Tutorial week 10
IO

By Sander Blijenberg

Actions vs Values - Which is correct?

```haskell
f1 = putStr x
  where  x = getLine

f2 = putStr (getLine ++ getLine)

f3 = getLine

f4 = do x <- getLine
        y <- getLine
        z <- x ++ y
        putStr z

f5 = length . getLine

f6 = getLine >>= length


f7 = do x <- getLine
        putStr "hello"
        return x
        putStr "world"
```

One-time pad

To show the use of IO, let us make a one-time pad program.

First we split the “pure” part of our program from our IO.

What is our pure part?

One-time pad

To show the use of IO, let us make a one-time pad program.

First we split the “pure” part of our program from our IO.

What is our pure part?

-
```haskell
vigenere :: [Int] -> String -> String
```

  (As we restrict to the letters A through Z, it’s really a Vigenère cipher with a random key)

One-time pad

To show the use of IO, let uss make a one-time pad program.

First we split the “pure” part of our program from our IO.

What is our pure part?

- `vigenere :: [Int] -> String -> String`

What could be our IO part?

One-time pad

To show the use of IO, let us make a one-time pad program.

First we split the “pure” part of our program from our IO.

What is our pure part?

- `vigenere :: [Int] -> String -> String`

What could be our IO part?

- Interactivity in the terminal
- Reading / writing to files
- Random number generation

vigenere :: [Int] -> String -> String

```haskell
vigenere :: [Int] -> String -> String
vigenere = zipWith shift
  where
    shift :: Int -> Char -> Char
    shift s c = iterate nextLetter c !! s

    nextLetter 'z' = 'a'
    nextLetter x = succ x
```

Let's try it in the command line

- Get input
- Encrypt it with some key
- Show the result

```haskell
getLine :: IO String
putStrLn :: String -> IO ()
return :: a -> IO a
vigenere :: [Int] -> String -> String
```

```haskell
try = do
```

Using do notation

Let’s try it in the command line

- Get input
- Encrypt it with some key
- Show the result

```haskell
getLine :: IO String
putStrLn :: String -> IO ()
return :: a -> IO a
vigenere :: [Int] -> String -> String
```

```haskell
try = do
  l <- getLine
```

Let's try it in the command line

- Get input
- Encrypt it with some key
- Show the result

```haskell
getLine :: IO String
putStrLn :: String -> IO ()
return :: a -> IO a
vigenere :: [Int] -> String -> String
```

```haskell
try = do
  l <- getLine
  c <- return $ vigenere [0..] l
```

Let’s try it in the command line

- Get input
- Encrypt it with some key
- Show the result

```haskell
getLine :: IO String
putStrLn :: String -> IO ()
return :: a -> IO a
vigenere :: [Int] -> String -> String
```

```haskell
try = do
  l <- getLine
  c <- return $ vigenere [0..] l
  putStrLn c
```

Note that we could’ve also used “putStrLn $ vigenere [0..] l”

Let's try it in the command line

```haskell
try = do
  l <- getLine
  c <- return $ vigenere [0..] l
  putStrLn c
```

```
doesitwork
dpgvmycvzt
ghci>
```

```haskell
getLine :: IO String
putStrLn :: String -> IO ()
return :: a -> IO a
vigenere :: [Int] -> String -> String
```

Let’s try it in the command line

```haskell
try = do
  l <- getLine
  c <- return $ vigenere [0..] l
  putStrLn c
```

```haskell
getLine :: IO String
putStrLn :: String -> IO ()
return :: a -> IO a
vigenere :: [Int] -> String -> String
```

Desugared:

```haskell
try' = getLine >>= \l -> return (vigenere [0..] l) >>= \c ->
putStrLn c
```

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

Let's try it in the command line

```
getLine :: IO String
putStrLn :: String -> IO ()
return :: a -> IO a
vigenere :: [Int] -> String -> String
```

```
try = do
  l <- getLine
  c <- return $ vigenere [0..] l
  putStrLn c
```

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

Desugared:

```
try' = getLine >>= \l -> return (vigenere [0..] l) >>= \c ->
putStrLn c
```

Cleaned up:

```
try'' = getLine >>= return . vigenere [0..]  >>= putStrLn
```

Let’s try it in the command line

```haskell
getLine :: IO String
putStrLn :: String -> IO ()
return :: a -> IO a
vigenere :: [Int] -> String -> String
```

Cleaned up more by using point-free style (hiding the explicit arguments):

```haskell
try'' = getLine >>= return . vigenere [0..]  >>= putStrLn
```

Notice that (>>=) is used to apply IO functions to IO arguments, just in reverse order.

Usual function application order:

```haskell
f (g (x))   f $ g $ x
```

```haskell
($)   :: (a ->    b) ->     a        ->    b
(=<<) :: (a -> IO b) ->  IO a        -> IO b

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

Notice the similarity between ($) and (=<<)

(=<<) is (>>=) with arguments swapped

Let’s try it in the command line

```haskell
getLine :: IO String
putStrLn :: String -> IO ()
return :: a -> IO a
vigenere :: [Int] -> String -> String
```

Applying (=<<):

```haskell
try'' = getLine >>= return . vigenere [0..]  >>= putStrLn
try''' = putStrLn =<< return . vigenere [0..] =<< getLine
```

Let’s try it in the command line

```haskell
getLine :: IO String
putStrLn :: String -> IO ()
return :: a -> IO a
vigenere :: [Int] -> String -> String
```

Applying (=<<):

```haskell
try'' = getLine >>= return . vigenere [0..]  >>= putStrLn
try''' = putStrLn =<< return . vigenere [0..] =<< getLine
```

What about composition?

```haskell
(f . g) x
```

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

($)   :: (a ->    b) ->     a ->    b
(=<<) :: (a -> IO b) ->  IO a -> IO b

(.)   :: (b ->    c) -> (a ->    b) -> a ->    c
(<=<) :: (b -> IO c) -> (a ->  IO b) -> a -> IO c
```

Let’s try it in the command line

```haskell
getLine :: IO String
putStrLn :: String -> IO ()
return :: a -> IO a
vigenere :: [Int] -> String -> String
```

Applying (=<<):

```haskell
try'' = getLine >>= return . vigenere [0..]  >>= putStrLn
try''' = putStrLn =<< return . vigenere [0..] =<< getLine
```

What about composition?

```haskell
(f . g) x
```

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

($)   :: (a ->    b) ->     a ->    b
(=<<) :: (a -> IO b) ->  IO a -> IO b

(.)   :: (b ->    c) -> (a ->    b) -> a ->    c
(<=<) :: (b -> IO c) -> (a ->  IO b) -> a -> IO c
```

```haskell
try'''' = (putStrLn <=< return . vigenere [0..]) =<< getLine
```

Moral of the story

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

($)   :: (a ->    b) ->    a ->    b
(=<<) :: (a -> IO b) -> IO a -> IO b

(.)   :: (b ->    c) -> (a ->    b) -> a ->    c
(<=<) :: (b -> IO c) -> (a -> IO b) -> a -> IO c
```

Applying IO functions to IO arguments require explicit operators.

Be aware of your options to tackle each situation appropriately.

And see the similarities with the operators from before.

Look at Data.Function and Control.Monad if you're interested. (More on this in the coming weeks)

Random numbers

```haskell
getLine      :: IO String
putStrLn     :: String -> IO ()
return       :: a -> IO a
vigenere     :: [Int] -> String -> String
randomRIOs   :: IO [Int]
```

Back to the one-time pad

Random numbers between 0 and 26 from the IO sin bin.

Random numbers

```haskell
getLine       :: IO String
putStrLn      :: String -> IO ()
return        :: a -> IO a
vigenere      :: [Int] -> String -> String
randomRIOs    :: IO [Int]
```

Back to the one-time pad

Random numbers between 0 and 26 from the IO sin bin.

```haskell
rand = do
  s <- randomRIOs
  l <- getLine
  c <- return $ vigenere s l
  putStrLn c
```

```haskell
rand' = randomRIOs >>=
          \s -> getLine >>=
            \l -> return (vigenere s l) >>=
              \c -> putStrLn c
```

Random numbers

```haskell
getLine      :: IO String
putStrLn     :: String -> IO ()
return       :: a -> IO a
vigenere     :: [Int] -> String -> String
randomRIOs   :: IO [Int]
```

Back to the one-time pad

Random numbers between 0 and 26 from the IO sin bin.

```haskell
rand' = randomRIOs >>=
          \s -> getLine >>=
            \l -> return (vigenere s l) >>=
              \c -> putStrLn c
```

Due to the binary nature of “vigenere” we lose the nice program flow that allowed for the slick code from before.

```haskell
rand'' = randomRIOs >>=
           \s -> getLine >>= return . vigenere s >>= putStrLn
```

Is this is the best we can do?

Random numbers

```haskell
getLine      :: IO String
putStrLn     :: String -> IO ()
return       :: a -> IO a
vigenere     :: [Int] -> String -> String
randomRIOs   :: IO [Int]
```

```haskell
rand' = randomRIOs >>=
          \s -> getLine >>=
            \l -> return (vigenere s l) >>=
              \c -> putStrLn c
```

Notice that now both of vigenere’s arguments come from IO.

Notice also, that we used return to get its output into IO as well.

Idea

“Lift” vigenere into IO

Random numbers

```haskell
getLine      :: IO String
putStrLn     :: String -> IO ()
return       :: a -> IO a
vigenere     :: [Int] -> String -> String
randomRIOs   :: IO [Int]
```

```haskell
rand' = randomRIOs >>=
          \s -> getLine >>=
            \l -> return (vigenere s l) >>=
              \c -> putStrLn c
```

Idea

"Lift" vigenere into IO

```haskell
liftM2 :: (a -> b -> c) -> IO a -> IO b -> IO c
```

Then applying liftM2 to vigenere:

```haskell
liftM2 vigenere :: IO [Int] -> IO String -> IO String
```

liftM, liftM2, liftM3, etc. for different amounts of arguments

Random numbers

```haskell
getLine     :: IO String
putStrLn    :: String -> IO ()
return      :: a -> IO a
vigenere    :: [Int] -> String -> String
randomRIOs  :: IO [Int]
```

```haskell
rand' = randomRIOs >>=
          \s -> getLine >>=
            \l -> return (vigenere s l) >>=
              \c -> putStrLn c
```

```haskell
liftM2 vigenere :: IO [Int] -> IO String -> IO String
```

Random numbers

```haskell
getLine      :: IO String
putStrLn     :: String -> IO ()
return       :: a -> IO a
vigenere     :: [Int] -> String -> String
randomRIOs   :: IO [Int]
```

```haskell
rand' = randomRIOs >>=
          \s -> getLine >>=
            \l -> return (vigenere s l) >>=
              \c -> putStrLn c
```

```haskell
liftM2 vigenere :: IO [Int] -> IO String -> IO String
```

```haskell
rand''' = putStrLn =<< liftM2 vigenere randomRIOs getLine
```

Indeed this allows us to pass random numbers and user input just like that!

We then pass this result to be printed like any other IO function.

Random numbers

```haskell
getLine       :: IO String
putStrLn      :: String -> IO ()
return        :: a -> IO a
vigenere      :: [Int] -> String -> String
randomRIOs    :: IO [Int]
```

```haskell
rand''' = putStrLn =<< liftM2 vigenere randomRIOs getLine
```

Does it work?

```
ghci> rand'''
doesitwork
okdsgmczrh
ghci> rand'''
doesitwork
qolawqkata
```

Seems so

Linked lists

```haskell
type ListRef elem = IORef (List elem)
data List elem = Nil | Cons elem (ListRef elem)
```

Let us use the our IO knowledge to create our own lists using IORef.

We can make new IORefs or read existing ones.

```haskell
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
```

So we make a new one as a start to our list.

```haskell
nil :: IO (ListRef e)
nil = newIORef Nil
```

Linked lists

```haskell
type ListRef elem = IORef (List elem)
data List elem = Nil | Cons elem (ListRef elem)
```

```haskell
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
```

```haskell
nil :: IO (ListRef e)
nil = newIORef Nil
```

To add to the list we need a cons:

```haskell
cons :: elem -> ListRef elem -> IO (ListRef elem)
cons x ref =
```

Linked lists

```haskell
type ListRef elem = IORef (List elem)
data List elem = Nil | Cons elem (ListRef elem)
```

```haskell
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
```

To add to the list we need a cons:

```haskell
cons :: elem -> ListRef elem -> IO (ListRef elem)
cons x ref = newIORef $ Cons x ref
cons x = newIORef . Cons x
```

In pointfull and point-free style

But I don’t feel the type looks similar enough to normal cons

```haskell
(:) :: a -> [a] -> [a]
```

(I want the last to arguments to match in type)

Linked lists

```
type ListRef elem = IORef (List elem)
data List elem = Nil | Cons elem (ListRef elem)
```

```
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
```

To add to the list we need a cons:

```
cons :: elem -> ListRef elem -> IO (ListRef elem)
cons x ref = newIORef $ Cons x ref
cons x = newIORef . Cons x
```

But with a different signature:

```
cons' :: e -> IO (ListRef e) -> IO (ListRef e)
```

Looks like the second argument is now in IO. We want to pass this to “Cons x”.

What do we do?

Linked lists

```haskell
type ListRef elem = IORef (List elem)
data List elem = Nil | Cons elem (ListRef elem)
```

```haskell
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
```

To add to the list we need a cons:

```haskell
cons :: elem -> ListRef elem -> IO (ListRef elem)
cons x ref = newIORef $ Cons x ref
cons x = newIORef . Cons x
```

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

($)   :: (a ->    b) ->    a ->    b
(=<<) :: (a -> IO b) -> IO a -> IO b

(.)   :: (b ->    c) -> (a ->    b) -> a ->    c
(<=<) :: (b -> IO c) -> (a -> IO b) -> a -> IO c
```

But with a different signature:

```haskell
cons' :: e -> IO (ListRef e) -> IO (ListRef e)
cons' e r =                 liftM (Cons e) r
cons' e =                 liftM (Cons e)
```

Lift it to IO again!

Now we have to solve the same problem as in the original cons, but now in IO.

Linked lists

```haskell
type ListRef elem = IORef (List elem)
data List elem = Nil | Cons elem (ListRef elem)
```

```haskell
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
```

To add to the list we need a cons:

```haskell
cons :: elem -> ListRef elem -> IO (ListRef elem)
cons x ref = newIORef $ Cons x ref
cons x = newIORef . Cons x
```

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

($)   :: (a ->    b) ->    a ->    b
(=<<) :: (a -> IO b) -> IO a -> IO b

(.)   :: (b ->    c) -> (a ->    b) -> a ->    c
(<=<) :: (b -> IO c) -> (a -> IO b) -> a -> IO c
```

But with a different signature:

```haskell
cons' :: e -> IO (ListRef e) -> IO (ListRef e)
cons' e r = newIORef =<< liftM (Cons e) r
cons' e = newIORef <=< liftM (Cons e)
```

Notice the similarities!

```haskell
type ListRef elem = IORef (List elem)
data List elem = Nil | Cons elem (ListRef elem)
```

Linked lists

```haskell
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
```

nil and cons' can make lists like normal

```haskell
fromList :: [e] -> IO (ListRef e)
fromList [] = nil
fromList (x:xs) = x `cons'` fromList xs
```

Let us make toList, to practise reading an IORef

```haskell
toList :: ListRef e -> IO [e]
toList =
  where
    f :: List e -> IO [e]
    f Nil =
    f (Cons e r) =
```

1. We need to read an IORef (causing IO)
2. We need to construct a list recursively (in IO)

Linked lists

```
type ListRef elem = IORef (List elem)
data List elem = Nil | Cons elem (ListRef elem)
```

```
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
```

nil and cons' can make lists like normal

```
fromList :: [e] -> IO (ListRef e)
fromList [] = nil
fromList (x:xs) = x `cons'` fromList xs
```

Let us make toList, to practise reading an IORef

```
toList :: ListRef e -> IO [e]
toList = f <=< readIORef
  where
    f :: List e -> IO [e]
    f Nil =
    f (Cons e r) =
```

1. We need to read an IORef (causing IO)
2. We need to construct a list recursively (in IO)

Linked lists

```haskell
type ListRef elem = IORef (List elem)
data List elem = Nil | Cons elem (ListRef elem)
```

```haskell
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
```

nil and cons' can make lists like normal

```haskell
fromList :: [e] -> IO (ListRef e)
fromList [] = nil
fromList (x:xs) = x `cons'` fromList xs
```

Let us make toList, to practise reading an IORef

```haskell
toList :: ListRef e -> IO [e]
toList = f <=< readIORef
  where
    f :: List e -> IO [e]
    f Nil = return []
    f (Cons e r) =
```

1. We need to read an IORef (causing IO)
2. We need to construct a list recursively (in IO)

Linked lists

```haskell
type ListRef elem = IORef (List elem)
data List elem = Nil | Cons elem (ListRef elem)
```

```haskell
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
```

nil and cons' can make lists like normal

```haskell
fromList :: [e] -> IO (ListRef e)
fromList [] = nil
fromList (x:xs) = x `cons'` fromList xs
```

Let us make toList, to practise reading an IORef

```haskell
toList :: ListRef e -> IO [e]
toList = f <=< readIORef
  where
    f :: List e -> IO [e]
    f Nil = return []
    f (Cons e r) = (e :) <$> toList r
```

1. We need to read an IORef (causing IO)
2. We need to construct a list recursively (in IO)

map is fmap for lists
<$> is infix fmap

```haskell
map   :: (a -> b) -> [a] -> [b]
fmap  :: (a -> b) -> IO a -> IO b
(<$>) :: (a -> b) -> IO a -> IO b
```

Linked lists

```haskell
type ListRef elem = IORef (List elem)
data List elem = Nil | Cons elem (ListRef elem)
```

```haskell
fromList :: [e] -> IO (ListRef e)
toList :: ListRef e -> IO [e]
mapM : : (a → IO b) → [a] → IO [b]
```

Now we put all our knowledge together.

To make a map function on our linked list.

To start with we can use the mapM given in the lecture slides.

```haskell
mapL :: (a -> IO b) -> ListRef a -> IO (ListRef b)
mapL f =
```

```haskell
type ListRef elem = IORef (List elem)
data List elem = Nil | Cons elem (ListRef elem)
```

Linked lists

Now we put all our knowledge together.

To make a map function on our linked list.

To start with we can use the mapM given in the lecture slides.

```haskell
mapL :: (a -> IO b) -> ListRef a -> IO (ListRef b)
mapL f = mapM f
```

Notice now all our relevant functions are unary functions.

What would be an appropriate approach to building mapL?

```haskell
fromList :: [e] -> IO (ListRef e)
toList :: ListRef e -> IO [e]
mapM f  :: [a] → IO [b]
```

Linked lists

```haskell
type ListRef elem = IORef (List elem)
data List elem = Nil | Cons elem (ListRef elem)
```

```haskell
fromList :: [e] -> IO (ListRef e)
toList :: ListRef e -> IO [e]
mapM f  :: [a] → IO [b]
```

Now we put all our knowledge together.

To make a map function on our linked list.

To start with we can use the mapM given in the lecture slides.

```haskell
mapL :: (a -> IO b) -> ListRef a -> IO (ListRef b)
mapL f = mapM f
```

Notice now all our relevant functions are unary functions.

What would be an appropriate approach to building mapL?

Composition! However, all our functions return in IO!

Linked lists

```haskell
type ListRef elem = IORef (List elem)
data List elem = Nil | Cons elem (ListRef elem)
```

```haskell
fromList :: [e] -> IO (ListRef e)
toList :: ListRef e -> IO [e]
mapM f  :: [a] → IO [b]
```

Now we put all our knowledge together.

To make a map function on our linked list.

To start with we can use the mapM given in the lecture slides.

```haskell
mapL :: (a -> IO b) -> ListRef a -> IO (ListRef b)
mapL f = mapM f
```

Notice now all our relevant functions are unary functions.

What would be an appropriate approach to building mapL?

Composition! However, all our functions return in IO! Luckily we know what to do:

```haskell
(.)   :: (b ->    c) -> (a ->    b) -> a ->    c
(<=<) :: (b -> IO c) -> (a -> IO b) -> a -> IO c
```

Linked lists

```haskell
type ListRef elem = IORef (List elem)
data List elem = Nil | Cons elem (ListRef elem)
```

```haskell
fromList :: [e] -> IO (ListRef e)
toList :: ListRef e -> IO [e]
mapM f  :: [a] → IO [b]
```

Now we put all our knowledge together.

To make a map function on our linked list.

```haskell
mapL :: (a -> IO b) -> ListRef a -> IO (ListRef b)
mapL f = fromList <=< mapM f <=< toList
```

Easy as pie!

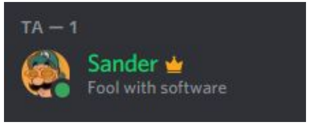
Or alternatively, applicatively:

```haskell
mapL :: (a -> IO b) -> ListRef a -> IO (ListRef b)
mapL f l = fromList =<< mapM f =<< toList l
```

Questions?

File will be on brightspace

Ask me questions on Discord




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

($)   :: (a ->    b) ->     a ->    b
(=<<) :: (a -> IO b) ->  IO a -> IO b

(.)   :: (b ->    c) -> (a ->     b) -> a ->    c
(<=<) :: (b -> IO c) -> (a ->  IO b) -> a -> IO c

liftM2 :: (a -> b -> c) -> IO a -> IO b -> IO c

getLine      :: IO String
putStrLn     :: String -> IO ()
return       :: a -> IO a
vigenere     :: [Int] -> String -> String
randomRIOs   :: IO [Int]

liftM2 vigenere :: IO [Int] -> IO String -> IO String
```