

CSC324 Lecture 22

A note about the final assessment:

Since there were a few questions about it on Piazza:

- It will be a three hour timed assessment in the style of the quizzes you've seen so far
- You will be able to write it during the time of your choice (but this does mean that there will be no invigilation, so if you have uncertainties during the exam you'll have to state any assumptions in your answer)
- Delivery mechanism still TBD: I think I'll write a tiny webapp where you enter your student ID and it gives you the questions, and then you'll submit it to MarkUs within the required time.

Plan for Monday

Next monday will be our last block of time together! :(It will either be a lecture session or a final lab. I haven't decided yet, but it probably will depend on how smooth today's material goes.

I'd like to have you get some experience playing with various monads before the final assessment, but if we're behind on course material then we might need that block of time to wrap up some final concepts.

If you have strong opinions either way, spam the livestream chat with them.

Last time: monads

A **monad** is a typeclass that might be defined in the following way:

```
-- Not exactly the Haskell implementation
class Functor m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

In Haskell, bind is named with this funny-looking operator. (Maybe it helps to imagine it as "the monad is being fed righwards, ">>=", into the effectful function?)

Today:

- do-notation
- The State Monad (to be concluded on Friday)

do-notation

Last time when we learned about functors, we saw how we could chain computations together with the `andThen` higher-order function.

```
*Main> (uncons [10,20,30]) `andThen` maybeSnd `andThen` uncons `andThen`  
maybeFst `andThen` (return . (+1))  
Just 21
```

do-notation

(We also saw that `andThen`, with respect to the `Monad` typeclass, is called `>>=`, which is an infix operator with the same type signature.)

```
*Main> (uncons [10,20,30]) >>= maybeSnd >>= uncons >>= maybeFst >>=
(return . (+1))
Just 21
```

do-notation

Two issues:

- Large chains of monadic operations might get difficult to read
- Structurally limiting: computation is assumed to be a chain of unary function applications

```
(uncons [10,20,30])  
  >>= maybeSnd  
  >>= uncons  
  >>= maybeFst  
  >>= (return . (+1))
```


do-notation

What if we wanted to write a similar expression that **summed the first and second elements** in the list (with the same monadic structure as before: if the list doesn't have enough elements then `Nothing` is produced)

```
(uncons [10,20,30])  
  >>= maybeSnd  
  >>= uncons  
  >>= maybeFst  
  >>= (return . (+1))
```

do-notation

We begin with unconsing the
head and the rest of the list...

```
(uncons [10,20,50])
```

do-notation

Now we are going to bind
an effect function to do
something with the head and
rest...

```
(uncons [10,20,50])  
  >>= (\ p -> ...)
```

do-notation

In particular, we are going to pattern match out the first element in the list and **bind an identifier** to it, namely `v1`.

```
(uncons [10,20,50])  
>>= (\ p -> case p of (v1,xs) -> ...)
```

We'll go ahead and now extract the head and rest of the rest of the list...

do-notation

In particular, we are going to pattern match out the first element in the list and **bind an identifier** to it, namely `v1`.

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

We'll go ahead and now extract the head and rest of the rest of the list...

do-notation

Is this right?

```
(uncons [10,20,50])
```

```
>>= (\ p -> case p of (v1,xs) -> uncons xs)
```

```
>>= (\ p -> case p of (v2,xs) -> ...v1+v2)
```

do-notation

Is this right?

v1 isn't in scope!

We want our final expression
to involve summing v1 and v2,
but they are scoped in their
own lambda functions... this
has to be nested

```
(uncons [10,20,50])
```

```
>>= (\ p -> case p of (v1,xs) -> uncons xs)
```

```
>>= (\ p -> case p of (v2,xs) -> ...v1+v2)
```

do-notation

OK!

We have $v1$ in the outer
lambda, and $v2$ in the inner
lambda, so we can produce
their sum...

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


do-notation

...but don't forget, `>>=`'s effect
function goes from `a -> m b`,
so the sum needs to be lifted
into a `Maybe`

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

do-notation

and we can use the general monadic lift function rather than calling `Just`, which is tied in particular to `Maybes`.

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

(Maybe you can begin to see here why that particular function has the name `return`!)

do-notation

do-notation is syntactic sugar for chains of `>>=` operations that lets us bind intermediary computations to values:

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

do-notation

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

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

do-notation

```
(uncons [10,20,50])
```

```
>>= (\ p -> case p of (v1,xs) ->  
    (uncons xs)
```

```
>>= (\ p -> case p of (v2,_) ->  
    return (v1+v2)))
```

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

do-notation

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

```
do p <- (uncons [10,20,50]); ...
```

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

do-notation

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

`do (v1,xs) <- (uncons [10,20,50]); ...` *(we can pattern-match right in a do)*

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

do-notation

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

```
do (v1,xs) <- (uncons [10,20,50]); ...
```

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


do-notation

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

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

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

do-notation

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

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

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

do-notation

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

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

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

do-notation

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

Each of these expressions
evaluate to Just 30.

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

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

Next: A monadic state data type

For days now I've promised you that there are more to functors than being "a container to modify the value in", but I haven't really given you good evidence for that.

For the rest of today, we'll explore another monad whose **computational context** contains a piece of mutable state, that, to the outside world, appears to be entirely side-effect free.

A functional random number generator

Consider a function that generates a random number. Can we have such a thing as a referentially-transparent random number generator?



A random number generator:

- takes no arguments as input
 - so its output can't entirely depend on its input
- doesn't return a constant, unchanging value
 - so there must be a piece of mutable state being updated somewhere...
- argh!!! Is this our worst-case scenario???

```
→ ~ python3
Python 3.7.3 (default, Jun  2 2020, 19:48:59)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import random
>>> random.random()
0.014965948421975739
>>> random.random()
0.7931094314233358
>>> random.random()
0.672897815321838
>>> random.random()
0.4469480460012545
>>> random.random()
0.08421920365041746
>>> █
```

A mutable PRNG implementation

→ ~ python3

Python 3.7.3 (default, Jun 2 2020, 19:48:59)

[Clang 11.0.3 (clang-1103.0.32.62)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

```
>>> seed = 31337
```

```
>>> def rand() -> int:
```

```
...     "Produces a pseudo-random integer given some global state."
```

```
...     global seed #tsk tsk!
```

```
...     seed = ((7**5) * seed) % (2**32 - 1)
```

```
...     return seed
```

```
>>>
```


A mutable PRNG implementation

```
>>> rand()
```

```
526680959
```

```
>>> rand()
```

```
4294250213
```

```
>>> rand()
```

```
832904711
```

```
>>> rand()
```

```
1331063372
```

```
>>> rand()
```

```
2992420844
```

```
>>> # sure, looks random enough!  Can we make this more FUNCTIONAL?
```

Well, we could pass in the state as an argument

```
>>> def rand(seed: int) -> int:
...     """A referentially-transparent PRNG!"""
...     seed = ((7*5) * seed) % (2**32 - 1)
...     return seed
>>> rand(526680959)
4294250213
>>> rand(4294250213)
832904711
>>> rand(832904711)
1331063372
>>> # ok, this works, but it's kinda awkward....
```

Problems with our solution

```
>>> def rand(seed: int) -> int:
...     """A referentially-transparent PRNG!"""
...     seed = ((7**5) * seed) % (2**32 - 1)
...     return seed
```

We've pushed the responsibility of handing the correct seed back to the caller, which is awkward: we have to remember the previous last value ourselves.

Dealing with awkwardness in functional programming

Trying to design richer programs while maintaining functional purity and referential transparency can feel awkward. Sometimes it feels like we should "bend the rules". Maybe this is an okay place to just insert a box and store the seed in it?

*"Awkwardness like this is almost always a sign of some missing abstraction waiting to be discovered. We encourage you to plow ahead and look for common patterns, **and you may even discover the standard solution yourself**"*

-- Runar Bjarnason

Our plan of attack:

How can we combine having a referentially-transparent function, while not requiring the caller to also keep track of the mutable state itself?

We tried adding an argument to the function, and that didn't give us the best solution. What if we added another value to the return type?

Our better PRNG function will return a tuple containing:

- the generated random number
- the state needed to generate the next random number

Returning a value and the next PRNG generator

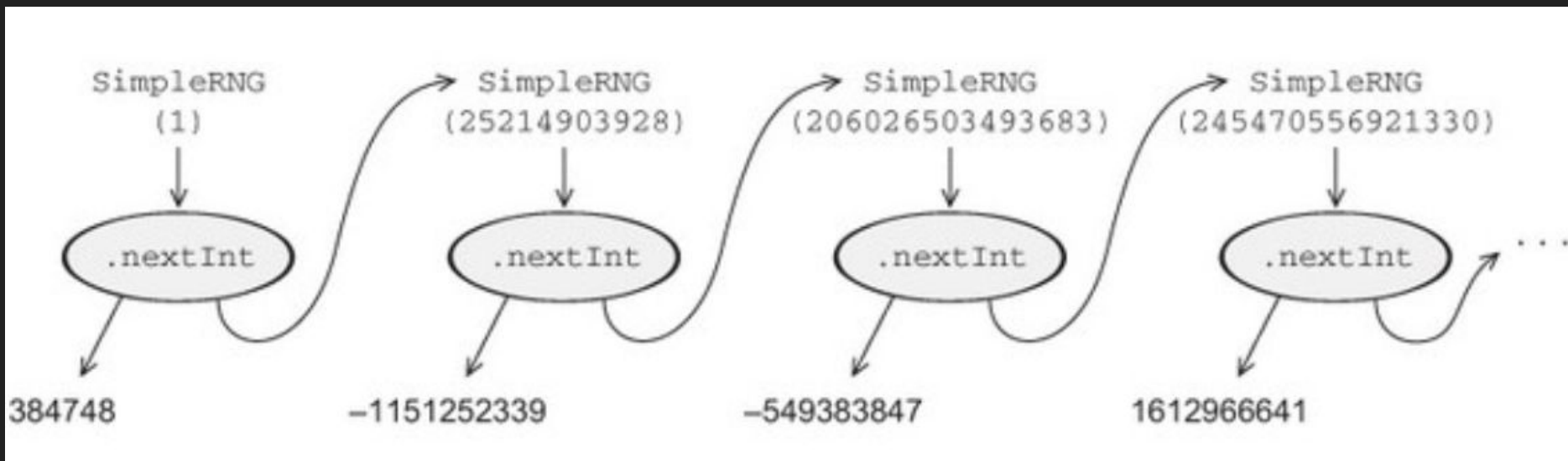


diagram credit: **Chiusano, Bjarnason. *Functional Programming in Scala***

A datatype for a stateful operation

Rather than constructing the type for the PRNG directly, we'll instead try to come up with a general form for a polymorphic stateful computation and then figure out how to specialise the datatype for the random number generator.

Our golden rule for this lecture (and for functional design in general) will be:
"Leave things general unless we have no choice, and see what design possibilities open up"

A datatype for a stateful operation

Let's wrap whatever state we need for a stateful computation into a type s .

Our stateful random number generator, given previous state of type s , produces both the return value of type a and the next state, also of type s .

What is the type of a function that performs the stateful computation?

A datatype for a stateful operation

Let's wrap whatever state we need for a stateful computation into a type s .

Our stateful random number generator, given previous state of type s , produces both the return value of type a and the next state, also of type s .

What is the type of a function that performs the stateful computation?

$\dots \rightarrow \dots$

A datatype for a stateful operation

Let's wrap whatever state we need for a stateful computation into a type s .

Our stateful random number generator, given previous state of type s , produces both the return value of type a and the next state, also of type s .

What is the type of a function that performs the stateful computation?

$s \rightarrow \dots$

A datatype for a stateful operation

The intuition with this function is that it consumes the current state, performs "some operation" that depends on the current state, producing a value of some type a and possibly changing the state, and returning a tuple containing both.

$$s \rightarrow (a, s)$$

Fitting the data definition to a monadic form

In order to make our stateful computation look more like the sorts of monads we've seen so far, we'll wrap the state function in a data constructor called `State`:

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



`State` is a record that wraps one value, the state transition function.

Fitting the data definition to a monadic form

In order to make our stateful computation look more like the sorts of monads we've seen so far, we'll wrap the state function in a data constructor called `State`:

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

Now consider our PRNG. What is the piece of state that we need in order to generate the next random number?

Type aliases and partial type application


In order to make our stateful computation look more like the sorts of monads we've seen so far, we'll wrap the state function in a data constructor called SM:

```
data State s a = State (s -> (a,s))  
type PrngState a = State Int a      -- aka State (Int -> (a, Int))
```

Type aliases and partial type application

In order to make our stateful computation look more like the sorts of monads we've seen so far, we'll wrap the state function in a data constructor called SM:

```
data State s a = State (s -> (a,s))  
type PrngState a = State Int a      -- aka State (Int -> (a, Int))
```



This type looks a lot like Maybe a, Either a, [a], ... feels rather *monadic*!

Type aliases and partial type application

In order to make our stateful computation look more like the sorts of monads we've seen so far, we'll wrap the state function in a data constructor called SM:

```
data State s a = State (s -> (a,s))
type PrngState a = State Int a      -- aka State (Int -> (a, Int))
```

Type parameters can be **partially-applied** to types, in the same way that function arguments can be partially-applied to functions.

Type aliases and partial type application

In order to make our stateful computation look more like the sorts of monads we've seen so far, we'll wrap the state function in a data constructor called SM:

```
data State s a = State (s -> (a,s))
type PrngState a = State Int a      -- aka State (Int -> (a, Int))
```

(Since the random number generator generates an Int, too, shouldn't we make the second type variable an Int, too? Remember the golden rule: let's see what we get by leaving that type variable unspecified.)

What can we do with a State?

We might want to get access to the state, too (so we could, for instance, use that to generate a new random number)

```
data State s a = State (s -> (a,s))  
type PrngState a = State Int a
```

We would like to:

- Make the output value the state value
- Keep the state value as-is

What can we do with a State?

We might want to get access to the state, too (so we could, for instance, use that to generate a new random number)

```
data State s a = State (s -> (a,s))  
type PrngState a = State Int a
```

```
get :: State (Int -> (Int, Int))  
get = State (\s -> (s,s))
```



The way to understand this function is: "get produces a function that consumes an old state and returns the tuple where where state can be read out of."

What can we do with a State?

We might want to get access to the state, too (so we could, for instance, use that to generate a new random number)

```
data State s a = State (s -> (a,s))  
type PrngState a = State Int a
```

```
get :: State Int Int  
get = State (\s -> (s,s))
```

What can we do with a State?

We might want to get access to the state, too (so we could, for instance, use that to generate a new random number)

```
data State s a = State (s -> (a,s))  
type PrngState a = State Int a
```

```
get :: State s s  
get = State (\s -> (s,s))
```

What can we do with State?

Certainly, given an existing state, we'd like to be able to update the state of the monad by passing in a function.

```
data State s a = State (s -> (a,s))  
type PrngState a = State Int a
```

```
get :: State s s  
get = State (\s -> (s,s))
```

What can we do with State?

Certainly, given an existing state, we'd like to be able to update the state of the monad by passing in a function.

```
data State s a = State (s -> (a,s))  
type PrngState a = State Int a
```

```
get :: State s s  
get = State (\s -> (s,s))
```

How should we finish this type signature?

```
update :: (s -> s) -> State ???  
update f = ...
```

What can we do with State?

Certainly, given an existing state, we'd like to be able to update the state of the monad by passing in a function.

```
data State s a = State (s -> (a,s))  
type PrngState a = State Int a
```

```
get :: State s s  
get = State (\s -> (s,s))
```

How should we finish this type signature?

```
update :: (s -> s) -> (State s a)  
update f = State (\s -> (????, f s))
```

Those were all great suggestions!!!

Some of you may have said "leave the type polymorphic"...



What can we do with State?


Certainly, given an existing state, we'd like to be able to update the state of the monad by passing in a function.

```
data State s a = State (s -> (a,s))  
type PrngState a = State Int a
```

```
get :: State s s  
get = State (\s -> (s,s))
```

How should we finish this type signature?

```
update :: (s -> s) -> (State s a)  
update f = State (\s -> (????, f s))
```



But we actually have to instantiate an value, which we can't do if the type is polymorphic... We have no type `a` on hand, nor do we have a function to take some expression and produce an `a`

What can we do with State?

Certainly, given an existing state, we'd like to be able to update the state of the monad by passing in a function.


```
data State s a = State (s -> (a,s))  
type PrngState a = State Int a
```

```
get :: State s s  
get = State (\s -> (s,s))
```

How should we finish this type signature?

```
update :: (s -> s) -> (State s ())  
update f = State (\s -> ((), f s))
```

In this example, we'll do what Haskell's built-in State does and have State hold a value of type Unit



The third and final operation...

We should probably have a way of unconditionally setting the state.

```
data State s a = State (s -> (a,s))  
type PrngState a = State Int a
```

```
get :: State s s  
get = State (\s -> (s,s))
```

```
update :: (s -> s) -> State s ()  
update f = State (\s -> ((), f s))
```

```
set :: s -> (State s ())  
set x = State (\_ -> ((), x))
```

The third and final operation...

The last thing we need is a way to extract the operation inside the State, so we can actually run it.

```
data State s a = State (s -> (a,s))  
type PrngState a = State Int a
```

```
get :: State s s  
get = State (\s -> (s,s))
```

```
update :: (s -> s) -> State s ()  
update f = State (\s -> ((), f s))
```

```
runState :: State s a -> (s -> (a,s))  
runState (State oper) = oper
```

A function as a computational context

Notice that whatever we do that depends on our stored state has to happen within the context of a function call (since a *State* *is* a function).

This feels a lot like the "apply an effect within a context" description of a functor that we had before!

```
data State s a = State (s -> (a,s))  
type PrngState a = State Int a
```

```
get :: State s s  
get = State (\s -> (s,s))
```

```
update :: (s -> s) -> State s ()  
update f = State (\s -> ((), f s))
```

```
runState :: State s a -> (s -> (a,s))  
runState (State oper) = oper
```

A function as a computational context

We can do simple operations on state:

```
*Main> :t runState get  
runState get :: a -> (a, a)
```

A function as a computational context

We can do simple operations on state:

```
*Main> runState get 42  
(42,42)
```

A function as a computational context

We can do simple operations on state:

```
State (s0 ->  
let  
  
in
```


A function as a computational context

We can do simple operations on state:

```
State (\s0 ->  
let (_, s1) = runState (put 41) s0  
  
in
```

A function as a computational context

We can do simple operations on state:

```
State (\s0 ->  
let (_, s1) = runState (put 41) s0  
    (v2, s2) = runState get s1  
in
```

A function as a computational context

We can do simple operations on state:

```
State (\s0 ->  
let (_, s1) = runState (put 41) s0  
    (v2, s2) = runState get s1  
    (v3, s3) = runState (update (+1)) s2  
in (v3, s3))
```

Running this state with some initial value produces `((), 42)` (why doesn't the initial value matter?)

A function as a computational context

Suppose we have our PRNG
implementation:

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

A function as a computational context

We can do simple operations on state:

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

```
State (\s0 ->
```

```
let (v1, s1) = runState get s0
```

```
in
```

A function as a computational context

We can do simple operations on state:

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

```
State (\s0 ->
```

```
  let (v1, s1) = runState get s0
```

```
      (_, s2) = runState (update rand) s1
```

```
in
```

A function as a computational context

We can do simple operations on state:

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

```
State (\s0 ->
```

```
  let (v1, s1) = runState get s0
      (_, s2) = runState (update rand) s1
      (v2, _) = runState get s2
```

```
in
```

A function as a computational context

We can do simple operations on state:

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

```
State (\s0 ->
```

```
  let (v1, s1) = runState get s0
      (_, s2) = runState (update rand) s1
      (v2, _) = runState get s2
      (_, s3) = runState (update rand) s2
```

```
in
```


A function as a computational context

We can do simple operations on state:

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

```
State (\s0 ->
```

```
  let (v1, s1) = runState get s0
      (_, s2) = runState (update rand) s1
      (v2, _) = runState get s2
      (_, s3) = runState (update rand) s2
      (v3, _) = runState get s3
```

```
in
```

A function as a computational context

We can do simple operations on state:

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

```
State (\s0 ->
```

```
  let (v1, s1) = runState get s0
      (_, s2) = runState (update rand) s1
      (v2, _) = runState get s2
      (_, s3) = runState (update rand) s2
      (v3, _) = runState get s3
      (_, sn) = runState (update rand) s3
```

```
in
```

A function as a computational context

We can do simple operations on state:

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

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

```
[MSFT] /tmp runghc Lecture22.hs
((31337,526680959,4294250213),832904711)
```

This current version has some serious downsides, though:

- The computation is wrapped in this State data constructor; in imperative programming we're used to this being "transparent"

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

This current version has some serious downsides, though:

- The right hand side of every assignment calls runState. That feels redundant and awkward, and something that the underlying mechanism should take care of for us.

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

This current version has some serious downsides, though:

- The biggest problem: the "mutable state" is obviously just being passed around here. From the point of view of presenting rand as a "mutable function", we're not quite there yet!

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

By the end of tomorrow's class, we'll have enough mechanism to perform stateful computations that look like this:

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