Tutorial 7A: Input / Output

In this tutorial we will practice input and output. Haskell has a specific data type `IO a`, which can be understood as follows: it performs some IO action, and contains a value of type `a`. We will use some of the following built-in functions:

`putStr`	`:: String -> IO ()`	prints a string to the console
`putStrLn`	`:: String -> IO ()`	prints a string and starts a new line
`print`	`:: Show a => a -> IO ()`	is `putStrLn . show`
`getLine`	`:: IO String`	asks for a line on the console
`return`	`:: a -> IO a`	wraps a value in the "do nothing" IO action
`randomIO`	`:: IO Int`	uses IO to generate a random integer

The last function, `randomIO`, lives in the module `System.Random`, which your tutorial file imports. It has a more general type in general, but we will use it as `IO Int`.

To create a sequence of IO interactions, Haskell uses a `do`**-block**:

```
do
    line_1
    line_2
     ...
    line_m
```

The lines must be aligned. If you start them on the next line, the indentation remains independent of the length of the line ending in `do`. A line can be of one of the following forms:

`x <- exp` Evaluates the IO expression `exp :: IO a`, carries out its `IO` action, and extracts the value as `x :: a`. Pattern-matching is allowed instead of `x`, for example as `(x,y) <- exp` or `x:xs <- exp`.

`exp` Evaluates the IO expression `exp :: IO a`, carries out its `IO` action, and discards the value. It is typically used for `IO ()`, as returned e.g. by `putStrLn`, since `()` is not a useful value. It is equivalent to `_ <- exp`.

`let x = exp` Evaluates the expression `exp :: a` and binds the result to `x :: a`. Pattern matching for `x` is allowed. The expression is equivalent to `x <- return exp`, which would first wrap `exp` in the empty IO action and then extract its value as `x`.

Each line in the `do`-block may use the variables introduced by previous lines. The last line must be of the second kind `exp :: IO a`, and its type becomes that of the whole `do`-block.

Exercise 1: Complete the function `repeatMe` which requests a line from the prompt, and repeats it back to you preceded by the message "You just told me:". Use a do-block with three lines: one using `x <- getLine` to get the user input, one with `putStr` to print the "You just told me:" message, and one with `putStrLn` to repeat the user's input.

In the below dialogue, the green text is given as input at the prompt.

```
*Main> repeatMe
I do not want people to be very agreeable, as it saves me the
trouble of liking them a great deal.
You just told me: I do not want people to be very agreeable,
as it saves me the trouble of liking them a great deal.
```

Exercise 2: We will build a small interactive program `lizzy` impersonating a psychologist, named Dr. Lizzy. Her lines of text are given to you as `welcome`, `exit`, and `response`. The first two are just strings. The second one is a function that takes a string `str`, the user's last input, and an integer `r` which we will generate at random, and selects one of 3×5 possible responses based on `r`.

a) Complete the function `lizzy`, which (for now) does nothing but print the `welcome` message to the console, using `putStrLn` in a do-block.

```
*Main> lizzy

Dr. Lizzy -- Good morning, how are you today. Please tell me
what's on your mind.
```

b) Complete the function `lizzyLoop`. It won't be a loop immediately, but we'll get to that. Give it a do-block of three lines. The first uses `getLine` to get a user response `str`. The second should be the line `r <- randomIO :: IO Int`, including the type signature, to draw a random integer `r`. (`randomIO` can make random values of many types; sometimes, as here, it needs to be told explicitly what type it should use.) The final line should give Dr. Lizzy's `response` using `str` and `r`. Note that the response is random, so not necessarily that below.

```
*Main> lizzyLoop
I'm not feeling so well today, doctor.

Dr. Lizzy -- Hmm... Please tell me more about that.
```

c) Combine `lizzy` and `lizzyLoop` into a continuous dialogue. First, call `lizzyLoop` at the end of `lizzy`'s do-block. Then, at the end of `lizzyLoop`, call `lizzyLoop` again to create a loop. Test it, and use `ctrl-c ctrl-c` to exit.

```
*Main> lizzy

Dr. Lizzy -- Good morning, how are you today. Please tell me
what's on your mind.

I'm hurting, doctor.

Dr. Lizzy -- Interesting that you say "I'm hurting, doctor."
Please tell me more about that.

I took an arrow to the knee.

Dr. Lizzy -- Hmm... How does that make you feel?
```

If–then–else

We don't yet have a way to exit the dialogue with Dr. Lizzy. We will build that now, for the input string `"Exit"`. We will need to have two cases, one where `str == "Exit"`, and one otherwise. We could use a new IO function and guards to create that, but another option is to use the conditional `if-then-else`. It creates an expression as follows:

```
b :: Bool      x :: a      y :: a      if b then x else y :: a
```

To create a choice in a `do`-block, you can use `if-then-else` with a new `do`-block after `then` or `else`, or both.

```
do
    line_1
     ...
    if bool
    then do
        line_n
         ...
    else do
        line_m
         ...
```

Exercise 3: Using `if-then-else`, adapt `lizzyLoop` so that if `str == "Exit"` it exits with `putStrLn exit`, and otherwise behaves as before.

```
*Main> lizzy

Dr. Lizzy -- Good morning, how are you today. Please tell me
what's on your mind.

Exit

Dr. Lizzy -- Thank you for coming in today. I think we made
good progress. I will see you next week at the same time.
```

Lambda-calculus: Beta-reduction

In the previous tutorials, we have implemented several components of the λ-calculus: terms, free and used variables, renaming, and capture-avoiding substitution. You will find these in your Haskell file. We now have all we need to implement beta-reduction. Here is the definition of beta-reduction, phrased to guide the implementation in Haskell. A top-level beta-step is of the form

$$(\lambda x.N)\,M \quad \to_\beta \quad N[M/x]\,.$$

A beta-step can be applied anywhere in a term. This is defined by: if $N_1 \to_\beta N_2$ then

$$\lambda x.N_1 \to_\beta \lambda x.N_2 \qquad N_1\,M \to_\beta N_2\,M \qquad M\,N_1 \to_\beta M\,N_2\,.$$

We will implement a beta-step with the function `beta`. Since a term may have many redexes, or none at all (if it is in normal form), `beta` will return the list of all possible reductions.

Exercise 4:

a) Complete the function `beta`, which returns the list of all beta-reducts of a term. **Hint:** you will need four pattern-matching cases: one to see if the term is a redex, and if not, the three usual cases for `Term` to look further down in the term. In the first case, don't forget to look for further redexes as well. Since `beta` returns a list, you will have to take special care with your recursive calls. List comprehensions are your friend!

b) Complete the IO function `normalize` which reduces a term to normal form (or continues indefinitely if there isn't one). Use a `do`-block with recursion and a conditional. Your function should do the following:

- output the current term (use `putStrLn` or `print`),
- apply `beta` to find its reducts (use a `let`-clause),
- if there are reducts, take the first one and continue normalizing that,
- if there are no reducts, stop.

(Your function may reduce different redexes as the example below.)

```
*Main> Apply example (numeral 1)
(\a. \x. (\y. a c) x b) (\f. \x. f x)
*Main> beta it
[\d. (\b. (\f. \x. f x) c) d b,(\a. \x. a c b) (\f. \x. f x)]
*Main> it !! 1
(\a. \x. a c b) (\f. \x. f x)
*Main> beta it
[\d. (\f. \x. f x) c b]
*Main> head it
\d. (\f. \x. f x) c b
*Main> head (beta it)
\d. (\a. c a) b
*Main> head (beta it)
\d. c b
*Main> beta it
[]
*Main> normalize (Apply (numeral 2) (numeral 2))
(\f. \x. f (f x)) (\f. \x. f (f x))
\a. (\f. \x. f (f x)) ((\f. \x. f (f x)) a)
\a. \b. (\f. \x. f (f x)) a ((\f. \x. f (f x)) a b)
\a. \b. (\b. a (a b)) ((\f. \x. f (f x)) a b)
\a. \b. a (a ((\f. \x. f (f x)) a b))
\a. \b. a (a ((\b. a (a b)) b))
\a. \b. a (a (a (a b)))
```