Lab 6

Functional Programming

2024-03-01

This week we learned how to do monadic I/O in typed *purely functional* programming languages. Unlike *imperative* programming languages these do not use *statements* to cause *effects*. 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. An element of type `IO a` is a computation that when run may perform some actions and then returns a result of type `a`.

We build up compound computations 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 be convenient, but it is important to understand that it is merely a syntactic transformation.

When `:exec`ing a computation in the interactive interpreter, you can see a textural representation of its result by sequencing it with `printLn : Show a => a -> IO Unit`.

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

```
get_number  :  IO Integer
```

For example:

```
Lab6> :exec get_number >>= printLn
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
```

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

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

Task 3
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:

```
Lab6> :exec get_numbers_checked >>= printLn
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]
```

Task 4
Write a computation that gets a list of numbers from the user and returns their sum, or zero if the list is empty.

```
add_numbers  :  IO Integer
```

For example:

```
Lab6> :exec add_numbers >>= printLn
Please enter a number: 1
Please enter a number: 2
Please enter a number: 3
Please enter a number: done
6
```

Hint: you can write this as a point-free one-liner using `get_numbers` from lecture together with computation sequencing, function composition, and the *fold* for lists, which you can find in the standard library as:

```
foldr  :  (c : a -> t -> t) -> (n : t) -> List a -> t
```

Task 5
Write a function that takes both a computation that when run produces a result of type `a` and a computation that when run produces a result of type `b`, and returns a computation that when run, runs the two computations in that order and produces the pair of their results:

```
bothIO  :  IO a -> IO b -> IO (Pair a b)
```

Now write a function `bothIO'` that has the same type as `bothIO`, but which runs the two argument computations in the opposite order:

```
Lab6> :exec bothIO (putStrLn "hello") (putStrLn "world")
hello
world
Lab6> :exec bothIO' (putStrLn "hello") (putStrLn "world")
world
hello
```

Task 6
Write a function that takes a string transformer function and paths to a source and target file, which reads the contents of the source file, transforms it by the function, and writes the result to the target file.

```
transform_file  :  (transform : String -> String) ->
  (src_path : String) -> (tgt_path : String) ->
  IO (Either FileError Unit)
```

You can read the contents of a file using
`readFile  :  String -> IO (Either FileError String)`
and write the contents of a file using one of
`writeFile , appendFile  :  String -> String -> IO (Either FileError Unit)`
which you can use by importing the `System.File` module.

Note: You can use `Data.String.toUpper : String -> String` to test your function.
Be careful not to overwrite any important files on your system.