COMP105 Lecture 25

# IO Examples

# Class Test 2

The second class test will take place in **Week 11**

Same format as Class Test 1

- 20 questions
- 35 minutes
- Multiple choice
- Answers filled in on a computer-readable sheet

# Class Test 2: Topics

### Higher order functions

- `map`
- `filter`
- `fold`
- `scan`
- Other higher order functions

### Types

- Anonymous functions
- The `.` and `$` operators
- Types of higher order functions
- Most general type annotations

# The class test: topics

## Custom types

- Custom data types
- The `Show`, `Read`, `Eq`, and `Ord` type classes
- `Maybe` and `Either`
- Custom lists and trees

## IO and Lazy Evaluation

- IO code
- Boxing and Unboxing the IO type
- Evaluation models – lazy vs strict (not covered yet in class)

# Preparation

A **practice test** is available on Canvas

- We will go through the solutions in a revision lecture
- In Week 10

# Outline

Today

- IO actions that return things
- Extended programming example

Relevant book chapters

- Programming In Haskell Chapter 10

# IO actions that return things

The **last line** of a `do` block determines the return type

```
get_two_ints :: IO Int
get_two_ints =
    do
        x <- getLine
        y <- getLine
        let ret = read x + read y
        return ret
```

# IO actions that return things

We can return **any type** in the IO box

```
get_char_bool :: IO (Char, Bool)
get_char_bool =
    do
        x <- readLn
        y <- readLn
        return (x, y)
```

# IO actions that return things

We can **build complex types** while doing IO

```
get_lines :: IO [String]
get_lines = do
    x <- getLine
    if x == ""
        then return []
        else do
            xs <- get_lines
            return (x : xs)
```

We have to **unbox** and **rebox** the output of the recursion

# Exercise

What does this IO action do?

```
mystery :: [a] -> IO [a]
mystery []     = return []
mystery (x:xs) = do
    bool <- readLn :: IO Bool
    rest <- mystery xs
    if bool
        then return (x : rest)
        else return rest
```

# The task

We will build a program to print out words in ASCII art

```
  #   ###   ###     #   ###
 # #  #  #  #  #   # #  #  #
#   # ###   ###   #   # ###
##### #  #  #  #  ##### #  #
#   # #  #  #  #  #   # #  #
#   # ###   ###   #   # ###
```

# Screens

We will use a **list of strings** to represent a screen

This list

```
["##  ##  ##", "  ##  ##  ", "##  ##  ##"]
```

represents this screen

```
##  ##  ##
  ##  ##
##  ##  ##
```

# Printing out a screen

We can **print** a screen with a recursive IO action

```
print_screen :: [String] -> IO ()
print_screen []     = return ()
print_screen (x:xs) =
    do
        putStrLn x
        print_screen xs
```

# Creating a screen

This code **creates** a blank screen

- x is the width
- y is the height

```haskell
make_screen :: Int -> Int -> [String]
make_screen x y = [replicate x ' ' | _ <- [1..y]]
```

```haskell
blank_screen = make_screen 40 6
```

# Modifying a list

When we **modify a list**, we replace a single element of that list

```
modify_list :: [a] -> Int -> a -> [a]
modify_list list pos new =
    let
        before = take  pos    list
        after  = drop (pos+1) list
    in
        before ++ [new] ++ after
```

```
ghci> modify_list [1,2,3,4,5] 3 100
[1,2,3,100,5]
```

# Modifying a screen

```
set :: [String] -> Int -> Int -> Char -> [String]
set screen x y char =
    let
        line       = screen !! y
        new_line   = modify_list line   x char
        new_screen = modify_list screen y new_line
    in
        new_screen
```

# Modifying a screen with a list

This code takes a **list of modifications**

- `(x, y, char)`

For example:

```
[(1, 1, '#'), (2, 1, '#'), (3, 0, '#')]
```

```
set_list :: [String] -> [(Int, Int, Char)] -> [String]
set_list screen []                   = screen
set_list screen ((x,y,c) : xs) =
    set (set_list screen xs) x y c
```

# Some letters

```
letter_a :: [(Int, Int, Char)]
letter_a = map (\ (x, y) -> (x, y, '#')) [
        (2, 0), (1, 1), (3, 1), (0, 2), (4, 2),
        (0, 3), (1, 3), (2, 3), (3, 3), (4, 3),
        (0, 4), (4, 4), (0, 5), (4, 5)
    ]

letter_b :: [(Int, Int, Char)]
letter_b = map (\ (x, y) -> (x, y, '#')) [
        (0, 0), (1, 0), (2, 0), (0, 1), (3, 1),
        (0, 2), (1, 2), (2, 2), (0, 3), (3, 3),
        (0, 4), (3, 4), (0, 5), (1, 5), (2, 5)
    ]
```

# Shifting letters to the right

To shift a letter to the right by `offset`

- Add `offset` to each `x` coordinate

```
shift_letter :: [(Int, Int, Char)] -> Int ->
                       [(Int, Int, Char)]
shift_letter letter shift =
    map (\ (x, y, c) -> (x + shift, y, c)) letter
```

# The IO loop

```
big_letters :: [String] -> Int -> IO ()
big_letters screen cursor =
    do
        c <- getLine
        let lett = case head c of
                'a'       -> letter_a
                'b'       -> letter_b
                otherwise -> []
            new_screen = set_list screen
                            (shift_letter lett cursor)
        print_screen new_screen
        big_letters new_screen (cursor + 6)
```

# A main function

Finally we can turn big_letters into a runnable program

```
main :: IO ()
main = big_letters blank_screen 0
```

# Summary

- Extended programming example

Next time: Evaluation strategies.