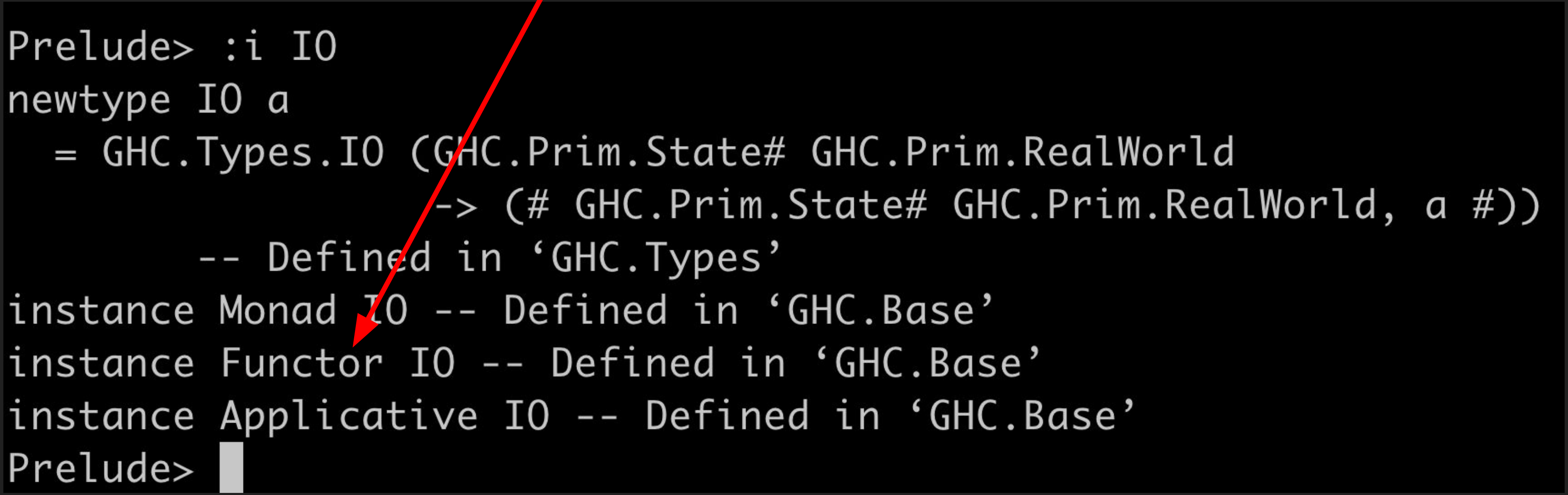
Haskell tells us that this is indeed a monad, so we know >>= and `return` have meaning with an IO...



```
Prelude> :i IO
newtype IO a
  = GHC.Types.IO (GHC.Prim.State# GHC.Prim.RealWorld
                  -> (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
        -- Defined in ‘GHC.Types’
instance Monad IO -- Defined in ‘GHC.Base’
instance Functor IO -- Defined in ‘GHC.Base’
instance Applicative IO -- Defined in ‘GHC.Base’
Prelude> 
```

The IO typeclasses

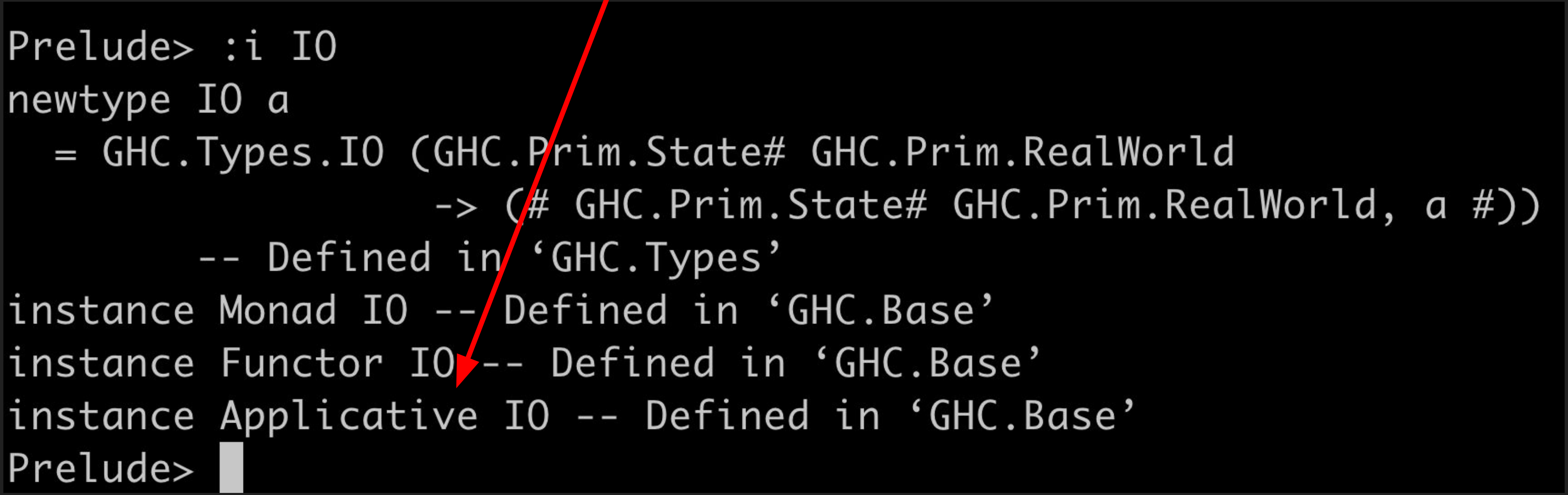
It's also a functor (recall: every monad is a functor), so `fmap` has meaning with IO, too...



```
Prelude> :i IO
newtype IO a
  = GHC.Types.IO (GHC.Prim.State# GHC.Prim.RealWorld
                  -> (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
        -- Defined in ‘GHC.Types’
instance Monad IO -- Defined in ‘GHC.Base’
instance Functor IO -- Defined in ‘GHC.Base’
instance Applicative IO -- Defined in ‘GHC.Base’
Prelude> 
```

The IO typeclasses

It's also also an applicative, which is unfortunately something we don't have time to talk about in this class (but you should look into it in your copious spare time!)



```
Prelude> :i IO
newtype IO a
  = GHC.Types.IO (GHC.Prim.State# GHC.Prim.RealWorld
                  -> (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
        -- Defined in ‘GHC.Types’
instance Monad IO -- Defined in ‘GHC.Base’
instance Functor IO -- Defined in ‘GHC.Base’
instance Applicative IO -- Defined in ‘GHC.Base’
Prelude> 
```

The IO Functor

Let's begin by talking about what `fmap` would do. Recall its type signature:

```
fmap :: (a -> b) -> IO a -> IO b
```

By now we should be able to write a high-level description of what IO does: given

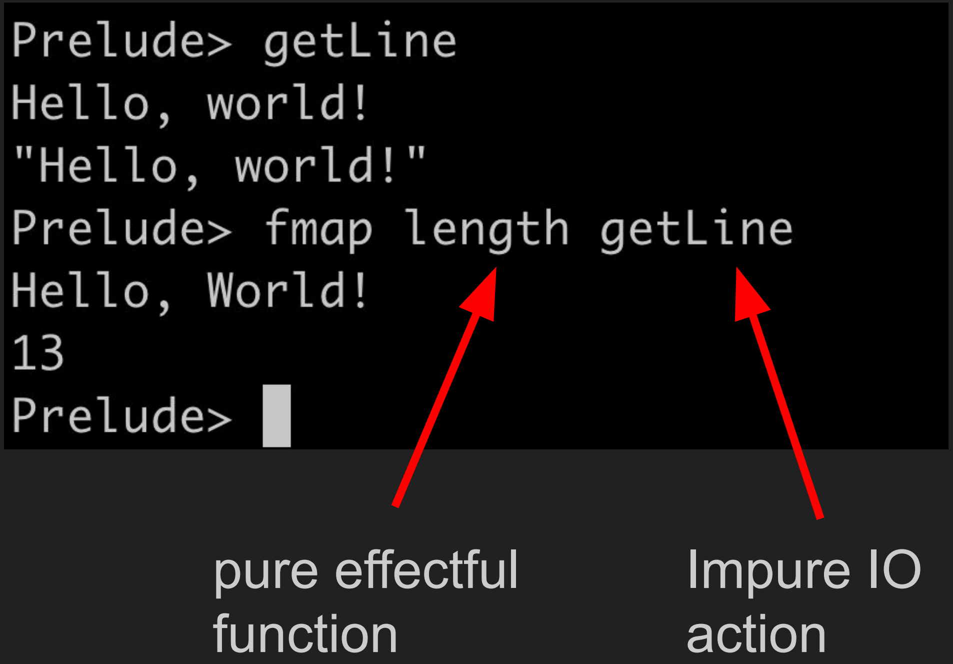
- a pure function from a to b
- an impure IO action returning a value of type a,

`fmap` will produce an IO action of `b` by applying the function to the input IO action's return value.

An IO-producing function

we can combine IO effect-producing functions with fmap to **apply an effct to the IO-returned value** which lives inside the IO computational context.

This also lets us compose pure and impure functions in a natural way.



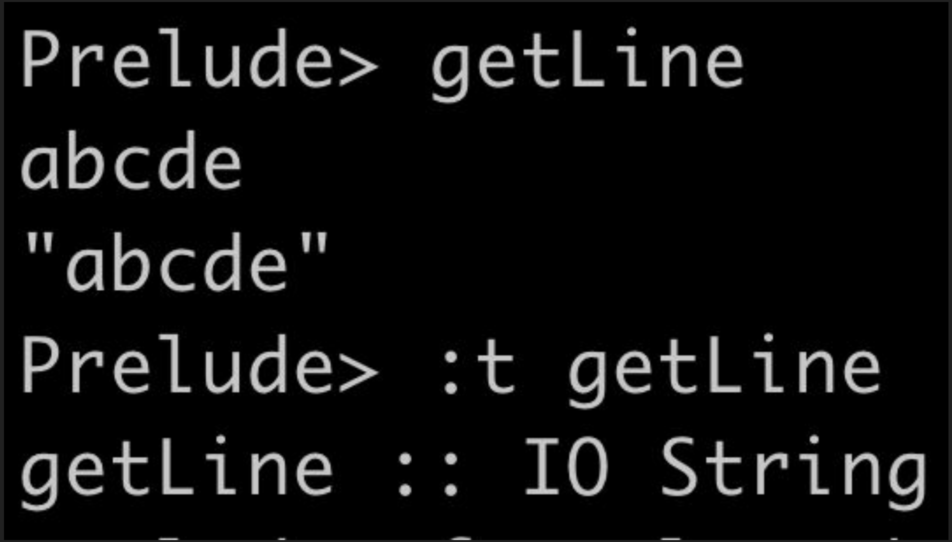
```
Prelude> getLine
Hello, world!
"Hello, world!"
Prelude> fmap length getLine
Hello, World!
13
Prelude> 
```

IO in the REPL

Hold on, shouldn't getLine have returned something like

```
IO "abcde"
```

instead of just the actual string?

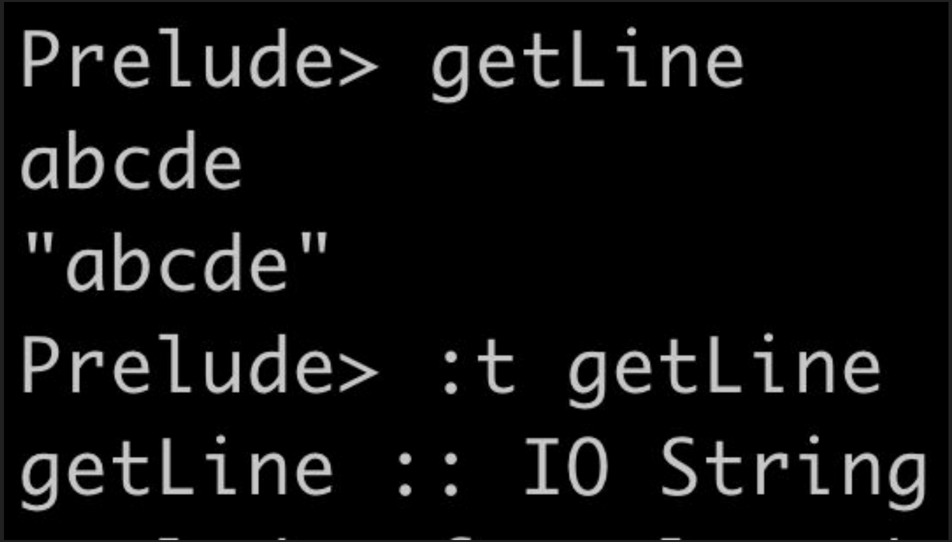


```
Prelude> getLine
abcde
"abcde"
Prelude> :t getLine
getLine :: IO String
```

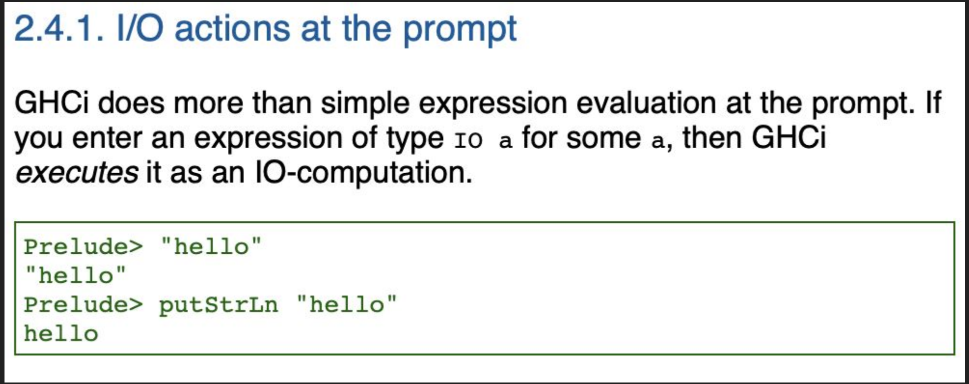
IO in the REPL

The REPL will implicitly execute any IO actions.

Given that entering commands into the REPL is itself something that depends on external state, this makes sense if you think about it.



```
Prelude> getLine
abcde
"abcde"
Prelude> :t getLine
getLine :: IO String
```



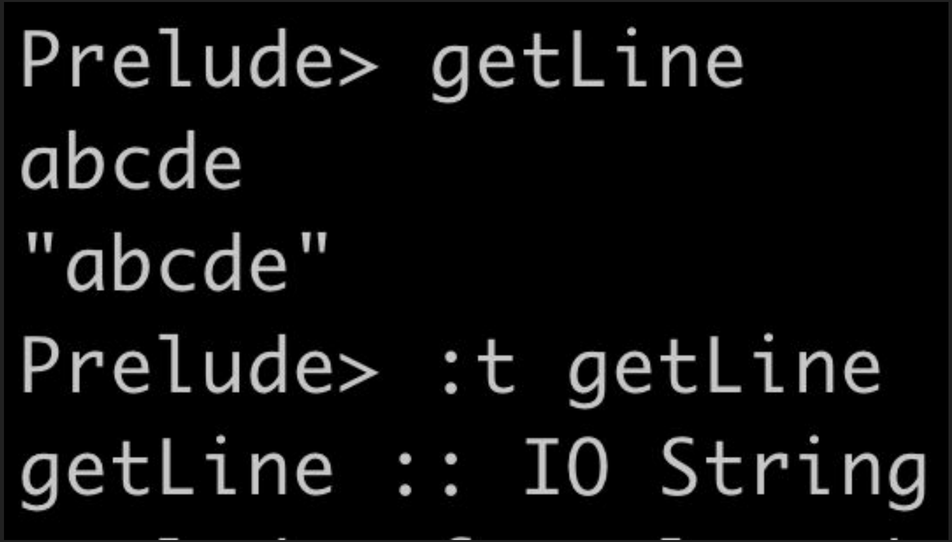
2.4.1. I/O actions at the prompt

GHCi does more than simple expression evaluation at the prompt. If you enter an expression of type `IO a` for some `a`, then GHCi *executes* it as an IO-computation.

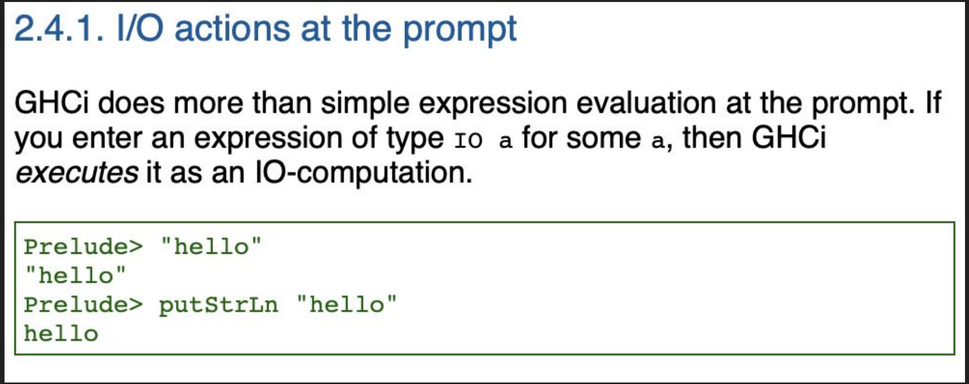
```
Prelude> "hello"
"hello"
Prelude> putStrLn "hello"
hello
```

IO in the REPL

"maybe our whole reality is actually just a pure function `fmap`ped into an IO monad"



```
Prelude> getLine
abcde
"abcde"
Prelude> :t getLine
getLine :: IO String
```



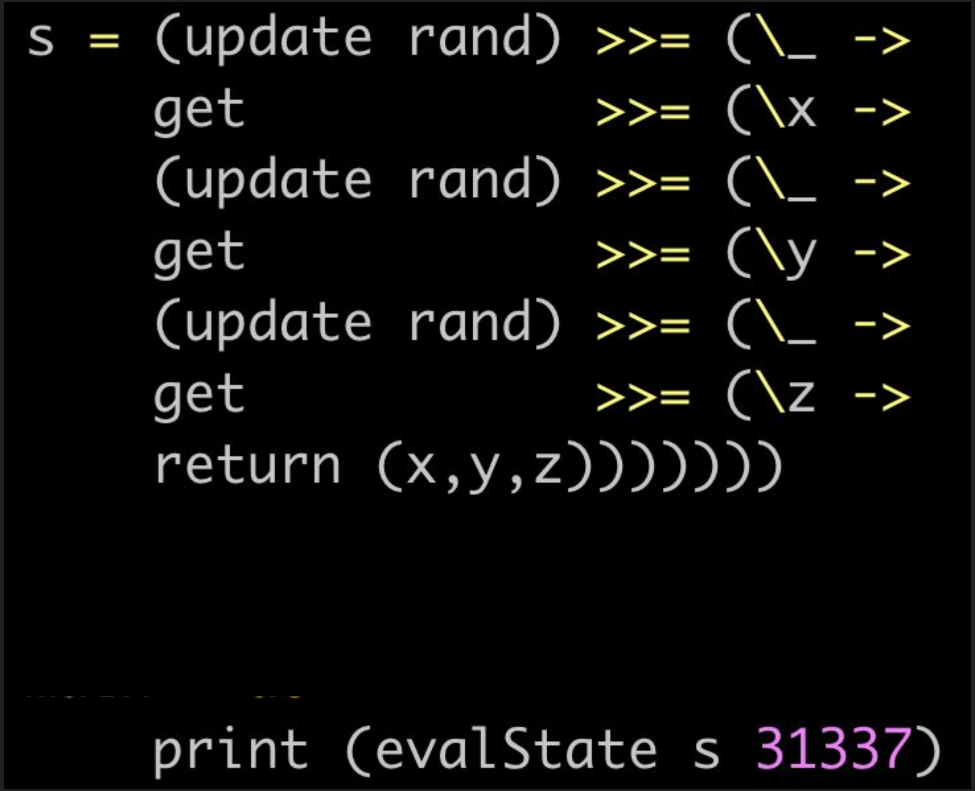
2.4.1. I/O actions at the prompt

GHCi does more than simple expression evaluation at the prompt. If you enter an expression of type `IO a` for some `a`, then GHCi *executes* it as an IO-computation.

```
Prelude> "hello"
"hello"
Prelude> putStrLn "hello"
hello
```

An IO-producing program

It's now time to reveal the last part of this example that I blacked out.



```
s = (update rand) >>= (\_ ->
    get           >>= (\x ->
    (update rand) >>= (\_ ->
    get           >>= (\y ->
    (update rand) >>= (\_ ->
    get           >>= (\z ->
    return (x,y,z)))))))
```

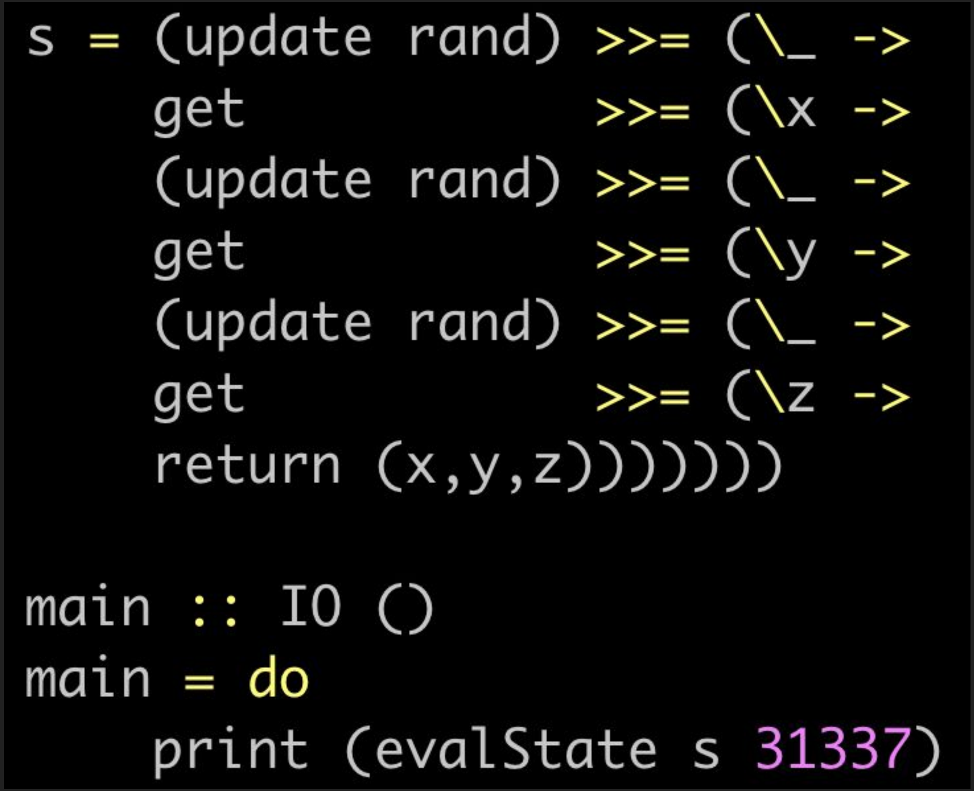
[redacted]

```
    print (evalState s 31337)
```

An IO-producing program

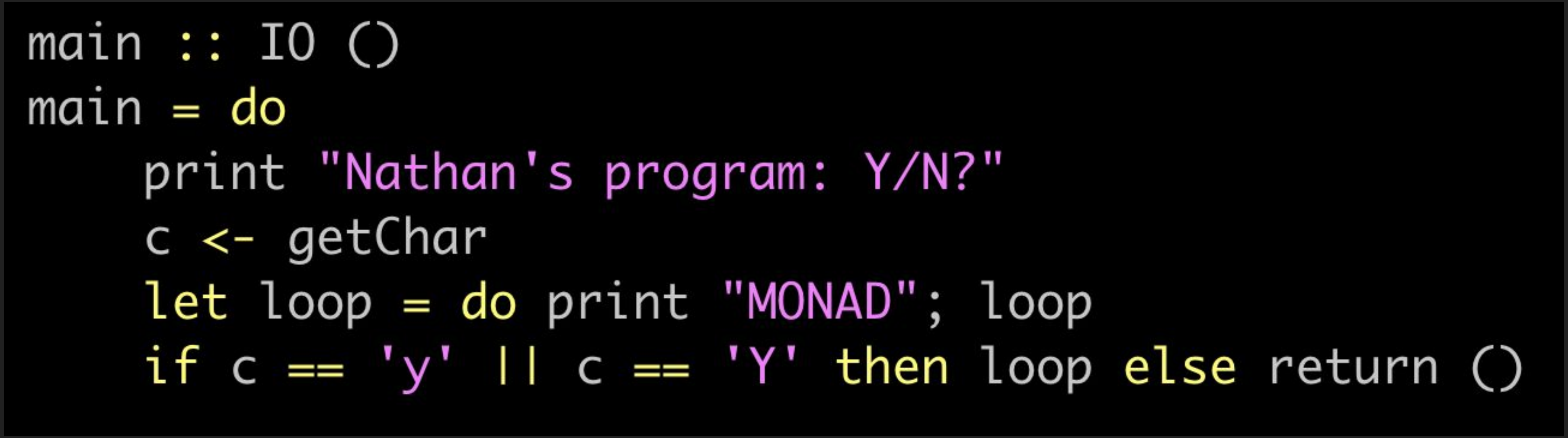
We've contorted outselves to do everything in the REPL, because until now we didn't have enough information to describe what the heck Haskell's `main` functions are.

main is a function that produces an IO action of `unit`!



```
s = (update rand) >>= (\_ ->
    get           >>= (\x ->
    (update rand) >>= (\_ ->
    get           >>= (\y ->
    (update rand) >>= (\_ ->
    get           >>= (\z ->
    return (x,y,z)))))))

main :: IO ()
main = do
    print (evalState s 31337)
```

"...I don't know what I expected…"

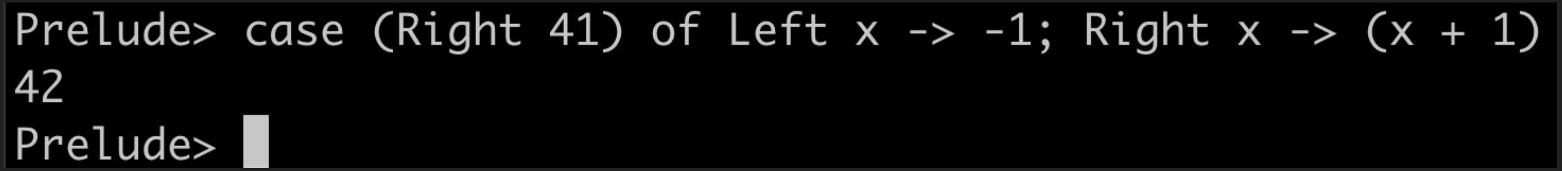


```haskell
main :: IO ()
main = do
    print "Nathan's program: Y/N?"
    c <- getChar
    let loop = do print "MONAD"; loop
    if c == 'y' || c == 'Y' then loop else return ()
```

What happens in the IO...stays in the IO

We've seen that with many other monads we're able to extract values out them through pattern matching.

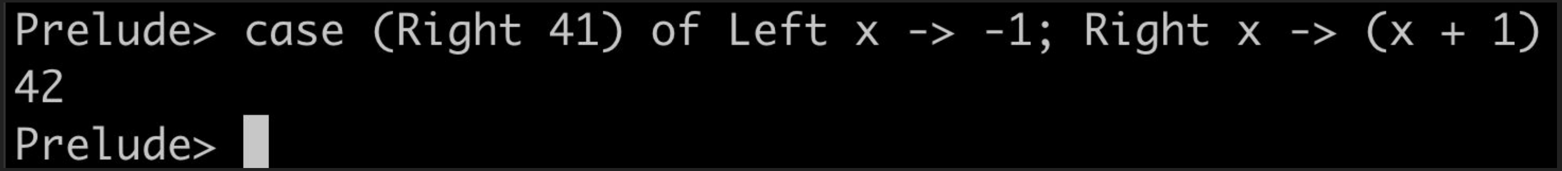
This is an expression, for instance, that operates on a `Maybe Int` and evaluates to an `Int`.



```
Prelude> case (Right 41) of Left x -> -1; Right x -> (x + 1)
42
Prelude>
```

What happens in the IO...stays in the IO

This operation is a bit like an inverse-`return`. (return goes from `a -> m a`, and this expression converts an `m a` to an `a`.)



```
Prelude> case (Right 41) of Left x -> -1; Right x -> (x + 1)
42
Prelude> 
```

What happens in the IO...stays in the IO

IO, by contrast, is different. Once you have an IO action, there's *no way* to pull the value it produces out of the IO "data structure". In fact, the only ways to ever get the actual result from an IO action are:

- Evaluate the action in the REPL
- Evaluate the action in your `main` function

What happens in the IO...stays in the IO

IO, by contrast, is different. Once you have an IO action, there's *no way* to pull the value it produces out of the IO "data structure". In fact, the only ways to ever get the actual result from an IO action are:

- Evaluate the action in the REPL
- Evaluate the action in your `main` function
- *...or, you could link your Haskell program with some C code and have the C code read the Haskell runtime's memory to peek at the action's value, but if you actually do this you will make me cry. Or be very impressed.*

Suffice to say that there are only two **typesafe** ways of getting a result from an IO action.

Next time

- Abstractions on monads
- A connection between monads and continuations
- Course wrapup and farewells