The IO Monad.
Lab 9

Recall that `IO a` is a type whose values are *actions that, once executed, produce a value of type* `a`.

For example, the function `putStrLn :: String -> IO ()` takes a string and returns an action which, once executed, produces a value of type *unit* (the *unit* type is written `()`).

Exercise 0.1. What is the only value of type `()`?

To make a program executable, it is enough to write a function named `main` that takes no arguments and returns a value of type `IO ()`, that is, an action.

```
main :: IO ()
main = putStrLn "Hello, World!"
```

If you compile a file that contains a `main` function, then when you run the executable, Haskell will evaluate the main function (the result will be an action) and then *execute the action.*

```
$ ghc file.hs -o executablefile
...
$ ./executablefile
Hello, World!
```

You can use the binary operator `(>>) :: IO a -> IO b -> IO b` to *sequentially compose two actions.*

```
main :: IO ()
main = (putStrLn "Hello, World!") >> (putStrLn "All good.")
```

Exercise 0.2. Use `>>` to create a sequence of three actions.

Use the `:info` command to find out whether the parser associates `>>` to the left or to the right by default.

The actions returned by the `putStrLn` function produce, after execution, a value of type unit, which is not very interesting. Other actions, such as the action returned by the `getLine` function, produce interesting values:

```
> :t getLine
getLine :: IO String
```

When executed, the action returned by the `getLine` function reads a string from the console. Here is an example of using the `getLine` function:

```
main :: IO ()
main = (putStrLn "What is your name?") >>
       (getLine) >>
       (putStrLn "Hello, ...!")
```

In a sequence of actions, how can we get hold of the value produced by one of the actions? We cannot if we only use the `(>>)` operator:

```
:t (>>) :: IO a -> IO b -> IO b
(>>) :: IO a -> IO b -> IO b :: IO a -> IO b -> IO b
```

As you can see, the value of type `a` produced by the first action *is lost*. If we need it, we have to use the `>>=` operator:

```
:t (>>=) :: IO a -> (a -> IO b) -> IO b
(>>=) :: IO a -> (a -> IO b) -> IO b :: IO a -> (a -> IO b) -> IO b
```

It takes two arguments:

1. the first argument (of type `IO a`) is an action which, if executed, produces a value of type `a`;
2. the second argument (of type `a -> IO b`) is a function that takes a value of type `a` and returns an action which, if executed, produces a value of type `b`.

The `>>=` operator returns an action which, if executed, first executes the action of type `IO a` and uses the value it produces as the argument of the function of type `a -> IO b`, and then executes the action returned by that function. Here is an example:

```
main :: IO ()
main = (putStrLn "What is your name?") >>
       (getLine >>=
        (\name -> putStrLn ("Hello, " ++ name ++ "!")))
```

Exercise 0.3. Write a program that reads the user's first name from the keyboard, then their last name, and displays a greeting message (hint: you will use `>>=` twice).

We can rewrite the program above using `do` notation:

```
main :: IO ()
main = do
       putStrLn "What is your name?"
       name <- getLine
       putStrLn ("Hello, " ++ name ++ "!")
```

Exercise 0.4. Redo the first-name and last-name program above using `do` notation.

To repeat an action in a loop, we can use recursion:

```
main :: IO ()
main = do
       putStrLn "What is your name?"
       name <- getLine
       putStrLn ("Hello, " ++ name ++ "!")
       main
```

To exit the program above, use Ctrl-C or Ctrl-Z, depending on your operating system.

Exercise 0.5. Write the recursive program above using `>>` and `>>=`, without `do` notation.

Exercise 0.6. Modify the program above so that it asks (forever) for the first name and then the last name.

The function `return :: a -> IO a` takes a value as input and returns an action which, when executed, does nothing and produces the value given as the argument.

```
main :: IO ()
main = return ()
```

Warning! The `return` function is nothing like the `return` keyword in the C(++)/Java family of languages. To understand this, explain the behavior of the following programs:

```
main :: IO ()
main = do
       putStrLn "What is your name?"
       name <- return "Victor"
       putStrLn ("Hello, " ++ name ++ "!")

main :: IO ()
main = do
       putStrLn "What is your name?"
       return ()
       name <- getLine
       putStrLn ("Hello, " ++ name ++ "!")
```

To avoid the infinite loop in Exercises **??** and **??**, it is enough to add a termination condition:

```
main :: IO ()
main = do
       putStrLn "What is your name?"
       name <- getLine
       putStrLn ("Hello, " ++ name ++ "!")
       if name == "" then
         return ()
       else
         main
```

Exercise 0.7. Modify the program that reads first and last names so that it stops when the first name **or** the last name is the empty string.

Exercise 0.8. Write a program that reads strings from the keyboard and displays them in UPPERCASE.

Exercise 0.9. Look up the following functions in the standard library documentation (including which module they are in) and write down their types:

1. `openFile :: ...`
2. `hGetContents :: ...`

3. `hGetLine :: ...`
4. `hClose :: ...`
5. `getArgs :: ...`
6. `getProgName :: ...`
7. `hPutStr :: ...`

Exercise 0.10. Write a program that displays the contents of the text file "example.txt" on the screen.

Exercise 0.11. Modify the program so that the file name is the first command-line argument.

Exercise 0.12. Modify the program so that, if the command line is incorrect, it displays the program's usage syntax (use the `getProgName` function).

Exercise 0.13. Modify the program so that it displays the contents of the file in UPPERCASE.

Exercise 0.14. Create a program that guesses a number between 1 and 100 (using binary search). Here is an example interaction:

```
$ guess
Is the number to guess >= 50?
No
Is the number to guess >= 25?
No
Is the number to guess >= 12?
No
Is the number to guess >= 6?
Yes
Is the number to guess >= 9?
asdf
I didn't understand. Is the number to guess >= 9?
No
Is the number to guess >= 7?
Yes
Is the number to guess >= 8?
No
Then the number is 7.
```

Exercise 0.15. Explain why lazy evaluation is crucial for executing `IO` actions.

Exercise 0.16. Design an experiment that demonstrates that the `hGetContents` function does not read the entire contents of a file unless it is strictly necessary.

Exercise 0.17. Write a Haskell program that reads a number `n` from standard input and displays `n` pseudorandom numbers.