CSCI-UA 490

Monads

Edward Z. Yang

(w/ help from Simon Peyton Jones, Kathleen Fisher and John Mitchell)

Beauty

FP is mathematics; it is beautiful!

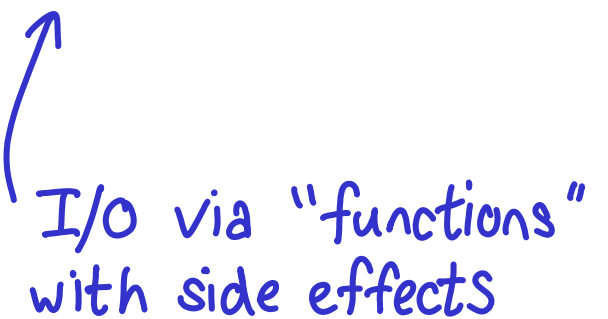
Elegant & powerful abstractions

the "Awkward squad":
input/output, error recovery, concurrency, FFI

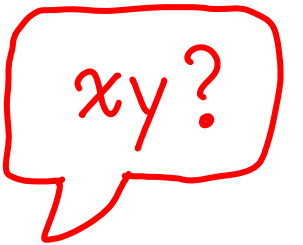
and the Beast

Direct approach

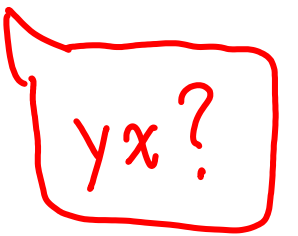
```
putchar 'x' + putchar 'y'
```



~~Direct approach~~ Laziness!




```
putchar 'x' + putchar 'y'
```

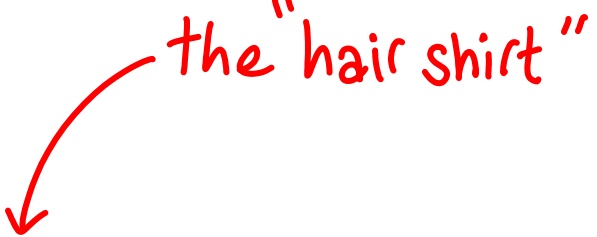


depends on evaluation order

~~Direct approach~~ Laziness!

```
ls = [putchar 'x', putchar 'y']

> length ls  ⇝ nothing
> head ls  ⇝ x
()
```



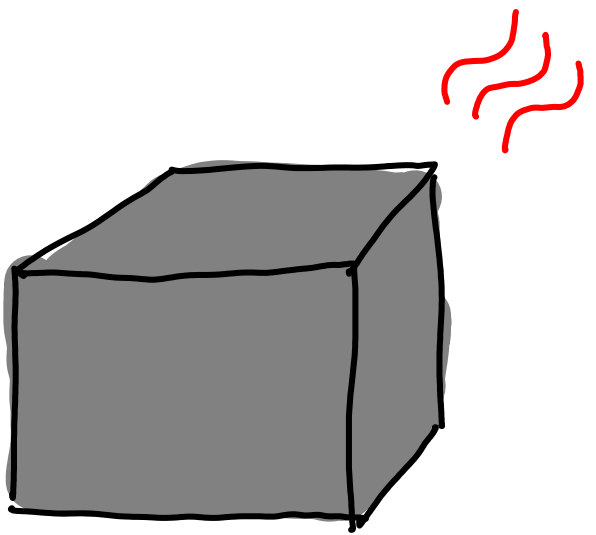
Laziness forces us to take a different, more principled approach

(lessons applicable to strict languages too)

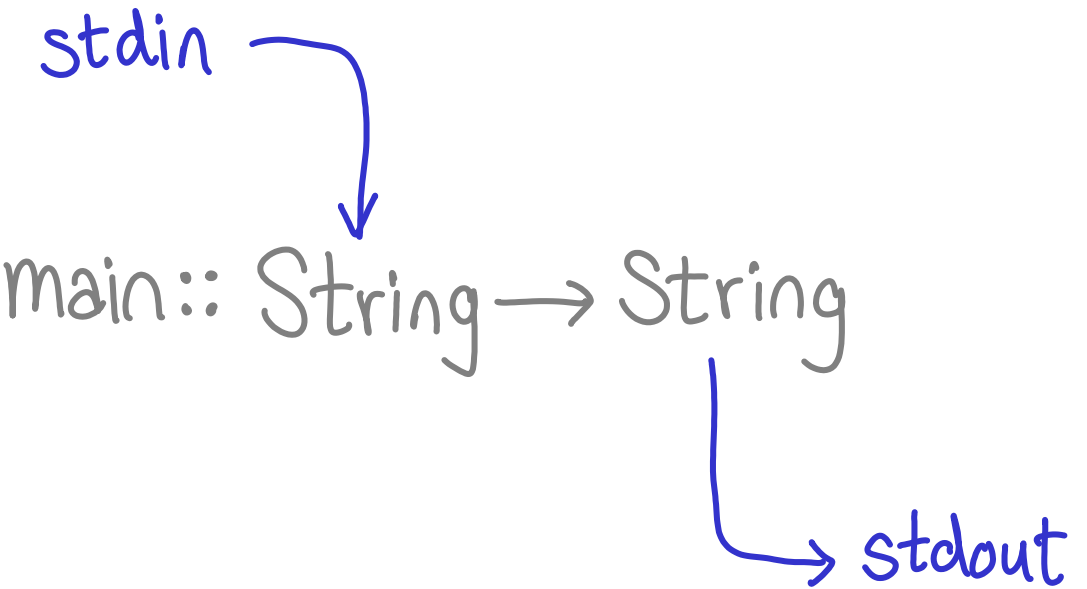
The tension

Functional programs define pure functions with no side effects

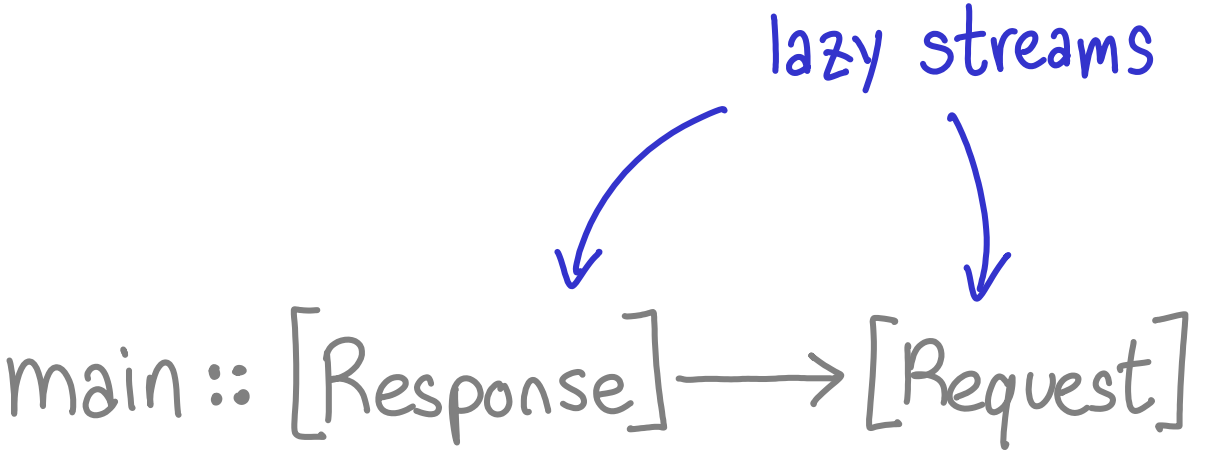
The whole point of running a program is to have a side effect



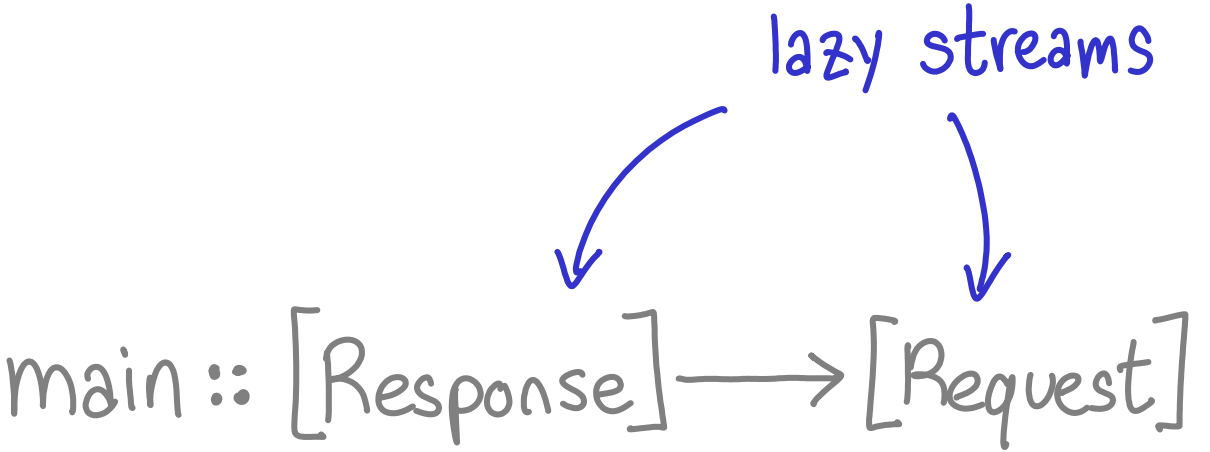
Functional I/O



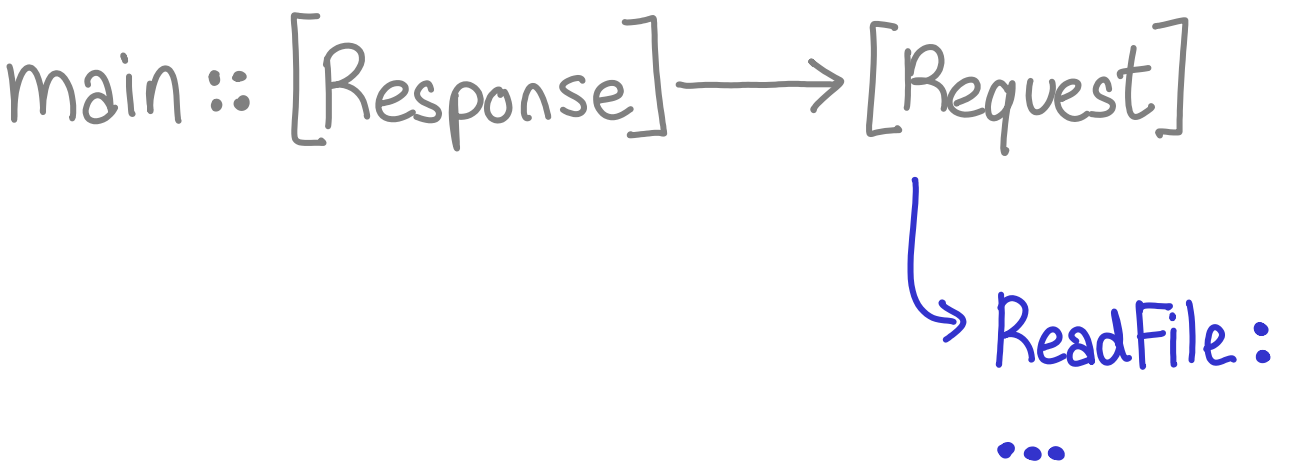
Functional I/O



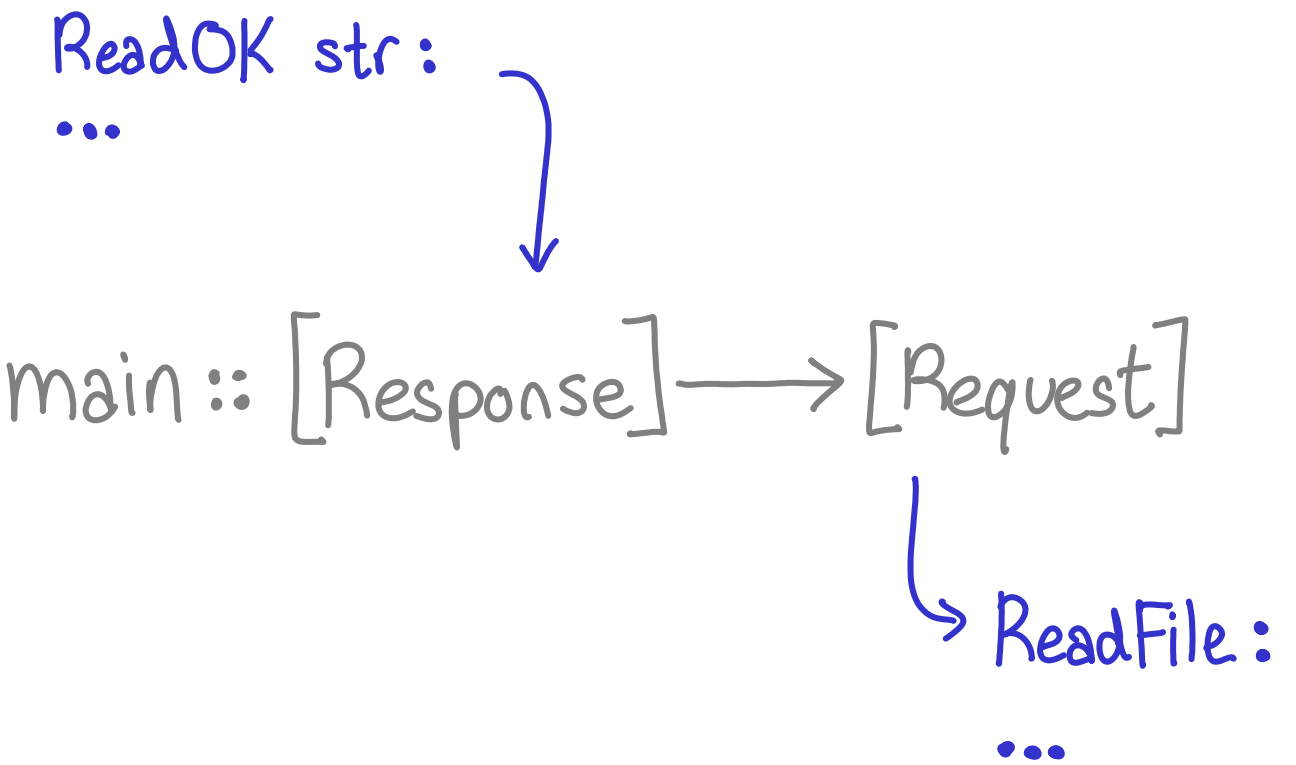
Functional I/O



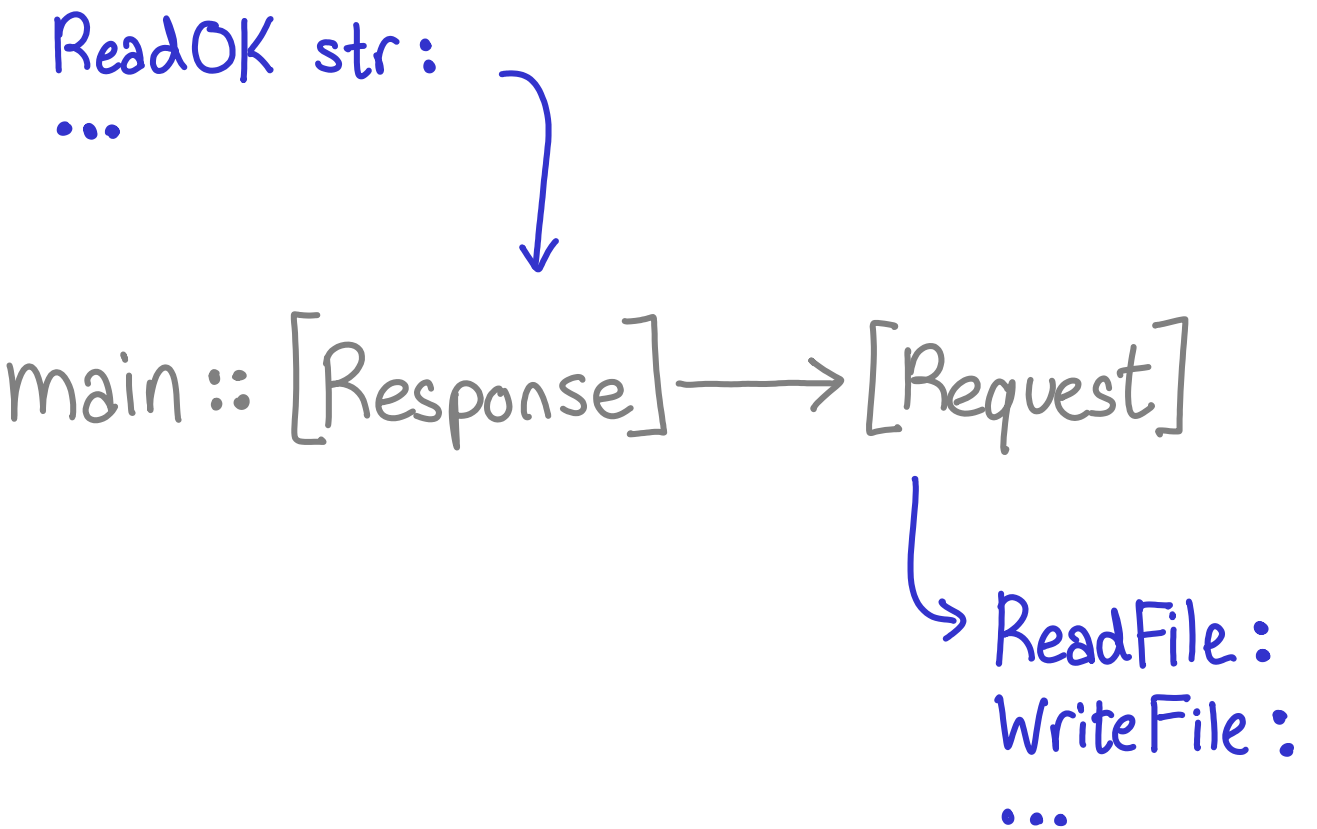
Functional I/O



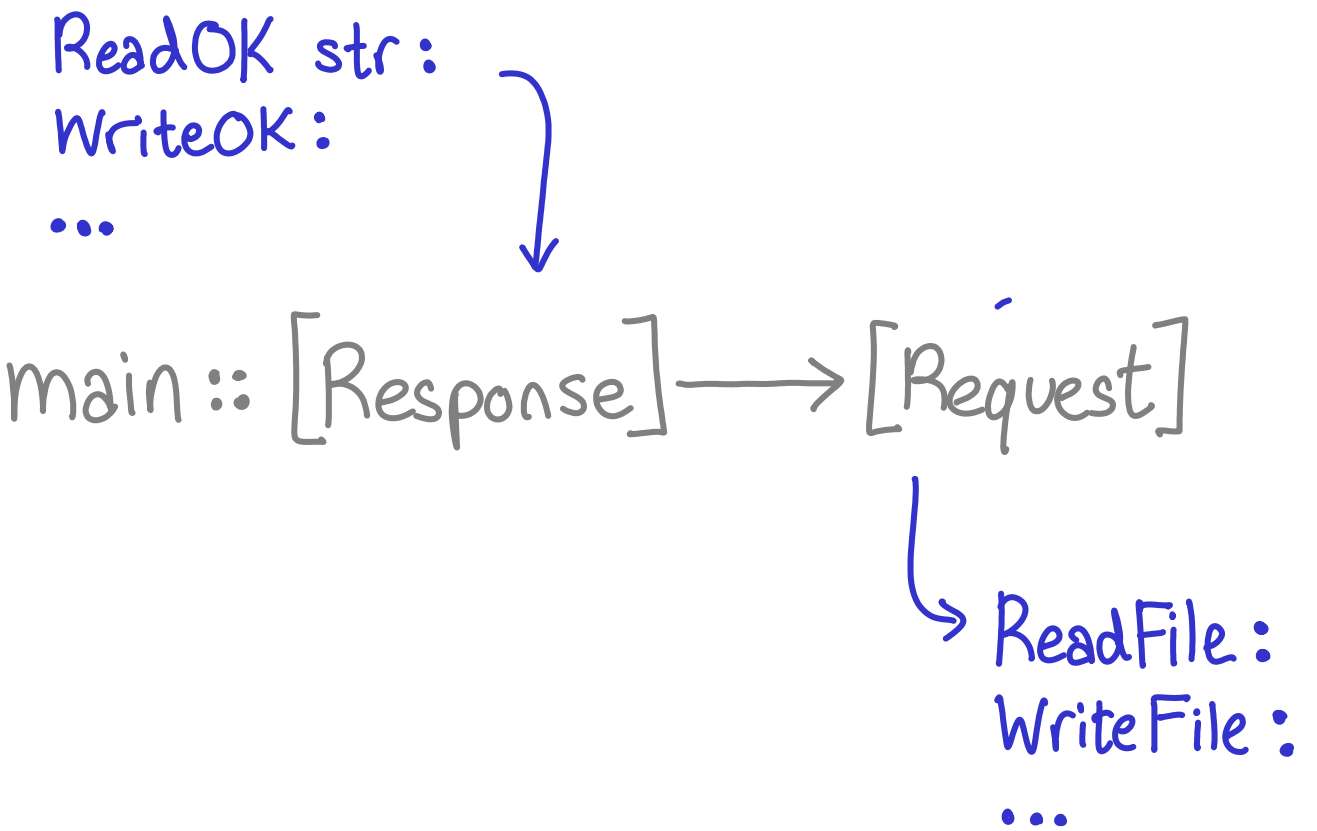
Functional I/O



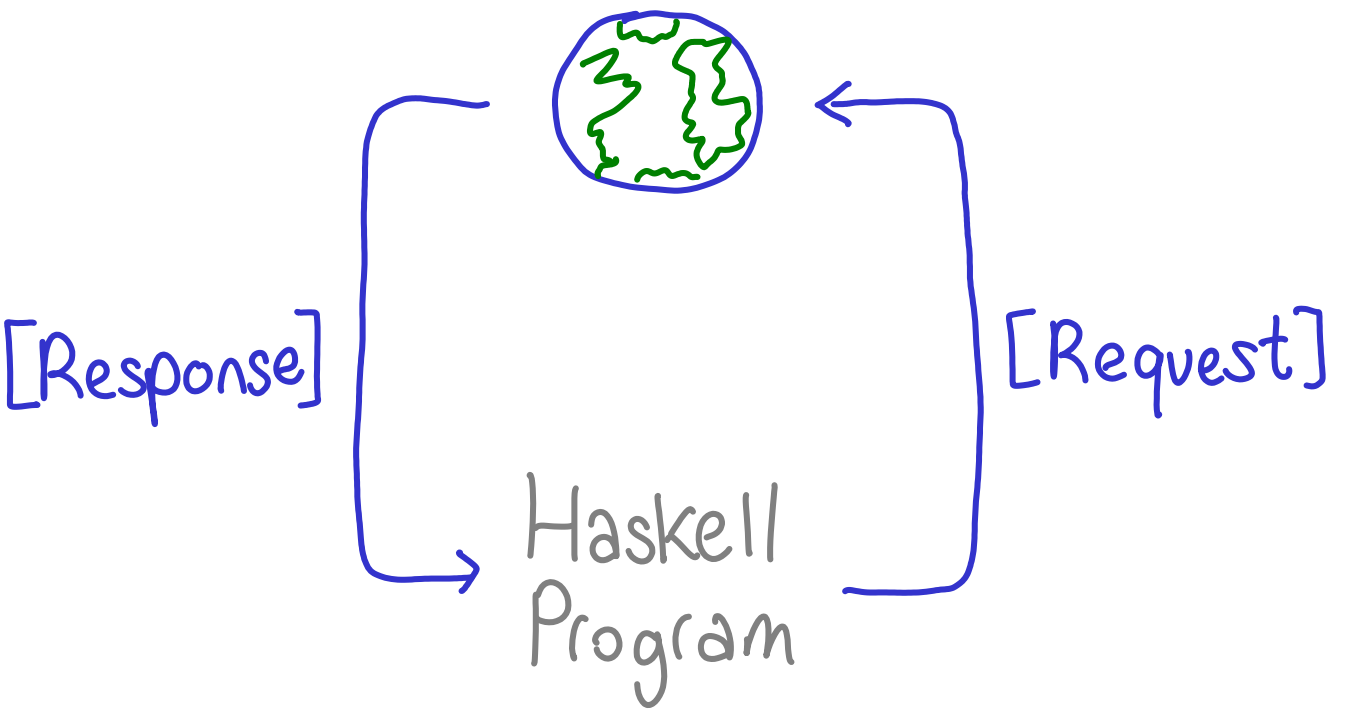
Functional I/O



Functional I/O



Functional I/O



Functional I/O: Awkward!

- Hard to extend
- Hard to use

Deadlock!

Monadic I/O: the Key idea

A value of type (IO a) is an "action" that, when performed, may do some I/O before delivering a result of type a.

Monadic I/O: the Key idea

A value of type (IO a) is an "action" that, when performed, may do some I/O before delivering a result of type a.

```
type IO a = World → (a, World)
```

Actions are first class

A value of type (IO a) is an "action" that, when performed, may do some I/O before delivering a result of type a.

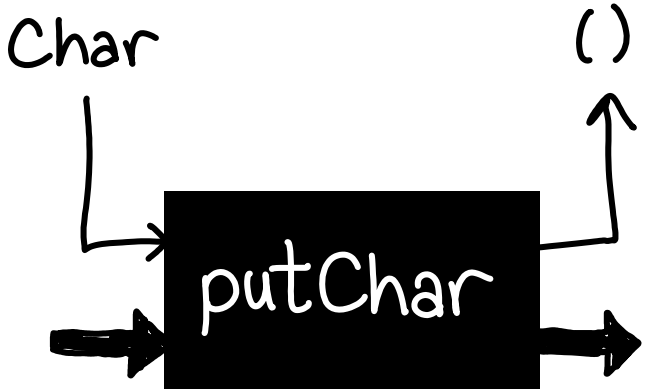
```
type IO a = World → (a, World)
```

Evaluating an action has no effect;
perfoming the action has an effect

Evaluating vs. Performing

Recipe for Swedish Meatballs

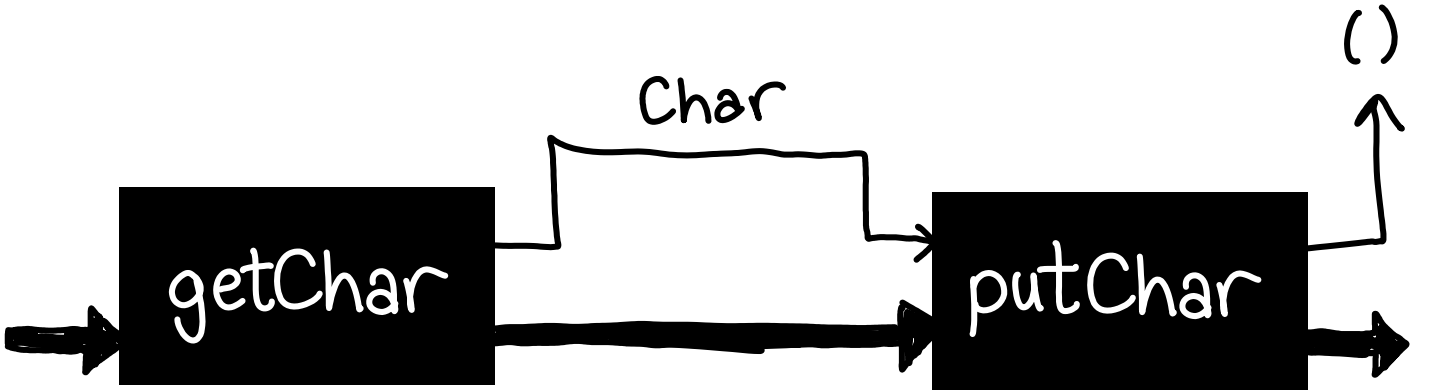
1. ~~~~~~~~~~
2. ~~~~~~~~~~
3. ~~~~~~~~~~
   ~~~~~~~~~~
   ~~~~~~~~~~



getChar :: IO Char

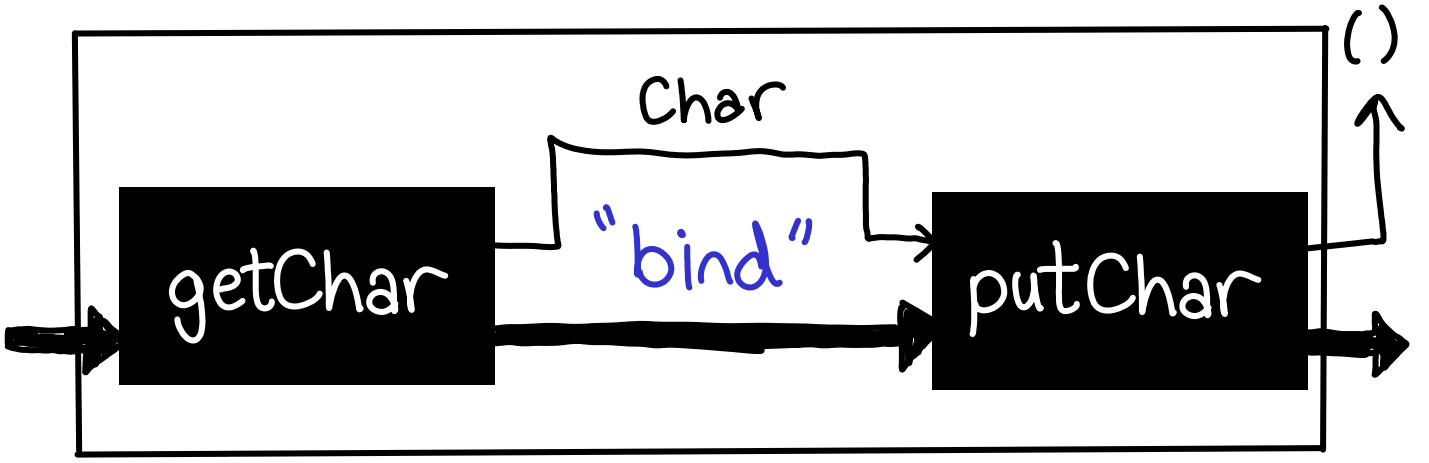
putChar :: Char → IO()

```
main :: IO ()
main = putChar 'x'
```



Read a character and write it back out

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



a new, bigger action!

```
echo :: IO ()
echo = getChar >>= putChar
```

Print a character twice

```
echoDup :: IO ()
echoDup = getChar >>= (\c ->
          putChar c >>= (\() ->
          putChar c ))
```

nothing interesting

(parentheses optional)

The (>>) combinator

```
echoDup :: IO ()
echoDup = getChar >>= \c ->
          putChar c >>
          putChar c
```

```
(>>) :: IO a -> IO b -> IO b
m >> n = m >>= \_ -> n
```

The return combinator

↑ not short circuiting

```
getTwoChars :: IO (Char, Char)
getTwoChars = getChar >>= \c1 ->
              getChar >>= \c2 ->
              ???
```

The return combinator

↑ not short circuiting

```
getTwoChars :: IO (Char, Char)
getTwoChars = getChar >>= \c1 ->
              getChar >>= \c2 ->
```

✗ (c1, c2)

Got (Char, Char), expecting IO (Char, Char)

The return combinator

↑ not short circuiting

```
getTwoChars :: IO (Char, Char)
getTwoChars = getChar >>= \c1 ->
              getChar >>= \c2 ->
              return (c1, c2)
```


```
return :: a -> IO a
```

Do notation

```
getTwoChars :: IO (Char, Char)
getTwoChars = getChar >>= \c1 ->
              getChar >>= \c2 ->
              return (c1, c2)
```

versus

```
c1 = getchar();
c2 = getchar();
return (c1, c2)
```

imperative

Do notation

```
getTwoChars :: IO (Char, Char)
getTwoChars = do { c1 ← getChar;
                   c2 ← getChar;
                   return (c1, c2) }
```

syntax sugar

Do notation

$$\texttt{do}\,\{\,x \leftarrow e;\; s\,\} \equiv e \gg\!= \backslash x \rightarrow \texttt{do}\,\{s\}$$

$$\texttt{do}\,\{e;\; s\} \equiv e \gg \texttt{do}\,\{s\}$$

$$\texttt{do}\,\{e\} \equiv e$$

GHCi examples

- Global state `configFile :: [String]`
- Configuration variables

Trouble?

```
unsafePerformIO :: IO a -> a
```

forall !

```
r :: IORef a
r = unsafePerformIO (newIORef undefined)
```

```
cast :: b -> IO c
cast x = do
    writeIORef r x
    readIORef r
```

Implementing the IO monad

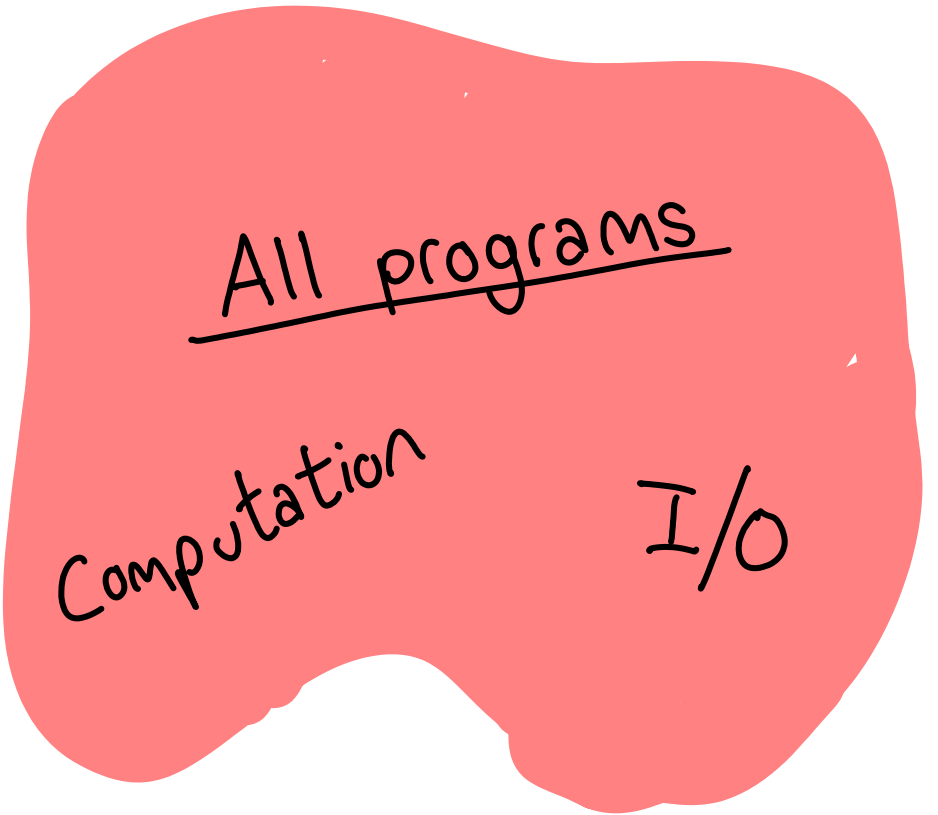
"type IO a = World → (a, World)"

... literally!

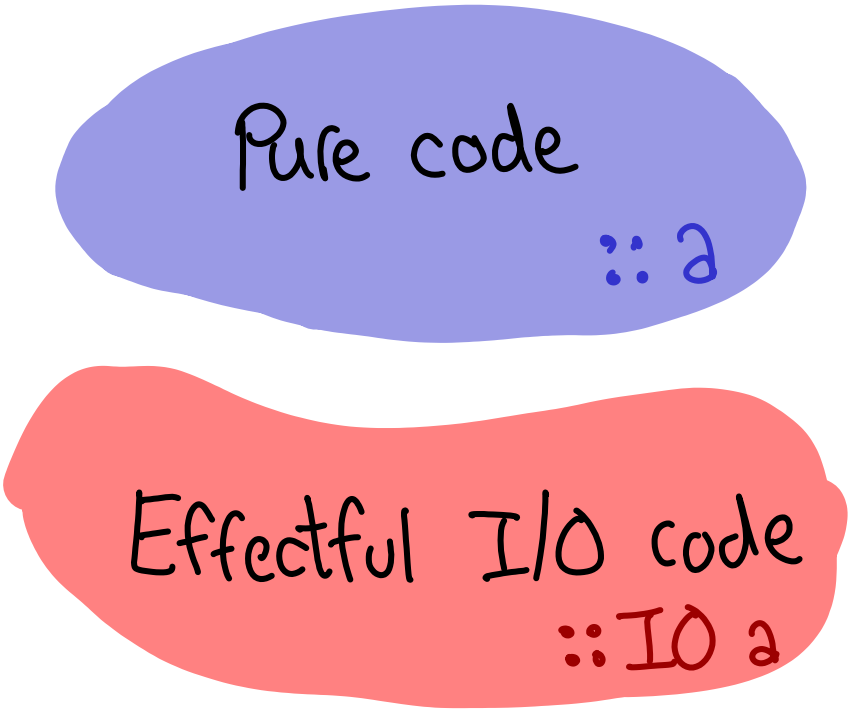
unforgeable token

Standard FP optimizations apply
(threading ensures linearity)

Comparison: Traditional Languages

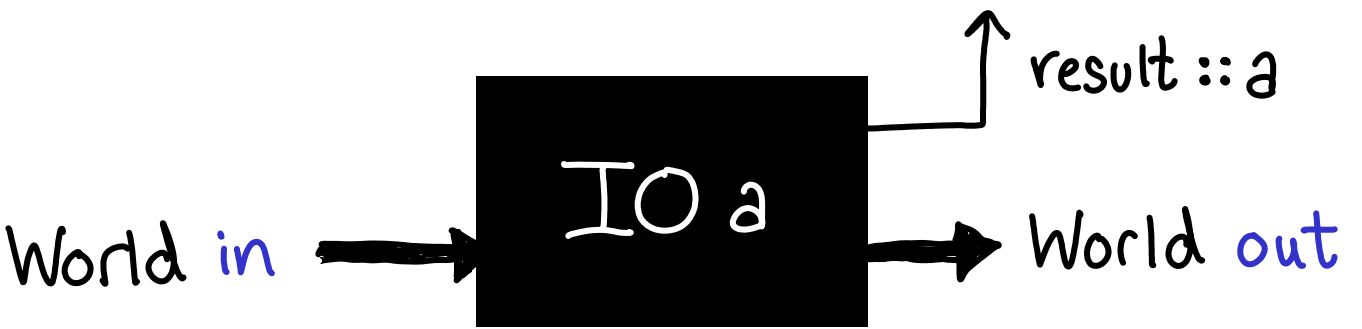


Comparison: Haskell



Monadic I/O: the Key idea

A value of type (IO a) is an "action" that, when performed, may do some I/O before delivering a result of type a.




```
class Monad m where
    (>>=) :: m a → (a → m b) → m b
    return :: a → m a
```

Not just for IO

- Maybe monad
- State monad

Just define >>= and return

Subject to laws

>>= is "associative"

return is "identity"