Warm Up (Drilling on IO)

Some Tips About IO

A few tips about IO:

- A few useful functions from the Prelude are:

  ```
  putStr   :: String -> IO ()
  putStrLn :: String -> IO ()
  print    :: Show a => a -> IO ()
  getLine  :: IO String
  show     :: Show a => a -> String
  read     :: Read a => String -> a
  ```

  Make sure you know what these do before you start.

- Make sure that in a `do`-block, the different actions are at the same indentation level. Haskell insists that the layout of your code is correct.

- Remember that a `do`-block is a normal expression. The type of a `do`-block is equal to the type of the last expression in that `do`-block.

- On each line of a `do`-block, there has to be a monadic value. If you want to give a name to the result of a normal function call, this is possible with `var <- return (normal call)` or `let var = normal call`.

- The following three expressions have the exact same meaning:
  ```
  do
  l <- getLine
  return l
  ```

  and

  ```
  do getLine
  ```

  and

  ```
  getLine
  ```

  A `do-block` can be omitted if the implementation only has one line.

- The type `()` (pronounce: "unit"), is a built-in trivial datatype that is defined as: `data () = ()`. This type is often used in monadic functions to indicate that there is no useful result.

- The result type of functions such as `print` is `IO ()` to indicate that no relevant value is returned. For these functions we care about the IO itself.
- Take a look at the entries for functions `show` and `read` in the manual. How do you explain the difference in the output of `print "Hello"` and `putStrLn "Hello"`?

When in doubt about library functions, you can always refer to the documentation. In the PC classes this documentation can be found at `file:///usr/share/doc/ghc-doc/html/libraries/base-4.6.0.1/index.html`. The default imported functions can be found under `Prelude`, a lot of useful functions are in `Data.List`, and when you use monads, a lot of things can be found under `Control.Monad`. This is the only documentation that is allowed on the exam.

Exercises

These exercises should not pose much of a challenge and serve to bring you up to speed with IO.

1. Write a program `prog1 :: IO()` that reads two natural numbers `m` and `n` from the standard input and writes `m` copies of `n` to the standard output. For example:

```
Main> prog1
3            -- input: m
7            -- input: n
7            -- output: m times n (each in separate line)
7
7
```

2. Now write a program `prog1b :: IO ()` that does the same thing as `prog1`, but instead of using **do**-notation, you should use `>>= :: (a -> IO b) -> IO b`.

   Hint remember that the following programs are equivalent (where `io1 :: IO a` and `io2 ::  a -> IO b`):

```
do x <- io1
   io2 y

-- and

io1 >>= \x -> io2 y

-- and

io1 >>= \x ->
io2 y
```

3. Write a program `prog2 :: IO()` that continuously reads a line and outputs that line reversed until a blank line is read. For example:

```
Main> prog2
Haskell         -- input
"lleksaH"       -- output
is fun!         -- input
"!nuf si"       -- output
                -- input (empty line)
Main>
```

4. Write a function `index :: [IO a] -> IO Int -> IO a` which indexes a list with an int which is obtained through an IO action:

```
Main> index [print "Hello World",print "Hello Galaxy"] readLn
1
"Hello Galaxy"
```