CIS 1904: Haskell

Input/Output

Input/Output

- Haskell is a *pure* language: no side effects
- Practical programs need to be able to print, read from memory, etc.

Q: How do we write programs with input/output if Haskell cannot have effects?

Input/Output

- Haskell is a *pure* language: no side effects
- Practical programs need to be able to print, read from memory, etc.

Q: How do we write programs with input/output if Haskell cannot have effects?

A: We use Haskell to construct a *description* of effectful code and make the runtime system do the effects for us!

IO Actions: What

- Writing a recipe for a cake is distinct from baking a cake
- We write Haskell code that, when executed, writes a *recipe* for I/O
    - We call a recipe for I/O an *IO action*

IO Actions: Why

- Constructing recipes works much better with laziness
  - It matters when in the program we read from terminal
  - It does not matter when in the program we add the line “read from terminal” to our recipe

IO Actions: Why

- IO actions can be passed around as values
    - The action of reading from terminal is not a value
    - The *instruction* to read from terminal is a value

The `IO` Type Constructor

```
getLine :: IO String
```

This is an *instruction* to read from terminal.
If the runtime system does so, it will get a `String`.

The `IO` Type Constructor

```
getLine :: IO String
```

This is an *instruction* to read from terminal.
If the runtime system does so, it will get a `String`.

Note: `IO` is not a normal container! There is no `String` “inside” this instruction.

- there are no ingredients inside the line “mix flour, sugar, and salt”
- we can still refer to “the dry ingredients” later in the recipe
- those ingredients will not exist unless someone decides to follow the recipe

Computing with `IO`

```
putStr :: String -> IO ()
```

This takes a string and uses it to build an *instruction* to put a string in terminal.
If the runtime system does this, it will get nothing back.

`()` is a type with exactly one inhabitant, also written `()`.
We use this as a dummy value in situations where there is no value.

Note: `putStr` is not an IO action. It is a function from a `String` to an IO action.

More IO functions

`putStrLn :: String -> IO ()`

This is like `putStr`, but adds `'\n'` to the end of the string.

`print :: Show a => a -> IO ()`

This is like `putStrLn`, but the input can have any type in the `Show` typeclass.

`readLn :: Read a => IO a`

This is like `getLine`, but the result when the runtime system executes it can have any type that implements `Read`, a simple typeclass for parseable types.

Combining recipes

Suppose we want a recipe for printing and then reading.

```
printAndRead :: IO String
printAndRead = putStrLn “Hello” ??? getLine
```

Combining recipes

Suppose we want a recipe for printing and then reading.

```
putAndRead :: IO String
putAndRead = putStrLn “Hello” >> getLine
```

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

This combines two recipes into a larger recipe by sequencing them.

Combining recipes

Suppose we want a recipe for printing and then reading.

```
putAndRead :: IO String
putAndRead = putStrLn “Hello” >> getLine
```

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

This combines two recipes into a larger recipe by sequencing them.

Combining recipes

```
putTwo :: String -> String -> IO ()
putTwo xs ys = putStrLn xs >> putStrLn ys
```

Combining recipes

What if we want to read and then print what we read?

```
readAndPut :: IO ()
readAndPut = getLine ??? putStrLn
```

Combining recipes

What if we want to read and then print what we read?

```
readAndPut :: IO ()
readAndPut = getLine >>= putStrLn
```

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

This combines a recipe and a function that uses the result of that recipe to make a new recipe.

The `IO` Type Constructor

```
getLine :: IO String
```

Note: `IO` is not a normal container! There is no `String` “inside” this instruction.

- there are no ingredients inside the line “mix flour, sugar, and salt”
- we can still refer to “the dry ingredients” later in the recipe
- those ingredients will not exist unless someone decides to follow the recipe

`(>>=)` is what lets us talk about “the dry ingredients” later in our recipe.

Examples

```
ex1 :: ???
ex1 = putStr "Hello, " >> putStr "world!"
```

Examples

```
ex1 :: IO ()
ex1 = putStr "Hello, " >> putStr "world!"
```

Examples

```haskell
ex2 :: IO Int
ex2 = putStr "Enter a number: " >> readLn
```

Note: this will crash if the user types in something that is not an `Int`.

Examples

```haskell
ex2 :: IO Int
ex2 = putStr "Enter a number: " >> readLn
```

Note: this will crash if the user types in something that is not an `Int`.

```haskell
ex2 :: IO String
ex2 = putStr "Enter your name: " >> readLn
```

This will crash if the user types in something that is not a `String`.

Examples

```
ex3 :: ???
ex3 = putStr "Enter a number: " >> readLn >>= (\n -> print (n + 1))
```

Examples

```
ex3 :: IO ()
ex3 = putStr "Enter a number: " >> readLn >>= (\n -> print (n + 1))
```

If the runtime system executes this IO action, it will:

1. print “Enter a number: ”
2. read what the user types
3. print the integer one more than what the user typed

Examples

```haskell
ex4 :: IO Int
ex4 = getLine >>= (\input -> return (length input))
```

`return` is a trivial case of IO that does not read or write.

It just tells the runtime system to create the value `(length input)`.

As before, this code creates that instruction; it does not directly create that value.

do notation

```
ex3 :: IO ()
ex3 = putStr "Enter a number: " >> readLn >>= (\n -> print (n + 1))
```

Haskell gives us special notation to make this easier to read.

```
ex3' :: IO ()
ex3' = do
    putStr "Enter a number: "          ← this has an implicit (>>) after it
    n <- readLn                        ← this combines >>= and \n
    print (n + 1)
```

do notation

```
ex1 :: IO ()
ex1 = putStr "Hello, " >> putStr "world!"



ex1’ = do
```

do notation

```
ex1 :: IO ()
ex1 = putStr "Hello, " >> putStr "world!"



ex1’ = do
    putStr "Hello, "
    putStr "world!"
```

do notation

```
ex2 :: IO Int
ex2 = putStr "Enter a number: " >> readLn



ex2’ = do
```

do notation

```
ex2 :: IO Int
ex2 = putStr "Enter a number: " >> readLn



ex2’ = do
    putStr "Enter a number: "
    readLn
```

do notation

```
ex4 :: IO Int
ex4 = getLine >>= (\input -> return (length input))



ex4’ = do
```

do notation

```
ex4 :: IO Int
ex4 = getLine >>= (\input -> return (length input))


ex4’ = do
    input <- getLine
    return (length input)
```

do notation

```
ex5 :: IO ()
ex5 = do
    putStr "Type 5"
    n <- readLn
    if n == 5
        then putStrLn "Thanks"
    else do
        putStrLn "Try again"
        ex5
```

do notation

Remember: do notation is just special syntax!
It is exactly equivalent to the previous versions with >> and >>=.

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

We cannot have arbitrary code as a line in a do block.

```
bad = do
    putStr "Hello"
    4                    <- 4 has type Int, not IO Int
```

do notation

Remember: do notation is just special syntax!
It is exactly equivalent to the previous versions with >> and >>=.

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

We cannot have arbitrary code as a line in a do block.

```
bad = do
    putStr "Hello"
    return 4
```

map

```
printList :: Show a => [a] -> ???
printList xs = map print xs
```

map

```
printList :: Show a => [a] -> [IO ()]
printList xs = map print xs
```

map

```
printList :: Show a => [a] -> IO ()
printList [] = return ()
printList (x : xs) = do
    print x
    printList xs
```

map

```haskell
printList :: Show a => [a] -> [IO ()]
printList xs = map print xs
```

mapM

```
map :: (a -> b) -> [a] -> [b]

mapM :: (a -> IO b) -> [a] -> IO [b]
```

`mapM` “adds effects” to the result types of `map`.

mapM

```
printList :: Show a => [a] -> IO [()]
printList xs = mapM print xs
```

mapM

```
map :: (a -> b) -> [a] -> [b]

mapM :: (a -> IO b) -> [a] -> IO [b]

mapM_ :: (a -> IO b) -> [a] -> IO ()
```

`mapM_` “adds effects” to the result types of `map`, and then ignores the results.

This is useful when we **only** care about the side effects.

mapM

```
printList :: Show a => [a] -> IO ()
printList xs = mapM_ print xs
```

Running code with IO

Typically, code *only* gets run through `main`.

`main` the master recipe that the runtime system follows.

```
main :: IO ()
main = do
    putStrLn “Running main”
```

For a file FileName.hs, you can run it with `runhaskell FileName.hs`

Running code with IO

GHCi gives us another option.

```
foo :: IO ()
foo = do
    putStrLn “Running foo!”
```

For a file FileName.hs, you can load FileName.hs with `stack ghci FileName.hs` and then, in ghci, type `foo` to run it.