CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

Lecture 12: State Monads

- State Monad: Creating a Stack-Based Evaluator
- State Monad: General Principles
- Control.Monad.State
- Examples using the State Monad

The State Monad

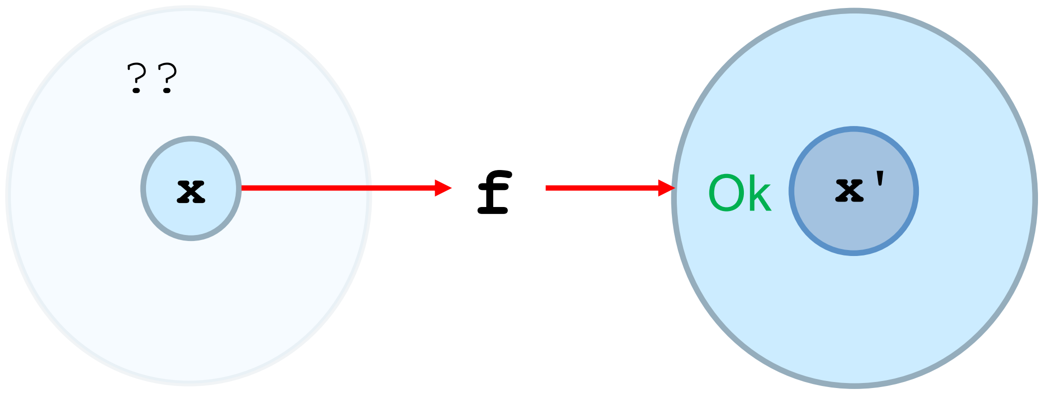
```
data Checked a = Ok a
               | Warning a String
               | Error String deriving (Show,Eq)
```

```
divide :: Integer -> Integer -> Checked Integer
divide x y | y == 0    = Error "Division by Zero!"
           | otherwise = if x >= 0
                         then Warning x $ "Negative!"
                         else Ok (x `div` y))
```

There's a problem brewing in Monad World, and it has to do with our assumption that functions should take pure (non-monadic) values and return monadic types, for example, a function

```
f :: a -> Checked
```

can decide whether to generate an `Ok`, a `Warning`, or an `Error`, but can't know whether such was generated in the past.

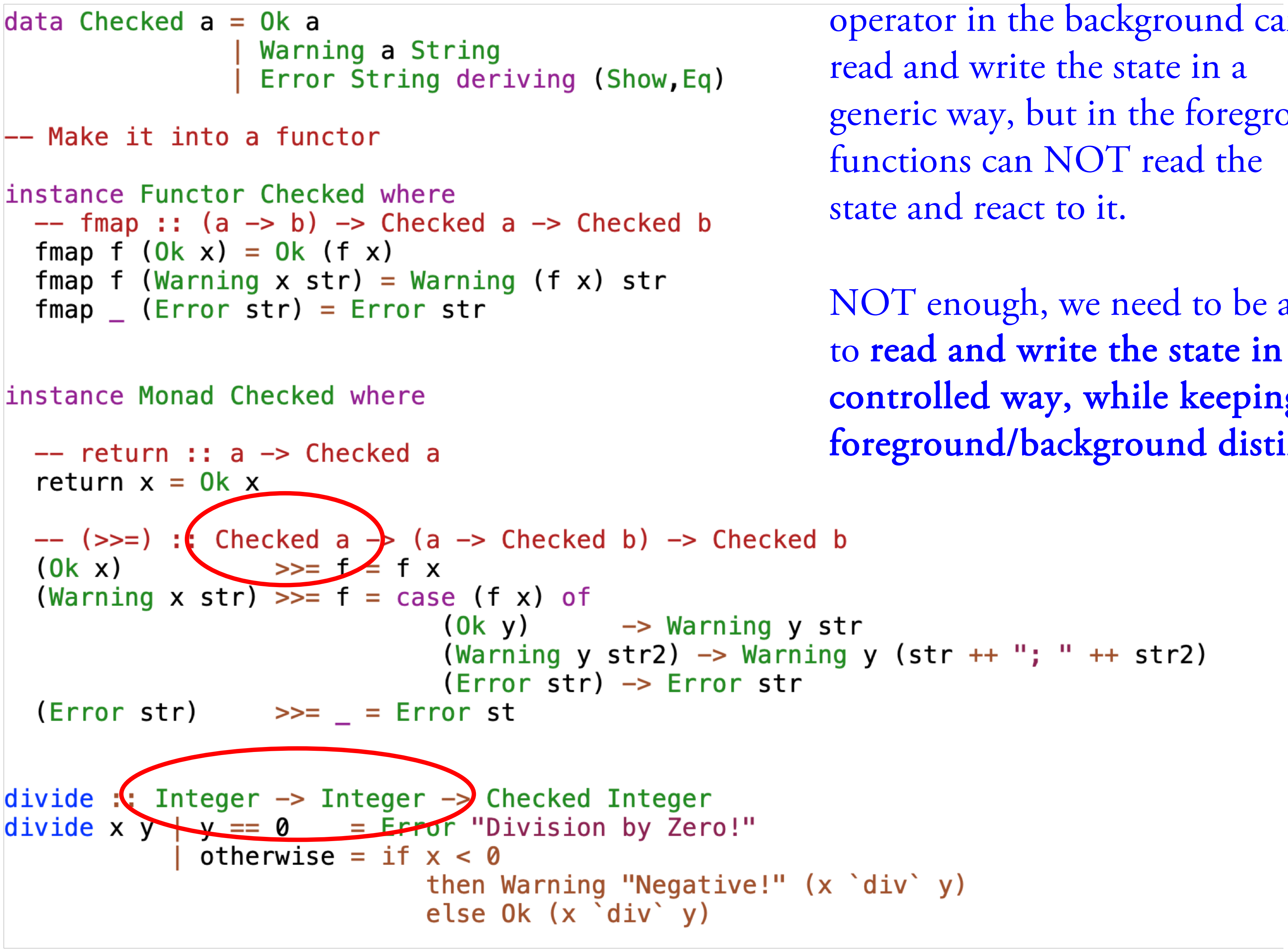


What if **`f`** wants to modify its behavior based on whether it's input is wrapped in a `Warning`?

So far, Monads allow us to **focus on a foreground value while passing along a background state, and**

- We can WRITE to the state (modify the future state), but
- We can't READ the state (get information from the past).

The State Monad



```haskell
data Checked a = Ok a
               | Warning a String
               | Error String deriving (Show,Eq)

-- Make it into a functor

instance Functor Checked where
  -- fmap :: (a -> b) -> Checked a -> Checked b
  fmap f (Ok x) = Ok (f x)
  fmap f (Warning x str) = Warning (f x) str
  fmap _ (Error str) = Error str


instance Monad Checked where

  -- return :: a -> Checked a
  return x = Ok x

  -- (>>=) :: Checked a -> (a -> Checked b) -> Checked b
  (Ok x)          >>= f = f x
  (Warning x str) >>= f = case (f x) of
                            (Ok y)        -> Warning y str
                            (Warning y str2) -> Warning y (str ++ "; " ++ str2)
                            (Error str) -> Error str
  (Error str)     >>= _ = Error st


divide :: Integer -> Integer -> Checked Integer
divide x y | y == 0      = Error "Division by Zero!"
           | otherwise = if x < 0
                           then Warning "Negative!" (x `div` y)
                           else Ok (x `div` y)
```

In Detail:

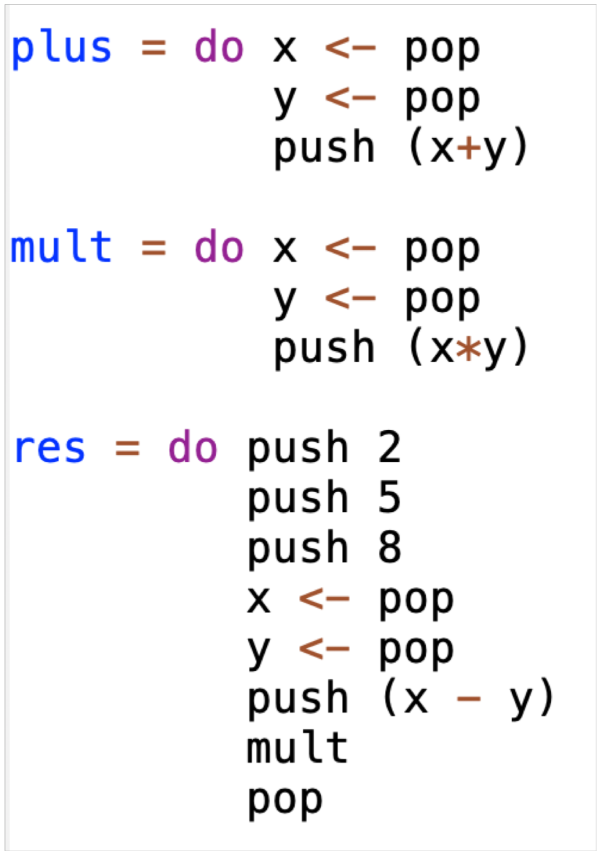
In the Checked Monad, the bind operator in the background can read and write the state in a generic way, but in the foreground, the functions can NOT read the state and react to it.

NOT enough, we need to be able to **read and write the state in a controlled way, while keeping the foreground/background distinction.**

The State Monad: A Stack-based Evaluator

Let us consider a concrete example of why we might want to do this: **a stack-based evaluator for arithmetic expressions.**

We need to read and write the stack, but we want to keep all the code for the stack in the background, and focus on our foreground computation:



```haskell
plus = do x <- pop
          y <- pop
          push (x+y)

mult = do x <- pop
          y <- pop
          push (x*y)

res = do push 2
         push 5
         push 8
         x <- pop
         y <- pop
         push (x - y)
         mult
         pop
```

```
=> 6
```

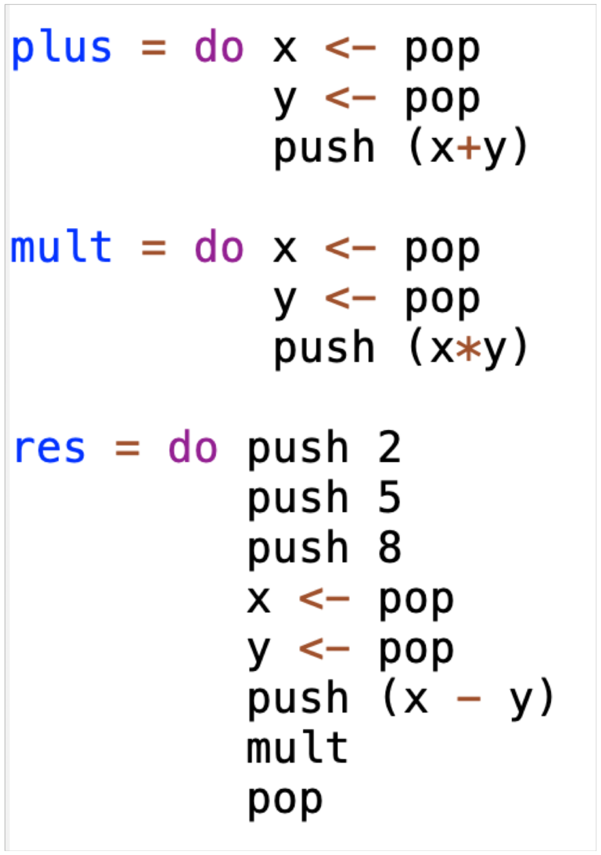
Every `push` writes to the background stack, and every `pop` reads from the background stack.

There is constant interaction with the background state!

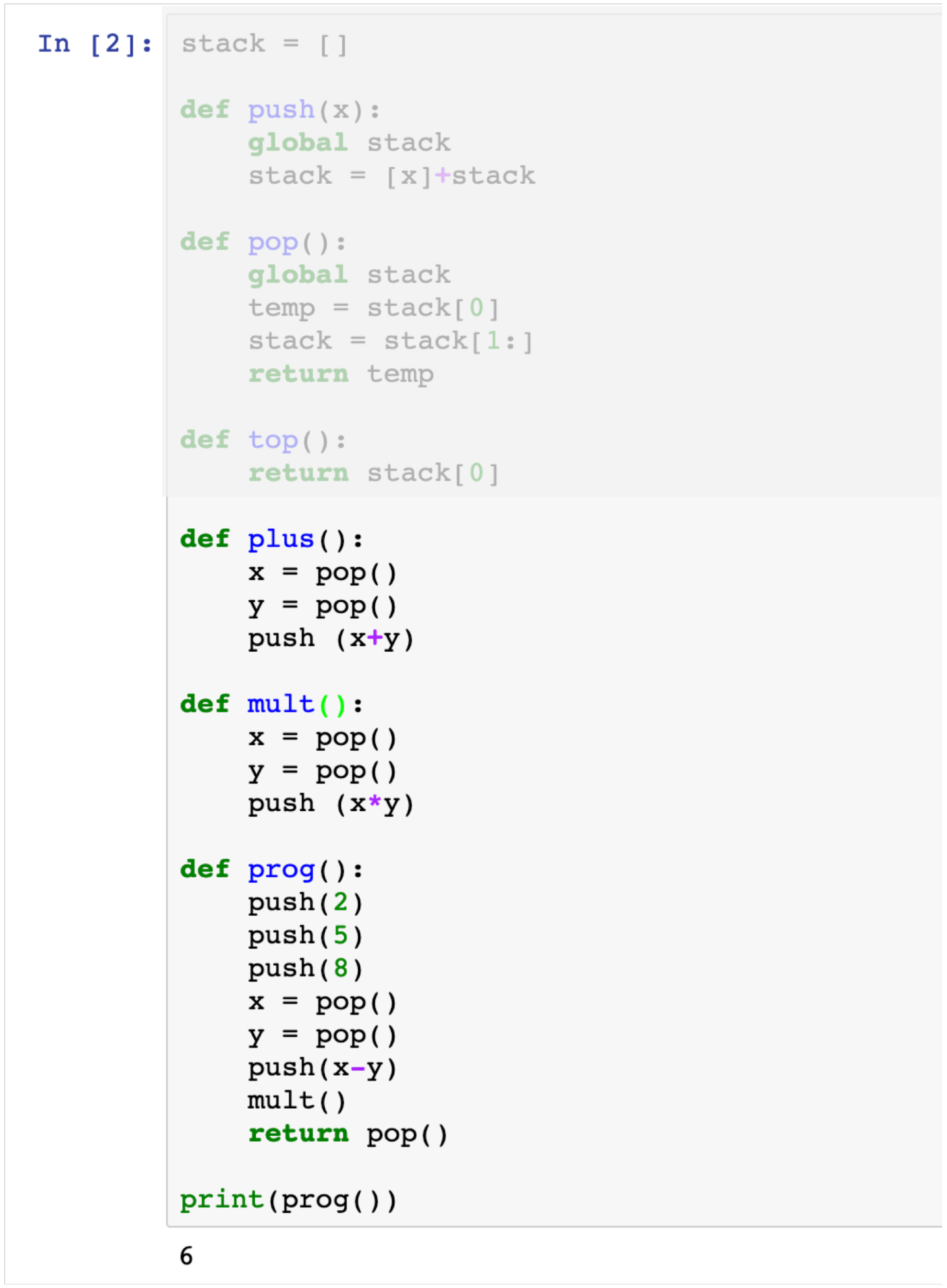
```
 8
 6
 2
---

 2           x = 8
---          y = 5

 3
 2           x = 8
---          y = 5

 6           x = 8
---          y = 5
```

The State Monad: A Stack-based Evaluator

This is exactly the kind of code that you would write in an imperative language, in which you have some data structure in the background and you concentrate on your foreground computation:



```
plus = do x <- pop
          y <- pop
          push (x+y)

mult = do x <- pop
          y <- pop
          push (x*y)

res = do push 2
         push 5
         push 8
         x <- pop
         y <- pop
         push (x - y)
         mult
         pop
```



```
In [2]: stack = []

        def push(x):
            global stack
            stack = [x]+stack

        def pop():
            global stack
            temp = stack[0]
            stack = stack[1:]
            return temp

        def top():
            return stack[0]

        def plus():
            x = pop()
            y = pop()
            push (x+y)

        def mult():
            x = pop()
            y = pop()
            push (x*y)

        def prog():
            push(2)
            push(5)
            push(8)
            x = pop()
            y = pop()
            push(x-y)
            mult()
            return pop()

        print(prog())

        6
```

The State Monad: A Stack-based Evaluator

What we will do is consider the whole computation to be passing a long a pair consisting of the foreground value and the background state:

$$\left(\ \underbrace{\text{Integer}}_{\substack{\text{foreground}\\ \text{value}}}\ ,\ \underbrace{[\ \text{Integer}\ ]}_{\substack{\text{background}\\ \text{stack}}}\ \right)$$

Supposing we start with an empty stack and a 0 value, we could think of the computation going like this, with the background stack being hidden:

```
                                  ( 0, []      )
res = do push 2                   ( 2, [2]     )
         push 5                   ( 5, [5,2]   )
         push 8                   ( 8, [8,5,2] )
         x <- pop                 ( 8, [5,2]   )
         y <- pop                 ( 5, [2]     )
         push (x - y)             ( 3, [3,2]   )
         mult                     ( 6, [6]     )
         pop                      ( 6, []      )
```

```
  8
  6
  2
 ---

  2           x = 8
 ---          y = 5

  3
  2           x = 8
 ---          y = 5

  6           x = 8
 ---          y = 5
```

The State Monad: A Stack-based Evaluator

But then how to write our code?   We can't just do this:

```haskell
push :: (Integer, [Integer]) -> (Integer, [Integer])
push (x, xs) = (x, x:xs)

pop :: (Integer, [Integer]) -> (Integer, [Integer])
pop (_, (x:xs)) = (x, xs)

mult :: (Integer, [Integer]) -> (Integer, [Integer])
mult (_,(x:y:ys)) = (x*y,(x*y):ys)



res = let (_,stack1) = push (2,[])
          (_,stack2) = push (5,stack1)
          (_,stack3) = push (8,stack2)
          (x,stack4) = pop (0,stack3)
          (y,stack5) = pop (0,stack4)
          (z,stack6) = push (x-y,stack5)
          (_,stack7) = mult (0,stack6)
      in pop (0,stack7)
```

There is no monad, no foreground/background, and we are back where we started from two weeks ago!

The State Monad: A Stack-based Evaluator

The solution is to use a **curried** version of push that accepts two arguments, instead of a pair:

```
push :: (Integer, [Integer]) -> (Integer, [Integer])
push (x, xs) = (x, x:xs)
```

```
push :: Integer -> [Integer] -> (Integer, [Integer])
push    x       =  \xs        -> (x ,      x:xs     )
```

Notice that this version keeps the foreground parameter, and returns a background data type – a function on the background stack:

```
push :: Integer -> ([Integer] -> (Integer, [Integer]))
push    x       =  (\xs        -> (x ,      x:xs     ))
```

foreground value

background **function on** a stack

The State Monad: A Stack-based Evaluator

```
push :: Integer -> ([Integer] -> (Integer, [Integer]))
push    x       =  (\xs       -> (x ,       x:xs     ))
```

foreground
value

background
function on a stack

How does this function work?   It returns a function which pushes its value on a stack:

```
Main> push 1 [2,3,4]
(1,[1,2,3,4])
Main> sf = push 1
Main> sf []
(1,[1])
Main> sf [2,3,4]
(1,[1,2,3,4])
Main> :t sf
sf :: [Integer] -> (Integer, [Integer])
```

```
push 1 => sf = \xs -> (1, 1:xs)
```

$$\begin{tikzcd}
{[2,3,4]} \arrow[d] \\
\text{sf} \quad \boxed{\backslash xs \to (1,1{:}xs)} \arrow[d] \\
(1,[1,2,3,4])
\end{tikzcd}$$

The State Monad: A Stack-based Evaluator

But this isn't much better without monads:

```
push :: Integer -> ([Integer] -> (Integer, [Integer]))
push x  = \xs -> (x, x:xs)

pop ::  ([Integer] -> (Integer, [Integer]))
pop    = \(x:xs) -> (x, xs)

mult :: [Integer] -> (Integer, [Integer])
mult = \(x:y:ys) -> (x*y,(x*y):ys)


res = let (_,stack1) = push 2 []
          (_,stack2) = push 5 stack1
          (_,stack3) = push 8 stack2
          (x,stack4) = pop stack3
          (y,stack5) = pop stack4
          (z,stack6) = push (x-y) stack5
          (_,stack7) = mult stack6
      in pop stack7
```

The State Monad: A Stack-based Evaluator

Clearly we want to make this a monad so we can use do expressions:

```
type StackFunction a =  ([a] -> (a, [a]))

return :: a -> StackFunction a
return x = \xs -> (x, xs)

(>>=) :: StackFunction a -> (a -> StackFunction a) -> StackFunction a

sf >>= f = \stack -> let (x,newstack) = sf stack
                         sf2 = f x
                     in sf2 newstack


sf3 :: StackFunction Integer
sf3 = push 2 >>= (\_ -> (push 5)
             >>= (\_ -> (push 8)
             >>= (\_ -> pop
             >>= (\x -> pop
             >>= (\y -> (push (x-y))
             >>= (\z -> mult
             >>= (\_ -> pop)))))))

prog :: StackFunction Integer
prog = push 2      >>= \_ ->
       push 5      >>= \_ ->
       push 8      >>= \_ ->
       pop         >>= \x ->
       pop         >>= \y ->
       push (x-y) >>= \z ->
       mult        >>= \_ ->
       pop
```

```
prog2 :: StackFunction Integer
prog2 = do push 2
           push 5
           push 8
           x <- pop
           y <- pop
           z <- push (x-y)
           mult
           pop
```

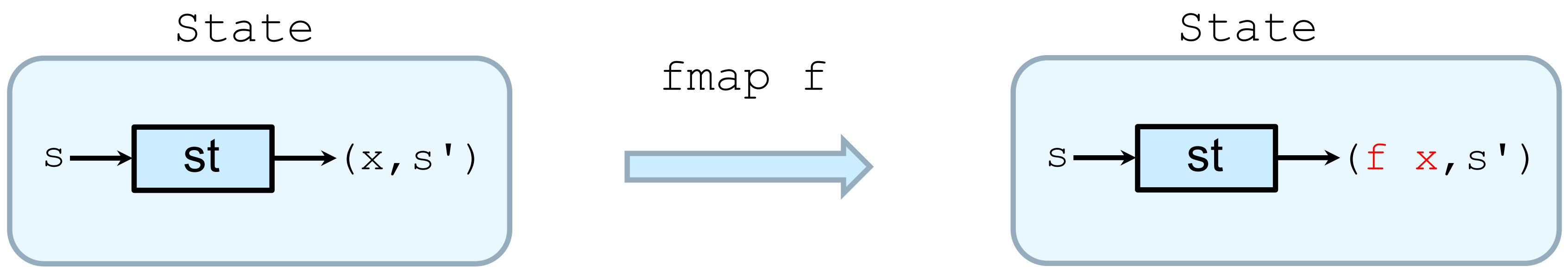
The State Monad: A General Framework

But let's generalize it so that it does not have to work with stacks as lists or even stacks at all! Let's just assume that we have a **foreground value** and a **background state.** Remember that we have to create a data type that we make an instance of `Monad`, so here is a polymorphic version, with a foreground type **`a`** and a background type **`s`**:

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

Now we have to make it a `Functor`, which means applying a function `f` to the foreground value being passed along, without changing the actual state:

```
instance Functor (State s) where
  -- fmap :: (a -> b) -> State s a -> State s b
  fmap f (State st) = State $ \s -> let (x,s') = st s
                                    in (f x, s')
```



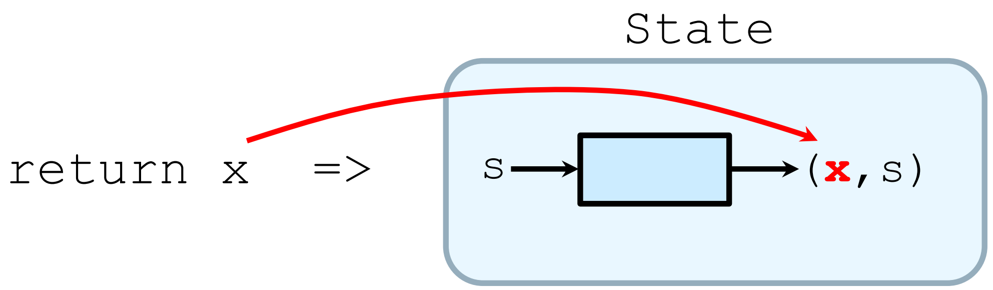
The State Monad: A General Framework

Then we have to make it an instance of `Monad` by defining the usual functions:

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

```haskell
instance Monad (State s) where

  -- return :: a -> State s a
  return x = State $ \s -> (x,s)
```

Return just inserts a value into the foreground without changing the state at all:

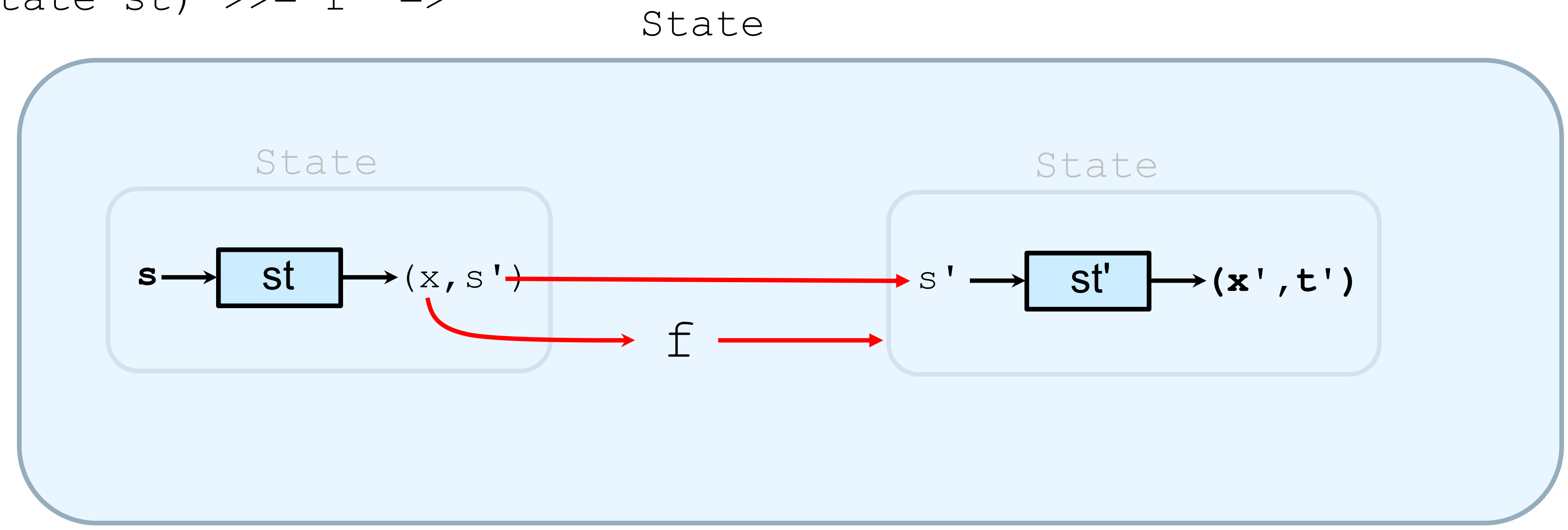


The State Monad: A General Framework

Bind more or less composes the state transition functions:

```haskell
instance Monad (State s) where

  -- return :: a -> State s a
  return x = State $ \s -> (x,s)

  -- (>>=) :: State s a -> (a -> State s b) -> State s b
  (State st) >>= f = State $ \s -> let (x,s') = st s
                                       (State st') = f x
                                   in st' s'
```

```
(State st) >>= f  =>
```



The State Monad: A Stack-based Evaluator

Now we can program our stack-based evaluator quite nicely:

```haskell
-- Basic Stack Operations

push :: Integer -> State [Integer] Integer
push n = State $ \s -> (n,n:s)

pop :: State [Integer] Integer
pop = State $ \(x:xs) -> (x,xs)

top :: State [Integer] Integer
top = State $ \(x:xs) -> (x,x:xs)


-- These do arithmetic on the stack

plus :: State [Integer] Integer
plus = do x <- pop
          y <- pop
          push (x+y)

mult :: State [Integer] Integer
mult = do x <- pop
          y <- pop
          push (x*y)
```

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

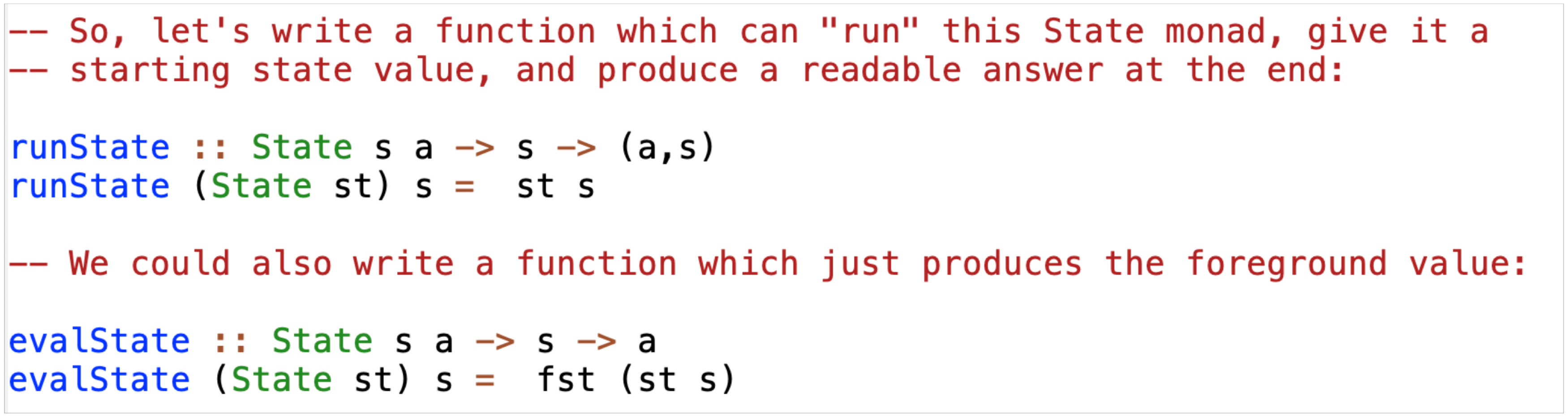
```haskell
                                 -- ( 0, []      )
prog = do push 2                 -- ( 2, [2]     )
          push 5                 -- ( 5, [5,2]   )
          push 8                 -- ( 8, [8,5,2] )
          x <- pop               -- ( 8, [5,2]   )
          y <- pop               -- ( 5, [2]     )
          push (x - y)           -- ( 3, [3,2]   )
          mult                   -- ( 6, [6]     )
          pop                    -- ( 6, []      )
```

```
Main> runState prog []
(6,[])

Main> runState prog [2,3,4]
(6,[2,3,4])
```

The State Monad: Basic Utility Functions

There are a number of basic functions for manipulating states that come in handy and are defined in Control.Monad.State:

First we have functions to initialize ("run") the state monad by providing an initial value for the state, and return either the pair or just the foreground value:



```haskell
-- So, let's write a function which can "run" this State monad, give it a
-- starting state value, and produce a readable answer at the end:

runState :: State s a -> s -> (a,s)
runState (State st) s =  st s

-- We could also write a function which just produces the foreground value:

evalState :: State s a -> s -> a
evalState (State st) s =  fst (st s)
```

The State Monad: Basic Utility Functions

Next, we have functions for basic communication between the background and foreground:

```haskell
-- Insert a new state into the computation (modify
-- the background from the foreground). Because
-- we don't need the value, we assume it is unit (Haskell ()).
-- Unit is like a null value: unit is what a function returns that doesn't
-- actually return anything. The same notation is used for both
-- the null data data value and its type:   ()::().

put :: s -> State s ()
put s = State $ \_ -> ((), s)

-- Extract the state from background into the foreground

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

```haskell
example :: (Integer,[Integer])
example = runState (do put [2,3]
                       stack <- get
                       put (1:stack)
                       return 0)
                    []
```

```
Main> example
(0,[1,2,3])
```

The State Monad: A Improved Stack-based Evaluator

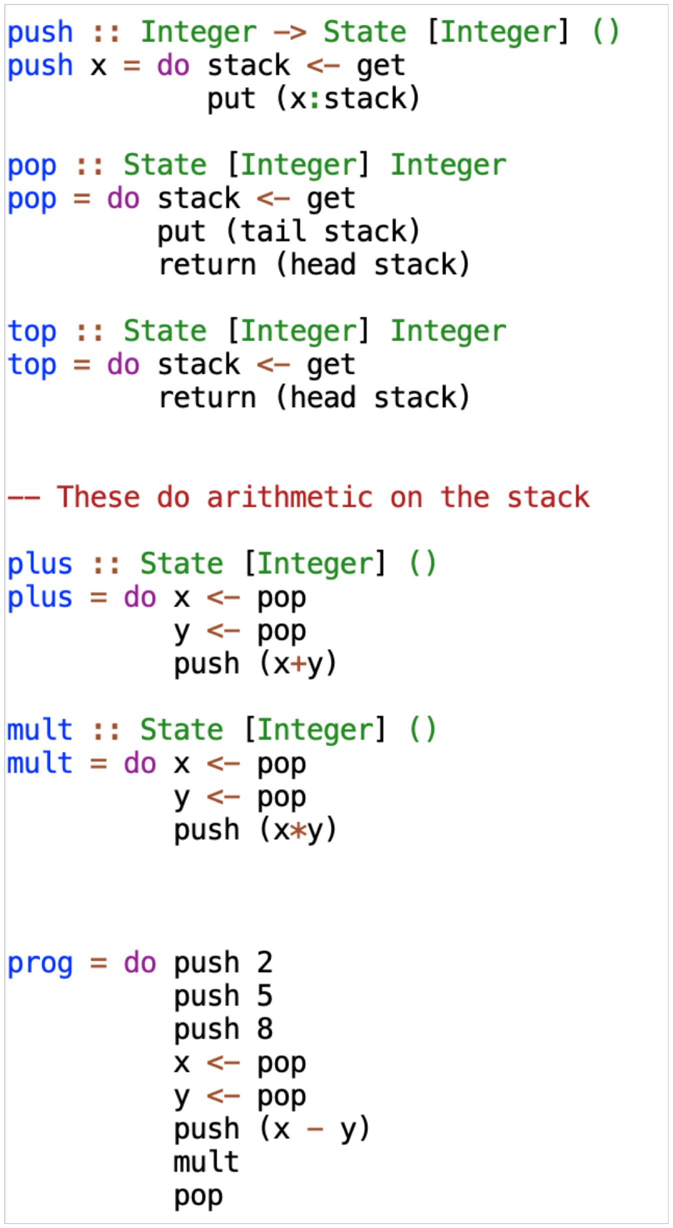
Now we COULD rewrite our stack code so that it doesn't refer to the State at all:

```haskell
push :: Integer -> State [Integer] ()
push x = do stack <- get
            put (x:stack)

pop :: State [Integer] Integer
pop = do stack <- get
         put (tail stack)
         return (head stack)

top :: State [Integer] Integer
top = do stack <- get
         return (head stack)


-- These do arithmetic on the stack

plus :: State [Integer] ()
plus = do x <- pop
          y <- pop
          push (x+y)

mult :: State [Integer] ()
mult = do x <- pop
          y <- pop
          push (x*y)



prog = do push 2
          push 5
          push 8
          x <- pop
          y <- pop
          push (x - y)
          mult
          pop
```

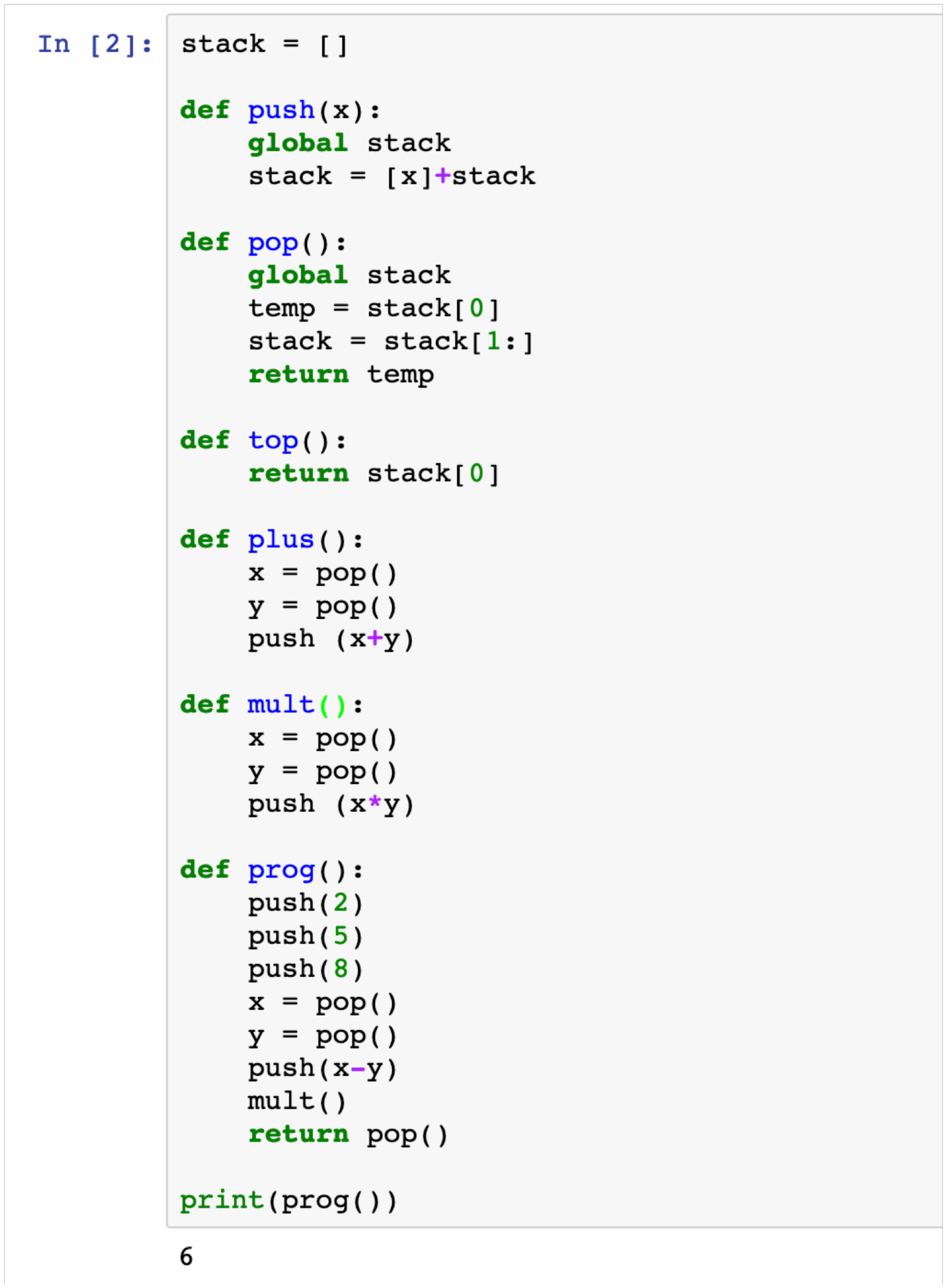
```
Main> evalState prog []
6

Main> runState prog []
(6,[])

Main> runState prog [2,3,4]
(6,[2,3,4])
```

The State Monad: A Improved Stack-based Evaluator

It is instructive to compare this with Python:



```haskell
push :: Integer -> State [Integer] ()
push x = do stack <- get
            put (x:stack)

pop :: State [Integer] Integer
pop = do stack <- get
         put (tail stack)
         return (head stack)

top :: State [Integer] Integer
top = do stack <- get
         return (head stack)


-- These do arithmetic on the stack

plus :: State [Integer] ()
plus = do x <- pop
          y <- pop
          push (x+y)

mult :: State [Integer] ()
mult = do x <- pop
          y <- pop
          push (x*y)



prog = do push 2
          push 5
          push 8
          x <- pop
          y <- pop
          push (x - y)
          mult
          pop
```



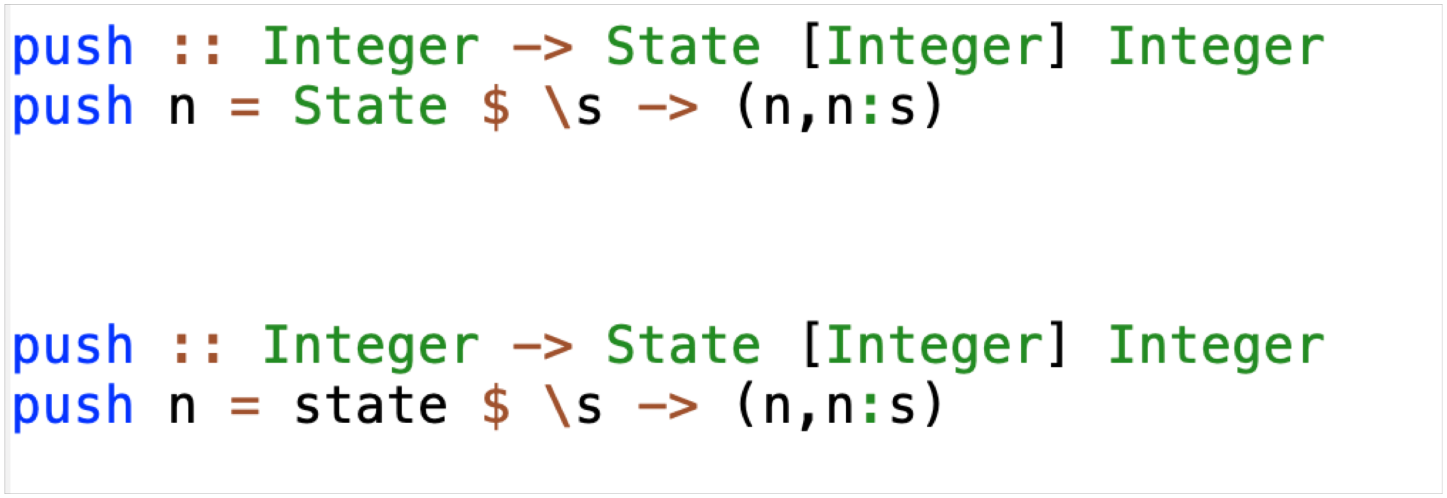
```python
In [2]: stack = []

        def push(x):
            global stack
            stack = [x]+stack

        def pop():
            global stack
            temp = stack[0]
            stack = stack[1:]
            return temp

        def top():
            return stack[0]

        def plus():
            x = pop()
            y = pop()
            push (x+y)

        def mult():
            x = pop()
            y = pop()
            push (x*y)

        def prog():
            push(2)
            push(5)
            push(8)
            x = pop()
            y = pop()
            push(x-y)
            mult()
            return pop()

        print(prog())
```

```
6
```

Control.Monad.State

Everything we have done is consistent with the Haskell library `Control.Monad.State` with one small exception:

The `Control.Monad.State` library does not use a constructor `State`, but a function `state`, so anywhere you would use the constructor `State` you have to use `state`:



```haskell
push :: Integer -> State [Integer] Integer
push n = State $ \s -> (n,n:s)


push :: Integer -> State [Integer] Integer
push n = state $ \s -> (n,n:s)
```

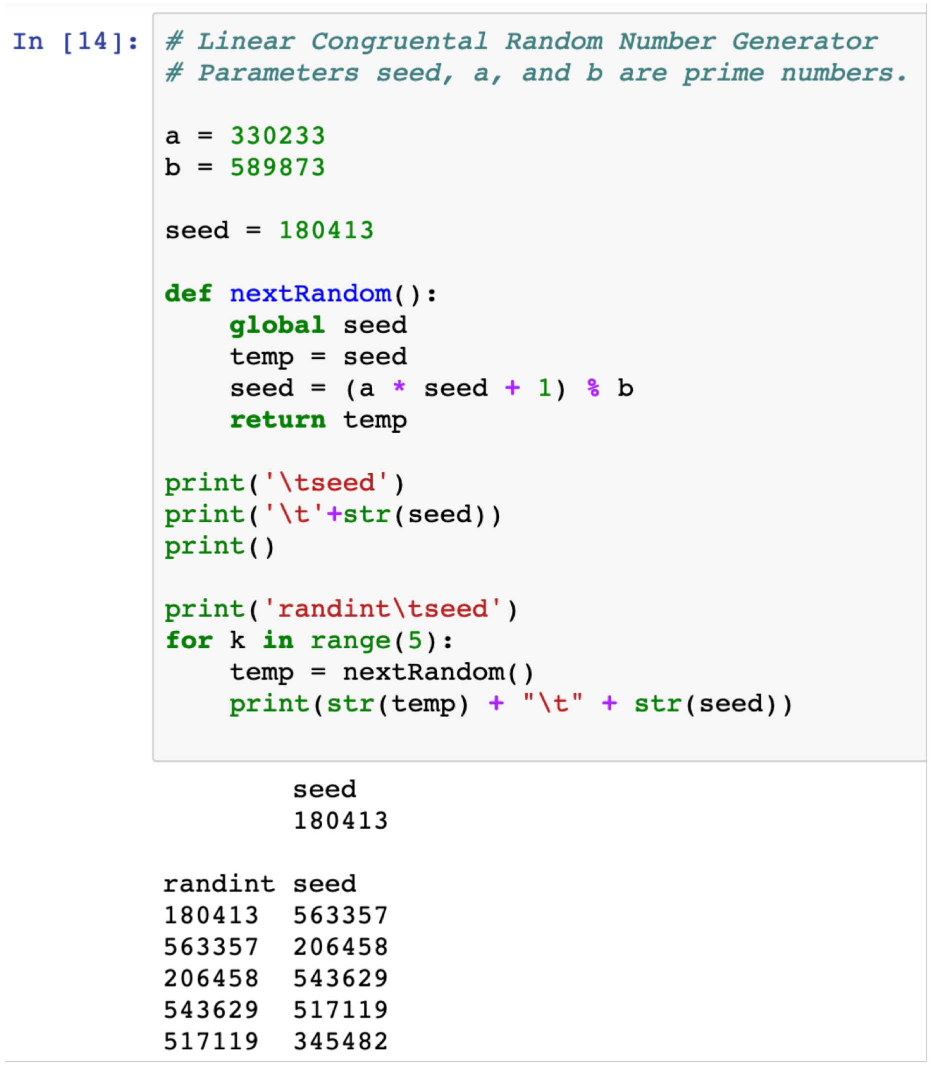
That's it! Otherwise it is just as we have seen.... let's try a few more examples!

Control.Monad.State

```haskell
-- Example 2, essentially same as writer monad

write :: String -> State [String] ()
write str = do strs <- get
               put (strs ++[str])

plus' :: Integer -> Integer -> State [String] Integer
plus' x y = return $ x + y

mult' :: Integer -> Integer -> State [String] Integer
mult' x y = return $ x * y




ex7 :: (Integer,[String])
ex7 = runState (do x <- plus' 8 7
                   write $ "8 + 7 => " ++ show x
                   y <- mult' x 2
                   write $ show x ++ " * 2 => " ++ show y
                   return y
               ) []

-- Main> ex7
-- (30,["8 + 7 => 15","15 * 2 => 30"])
```

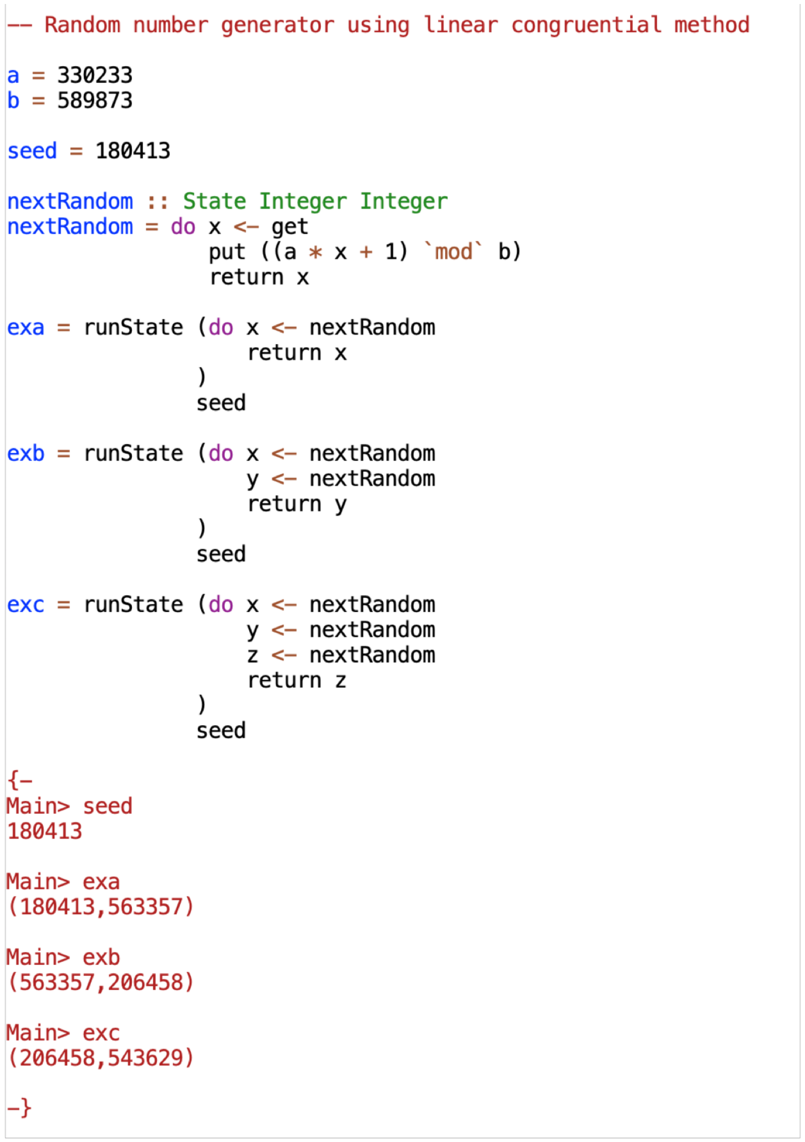
Control.Monad.State

As a final exercise, let's compare a Python Random-Number Generator with the same exact algorithm in Haskell:



```
In [14]: # Linear Congruential Random Number Generator
         # Parameters seed, a, and b are prime numbers.

         a = 330233
         b = 589873

         seed = 180413

         def nextRandom():
             global seed
             temp = seed
             seed = (a * seed + 1) % b
             return temp

         print('\tseed')
         print('\t'+str(seed))
         print()

         print('randint\tseed')
         for k in range(5):
             temp = nextRandom()
             print(str(temp) + "\t" + str(seed))
```

```
        seed
        180413

randint seed
180413  563357
563357  206458
206458  543629
543629  517119
517119  345482
```



```
-- Random number generator using linear congruential method

a = 330233
b = 589873

seed = 180413

nextRandom :: State Integer Integer
nextRandom = do x <- get
                put ((a * x + 1) `mod` b)
                return x

exa = runState (do x <- nextRandom
                   return x
               )
               seed

exb = runState (do x <- nextRandom
                   y <- nextRandom
                   return y
               )
               seed

exc = runState (do x <- nextRandom
                   y <- nextRandom
                   z <- nextRandom
                   return z
               )
               seed

{-
Main> seed
180413

Main> exa
(180413,563357)

Main> exb
(563357,206458)

Main> exc
(206458,543629)

-}
```