Background – Operators (1D)

Copyright (c) 2016 - 2017 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

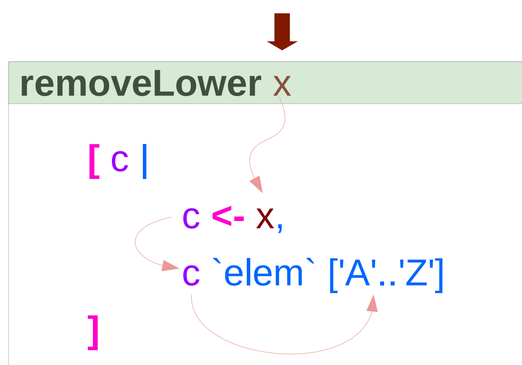
Based on

Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

A List Comprehension Function

```
let removeLower x = [c | c <- x,  c `elem` ['A'..'Z']]
```

a **list comprehension**



“Hello”

[c: ‘H’
c: ‘e’
c: ‘l’
c: ‘l’
c: ‘o’]

“H”

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```

x1 : Return value of action1

x2: Return value of action2

https://stackoverflow.com/questions/35198897/does-mean-assigning-a-variable-in-haskell

Pattern and Predicate

```
let removeLower x = [c | c <- x,  c `elem` ['A'..'Z']]
```

a **list comprehension**

[c | c <- x, c `elem` ['A'..'Z']]

c <- x is a **generator**

(x : argument of the function **removeLower**)

c is a **pattern**

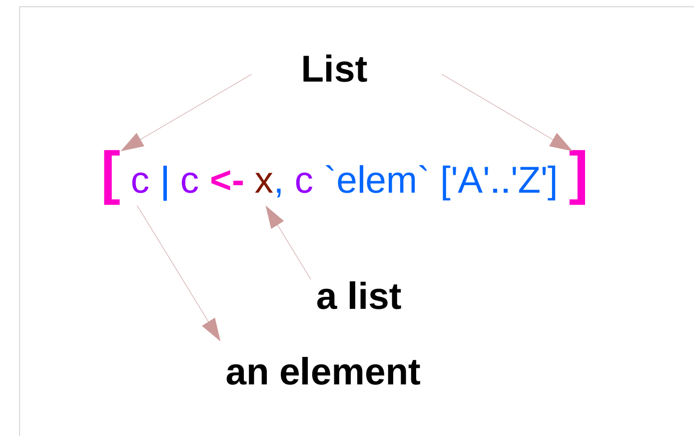
matching from the **elements** of the **list x**

successive binding of c to the **elements** of the **list x**

c `elem` ['A'..'Z']

is a **predicate** which is applied to each successive binding of c

Only c which <u>passes</u> this predicate will appear in the output list



https://stackoverflow.com/questions/35198897/does-mean-assigning-a-variable-in-haskell

Assignment in Haskell

Assignment in Haskell : declaration with initialization:

- no uninitialized variables,
- must declare with an initial value
- no mutation
- a variable keeps its initial value throughout its scope.

https://stackoverflow.com/questions/35198897/does-mean-assigning-a-variable-in-haskell

Generator

```
[c| c <- x, c `elem` ['A'..'Z']]

filter (`elem` ['A' .. 'Z']) x
```

```
[ c | c <- x ]
```

c: an element
x: a list

```
do c <- x
  return c
```

```
x >>= ( \c -> return c )
```

```
x >>= return
```

c: an element
x: an element

or

c: a list
x: a list

```
action1 >>= (\ x1 ->
  action2 >>= (\ x2 ->
    mk_action3 x1 x2 ))
```

https://stackoverflow.com/questions/35198897/does-mean-assigning-a-variable-in-haskell

Anonymous Functions

```
(\x -> x + 1) 4
5 :: Integer


(\x y -> x + y) 3 5
8 :: Integer

inc1 = \x -> x + 1
```

```
incListA lst = map inc2 lst
   where inc2 x = x + 1
```

```
incListB lst = map (\x -> x + 1) lst
```

```
incListC = map (+1)
```

https://wiki.haskell.org/Anonymous_function

Then Operator (>>) and do Statements

a chain of actions

to *sequence* input / output operations

the (>>) (**then**) operator works almost identically in **do** notation

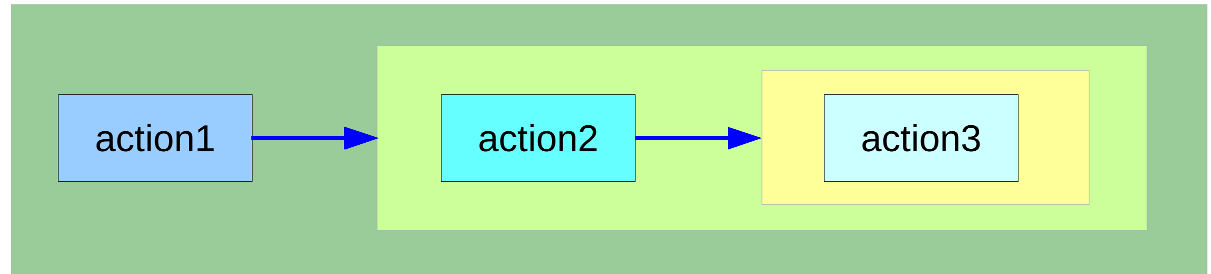
```
putStr "Hello"  >>
putStr " "      >>
putStr "world!" >>
putStr "\n"
```

```
do { putStr "Hello"
   ; putStr " "
   ; putStr "world!"
   ; putStr "\n" }
```

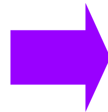
https://en.wikibooks.org/wiki/Haskell/do_notation

Chaining in **do** and >> notations

```
do { action1
     ; action2
     ; action3 }
```



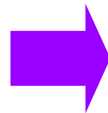

```
do { action1
     ; do { action2
          ; action3 } }
```



```
action1 >>
do { action2
     ; action3 }
```

can **chain** any actions
all of which are in **the same monad**

```
do { action1
     ; do { action2
          ; do { action3 } } }
```



```
action1 >>
  action2 >>
    action3
```

https://en.wikibooks.org/wiki/Haskell/do_notation

Bind Operator (>==) and **do** statements

The bind operator (>>=)

passes a value ->

(the result of an action or function),

downstream in the binding sequence.

```
action1 >>= (\ x1 ->
  action2 >>= (\ x2 ->
    mk_action3 x1 x2 ))
```

anonymous function

(lambda expression)

is used

do notation assigns a variable name

to the passed value using the <-

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```

https://en.wikibooks.org/wiki/Haskell/do_notation

Chaining >>= and do notations

->

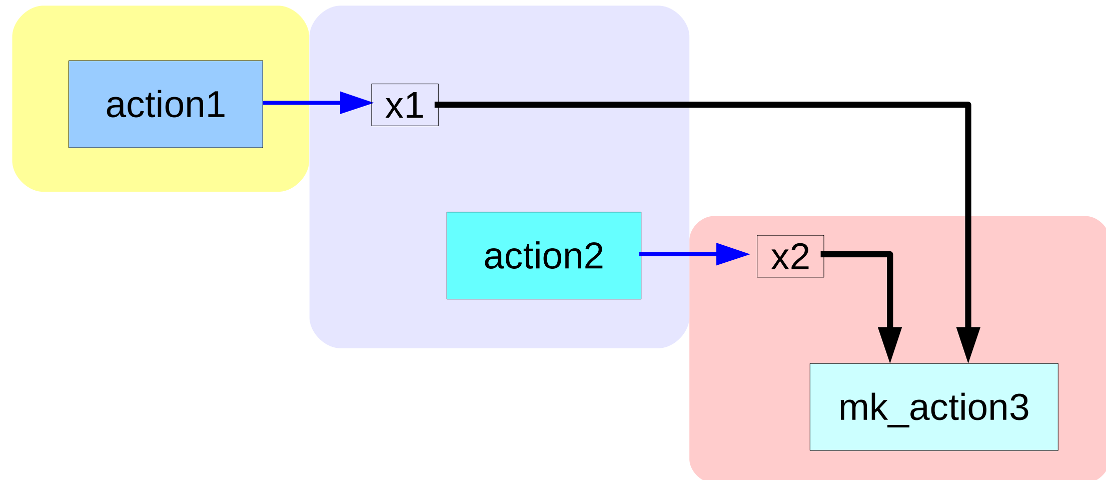
```
action1 >>= (\ x1 -> action2 >>= (\ x2 -> mk_action3 x1 x2 ))
```

```
action1
  >>=
    (\ x1 -> action2
       >>=
         (\ x2 -> mk_action3 x1 x2 ))
```

```
action1 >>= (\ x1 ->
  action2 >>= (\ x2 ->
    mk_action3 x1 x2 ))
```

<-

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```



https://en.wikibooks.org/wiki/Haskell/do_notation

fail method

```
do { Just x1 <- action1
   ;        x2 <- action2
   ; mk_action3 x1 x2  }
```

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```

O.K. when action1 returns Just x1

when action1 returns Nothing
crash with an non-exhaustive patterns error

Handling failure with **fail** method

```
action1 >>= f where
    f (Just x1) = do { x2 <- action2
                     ; mk_action3 x1 x2 }
    f _         = fail "..."
```

-- *A compiler-generated message.*

https://en.wikibooks.org/wiki/Haskell/do_notation

Example

```
nameDo :: IO ()
nameDo = do { putStr "What is your first name? "
            ; first <- getLine
            ; putStr "And your last name? "
            ; last <- getLine
            ; let full = first ++ " " ++ last
            ; putStrLn ("Pleased to meet you, " ++ full ++ "!") }
```

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```

using the **do** statement

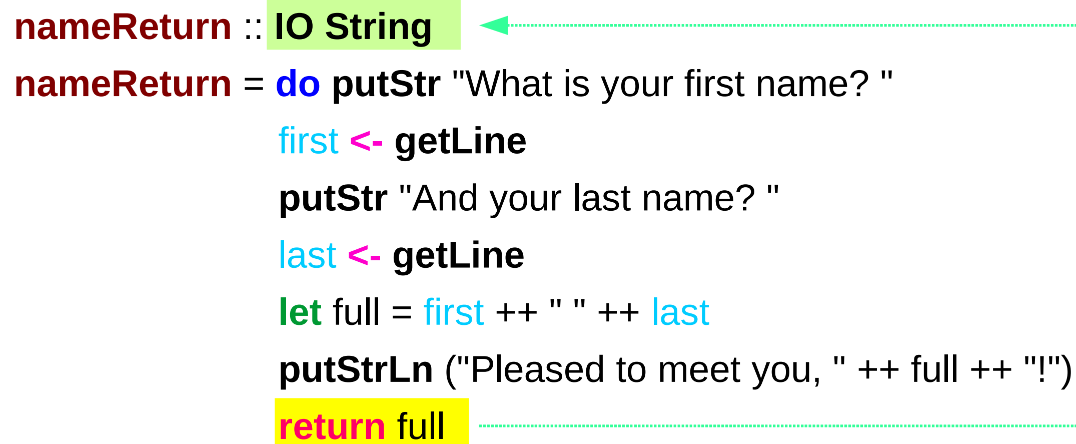
A possible translation into vanilla monadic code:

```
nameLambda :: IO ()
nameLambda = putStr "What is your first name? " >>
             getLine >>= \ first ->
             putStr "And your last name? " >>
             getLine >>= \ last ->
             let full = first ++ " " ++ last
             in putStrLn ("Pleased to meet you, " ++ full ++ "!")
```

using **then (>>)** and **Bind (>>=)** operators

https://en.wikibooks.org/wiki/Haskell/do_notation

return method



```
nameReturn :: IO String
nameReturn = do putStr "What is your first name? "
                first <- getLine
                putStr "And your last name? "
                last <- getLine
                let full = first ++ " " ++ last
                putStrLn ("Pleased to meet you, " ++ full ++ "!")
                return full
```

```
greetAndSeeYou :: IO ()
greetAndSeeYou = do name <- nameReturn
                    putStrLn ("See you, " ++ name ++ "!")
```

https://en.wikibooks.org/wiki/Haskell/do_notation

Without a **return** method

```
nameReturn :: IO String
nameReturn = do putStr "What is your first name? "
                first <- getLine
                putStr "And your last name? "
                last <- getLine
                let full = first ++ " " ++ last
                putStrLn ("Pleased to meet you, " ++ full ++ "!")
                return full
```

explicit return statement

returns **IO String** monad

```
nameDo :: IO ()
nameDo = do { putStr "What is your first name? "
            ; first <- getLine
            ; putStr "And your last name? "
            ; last <- getLine
            ; let full = first ++ " " ++ last
            ; putStrLn ("Pleased to meet you, " ++ full ++ "!") }
```

no return statement

returns **empty IO** monad

https://en.wikibooks.org/wiki/Haskell/do_notation

return method – not a final statement

```
nameReturnAndCarryOn :: IO ()
nameReturnAndCarryOn = do putStr "What is your first name? "
                          first <- getLine
                          putStr "And your last name? "
                          last <- getLine
                          let full = first++" "++last
                          putStrLn ("Pleased to meet you, "++full++"!")
                          return full
                          putStrLn "I am not finished yet!"
```

the return statement does not interrupt the flow

the last statements of the sequence returns a value

https://en.wikibooks.org/wiki/Haskell/do_notation

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf