Lab 5

Functional Programming (ITI0212)

2021.02.23

Recall that in the lecture, we imported the `Data.Strings` module from the standard library in order to use `parsePositive`. In particular, we made the following function definition:

```
parseNat : String -> Maybe Nat
parseNat = parsePositive
```

It is reccommended that you do the same for this exercise set. You will also want to recall the functions `getLine` and `putStr` (also `purStrLn`) from the lecture. For the final exercise, you will need to use the function `writeFile` from the `System.File` module of the standard library.

1. (a) Write a function `getNat : IO Nat` which, when executed, prompts the user to enter a natural number. Then, if the user enters a valid natural number, the program should return that number. Otherwise, the program should return zero. For example

```
REPL> :exec (getNat >>= printLn)
Please Enter a Nat: 15
15
REPL> :exec (getNat >>= printLn)
Please Enter a Nat: don't tell me what to do!
0
```

   (b) Write a function `insistNat : IO Nat` which, when executed, prompts the user to enter a natural number. If the user enters a valid natural number, the program should return that number. Otherwise, the program should prompt the user again, until a valid natural number is entered. For example:

```
REPL> :exec (insistNat >>= printLn)
Please Enter a Nat: 15
15
REPL : exec (insistNat >>= printLn)
Please Enter a Nat: no
Please Enter a Nat: i'd rather not
Please Enter a Nat: please let me go
Please Enter a Nat: why are you doing this?
Please Enter a Nat: 0
0
```

(c) Write a function `insistAdd : Nat -> IO Nat` such that for `n : Nat` the program `insistAdd n`, when executed, prompts the user for a natural number until a valid number is entered (use `insistNat`), and then returns the result of adding `n` to that number.

```
REPL> :exec (insistAdd 3 >>= printLn)
Please Enter a Nat: 5
8
REPL> :exec (insistAdd 10 >>= printLn)
Please Enter a Nat: why
Please Enter a Nat: 10
20
```

(d) Write a function `addAfter : (IO Nat) -> Nat -> IO Nat` such that `addAfter insistNat` does the same thing as `insistAdd`. How does the behaviour of `addAfter insistNat` differ from the behaviour of `addAfter getNat`? Give an example where they differ.

2. (a) Write a function `natsGet : IO (Maybe (List Nat))` that reads a line of user input consisting of space separated natural numbers, and returns the corresponding `List Nat`. If the user input cannot be parsed as a list of natural numbers, the program should return `Nothing`. For example:

```
REPL> :exec natsGet >>= printLn
10 6 7 28
Just [10,6,7,28]
REPL> :exec natsGet >>= printLn
10 6 no 28
Nothing
REPL> :exec natsGet >>= printLn
<empty line>
Just []
```

(b) Write a function `tryNats : IO (List Nat)` that read a line of user input consisting of space separated natural numbers, and returns the corresponding `List Nat`. If some part of the user input cannot be parsed as a natural number, it should be omitted from the list, but the rest of the list should still be returned. For example:

```
REPL> :exec tryNats >>= printLn
10 no 5 3 no 2
[10,5,3,2]
REPL> :exec tryNats >>= printLn
no 23 10 5 nope
```

```
[23,10,5]
REPL> :exec tryNats >>= printLn
go away
[]
```

3. (a) Write a function `getLines : IO (List String)` that reads lines of user input until the user enters `done`, and returns the lines as a `List String`. For example

```
REPL> :exec getLines >>= printLn
Enter Line: Hello computer
Enter Line: I have letters to feed you
Enter Line: How many letters can you eat?
Enter Line: done
["Hello computer","I have letters to feed you","How many letters can you eat?"]
REPL> :exec getLines >>= printLn
Enter Line: done
[]
REPL> :exec getLines >>= printLn
Enter Line: done must be the only thing on the line to end the process
Enter Line: done
["done must be the only thing on the line to end the process"]
```

(b) Write a function `dictate : IO ()` and compile it to obtain an executable (here also named `dictate`). When run, `dictate` should read lines until the user enters `done`, then prompt the user for the name of a file to store those lines in. If the user enters `none`, the lines are instead thrown away. If the user enters a valid file name, then the program should attempt to store the lines of user input as a file with that name. A message relaying the success of failure of this operation should be printed before the program exits. For example:

```
$> ./dictate
Enter Line: how many lines
Enter Line: should we put in our file?
Enter Line: there is no right answer
Enter Line: let your heart decide!
Enter Line: done
Enter Storage Location: amazing-poem.txt
Success!
$> cat amazing-poem.txt
how many lines
should we put in our file?
there is no right answer
let your heart decide!
$> ./dictate
Enter Line: if we write a bad poem
Enter Line: we might not want to keep it
Enter Line: in that case, we write
Enter Line: done
Enter Storage Location: none
Throwing Lines Away!
```

These examples are not exhaustive: there are many things that can go wrong when opening a file. For your program it is okay to print a generic error message if one of them happens.