Input and Output

Main concepts of this unit:

> The *World*
> - `IO` t
> Actions
> Composition
> - `>>`
> - `>>=`
> - `return`
> Do notation
> - `<-`
> - `let`
> An example

The World

Referential transparency requires that the *same* expression evaluates always to the *same* value. Suppose that a programming language has a function, say `getChar`, to read a character from a stream. How can this be consistent with the requirement of referential transparency?

> `bad = (getChar,getChar)`

An option is that `getChar` takes an argument, referred to as the *World*, and returns the character read from the stream plus a new *World*.

The next time `getChar` is called, the *World* has changed, thus returning a different character does not violate the requirement of referential transparency?

The type `World` is hidden. There is a type `IO` t which is an abbreviation for

> `World -> (`t`,World)`

The *initial* *World* is supplied automagically by the run-time environment.

Actions

An expression that "changes" the *World* is called an *action*. The following actions read a character or a line from standard input:

> ```
> getChar ::  IO Char
> getLine ::  IO String
> ```

The following actions take an argument and put it on standard output:

> ```
> putChar  ::  Char   -> IO ()
> putStr   ::  String -> IO ()
> putStrLn ::  String -> IO ()
> ```

Contrary to all other expressions, the *order* in which actions are executed is relevant. E.g., consider `bad` in the previous page. The operation `>>` composes actions so that they occur in the right *order*, e.g.,

> ```
> putStr "Hello"
>   >> putChar ' '
>   >> putStrLn "world."
> ```

Composition

The type of the operation `>>` is:

> `>> ::  IO a -> IO b -> IO b`

The value returned by the first action is *ignored* by the second action. When the value returned by the first action is to be *used* by the second action, a different composition is available:

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

For example:

> ```
> getChar >>= putChar
> getLine >>= putLine
> ```

copy a character and a line from standard input to standard output, respectively.

There is one last operation to *constructs* `IO` values from ordinary values:

> `return ::  a -> IO a`

e.g.:

> `return "hello world" >>= putLine`

Do notation

Values read by actions can be used by computations, e.g.,

```
getLine >>=
  \line -> putStr "Your input:  " >>
           putStrLn line
```

A special notation is available to ease the above:

```
do line <- getLine
   putStr "Your input:  "
   putStrLn line
```

The indentation must follow the off-side rule. There is also an abbreviated `let` construct for ordinary binding:

```
do line <- getLine
   let prefix = "Your input:  "
   putStrLn (prefix ++ line)
```

An example

The following program, similar to Unix's *wc*, counts the number of lines, words, and character in a file. The efficiency of the program is not an issue in this example.

```
import IO -- readFile

main fileName = do
    content <- readFile fileName
    let (c,w,l) = process content
    putStrLn (show l ++ " " ++
              show w ++ " " ++
              show c ++ " " ++
              fileName)

process content = (c,w,l)
    where c = length content
          w = length (words content)
          l = length (lines content)
```