COMP105: Programming Paradigms
Lab Sheet 9

Filename: Lab9.hs

This lab covers material on IO.

1. **Compiling and running programs.** Save the following code into a file named `M:\prog.hs`.

   ```
   main = putStrLn "Hello world!"
   ```

   Open the Command Prompt (you can search for `cmd` to find it). By default, the current working directory should be `M:\`, but if it is not, you can switch by typing `M:`. Type:

   ```
   ghc prog
   ```

   This will compile `prog.hs` creating the executable file `prog.exe`. You can then type:

   ```
   prog
   ```

   This will run the program, and print out `Hello world!`

2. **Changing directory**. Create a directory called `M:\comp105` (or similar) and move `prog.hs` to `M:\comp105\prog.hs`. At the command prompt, make sure that you are in `M:\` and then type:

   ```
   cd comp105
   ```

   This changes the current working directory to `M:\comp105\`. You can now compile and run `prog.hs` just as before.

3. **getLine and putStrLn**. The following questions should be solved using ghci.

   (a) Open ghci and run the `getLine` IO action. Type any string and press enter. Note that `getLine` returns the string that you entered.

   (b) Run `putStrLn "hello"`, and observe that it prints out `hello` with no quotes.

   (c) Copy the following code into a Haskell file

```
echo :: IO ()
echo = do
    str <- getLine
    putStrLn str
```

Notice that we used a `do` block here, because we needed to unbox the result of `getLine` (which returns type `IO String`). **Make sure that the do block is indented** or you will get confusing errors later on. Run the `echo` action to check that it works.

(d) Write an IO action `double_echo :: IO ()` that reads a string from the user, and then prints it out twice.

(e) Write an IO action `put_two_strs :: String -> String -> IO ()` that takes two strings, and prints them both on different lines.

4. **Let in do blocks** Recall that we can use `let` in a `do` block like so

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

The code above asks the user for a number, and then adds one to that number. Make sure that you understand this code before continuing.

(a) Write an IO action `times_two :: IO ()` that asks the user for a number, and the prints out two-times that number.

(b) Write an IO action `add :: IO ()` that asks the user for two numbers (on two different lines), and then prints out the sum of those two numbers.

(c) Write an IO action `io_reverse :: IO ()` that asks the user for a string and prints out the reverse of that string. The prelude function `reverse` will reverse a string.

(d) Write an IO action `guess_42 :: IO ()` that asks the user for a number. If the number is `42` then `correct` should be printed to the screen. Otherwise `wrong` should be printed.

5. **Return.** Recall that `return` lets us "box" a value in the IO type. Look at the following code

```
get_int :: IO Int
get_int = do
    str <- getLine
    let n = read str :: Int
    return n
```

The code asks the user for a number, converts it to an integer, and then returns that integer. Note that we needed to use `return`, in order to return `IO Int`, rather than `Int`. Make sure that you understand this code before continuing.

(a) Write a function `get_bool :: IO Bool` that asks the user to input either `True` or `False` and returns the boolean value that they input. Remember that `read` can be used to parse `Bool`s.

(b) Write a function `get_two_and_add :: IO Int` that asks the user for two integers, and returns the sum of those integers.

(c) Write a function `gt10 :: IO Bool` that asks the user for an integer, and returns `True` if that number is strictly greater than 10, and False otherwise.

(d) Write a function `get_two_strings :: IO (String, String)` that asks the user for two strings (on two different lines), and returns both strings that the user entered.

6. **Looping in IO code.** Recall that we can use recursion in IO code.

```
echo_forever :: IO ()
echo_forever = do
    str <- getLine
    putStrLn str
    echo_forever
```

The code above will continually ask the user for input, and then repeat that input, until the user presses control+c.

(a) Write a function `add_one_forever :: IO ()` that continually asks the user for a number, and then prints out that number plus 1.

(b) Write a function `echo_until_quit :: IO ()` that continually asks the user for input, and repeats that input, until the user enters `quit`.

(c) Write a function `print_numbers_between :: Int -> Int -> IO ()` that takes two numbers $a < b$, and prints out all the numbers between `a` and `b` (inclusive), each on a different line.

7. **Reading files.** Put the following into a file called `M:\file.txt`

```
Line one
Line two
Line three
```

In ghci type

```
ghci> readFile "M:\\file.txt"
```

You can also type `:cd M:\` in ghci to switch to the `M:` drive in ghci and then use `readFile "file.txt"`. This is handy if you want to work in a subdirectory.

(a) Write a function `print_file :: String -> IO ()` that takes a file name, and prints the contents of that file to the screen.

(b) Recall that you can use `lines` to turn a string with `\n` characters, into a list of strings. Write a function `first_line :: String -> IO ()` that takes a file name, and prints the first line of that file to the screen.

(c) Write a function `get_lines :: String -> IO [String]` that takes a file name and returns the list of strings in that file.

8. **Writing files.** Make sure that there is no file named `M:\file2.txt`. Enter the following into ghci:

```
ghci> writeFile "M:\\file2.txt" "hello\nthere\n"
```

Open `M:\file2.txt` and check its contents. Remember that `writeFile` will **overwrite** any existing files, so use it with caution!

(a) Write a function `write_to :: String -> Int -> IO ()` that takes a file name and an integer, and writes the integer to the given file name. You could test this function using `write_to "M:\\file2.txt" 1`, for example.

(b) Write a function `copy_file :: String -> String -> IO ()` that takes two file paths `a` and `b` and copies the contents of the file `a` into `b`.

(c) Write a function `write_lines :: String -> [String] -> IO ()` that takes a file path `a` and a list of strings, and writes those strings to `a` with one string on each line. Remember that `unlines` does the opposite of `lines`

9. **Getting command line arguments.** Copy the following code to `M:\prog2.hs`.

```
import System.Environment

main = do
    args <- getArgs
    putStrLn (show args)
```

This uses `getArgs` to read the command line arguments to the program, and then prints them out. Compile the program and then run

```
prog2 hello there
```

Make sure that you understand the code above before continuing. Comment out the existing version of `main` before continuing.

(a) Write a program that takes one command line argument, and then prints it to the screen. Rename `main` to `one_arg` after you have tested the program.

(b) Write a program that takes two command line arguments that are both numbers, and then prints out the sum of those numbers. Rename `main` to `sum_two` after you have tested the program.

(c) Write a program that takes one command line argument that is the name of a file, and then prints the contents of that file to the screen. Rename `main` to `read_file_and_print` after you have tested the program.

(d) Write a program that takes two command line arguments that are both file names, and then copies the contents of the first file into the second file. Rename `main` to `copy` after you have tested the program.

When you submit to Codegrade, make sure that your file still imports `System.Environment`, or Codegrade will not be able to test your programs.

Lab complete.