ECS713: Functional Programming
week 10: More Monads

Part 1: Do notation

Aims

- To explain how the do notation can be used with general monads.
- To explain the link between it and bind.

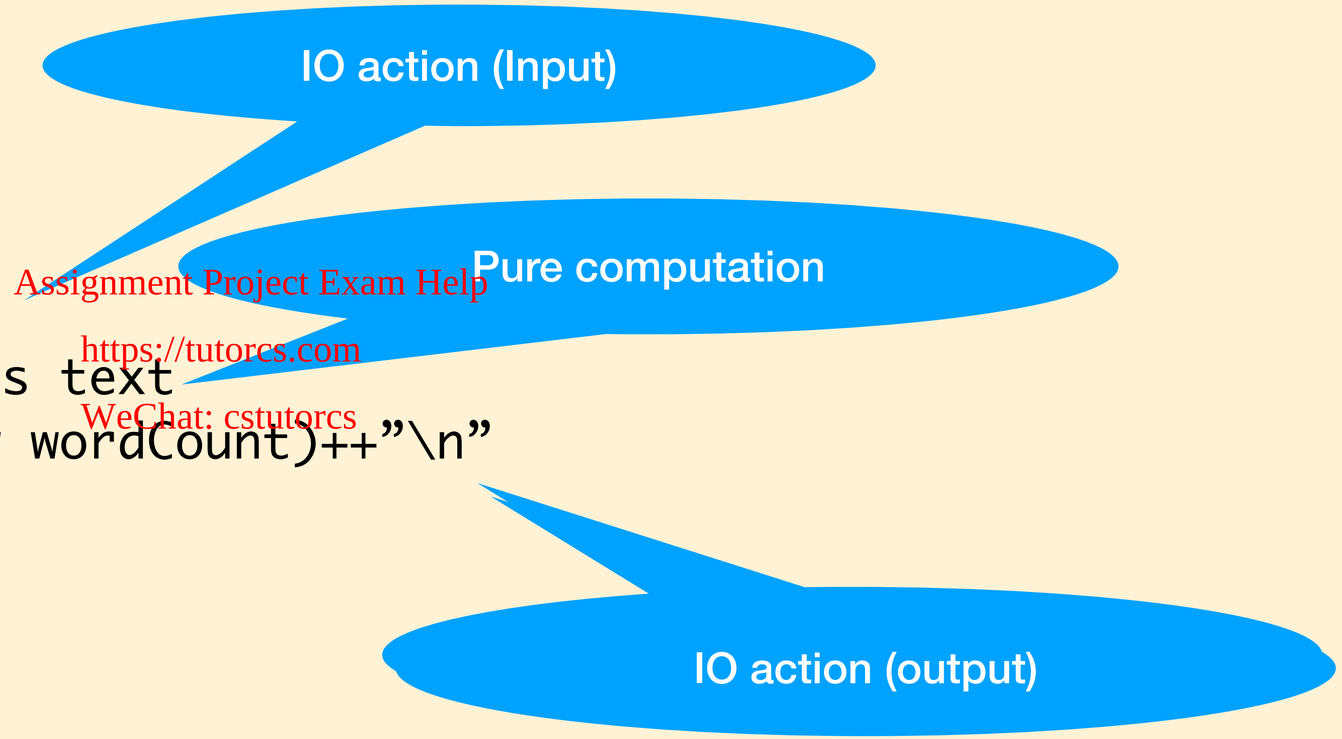
Learning Objectives

- Understand how to use the do notation with a general monad

- Understand the systematic relation between it and bind.

do and IO

```
do
  text <- readFile “myfile.txt”
  let wordCount = length $ words text
  print “Word Count is “++(show wordCount)++”\n”
```



translate to bind

```
do
  text <- readFile “myfile.txt”
  let wordCount = length $ words text
  print “Word Count is “++(show
wordCount)++”\n”
```



```
  print “Word Count is “++(show
wordCount)++”\n”
```

Start at the bottom: last line is IO action

translate to bind

```
do
  text <- readFile “myfile.txt”
  let wordCount = length $ words text
  print “Word Count is “++(show
wordCount)++”\n”
```



```
let wordCount = length $ words text
  in
  print “Word Count is “++(show
wordCount)++”\n”
```

Move up: next line is let

translate to bind

```
do
  text <- readFile “myfile.txt”
  let wordCount = length $ words text
  print “Word Count is “++(show
wordCount)++”\n”
```



```
readFile “myfile.txt”
  >>=
  (\text ->
  let wordCount = length $ words text
    in
  print “Word Count is “++(show
wordCount)++”\n”)
```

Move up: next line is bind

translate to bind

```
do
  text <- readFile “myfile.txt”
  let wordCount = length $ words text
  print “Word Count is “++(show
wordCount)++”\n”
```



```
readFile “myfile.txt”
  >>=
  (\text ->
  let wordCount = length $ words text
    in
  print “Word Count is “++(show
wordCount)++”\n”)
```

Done: next line is do

translate to do

```
do
  x <- e
  f x
```



```
e >>= f
```

Translate right to left

Another example

```
tf <*> ta =
 do
  f <- tf
  a <- ta
  return (f a)
```



```
tf <*> ta =
 tf >>= (\f ->
  ta >>= (\a ->
   return (f a)))
```

Sequential application as an example: RHS is as given in video 3

do notation and arbitrary monads

- The formal definition of the meaning of the do notation uses this transformation (that's why it's a notation)

- It follows we can use it for arbitrary monads.

do notation and arbitrary monads (Either)

```
safeHead [] = Left "exception: head []"
safeHead (x:xs) = Right x

safeLog x | x<=0 = Left "exception: log of negative"
          | otherwise = Right $ log x
```

```
foo list =
  do
    x <- safeHead list
    lg <- safeLog x
    return lg
```



```
foo []
Left "exception: head []"
```

```
foo [2..4]
Right 0.6931471805599453
```

```
foo [-1..3]
Left "exception: log of negative"
```

do notation and arbitrary monads (Lists)

```
newMap f xs = do { x <- xs; return $ f x }
```

```
newMap (+1) [1..4]
[2,3,4,5]
```



```
do { c1 <- "abc"; c2 <- "xy"; return $ c1:c2:[] }
["ax","ay","bx","by","cx","cy"]
```

```
do { c2 <- "xy"; c1 <- "abc"; return $ c1:c2:[] }
["ax","bx","cx","ay","by","cy"]
```

Summary

- The do notation can be expressed using bind instead.
- But the do notation can be used for arbitrary monads, not just IO
- And it can be much clearer than a functional expression with bind.