

Tutorial 6: 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: **first** it performs some IO actions, and **then** it releases a value of type `a`. We will use some of the following built-in functions:

<code>putStr</code>	<code>:: String -> IO ()</code>	prints a string to the console
<code>putStrLn</code>	<code>:: String -> IO ()</code>	prints a string and starts a new line
<code>print</code>	<code>:: Show a => a -> IO ()</code>	is <code>putStrLn . show</code>
<code>getLine</code>	<code>:: IO String</code>	asks for a line in the console
<code>return</code>	<code>:: a -> IO a</code>	wraps a value in the “do nothing” IO action
<code>randomIO</code>	<code>:: IO Int</code>	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`.

Note: when loading the file, if you get the error

```
Could not find module 'System.Random'
```

then the **Random** module is not installed on your system. You could try to install it, or you can follow the instructions in this document on how to work around it. First, comment out the import of the random module. Then, un-comment the alternative implementation of `randomIO`. If it works, you're good to go; otherwise, if you get the error

```
Could not find module 'Data.Time.Clock'
```

then the module that accesses the system clock is also not installed. In that case, follow the instructions in the tutorial that avoid using random inputs.

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.

```
ghci> 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` or `randomresponse`. The first two are just strings. The last two are functions that take a string `str`, the user’s last input, and select one of 3×5 possible responses. Here, `response` chooses a response based on the length of `str`, whereas `randomresponse` asks for an integer `r` as input, that we will generate at random.

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

```
ghci> 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 two lines: the first uses `getLine` to get a user response `str`, and the second should use `response` give a response.

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

Dr. Lizzy -- Let's examine that more closely, shall we.
Do you think this has something to do with your mother?

- c) If using `Random`, insert the line `r <- randomIO :: IO Int` in the middle, 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.) Use `r` to call `randomresponse` instead of `response`. Note that the response becomes random, so not necessarily that above.
- d) 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` to exit.

```
ghci> lizzy
```

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

```
I'm hurting, doctor.
```

Dr. Lizzy -- Let's examine that more closely, shall we.
Please tell me more about that.

```
I took an arrow to the knee.
```

Dr. Lizzy -- Hmm... Do you think this has something to do with
your mother?

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
x :: 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.

```
ghci> 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.
```

Tic-tac-toe

This part of the tutorial is more open-ended. Use what you have learned so far to complete the following assignment.

Exercise 4: Implement the game tic-tac-toe where the computer opponent plays random moves. (If you are working without random input, you can use the 1st available move, or 2nd, or 3rd, etc.) It could look as follows (player is `X` and opponent is `O`):

```
ghci> tictactoe
```

```
  |  |  
---+---+---  
  |  |  
---+---+---  
  |  |
```

Choose your move from 0 1 2 3 4 5 6 7 8: 4

```
  |  |  
---+---+---  
  | X |  
---+---+---  
0 |  |
```

Choose your move from 0 1 2 3 5 7 8: 8

```
  |  | 0  
---+---+---  
  | X |  
---+---+---  
0 |  | X
```

Choose your move from 0 1 3 5 7: 0

```
X |  | 0  
---+---+---  
  | X |  
---+---+---  
0 |  | X
```

You won!

Extra

Exercise 5: Implement an opponent for tic-tac-toe that plays optimally.