Monadic IO Lab

Functional Programming Curriculum

In this lesson we learned how to do monadic IO in a typed *purely functional* programming language. Unlike *imperative* programming languages such as Python, these languages do not have a syntactic class of *statements*. Instead, some *expressions* represent *computations*, which are instructions to the run-time system to perform various *actions*. These are distinguished in the type system by being elements of `IO` types.

We can build up compound computations from simpler ones using two monadic combinators, `pure : a -> IO a`, which produces a trivial computation, and `(>>=) : IO a -> (a -> IO b) -> IO b`, which sequences computations by running the first and passing the resulting value to the next.

There is syntactic sugar called `do`-notation, in which a sequence of computations can be written to resemble a block of statements in an imperative programming language. This can sometimes be convenient, but it is important to understand that this is merely a syntactic transformation: functional programming languages do not have statements.

Task 1
Write a function that doesn't give up until it gets a number from the user.

```
get_number  :  IO Integer
```

For example:

```
> :exec get_number
Please enter a number: forty two
I'm sorry, I didn't understand that.
Please enter a number: You know, the answer to life, the universe and everything.
I'm sorry, I didn't understand that.
Please enter a number: 42
42
```

Solution

```
get_number  =
  putStr "Please enter a number: " >>=
  const getLine >>=
  \ str => case parseInteger str of
    Nothing =>
      putStrLn "I'm sorry, I didn't understand that." >>=
      const get_number
    Just num => pure num
```

Task 2
Desugar the following function to use the computation sequencing operator `(>>=)` rather than `do`-notation:

```
add_pair  : IO Integer
add_pair  =  do
  putStr "Please enter the first number: "
  x <- get_number
  putStr "Please enter the second number: "
  y <- get_number
  pure (x + y)
```

Solution

```
add_pair'  :  IO Integer
add_pair'  =
  putStr "Please enter the first number: " >>=
  const get_number >>=
  \ x => putStr "Please enter the second number: " >>=
  const get_number >>=
  \ y => pure (x + y)
```

Task 3

Write a function that gets a list of numbers from the user, like `get_numbers` from the lecture, and returns their sum, or zero if the list is empty.

```
add_numbers  :  IO Integer
```

hint: You can write this as a short one-liner using `get_numbers` together with computation sequencing and higher-order functions.

Solution

```
add_numbers  =  get_numbers >>= pure . (foldr (+) 0)
```

Task 4

Write a checked version of `get_numbers`, which prompts the user to re-enter their input if it is unable to parse an integer from it.

```
get_numbers_checked  :  IO (List Integer)
```

For example:

```
> :exec get_numbers_checked
Please enter a number or 'done': 1
Please enter a number or 'done': two
I'm sorry, I didn't understand that.
Please enter a number or 'done': 2
Please enter a number or 'done': 3
Please enter a number or 'done': done
[1, 2, 3]
```

Solution

```
get_numbers_checked  =  do
  putStr "Please enter a number or 'done': "
  str <- getLine
  if str == "done"
```

```
then pure []
else case parseInteger str of
  Nothing => do
    putStrLn "I'm sorry, I didn't understand that."
    get_numbers_checked
  Just num => do
    nums <- get_numbers_checked
    pure (num :: nums)
```