Functional programming

I/O

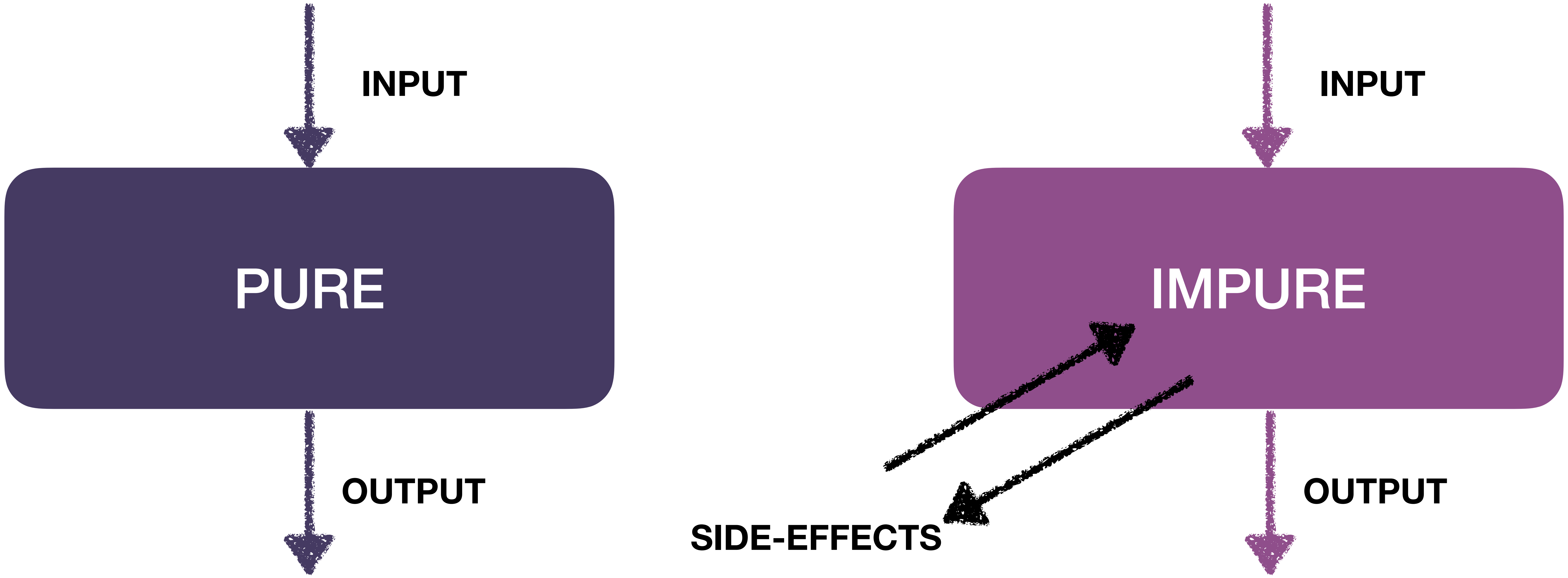
Function application

- sqrt 3 + 4 + 9
- sqrt $ 3 + 4 + 9

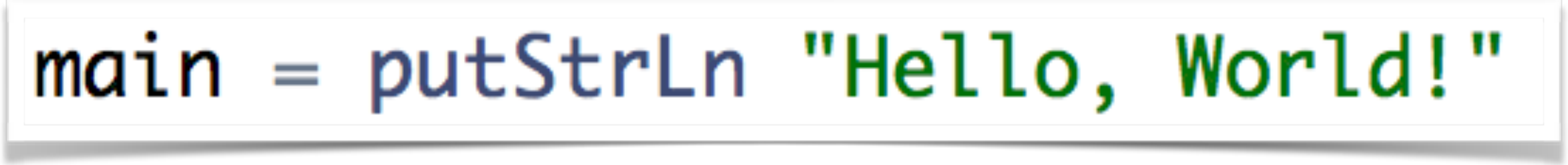
Function composition

- map (\xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]
- map (negate . sum . tail) [[1..5],[3..6],[1..7]]

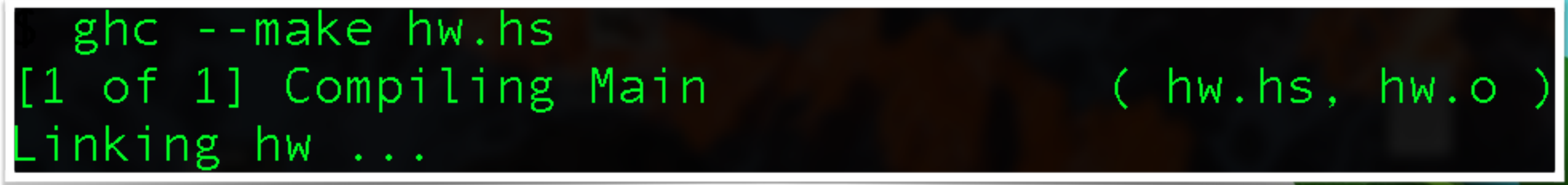
Practice makes perfect



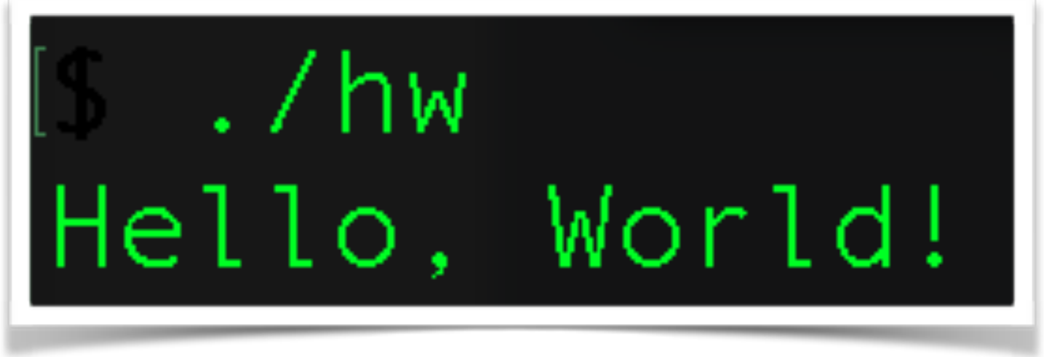
Meet an outer world



```haskell
main = putStrLn "Hello, World!"
```

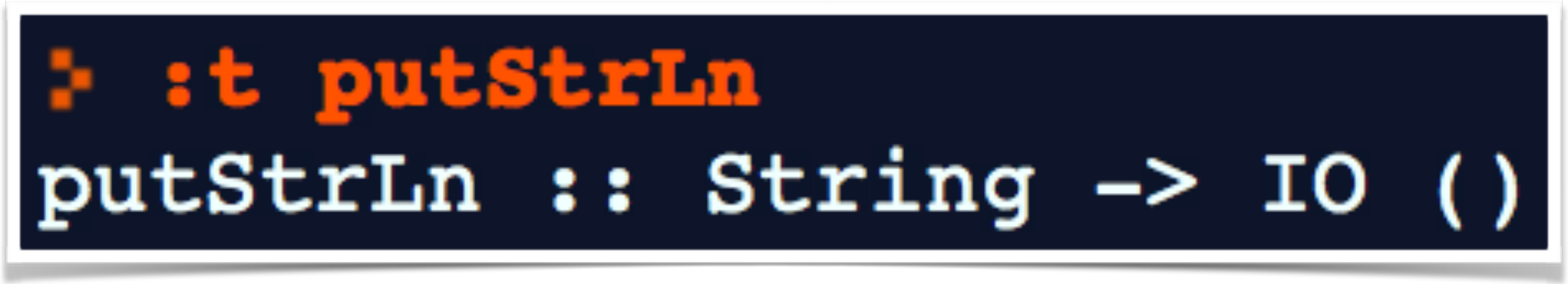


```
$ ghc --make hw.hs
[1 of 1] Compiling Main             ( hw.hs, hw.o )
Linking hw ...
```

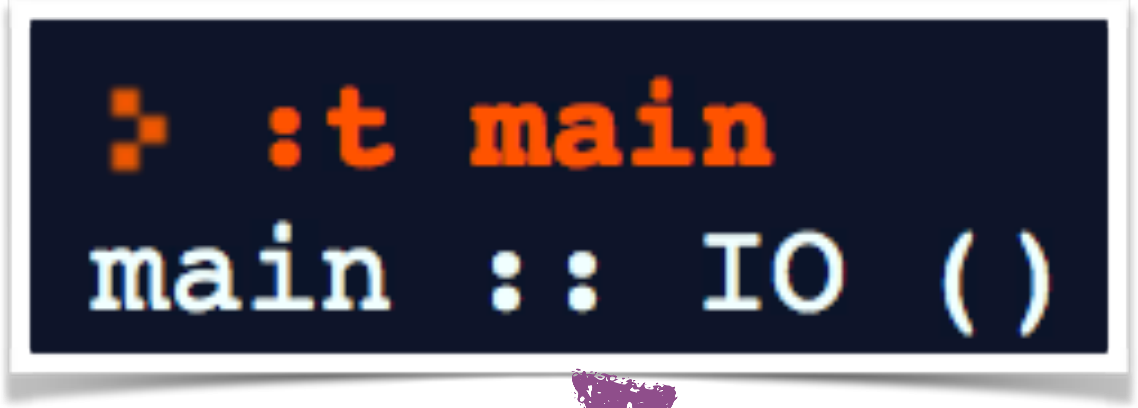


```
$ ./hw
Hello, World!
```

IO type



```
> :t putStrLn
putStrLn :: String -> IO ()
```

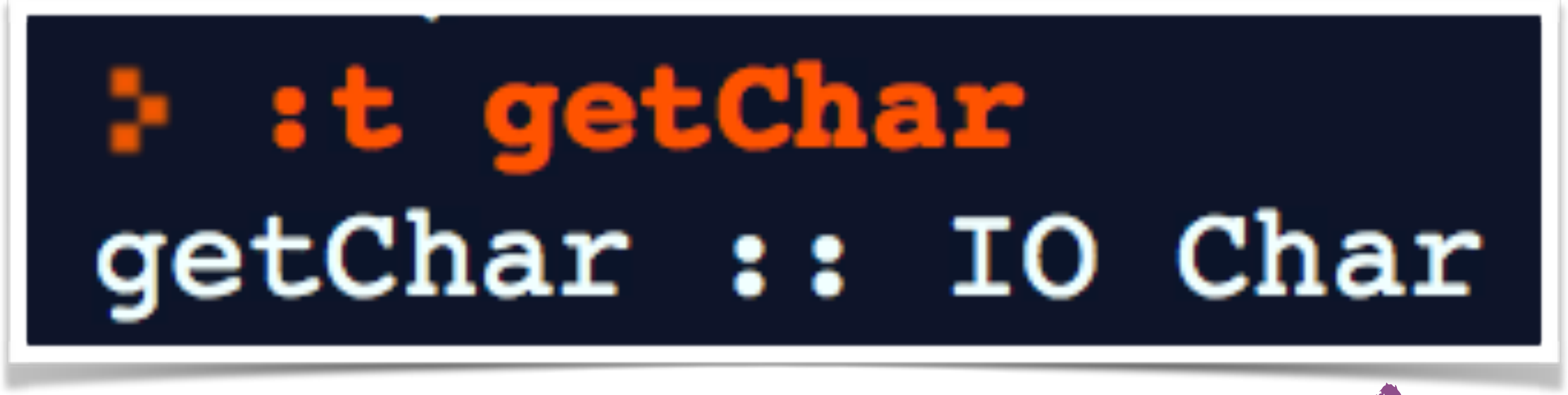


```
> :t main
main :: IO ()
```

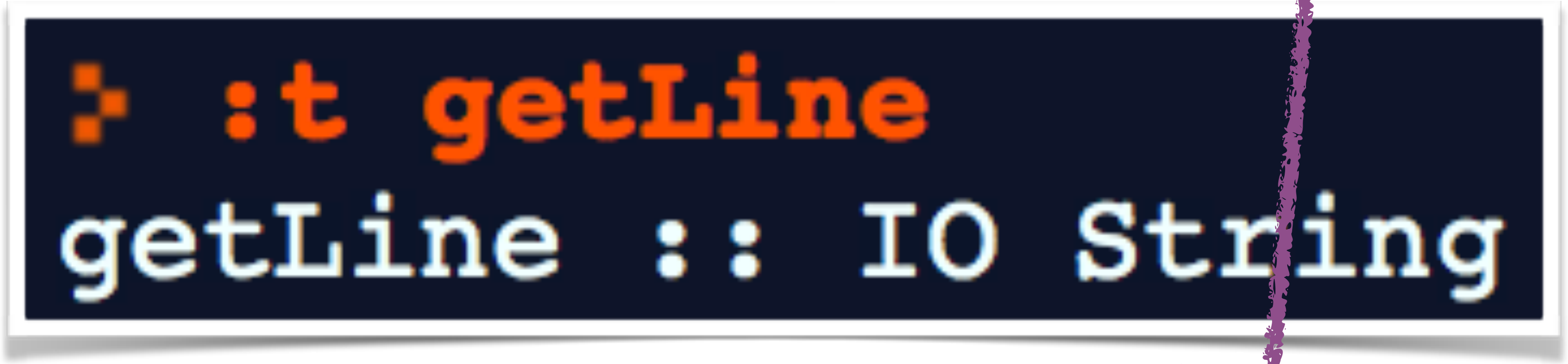
OUTPUT TYPE

Every I/O action returns a value.
In the type system, the return value is `tagged' with `IO` type,
distinguishing actions from other values.

Different IO types

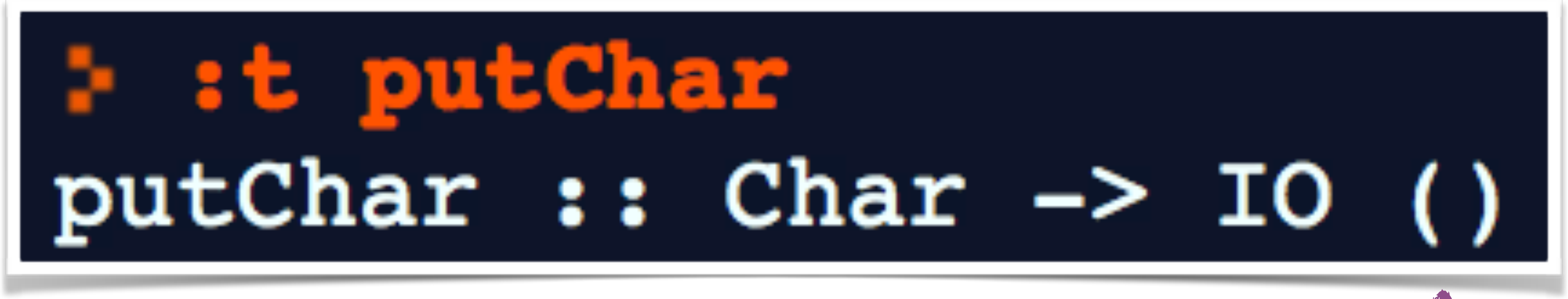


```
> :t getChar
getChar :: IO Char
```

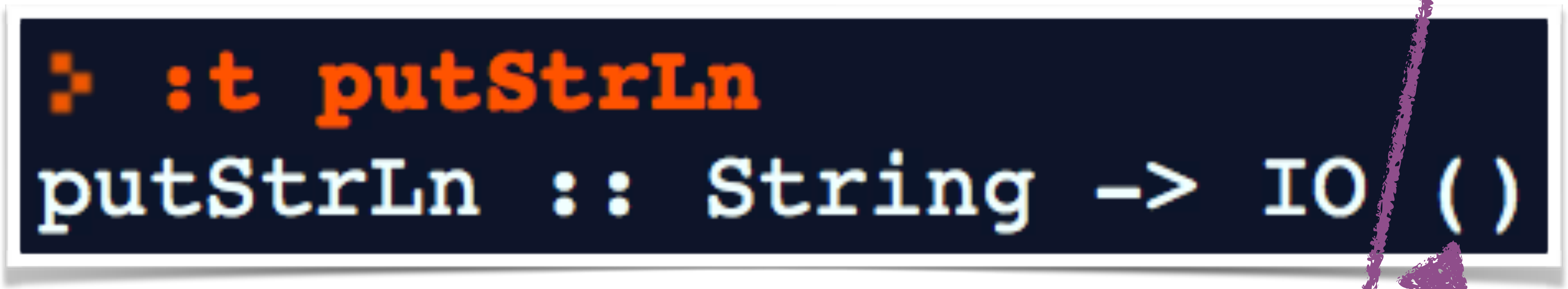


```
> :t getLine
getLine :: IO String
```

When invoked, performs some action which returns a



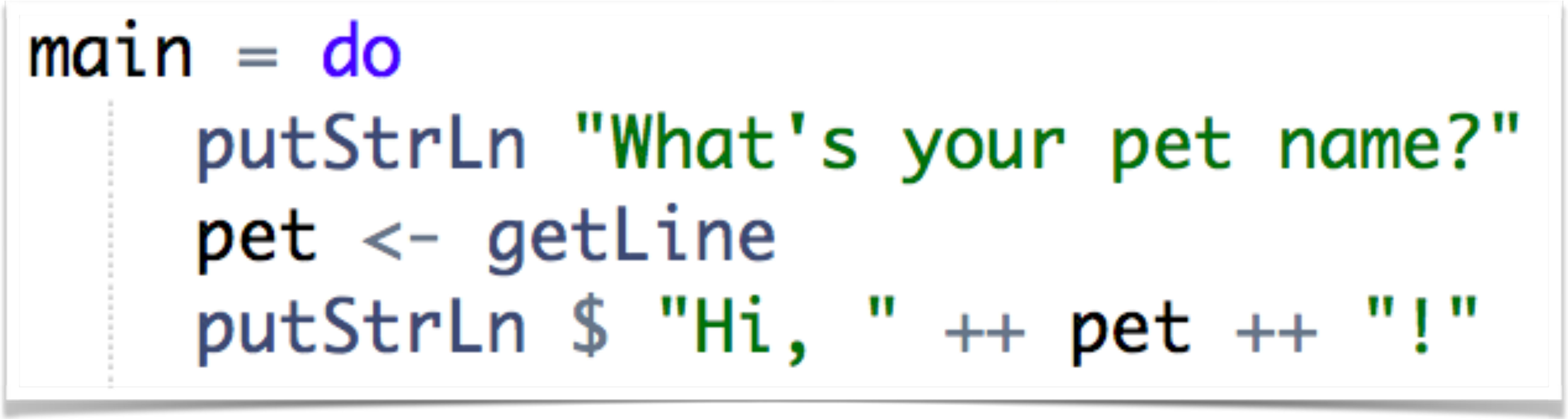
```
> :t putChar
putChar :: Char -> IO ()
```



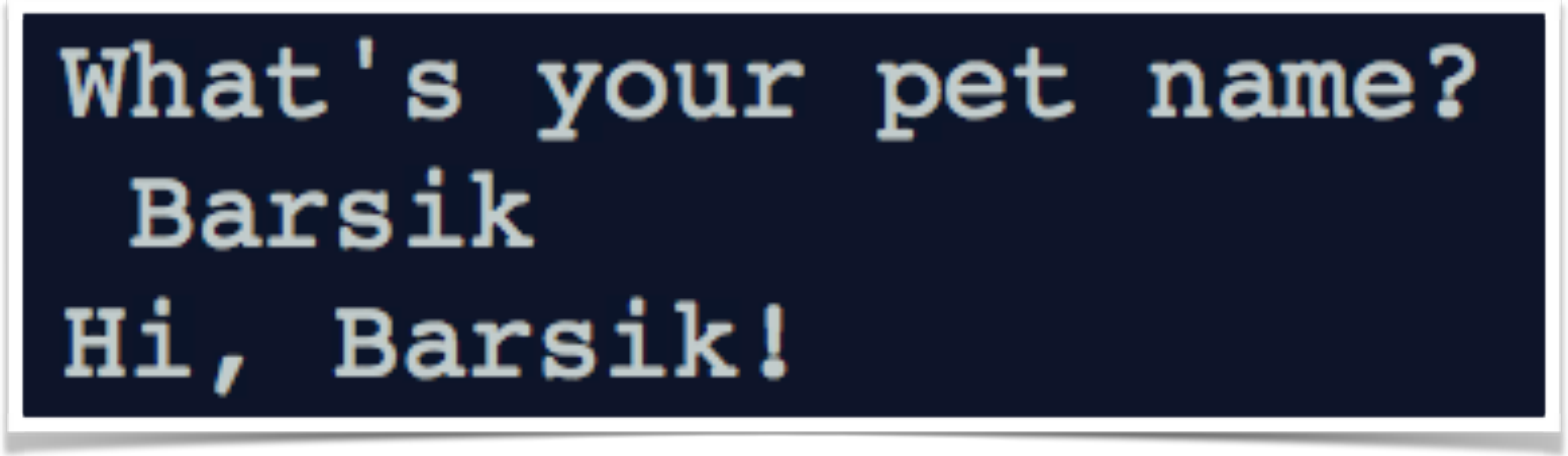
```
> :t putStrLn
putStrLn :: String -> IO ()
```

Actions which return no interesting values use the unit type, ( )

Let’s **do** it



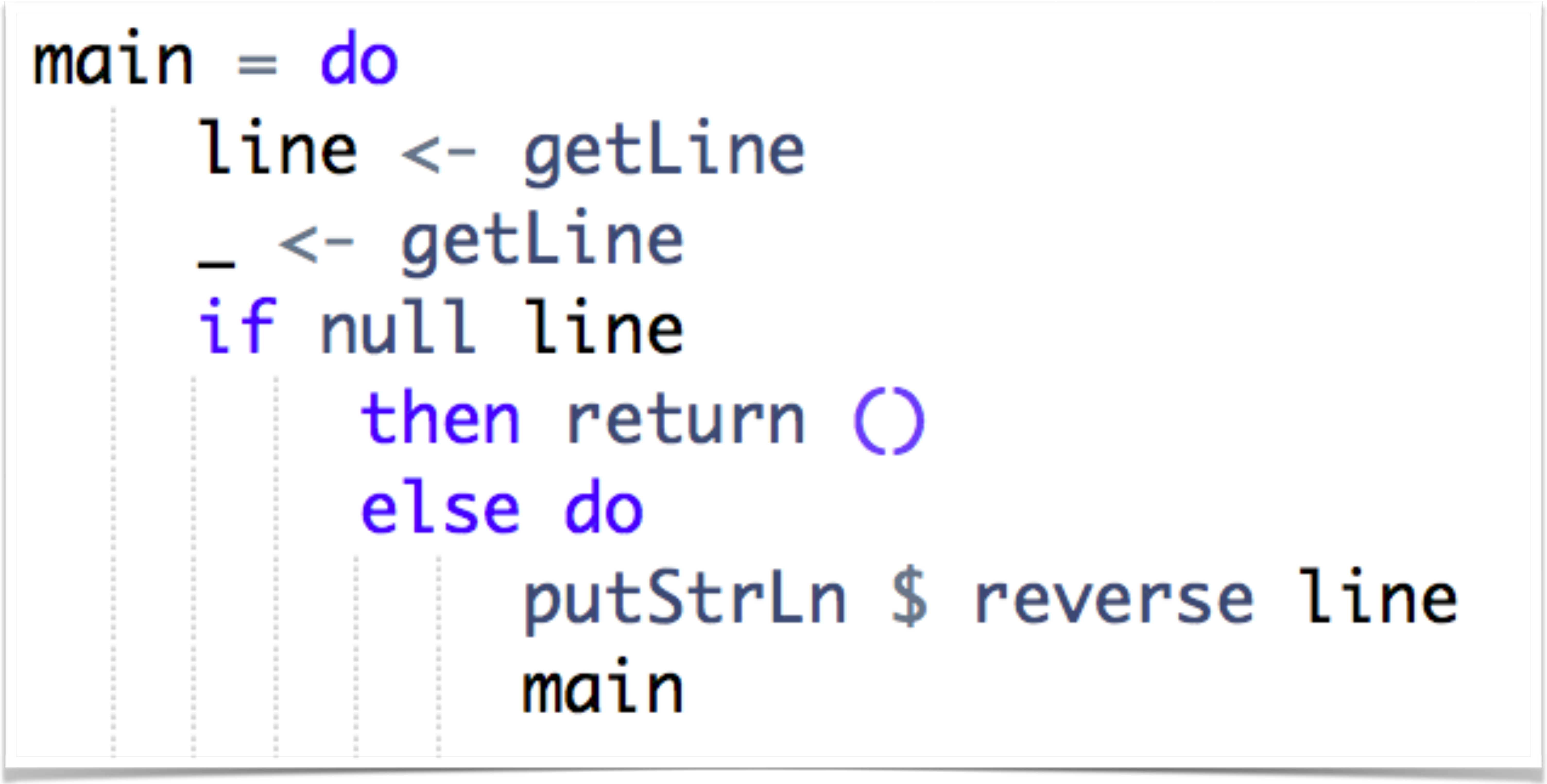
```haskell
main = do
    putStrLn "What's your pet name?"
    pet <- getLine
    putStrLn $ "Hi, " ++ pet ++ "!"
```



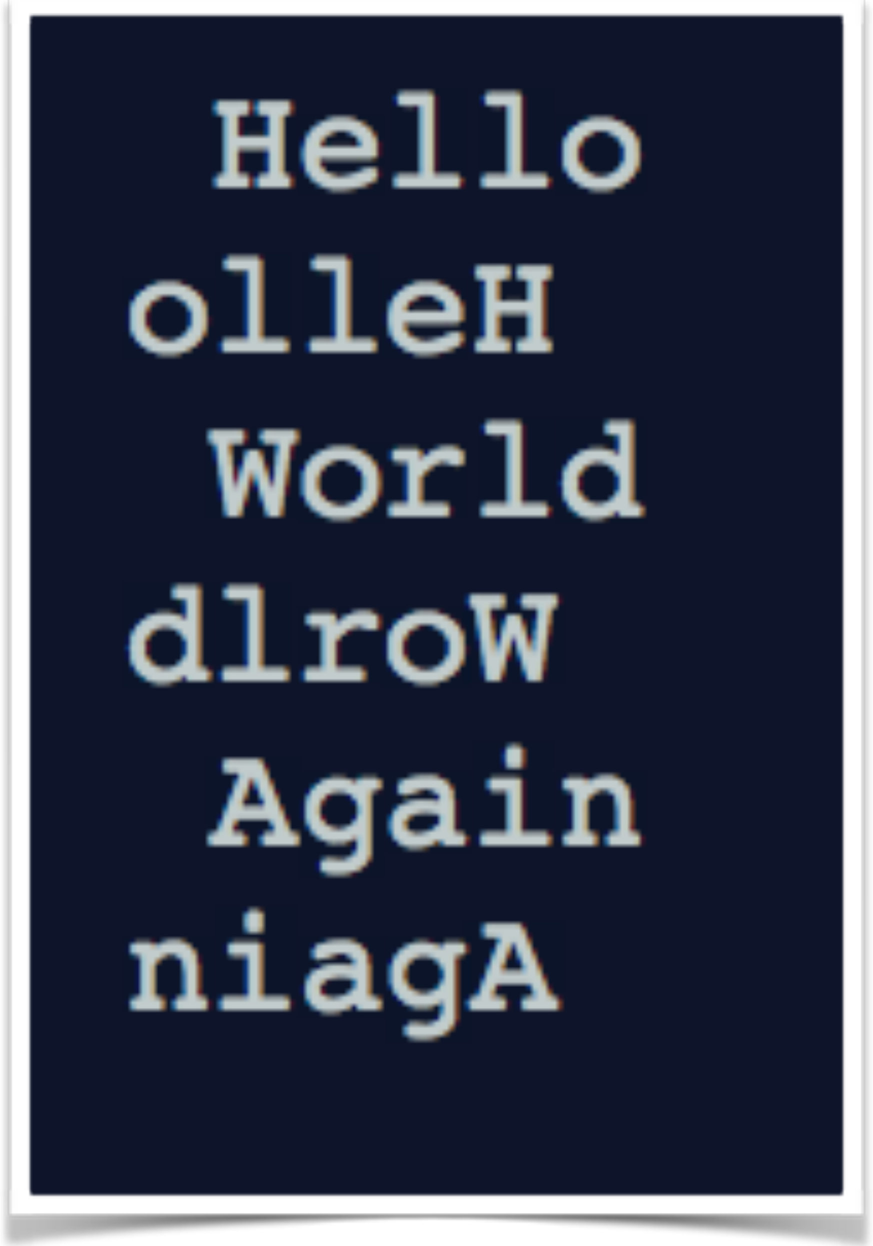
```
What's your pet name?
 Barsik
Hi, Barsik!
```

do notation provides a convenient means of putting actions together

Let’s do it once more



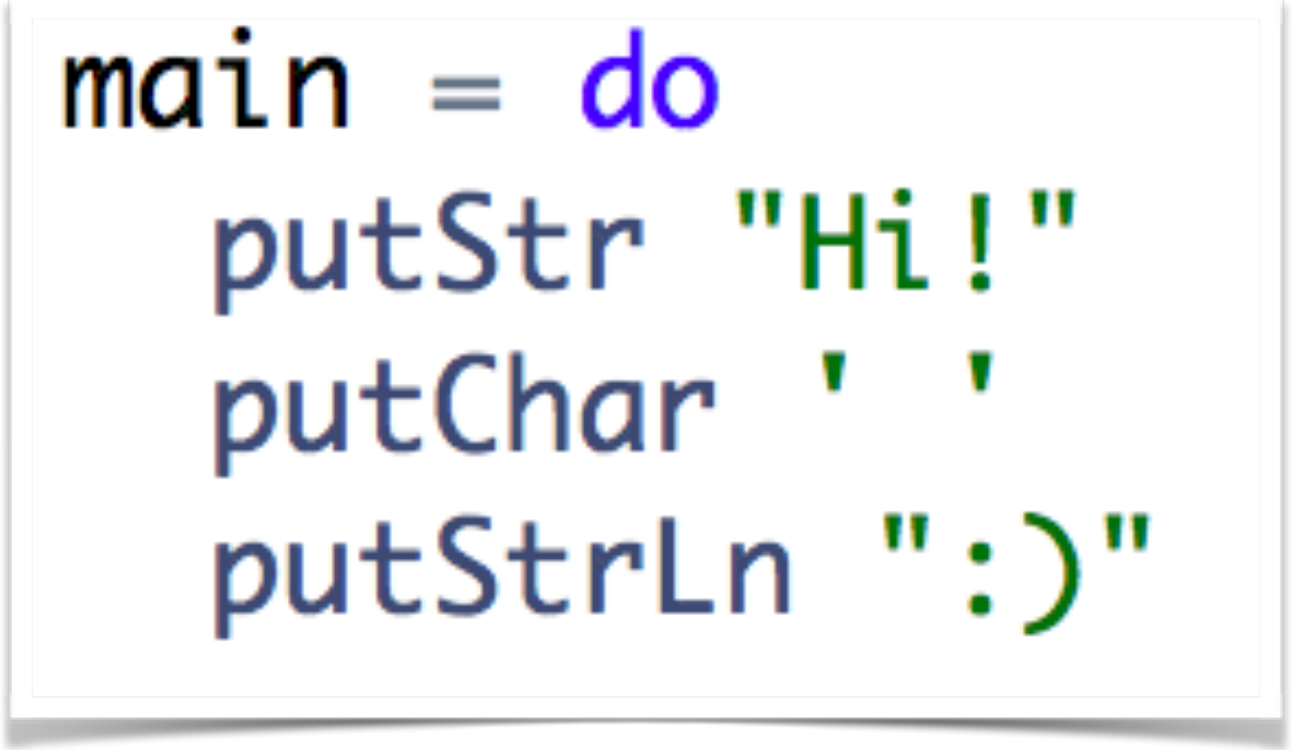
```haskell
main = do
    line <- getLine
    _ <- getLine
    if null line
        then return ()
        else do
            putStrLn $ reverse line
            main
```



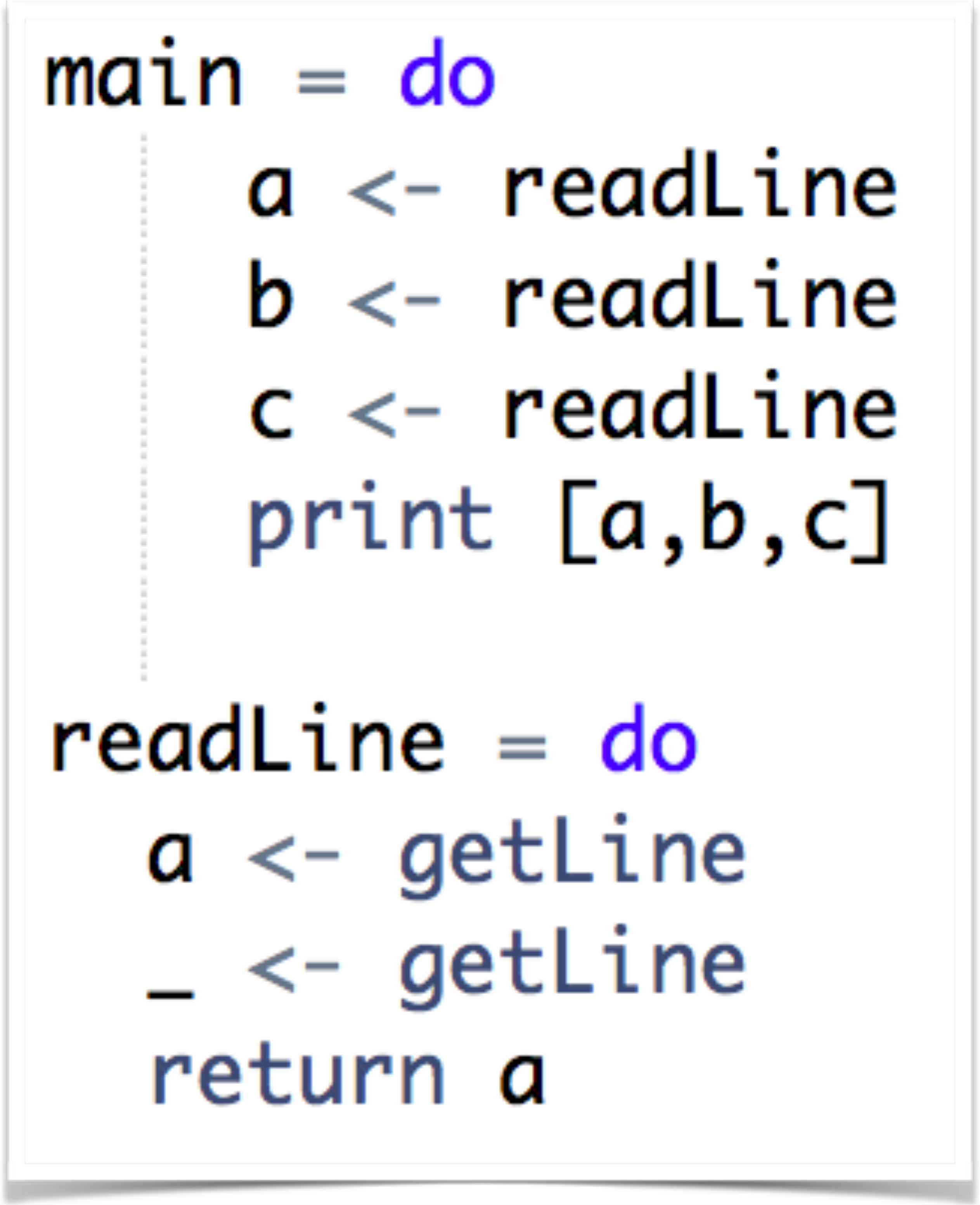
```
 Hello
olleH
 World
dlroW
 Again
niagA
```

The `return` function completes the set of sequencing primitives.

Some more IO funtions

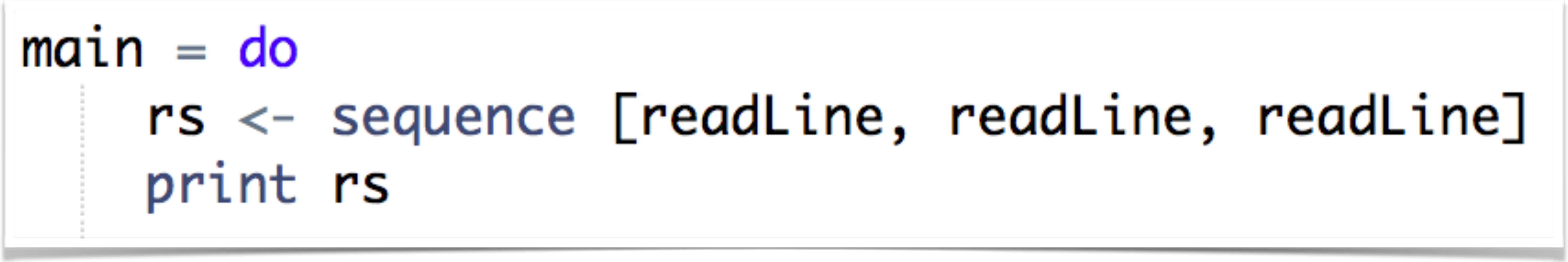


```haskell
main = do
  putStr "Hi!"
  putChar ' '
  putStrLn ":)"
```



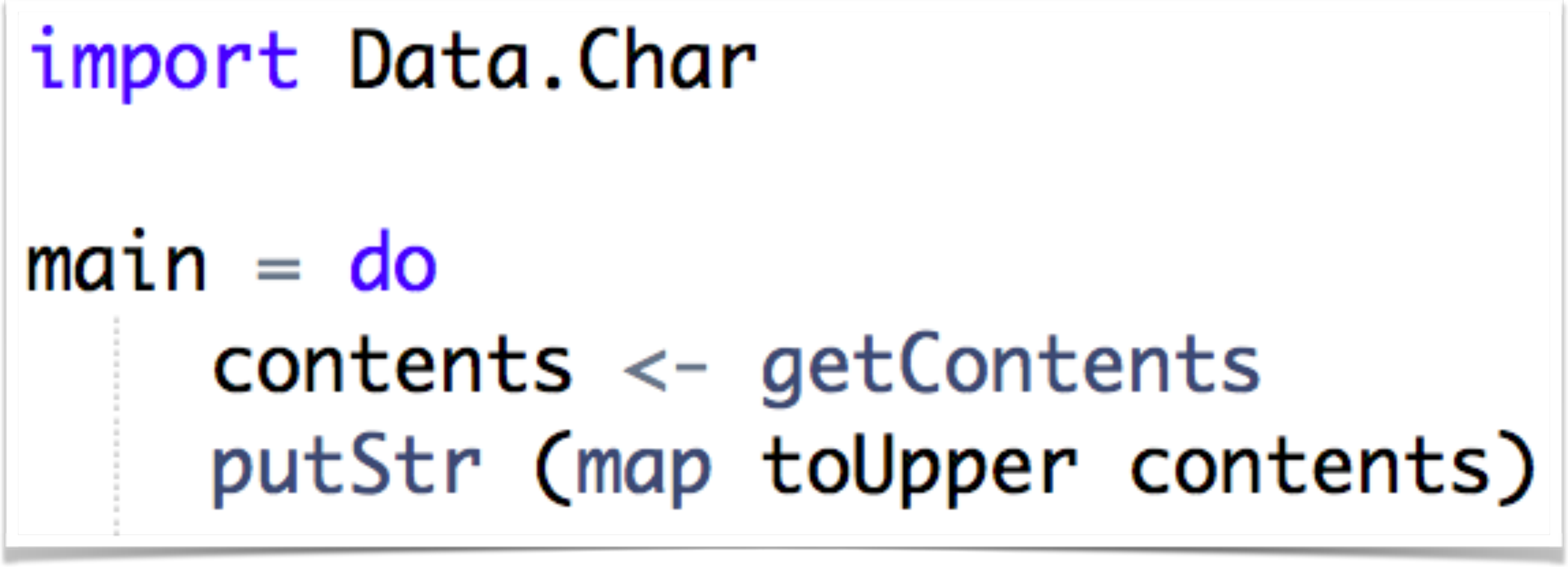
```haskell
main = do
    a <- readLine
    b <- readLine
    c <- readLine
    print [a,b,c]

readLine = do
  a <- getLine
  _ <- getLine
  return a
```

- putStr: : String -> IO ()
- putChar :: Char -> IO ()
- print :: Show a => a -> IO ()
- sequence

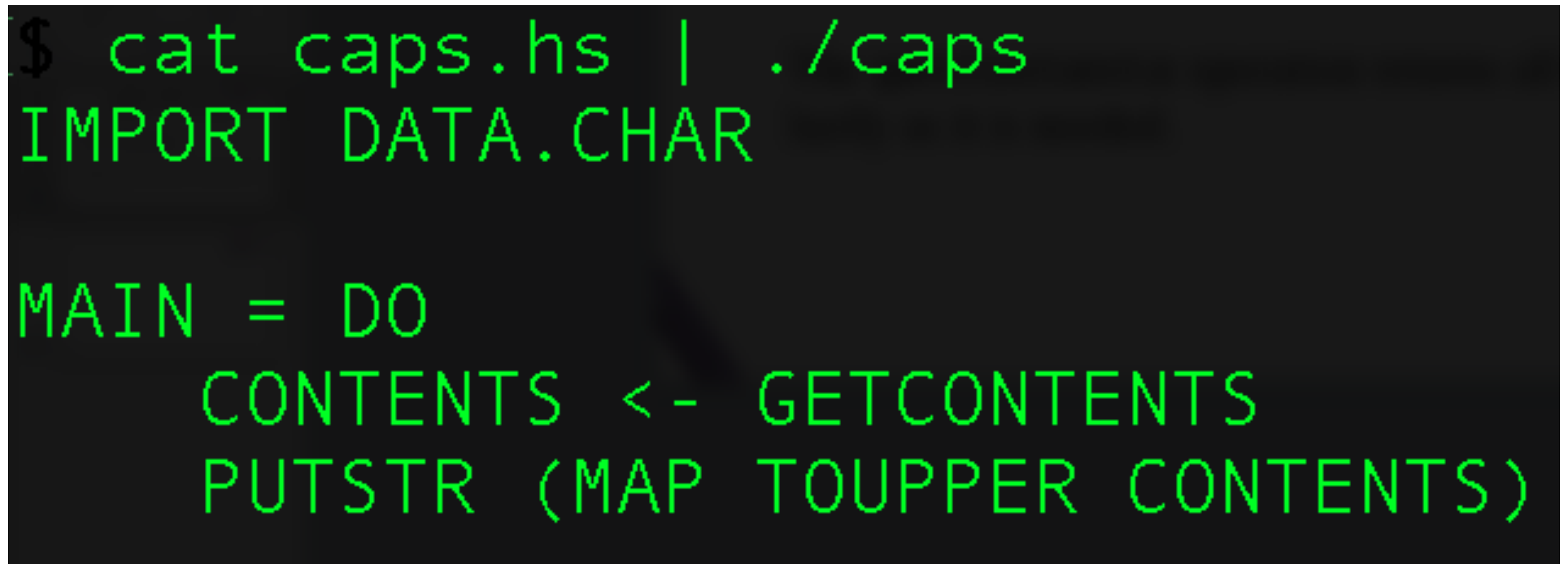


```haskell
main = do
    rs <- sequence [readLine, readLine, readLine]
    print rs
```

Getting all the input



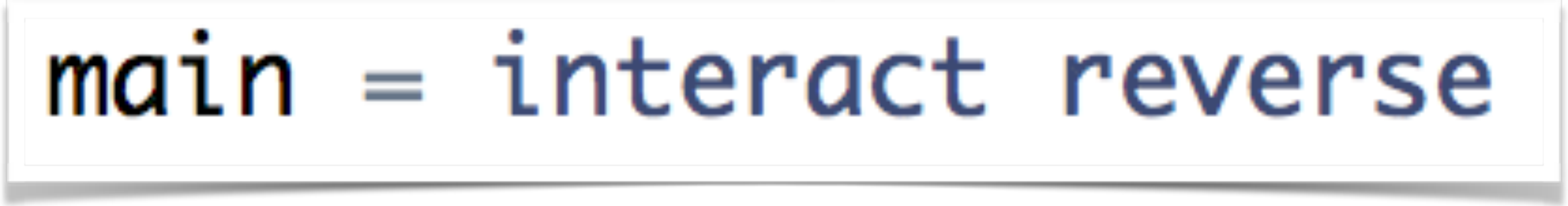
```haskell
import Data.Char

main = do
    contents <- getContents
    putStr (map toUpper contents)
```



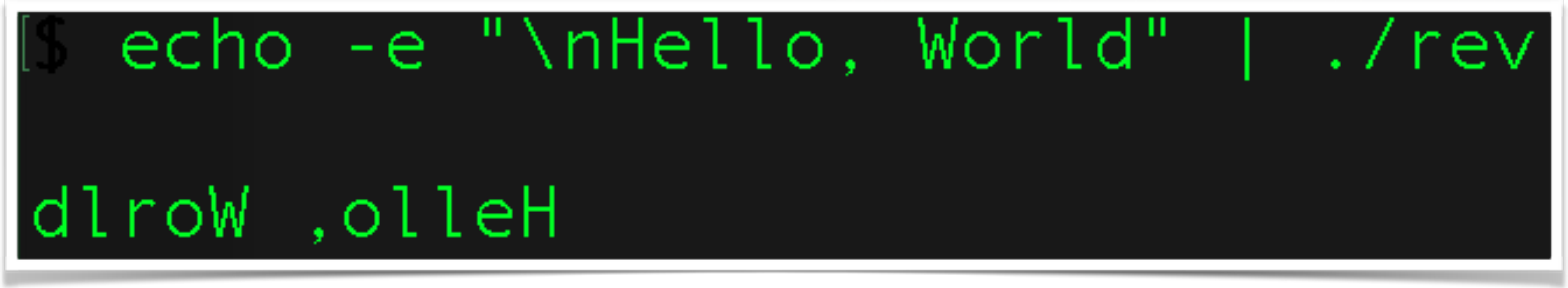
```
$ cat caps.hs | ./caps
IMPORT DATA.CHAR

MAIN = DO
    CONTENTS <- GETCONTENTS
    PUTSTR (MAP TOUPPER CONTENTS)
```

The `getContents` operation returns all user input as a single string, which is read lazily as it is needed.

Some interaction



```haskell
main = interact reverse
```



```
$ echo -e "\nHello, World" | ./rev

dlroW ,olleH
```

The `interact` function takes a function of type `String->String` as its argument. The entire input from the standard input device is passed to this function as its argument, and the resulting string is output on the standard output device.