(Not) Monads

Mainly `IO`, plus `Maybe`

LYaH:

- Chapter 9, *Input/Output*, section 1 ("Hello World").

Monads

A *monad* is a type constructor `m` for which the following two operations are defined.

```
return :: a -> M a
(>>=) :: m a -> (a -> m b) -> m b
```

???

E.g. for `m` the list-type constructor:

```
return x = [x]
xs >>= f = concat (map f xs)
```

???

We'll focus on two main applications: IO and Maybe-chaining, *ignoring* their monad basis.

IO in other languages

Most programming languages, e.g. Python, Java, C/C++:

1. Allow free mix of computation and IO.
2. Give no way to ensure imported methods are side-effect-free.

This makes parallelization and reasoning difficult.
Consider a function/method call `f(e1,e2)`.

1. Evaluate arguments e1 and e2 in parallel? Race condition possible.
2. Order of evaluation matters: can't reason by just plugging in argument expressions for variables in the body of `f`.

Haskell's approach to IO

In a nutshell: use the type system to prevent all IO within normal computation, forcing it to the "top level" of the program.

- `IO a` is the type of *IO actions* producing a value of type `a` only when *run*.
- computing a value of type `IO a` does *not* do any IO.
- E.g. `putStrLn:: String -> IO()` is a "built-in" that prints a line.
  Evaluating `putStrLn "foo"` creates an IO action value but does *not* do any IO.
- What can we do with an IO action besides combine it with other ones?
  i. Run it as the *main* program.
  ii. Run it at the `ghci` prompt.

What is an action?

An action

1. Specifies some IO to do, input and/or output, when the action is run.
2. Specifies what *value* should be produced when the action is run. E.g. for a "read" action, the input value is produced.

Key idea: *evaluating* an expression of type `IO a` produces an *action* which can be *run* only as *main* or at the `ghci` prompt.

Some built-in actions:

```
readFile :: String -> IO String  -- value of (readFile f) is the contents
putStrLn :: String -> IO ()      -- () is for no, or "don't care" value
```

Building IO actions

We use a special *do* notation (for monads).

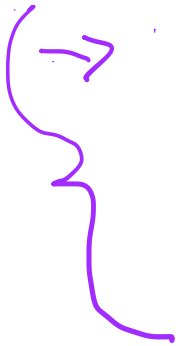
```
do
  -- line 1
  -- line 2
  -- ...
  -- line n
```

where each line is one of the following.

1. `e` where `e` is any expression of type `IO a`.
2. `x <- e` where `e` is any expression of type `IO a`.
3. `let x = e` where `e` is an expression of any type.

The `do` expression, when *evaluated*, evaluates to an action.

When the action is *run*, as `main` or at the `ghci` prompt, the lines are *run* in order.



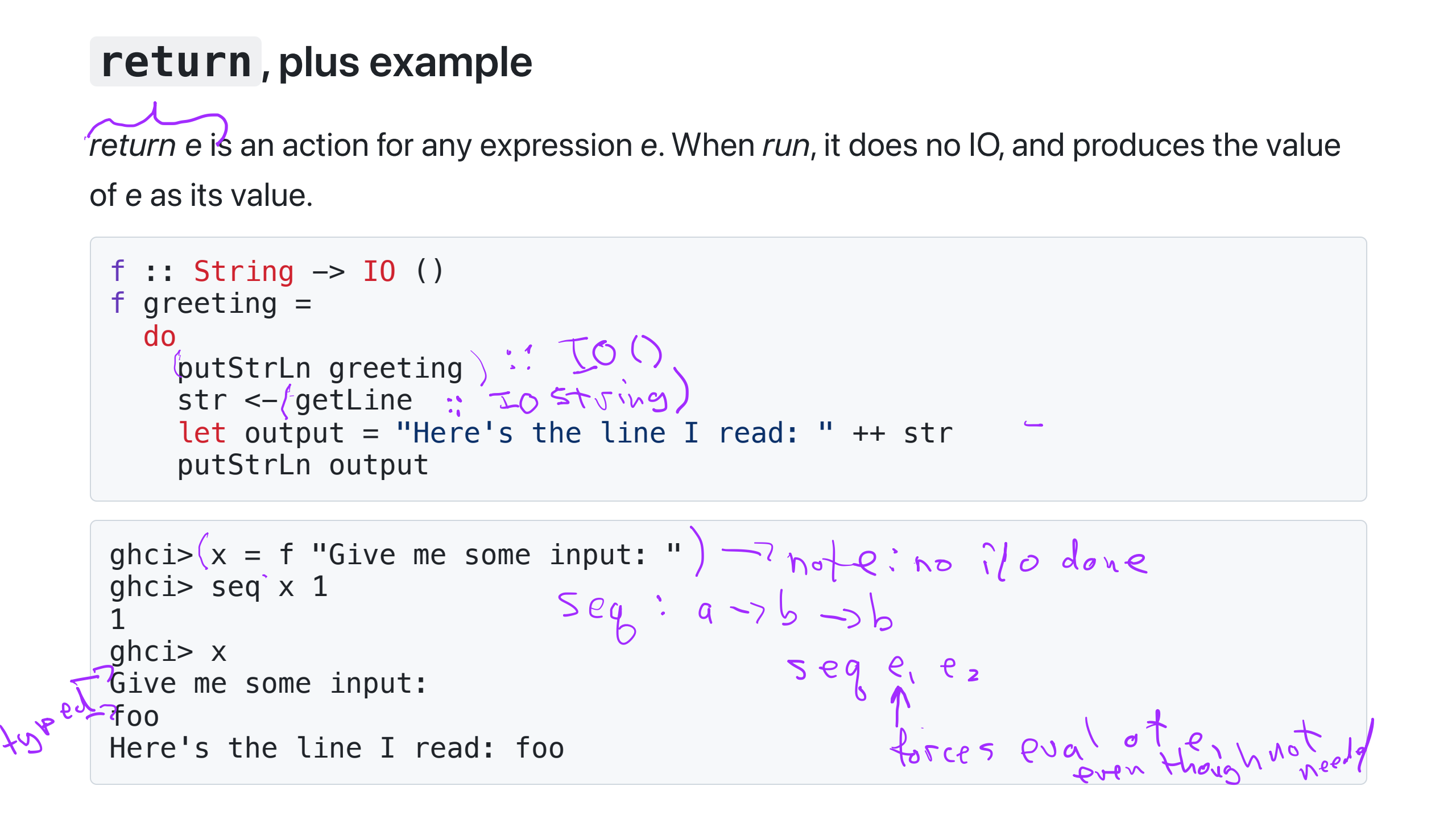
1. `x <- e`:
   The action `e` is run, and the value produced by running it is named `x`
2. `e`
   Must be a IO action -- run it. (Special case of 1.)
3. `let x = e`: the value of *e* is a "normal" value and is named `x`

`return`, plus example

return e is an action for any expression *e*. When *run*, it does no IO, and produces the value of *e* as its value.

```
f :: String -> IO ()
f greeting =
  do
    putStrLn greeting
    str <- getLine
    let output = "Here's the line I read: " ++ str
    putStrLn output
```

```
ghci> x = f "Give me some input: "
ghci> seq x 1
1
ghci> x
Give me some input:
foo
Here's the line I read: foo
```



Example: interactive stack-based calculator.

Program maintains a stack. User enters an "operator", resulting stack is displayed.

Op	Action
<number>	push the number onto the stack
*	replace top two elements with their product
+	replace top two elements with their sum
/	replace top two elements with the result of dividing
-	negate the top element
c	clear the stack

Example interaction.

User Input	Displayed stack
c	
17	17
3	3:17
1	1:3:17
-	-1:3:17
+	2:17
*	34