COMP105 Lecture 23

Writing IO Code

Writing our own IO code

We can write **our own** IO actions

```
print_two :: String -> String -> IO ()
print_two s1 s2 = putStrLn (s1 ++ s2)
```

```
ghci> print_two "abc" "def"
abcdef
```

Note that the return type is `IO ()`

Combining multiple IO calls

The **do** syntax allows us to combine multiple IO actions

```
get_and_print :: IO ()
get_and_print =
    do
        x <- getLine
        y <- getLine
        putStrLn (x ++ " " ++ y)
```

The do syntax

A do block has the following syntax

```
do
    v1 <- [IO action]
    v2 <- [IO action]
    ...
    vk <- [IO action]
    [IO action]
```

- v1 through vk unbox the results of IO actions
- The final IO action is the return value

The do syntax

The `v <-` portion can be **skipped** if you don't want to unbox

```
echo_two :: IO ()
echo_two =
    do
        x <- getLine
        putStrLn x
        y <- getLine
        putStrLn y
```

The do syntax

`let` expressions can be used inside do blocks

```
add_one :: IO ()
add_one =
    do
        n <- getLine
        let num = (read n) :: Int
            out = show (num + 1)
        putStrLn out
```

This is useful to do **pure** computation between IO actions

The do syntax

`if` expressions can be used inside do blocks

```
guess :: IO ()
guess = do
    x <- getLine
    if x == "42"
        then putStrLn "correct!"
        else putStrLn "try again"
```

Both branches of the if must have the same type

`do` blocks

`do` blocks let you sequence multiple actions

- Works with IO actions
- Will **not work** in pure functional code

Functional programs consist of

- a small amount of IO code
- a large amount of pure functional code

Don't try to write your entire program in IO code!

Putting values in the IO box

Sometimes we need to put a pure value **into** `IO`

- We can use the `return` function to do this

```
ghci> :t "hello"
"hello" :: [Char]
```

```
ghci> :t return "hello"
IO [Char]
```

`return` example

```
print_if_short :: String -> IO ()
print_if_short str =
    if length str <= 2
        then putStrLn str
        else return ()
```

Both sides of the `if` must have type `IO ()`

- So we use `return ()` in the else part

return

Note that `return` does **not** stop execution

- It just converts pure values to IO values
- It is nothing like `return` from imperative languages

```
print123 =
    do
        x <- return 1
        y <- return 2
        z <- return 3
        putStrLn (show x ++ show y ++ show z)
```

Monads

The type of `return` mentions monads

```
ghci> :t return
return :: Monad m => a -> m a
```

This is because IO is a **monad**

- Whenever you see `Monad m =>` substitute IO for m
- So `return :: a -> IO a`

You **don't need to know** anything about monads for COMP105

Exercises

1. Write an IO action `doubleEcho :: IO ()` that allows the user to enter a string and then prints that string out twice on two separate lines

2. Write an IO action `firstWord :: IO ()` that allows the user to enter a string, and then prints out the first word of that string

3. Write an IO action `printEven :: Int -> IO ()` that takes an integer argument x. It should print x out if it is even, and it should do nothing if x is odd