COMP105 Lecture 24

# Writing Programs

# Outline

Today

- Writing and compiling programs
- File IO
- Useful IO functions

Relevant book chapters

- Programming In Haskell Chapter 10
- Learn You a Haskell Chapter 9

# Writing programs

To write a program in Haskell we write a **main** function

```haskell
main :: IO ()
main = putStrLn "Hello world!"
```

`main` always

- Takes no arguments
- Returns an IO type

# Writing programs

To run the program, we first need to **compile** it

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

$ ./hello
Hello world!
```

# Compiling on Windows

1. Save your code as `M:\code.hs`
2. Open the Command Prompt (search for `cmd`)
3. Switch to the M drive with `"M:"`
4. Compile with `"ghc code.hs"`
5. To run, just type `"code"`

You can also compile and run code in subdirectories, but you will need to use `"cd"` to first switch to the right directory

# Command line arguments

Most command line programs take **arguments**

- `getArgs :: IO [String]` returns those arguments
- This function is in `System.Environment`

```
import System.Environment

main :: IO ()
main = do
    args <- getArgs
    let output = "Command line arguments: " ++ show args
    putStrLn output
```

# Looping in IO code

The only way to **loop** in IO code is to use recursion

```
printN :: String -> Int -> IO ()

printN _   0 = return ()
printN str n =
    do
        putStrLn str
        printN str (n-1)
```

- No variables!
- No loops!

# Using command line arguments

```
main :: IO ()
main = do
    args <- getArgs
    let n = read (args !! 0) :: Int
    printN (args !! 1) n
```

```
$ ./repeat_string 3 hello
hello
hello
hello
```

# Exercise

What does this IO action do?

```haskell
mystery :: Int -> IO ()
mystery n = do
    ans <- getLine
    let parsed = read ans
    if parsed == n
        then putStrLn "!"
        else do
            if parsed > n
                then putStrLn ">"
                else putStrLn "<"
            mystery n
```

# File IO

`readFile :: String -> IO String` reads the contents of a file

Suppose that `example.txt` contains:

```
line one
line two
line three
```

```
ghci> readFile "example.txt"
"line one\nline two\nline three\n"
```

# writeFile

`writeFile` writes a string **to a file**

- `writeFile :: String -> String -> IO ()`
- The file will be overwritten!

```
ghci> writeFile "output.txt" "hello\nthere\n"
```

The file `output.txt` contains:

```
hello
there
```

# Finishing the marks.csv example

We wrote the **report** function in Lecture 18

- Now we can turn it into a program

```
main :: IO ()
main = do
    args <- getArgs
    let infile = args !! 0
        outfile = args !! 1
    input <- readFile infile
    writeFile outfile (report input)
```

# Exercise

What does this program do?

```haskell
import System.Environment

mystery2 :: String -> String -> IO ()
mystery2 r w = do
    d <- readFile r
    let l = lines d
        p = filter (\x -> length x <= 5) l
        o = unlines p
    writeFile w o

main = do
    args <- getArgs
    mystery2 (args !! 0) (args !! 1)
```

# Useful IO functions

**print** is the same as `(putStrLn . show)`

```
print_stuff = do
    print "hi"
    print 1
    print [1,2,3]
    print False
```

# Useful IO functions

**putStr** prints a string **without** a new line

```
print_three a b c = do
    putStr a
    putStr b
    putStr c
    putStr "\n"
```

```
ghci> print_three "one" "two" "three"
onetwothree
```

# Useful IO functions

**readLn** gets a line of input and then calls `read`

```
readLn' :: Read a => IO a
readLn' = do
    x <- getLine
    return (read x)
```

```
add_one :: IO ()
add_one = do
    x <- readLn
    putStrLn (show (x + 1))
```

# Useful IO functions

**forever** repeats an IO action forever

- It's in the `Control.Monad` package

```
ghci> import Control.Monad

ghci> forever (putStrLn "hi")
hi
hi
hi
hi
...
```

# Interactive code with forever

```
import Control.Monad
import Data.Char

process :: IO ()
process = do
    putStrLn "Give me some input: "
    l <- getLine
    putStrLn (map toUpper l)

main = forever process
```

# sequence

`sequence` performs a list of IO actions

```
ghci> sequence [getLine, getLine, getLine]
one
two
three
["one","two","three"]
```

The final line is the return value of `sequence`

```
sequence :: [IO a] -> IO [a]
```

# sequence

sequence works well with map

```
ghci> sequence (map print [1,2,3])
1
2
3
[(),(),()]
```

# mapM

Alternatively, you can use mapM

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

```
ghci> mapM print [1,2,3,4]
1
2
3
4
[(),(),(),()]
```

# when

when executes an IO action **if a condition** is true

```
ghci> when True (print "hi")
"hi"

ghci> when False (print "hi")
ghci>
```

```
when :: Bool -> IO () -> IO ()
```

# unless

unless executes an IO action if a condition is **false**

```
ghci> unless True (print "hi")

ghci> unless False (print "hi")
"hi"
```

```
unless :: Bool -> IO () -> IO ()
```

# Exercises

1. Download the sample code for todays lecture. Compile and run one of these examples.

2. Write a program that asks the user to input a line of text, and then prints out the reverse of that line of text.

3. Write a program that takes two command line arguments that are numbers, and then prints out the sum of those two arguments.

# Exercises

4. Write a program that takes one command line argument that is the name of a file, and then prints out the contents of the file.

5. (*) Write a program that takes one command line argument that is the name of a file. It should repeatedly ask the user to give input lines, until the user enters the empty string. It should then write all of those lines to the file.

# Summary

- Writing and compiling programs
- File IO
- Useful IO functions

Next time: Extended programming example