# Background – Operators (1E)

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

# Based on

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

# **zip** function

```
zip :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs) = (a,b) : zip as bs
zip _      _      = []
```

```
Prelude> zip [1..3] [10..30]
[(1,10),(2,11),(3,12)]

Prelude> zip [1..3] [10..11]
[(1,10),(2,11)]
```

https://stackoverflow.com/questions/5776322/zip-function-in-haskell

# zipwith function

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _ _      _      = []
```

```
Prelude> zipWith (+) [1..3] [10..30]
[11,13,15]

Prelude> zipWith (+) [1..3] [10..11]
[11,13]
Prelude>
```

https://stackoverflow.com/questions/5776322/zip-function-in-haskell

# Set Builder Notation

$$S = \{\, 2 \cdot x \mid x \in \mathbb{N},\ x^2 > 3 \,\}$$

$$S = \{ \underbrace{2 \cdot x}_{\text{output expression}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\mathbb{N}}_{\text{input set}} , \underbrace{x^2 > 3}_{\text{predicate}} \}$$

This can be read,
"**S** is the set of all numbers **2x**
where **x** is an item in the set of natural numbers (**N**),
for which **x** squared is greater than 3

https://en.wikipedia.org/wiki/List_comprehension

# List Comprehension

$$S = \{ \underbrace{2 \cdot x}_{\text{output expression}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\mathbb{N}}_{\text{input set}}, \underbrace{x^2 > 3}_{\text{predicate}} \}$$

A **list comprehension** has the same syntactic components to represent generation of a list in order from an input list or iterator:

- A **variable** representing members of an input list.
- An **input list** (or iterator).
- An optional **predicate expression**.
- And an **output expression**
  producing members of the output list
  from members of the input iterable that satisfy the predicate

https://en.wikipedia.org/wiki/List_comprehension

# Left Arrow **<-** in List Comprehension

**s = [ 2*x | x <- [0..], x^2 > 3 ]**

the input list **[0..]** represents ***N***

**x^2>3** the predicate

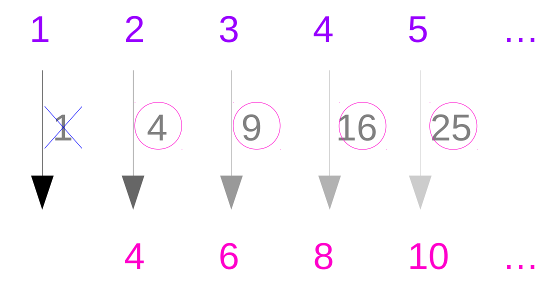
**2*x** the output expression

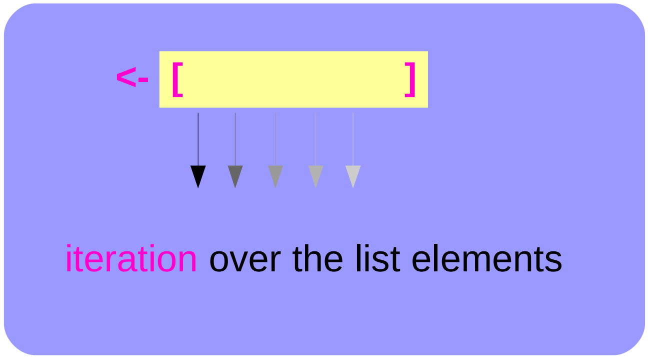
results in a defined order

may generate the members of a list in order,

rather than produce the entirety of the list

thus allowing the members of an infinite list.




https://en.wikipedia.org/wiki/List_comprehension

# Parallel List Comprehension

The Glasgow Haskell Compiler has an extension called **parallel** list comprehension (**zip**-comprehension) permits multiple independent branches of qualifiers

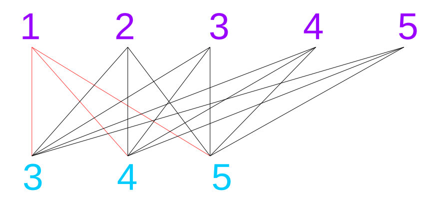
- qualifiers separated by commas are dependent ("nested"),
- qualifiers separated by pipes are evaluated in parallel (it merely means that the branches are zipped).

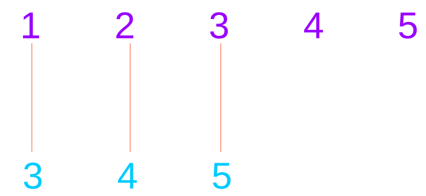
https://en.wikipedia.org/wiki/List_comprehension

# Parallel List Comprehension Examples

```
[(x,y) | x <- [1..5], y <- [3..5]]          -- regular list comprehension
-- [(1,3),(1,4),(1,5),(2,3),(2,4) ...

[(x,y) | x <- [1..5] | y <- [3..5]]         -- parallel list comprehension
[(x,y) | (x,y) <- zip [1..5] [3..5]]        -- zipped list comprehension
-- [(1,3),(2,4),(3,5)]
```




https://en.wikipedia.org/wiki/List_comprehension

# A List Comprehension Function

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

a **list comprehension**

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

“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**

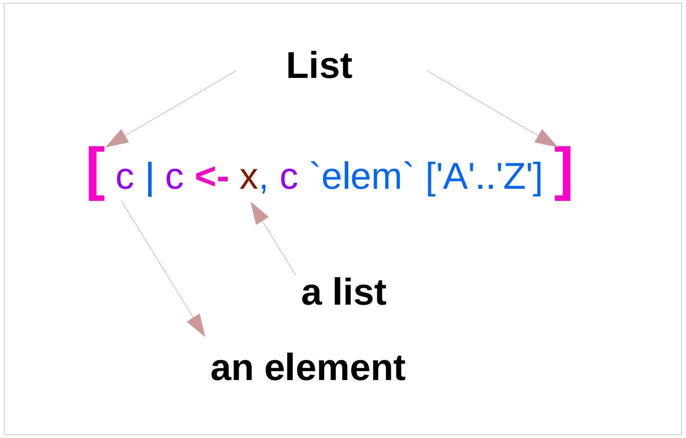
matching from the **elements** of the **list x**

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

# Equal = vs. Left Arrow <-

**let x = readFile file1**

This takes the action "readFile file1"
and stores the action in x.
**x** is an unexecuted I/O action object.

**x <- readFile file1**

This **executes** the action "readFile file1"
and stores the **result** of the action in x.
**x** is the contents of a file on disk.

**let x = action**

defines **x** to be equivalent to action,
but does not run anything.
Later on, you can use
y <- x meaning y <- action.

**x <- action**

runs the IO action,
gets its result,
and binds it to **x**

https://stackoverflow.com/questions/28624408/equal-vs-left-arrow-symbols-in-haskell

# Binding the execution result of actions

**x <- action**

stateful computation **x**

runs the IO action,
gets its result,
and binds it to **x**

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

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

```
x >>= return
```

**c** gets the result of the execution of the action **x**

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

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

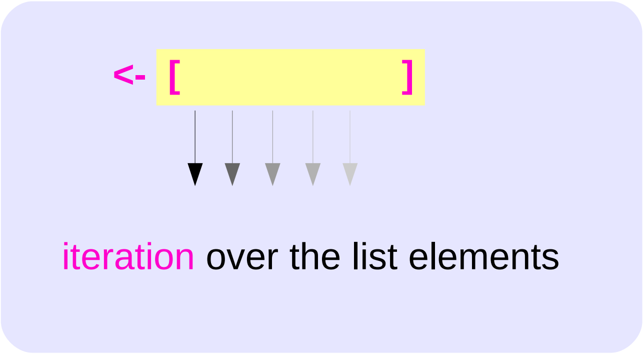
# Generator

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

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

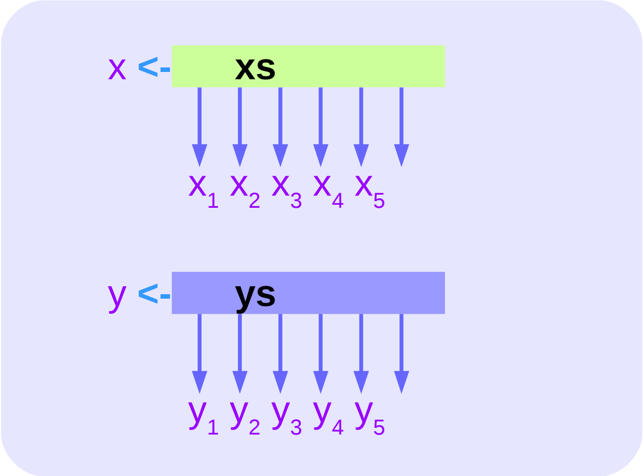
```
[ c | c <- x ]
```

c: an element
x: a list




```
pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys = do x <- xs
                 y <- ys
                 return (x, y)
```

x, y : elements
xs, ys : lists




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

# **do** Statements (1)

**exp** -> **do {** *stmts* **}** (do expression)

*stmts* -> $stmt_1$ **...** $stmt_n$ **exp [;]** (n>=0)

*stmts* -> **exp ;**
| **pat <- exp ;**
| **let decls ;**
| **;** (empty statement)

https://www.haskell.org/onlinereport/exps.html#sect3.11

# **do** Statements (2)

A do expression provides
a more <u>conventional</u> syntax

```
do  putStr "x: "
    l <- getLine
    return (words l)
```

monadic way

```
putStr "x: "   >>
getLine        >>= \l ->
return (words l)
```

https://www.haskell.org/onlinereport/exps.html#sect3.11

# **do** Statements (3)

Do expressions satisfy these identities,
which may be used as a translation into the kernel,
after eliminating empty stmts:

```
do {e}                   =    e
do {e; stmts}            =    e >> do {stmts}
do {p <- e; stmts}       =    let   ok p  = do {stmts}
                                    ok _  =  fail "..."
                                    in e >>= ok
do {let decls; stmts}    =    let decls in do {stmts}
```

The ellipsis "..." stands for a compiler-generated error message,
passed to fail, preferably giving some indication of the location of the pattern-match failure;
the functions >>, >>=, and fail are operations in the class Monad,
as defined in the Prelude; and **ok** is a fresh identifier.

https://www.haskell.org/onlinereport/exps.html#sect3.11

# **Then** Operator (**>>**) and **do** Statements

a chain of actions

to *sequence* input / output operations

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

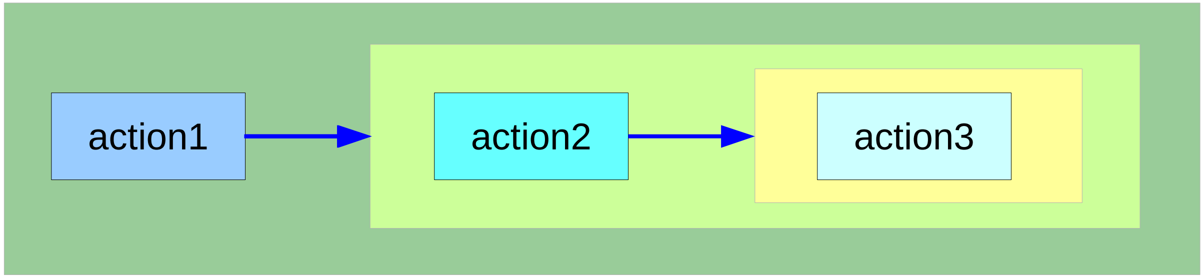
```
putStr "Hello"  >>
putStr " "      >>
putStr "world!" >>
putStr "\n"
```

```
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.

**do** notation assigns a variable name

to the passed value using the **<-**

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

anonymous function

(lambda expression)

is used

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

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

# Chaining >>= and do notations

->

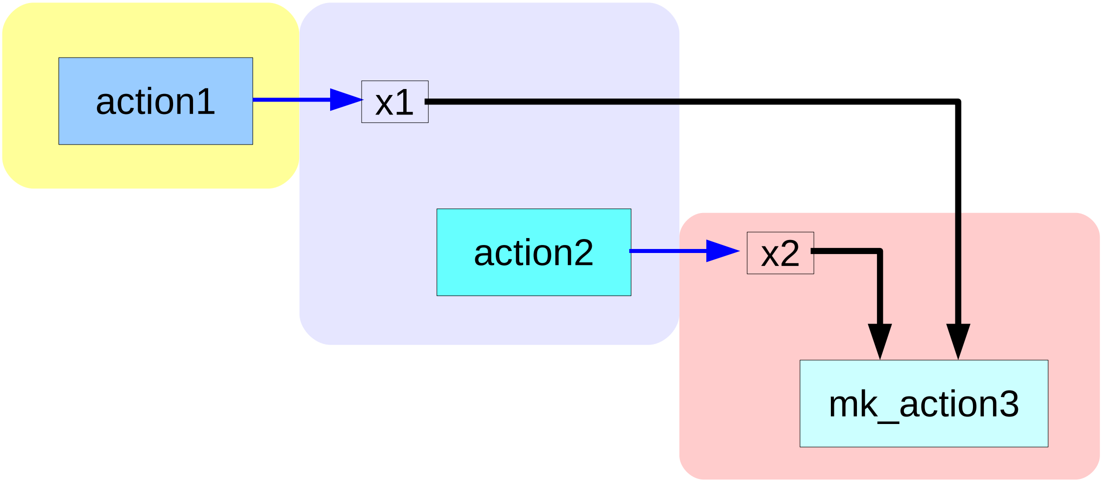
<-

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

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

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

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

```haskell
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
```

```haskell
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

# $ Operator

$ operator to avoid parentheses
anything appearing after $
will take precedence over anything that comes before.

B **$** A
higher precedence A

**putStrLn (show (1 + 1))**

**putStrLn (show $ 1 + 1)** (1+1) is the single argument to **show**
**putStrLn $ show $ 1 + 1** (show $ 1+1) is the single argument to **putStrLn**

**putStrLn $ show (1 + 1)** show (1+1) is the single argument to **putStrLn**
**putStrLn $ show $ 1 + 1** (1+1) is the single argument to **show**

https://stackoverflow.com/questions/940382/haskell-difference-between-dot-and-dollar-sign

# ( . ) Operator

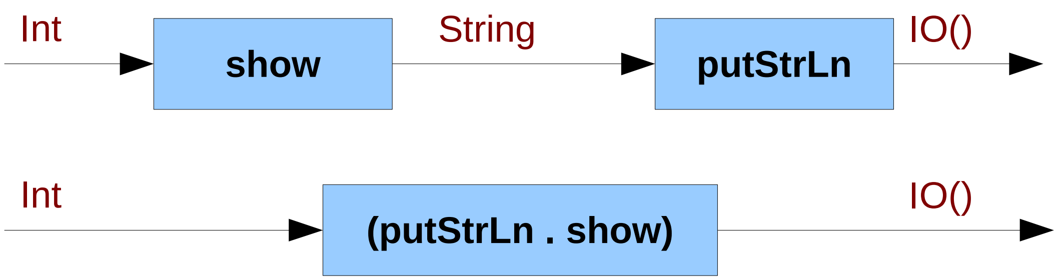
. operator to chain functions

**putStrLn (show (1 + 1))**

**show** can take an Int and return a String.
**putStrLn** can take a String and return an IO().

**(putStrLn . show) (1 + 1)**

**putStrLn . show $ 1 + 1**




(1 + 1) is not a function,
so the **.** operator cannot be applied

https://stackoverflow.com/questions/940382/haskell-difference-between-dot-and-dollar-sign

# **($)** vs **(.)** Operators

**($)** calls the function which is its left-hand argument on the value which is its right-hand argument.

**left_func $ right_value**

**( . )** composes the function which is its left-hand argument on the function which is its right-hand argument.

**left_func . right_func**

https://stackoverflow.com/questions/940382/haskell-difference-between-dot-and-dollar-sign

# ( . ) Operator

**( . )** : for a composite function

**result = (f . g) x**

is the same as building a function
that passes the result (**g x**)
of its argument **x** passed to **g** on to **f**.

```
h = \x -> f (g x)
result = h x
```

https://stackoverflow.com/questions/940382/haskell-difference-between-dot-and-dollar-sign

# **($)** calculates the right argument first

**($)** is intended to replace normal function application
but at a different precedence
to help avoid parentheses.

**($)** is a right-associative apply function
with low binding precedence.
So it merely calculates the things to the right of it first.

this matters because of Haskell's **lazy computation**,
**f** will be evaluated first

**result = f $ g x**

is the same as this,
procedurally **result = f (g x)**

**h = f**
**gx = g x**
**hgx = h gx**
**result = hgx**

**h = \x -> f (g x)**
**result = h x**

https://stackoverflow.com/questions/940382/haskell-difference-between-dot-and-dollar-sign

# **($)** operator as an identity function

Can consider **($)** as an identity function for function types.

**id :: a -> a**

**id x = x**

```
($) :: (a -> b) -> (a -> b)
($) = id
```

**– intentional parenthesis**

https://stackoverflow.com/questions/940382/haskell-difference-between-dot-and-dollar-sign

# Eliminating **($)** and **( . )**

**($)** can usually be eliminated
by adding parenthesis
(unless the operator is used in a section)
**f $ g x** ⟹ **f (g x)**.

**( . )** are often slightly harder to replace;
they usually need a lambda or the introduction
of an explicit function parameter.

**h = f . g** ⟹ **h x = (f . g) x** ⟹ **h x = f (g x)**

**h = \x -> f (g x)**
**result = h x**

https://stackoverflow.com/questions/940382/haskell-difference-between-dot-and-dollar-sign

# **($)** and **( . )** are operators

**($)** and **( . )** are not syntactic sugar for eliminating parentheses

- functions
- **infix**ed

thus we may call them operators.

```
infixr 9 .
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

```
infixr 0 $
($) :: (a -> b) -> a -> b
f $ x = f x
```

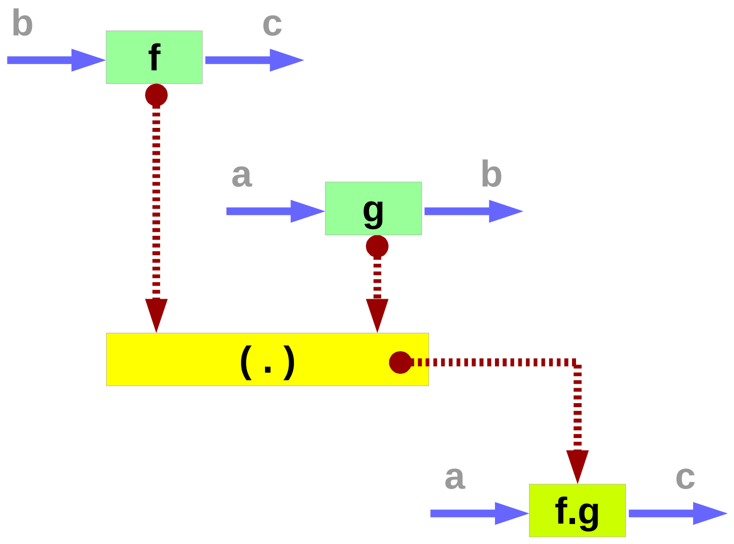
https://stackoverflow.com/questions/940382/haskell-difference-between-dot-and-dollar-sign

# ($) vs ( . ) Operator Types

**infixr 9 .**

**(.) :: (b -> c) -> (a -> b) -> (a -> c)**

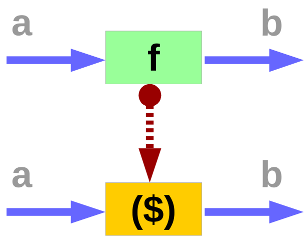
**(f . g) x = f (g x)**




**infixr 0 $**

**($) :: (a -> b) -> a -> b**

**f $ x = f x**




https://stackoverflow.com/questions/940382/haskell-difference-between-dot-and-dollar-sign

# Interchanging **($) vs ( . )** Operators

In some cases **($)** and **( . )** are interchangeable,
but this is not true in general.

**f $ g $ h $ x**

**f . g . h $ x**

a chain of **($)**s can be replaced by **( . )**
all but the last **($)**

https://stackoverflow.com/questions/940382/haskell-difference-between-dot-and-dollar-sign

# Fixity Declaration (1)

specifies a **precedence level** from 0 to 9

- with **9** being the strongest
- with **0** being the weakest
- normal application is assumed
  to have a precedence level of **10**

- **left-associativity (infixl)**
- **right-associativity (infixr)**
- **non-associativity (infix)**

http://zvon.org/other/haskell/Outputsyntax/fixityQdeclaration_reference.html

# Fixity Declaration (2)

**main = print (1 +++ 2 *** 3)**

**infixr 6 +++**
**infixr 7 ***,///**

**(+++) :: Int -> Int -> Int**
**a +++ b = a + 2*b**

**(***) :: Int -> Int -> Int**
**a *** b = a - 4*b**

**(///) :: Int -> Int -> Int**
**a /// b = 2*a - 3*b**

**(1 +++ 2 *** 3)**
**(1 +++ (2( *** 3)))**
**(1 +++ (2 - 4*3))**
**(1 +++ (-10))**
**1 – 20**
**-19**

http://zvon.org/other/haskell/Outputsyntax/fixityQdeclaration_reference.html

# Guard operator

**patterns** are a way of
making sure a **value** conforms to some **form**
and **deconstructing** it

**guards** are a way of
**testing** whether some **property** of a **value**
(or several of them) are **true** or **false**.

http://learnyouahaskell.com/syntax-in-functions

# !! operator

!! indexes lists

It takes a **list** and an **index**

and returns the **item** at that index

If the index is out of bounds, it returns ⊥

**:t (!!)**

**(!!) :: [a] -> Int -> a**

```
               0   1   2   3   4
Prelude> [11, 22, 33, 44, 55] !! 0
11
Prelude> [11, 22, 33, 44, 55] !! 10
*** Exception: Prelude.!!: index too large
Prelude> [11, 22, 33, 44, 55] !! 1
22
Prelude> [11, 22, 33, 44, 55] !! 4
55
```

http://learnyouahaskell.com/syntax-in-functions

## References

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