PROGRAMMING IN HASKELL

Part 4 - Interactive Programs and Monads
We would also like to use Haskell to write interactive programs that read from the keyboard and write to the screen, as they are running.
The Problem

Haskell programs are pure mathematical functions:

- Haskell programs **have no side effects**.

However, reading from the keyboard and writing to the screen are side effects:

- Interactive programs **have side effects**.
The Solution

Interactive programs can be written in Haskell by using types to distinguish pure expressions from impure actions that may involve side effects.

The type of actions that return a value of type \( a \).
For example:

\[
\textbf{IO Char}
\]

The type of actions that return a character.

\[
\textbf{IO ()}
\]

The type of purely side effecting actions that return no result value.

Note:

\[
\text{() is the type of tuples with no components.}
\]
Basic Actions

The standard library provides a number of actions, including the following three primitives:

- The action `getChar` reads a character from the keyboard, echoes it to the screen, and returns the character as its result value:

  `getChar :: IO Char`
The action `putChar c` writes the character `c` to the screen, and returns no result value:

```
putChar :: Char → IO ()
```

The action `return v` simply returns the value `v`, without performing any interaction:

```
return :: a → IO a
```
A sequence of actions can be combined as a single composite action using the keyword \texttt{do}.

For example:

\begin{verbatim}
  a :: IO (Char,Char)
  a = do x ← getChar
           getChar
           y ← getChar
           return (x,y)
\end{verbatim}
Derived Primitives

Reading a string from the keyboard:

```haskell
getLine :: IO String
getLine  = do x ← getChar
             if x == '\n' then
               return []
             else
               do xs ← getline
                  return (x:xs)
```
- Writing a string to the screen:

```
putStr :: String → IO ()
putStr [] = return ()
putStr (x:xs) = do putChar x
                putStr xs
```

- Writing a string and moving to a new line:

```
putStrLn :: String → IO ()
putStrLn xs = do putStr xs
                putChar '\n'
```
Example

We can now define an action that prompts for a string to be entered and displays its length:

```haskell
strlen :: IO ()
strlen = do putStr "Enter a string: "
            xs ← getLine
            putStrLn "The string has "
            putStrLn (show (length xs))
            putStrLn " characters"
```
For example:

> strlen

Enter a string: abcde
The string has 5 characters

Note:

- Evaluating an action executes its side effects, with the final result value being discarded.
The Monad Class

The IO type is an instance of the monad class.

\[
\text{class Monad m where}
\]

\[
\text{return :: a -> m a}
\]

\[
\text{(>>=) :: m a -> (a -> m b) -> m b}
\]

\[
\text{(>>=)} \text{ is the bind operator of the monad.}
\]
The do notation is just syntactic sugar for the bind operator >>=.

\[\begin{align*}
e1 & \gg= \ \backslash v1 \rightarrow \\
e2 & \gg= \ \backslash v2 \rightarrow \\
\text{return } (f \ v1 \ v2)
\end{align*}\]
The Maybe Monad

The Maybe data type is useful when interacting with databases, dictionaries, ....

data Maybe a = Nothing | Just a

instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f = f x
The List Monad

instance Monad [] where
    return x = [x]
    xs >>= f = concatMap f xs

where

concatMap :: (a -> [b]) -> [a] -> [b]
Homework!

Prepare Chap. 18.2 “The Monad Class” from *The Haskell School of Expression* by Paul Hudak till next time.