Haskell
High-Order Functions

http://igm.univ-mlv.fr/~vialette/?section=teaching

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

December 29, 2014

Curried functions

Every function in Haskell officially takes only one parameter.

A curried function is a function that, instead of taking several parameters, always takes exactly one parameter.

When it is called with that parameter, it returns a function that takes the next parameter, and so on.

Curried functions

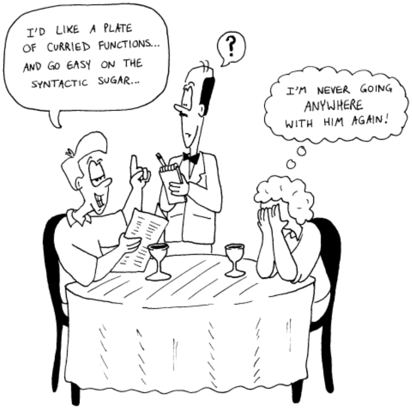
```
*Main> 1 + 2
3
*Main> :t +
<interactive>:1:1: parse error on input '+'
*Main> (+) 1 2
3
*Main> :t (+)
(+) :: Num a => a -> a -> a
*Main>
```

Curried functions

```
*Main> let add = (+)
*Main> :t add
add :: Num a => a -> a -> a
*Main> add 1 2
3
*Main> (add 1) 2
3
*Main> let add1 = add 1
*Main> :t add 1
add 1 :: Num a => a -> a
*Main> add1 2
3
*Main>
```

Curried functions

Whenever we have a type signature that features the arrow `->`, that means it is a function that takes whatever is on the left side of the arrow and returns a value whose type is indicated on the right side of the arrow.


I'D LIKE A PLATE OF CURRIED FUNCTIONS... AND GO EASY ON THE SYNTACTIC SUGAR...
?
I'M NEVER GOING ANYWHERE WITH HIM AGAIN!

Curried functions

When we have something like `a -> a -> a` (read `a -> (a -> a)`), we are dealing with a function that takes a value of type `a`, and it returns a function that also takes a value of type `a` and returns a value of type `a`.

Curried functions

```
*Main> let multThree x y z = x * y * z
*Main> :t multThree
multThree :: Num a => a -> a -> a -> a
*Main> let multTwoWithNine = multThree 9
*Main> :t multTwoWithNine
multTwoWithNine :: Num a => a -> a -> a
*Main> multTwoWithNine 2 3
54
*Main> let multWithNineAndFive = multTwoWithNine 5
*Main> :t multWithNineAndFive
multWithNineAndFive :: Num a => a -> a
*Main> multWithNineAndFive 2
90
*Main> multThree 2 5 9
90
*Main>
```

Curried functions

```
*Main> :t compare
compare :: Ord a => a -> a -> Ordering
*Main> :t (compare 100)
(compare 100) :: (Ord a, Num a) => a -> Ordering
*Main> let compareWithHundred x = compare 100 x
*Main> compareWithHundred 99
GT
*Main> :t compareWithHundred
compareWithHundred :: (Ord a, Num a) => a -> Ordering
*Main> let compareWithHundred' = compare 100
*Main> :t compareWithHundred'
compareWithHundred' :: (Ord a, Num a) => a -> Ordering
*Main> compareWithHundred' 99
GT
*Main>
```

Curried functions

```
*Main> let divideByTen = (/10)
*Main> :t divideByTen
divideByTen :: Fractional a => a -> a
*Main> divideByTen 200
20.0
*Main> (/ 10) 200
20.0
*Main> let isUpperAlphanum = (`elem` ['A'..'Z'])
*Main> :t isUpperAlphanum
isUpperAlphanum :: Char -> Bool
*Main> isUpperAlphanum 'k'
False
*Main> isUpperAlphanum 'K'
True
*Main>
```

Some Higher-Orderism Is in Order

In Haskell, function can take other functions as parameter, and as we have seen, they can also return functions as return value.

```haskell
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

`->` is naturally right-associative. Therefore, here parentheses are mandatory as `a -> a -> a -> a` is interpreted by Haskell as `a -> (a -> (a -> a))`.

Some Higher-Orderism Is in Order

```
*Main> applyTwice (+3) 10
16
*Main> (+3) ((+3) 10)
16
*Main> applyTwice (++ " HAHA") "HEY"
"HEY HAHA HAHA"
*Main> applyTwice ("HAHA " ++) "HEY"
"HAHA HAHA HEY"
*Main> let multThree x y z = x * y * z
       in applyTwice (multThree 2 2) 9
144
*Main> applyTwice (1:) [2]
[1,1,2]
*Main>
```

Some Higher-Orderism Is in Order

Implementing zipWith

zipWith takes a functionand two lists as parameters, and then joins the two lists by applying the function between corresponding elements (it's in the standard library).

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

Some Higher-Orderism Is in Order

Implementing `zipWith`

```
*Main> :t zipWith'
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
*Main> zipWith' (+) [1,2,3] [11,12,13]
[12,14,16]
*Main> zipWith' max [1,12,3] [11,2,13]
[11,12,13]
*Main> zipWith' (++) ["foo","bar"] ["fighther","hoppers"]
["foofighther","barhoppers"]
*Main> zipWith' (*) (replicate 5 2) [1..]
[2,4,6,8,10]
*Main> zipWith' (zipWith' (*)) [[1,2],[3,4]] [[5,6],[7,8]]
[[5,12],[21,32]]
*Main>
```

Some Higher-Orderism Is in Order

Implementing `flip`

`flip` takes a function and return a function that is like our original function, but with the first two arguments flipped (it's in the standard library).

```haskell
flip' :: (a -> b -> c) -> b -> a -> c
flip' f = g where g x y = f y x
```

Recall that the arrow `->` is right-associative, and hence
`(a -> b -> c) -> b -> a -> c` is the same as
`(a -> b -> c) -> (b -> a -> c)`.

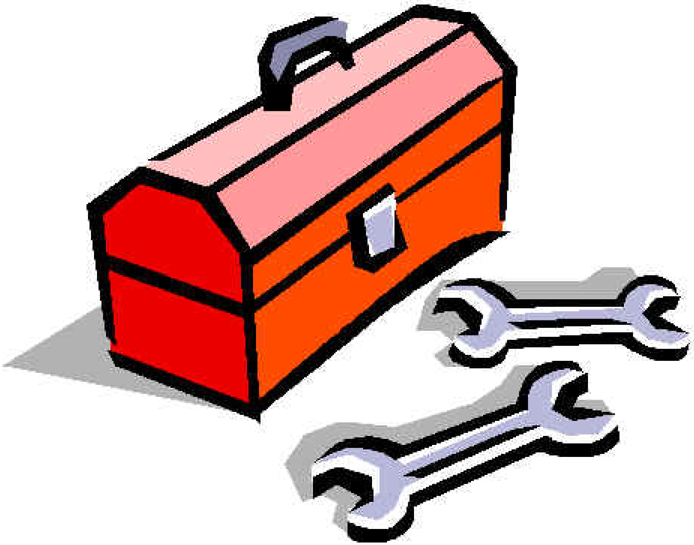
```haskell
flip' :: (a -> b -> c) -> b -> a -> c
flip' f x y = f y x
```

Some Higher-Orderism Is in Order

Implementing `flip`

```
*Main> zip [1..5] "hello"
[(1,'h'),(2,'e'),(3,'l'),(4,'l'),(5,'o')]
*Main> flip' zip [1..5] "hello"
[('h',1),('e',2),('l',3),('l',4),('o',5)]
*Main> zipWith div [2,2..] [10,8,6,4,2]
[0,0,0,0,1]
*Main> zipWith (flip' div) [2,2..] [10,8,6,4,2]
[5,4,3,2,1]
*Main>
```

The functionnal Programmer's Toolbox



The functionnal Proframmer’s Toolbox

The map function

The map function takes a function and a list, and applies that function to every element in the list, producing a new list.

```
map :: (a -> b) -> [a] -> [b]
map _ []      = []
map f (x:xs) = f x : map f xs
```

map is a versatile higher-order function that can be used in many different ways

The functionnal Proframmer's Toolbox

The map function

```
*Main> map (+1) [1,2,3,4,5]
[2,3,4,5,6]
*Main> map (++ "!") ["BIFF","BANG","POW"]
["BIFF!","BANG!","POW!"]
*Main> map (replicate 3) [1,2,3]
[[1,1,1],[2,2,2],[3,3,3]]
*Main> map (map (^2)) [[1,2],[3,4]]
[[1,4],[9,16]]
*Main> map fst [(1,2),(3,4),(5,6)]
[1,3,5]
*Main> map snd [(1,2),(3,4),(5,6)]
[2,4,6]
*Main>
```

The functionnal Proframmer's Toolbox

The filter function

The filter function takes a predicate and a list, and returns the list of elements that satify the predicate

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ []       = []
filer p (x:xs)
| p x         = x : filter p xs
| otherwise = filter p xs
```

If p x evaluates to True, the element is included in the new list. If it doesn't evaluate to True, it isn't included in the new list.

The functionnal Proframmer's Toolbox

The `filter` function

```
*Main> filter (> 3) [1,2,3,4,5,1,2,3,4,5]
[4,5,4,5]
*Main> filter (== 3) [1,2,3,4,5,1,2,3,4,5]
[3,3]
*Main> filter (< 3) [1,2,3,4,5,1,2,3,4,5]
[1,2,1,2]
*Main> filter even [1,2,3,4,5,1,2,3,4,5]
[2,4,2,4]
*Main> filter (`elem` ['a'..'z']) "I lOvE hAsKeLl"
"lvhsel"
*Main> filter (`elem` ['A'..'Z']) "I lOvE hAsKeLl"
"IOEAKL"
*Main>
```

The functionnal Proframmer's Toolbox

The filter function

The filter equivalent of applying several predicates in a list comprehension is either filtering something several times or joining predicates with the logical && function.

```
*Main> filter (< 15) (filter even [1..20])
[2,4,6,8,10,12,14]
*Main> let p x = x < 15 && even x in filter p [1..20]
[2,4,6,8,10,12,14]
*Main> filter (\x -> x < 15 && even x) [1..20]
[2,4,6,8,10,12,14]
*Main> [x | x <- [1..20], x < 15, even x]
[2,4,6,8,10,12,14]
*Main>
```

The functionnal Proframmer's Toolbox

The `filter` function

```haskell
quicksort :: (Ord a) => [a] -> [a]
quicksort []      = []
quicksort (x:xs) =
let smallerOrEqual = filter (<= x) xs
    larger         = filter (> x)  xs
in quicksort smallerOrEqual ++ [x] ++ quicksort larger
```

The functionnal Proframmer's Toolbox

More examples of `map` and `filter`

Let's find the largest number under 100 000 that is divisible by 3 829.

```haskell
largestDivisible :: Integer
largestDivisible = head (filter p [100000,99999..])
where p x = x `mod`3829 == 0
```

The functionnal Proframmer’s Toolbox

More examples of `map` and `filter`

Let’s find the sum of all odd squares that are smaller than 10 000.

```
*Main> sum (takeWhile (< 10000) (filter odd (map (^2) [1..])))
166650
*Main> sum (takeWhile (< 10000) [x | x <- [y^2 | y <- [1..]],
                                     odd x])
166650
*Main>
```

The functionnal Proframmer's Toolbox

More examples of `map` and `filter`

A *Collatz* sequence is defined as follows:

- Start with any natural number.
- If the number is 1, stop.
- If the number is even, divide it by 2.
- If the number id odd, multiply it by 3 and add 1.
- Repeat the algorithm with the resulting number.

Mathematicians theorize that for all starting number, the chain will finish at the number 1.

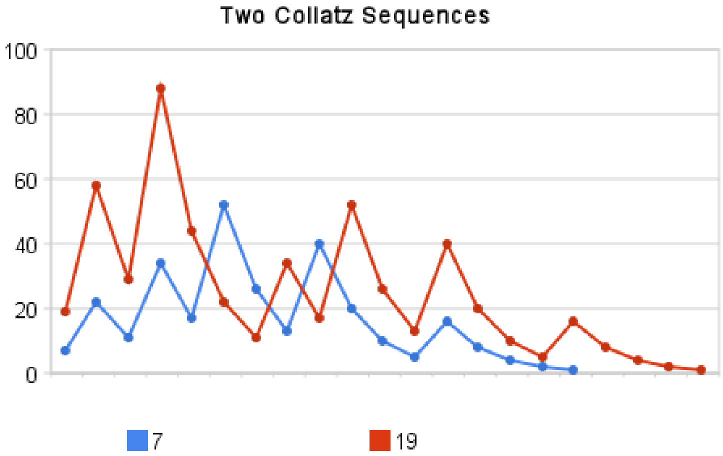
The functionnal Proframmer's Toolbox

More examples of `map` and `filter`

$$F(n) = \begin{cases} \frac{n}{2} & if\ n\%2 \ = 0 \\ \\ 3n+1 & if\ n\%2 \ = 1 \end{cases}$$

The functionnal Proframmer's Toolbox

More examples of `map` and `filter`



The functionnal Proframmer's Toolbox

More examples of `map` and `filter`

```haskell
collatz :: Integer -> [Integer]
collatz 1 = [1]
collatz n
| even n = n : collatz (n `div` 2)
| odd n  = n : collatz (n*3 + 1)
```

```
*Main> collatz 10
[10,5,16,8,4,2,1]
*Main> collatz 20
[20,10,5,16,8,4,2,1]
*Main> length $ collatz 100
26
*Main> length $ collatz 1000
112
*Main>
```

The functionnal Proframmer's Toolbox

Mapping functions with Multiple Parameters

```
*Main> let listOfFuns = map (*) [0..]
*Main> :t listOfFuns
listOfFuns :: (Num a, Enum a) => [a -> a]
*Main> (listOfFuns !! 0) 5
0
*Main> (listOfFuns !! 1) 5
5
*Main> (listOfFuns !! 2) 5
10
*Main> (listOfFuns !! 3) 5
15
*Main> (listOfFuns !! 4) 5
20
*Main>
```

Lambdas

Lambdas

Lambdas are anonymous fucntions that we use when we need a function only once.

Normally, we make a lambda with the sole purpose of passing it to a higer-order function.

To declare a lambda, we write `\` (because it kind of looks like the Greek letterlambda (λ) if you squint hard enough), and then we write the function's parameters, separated by spaces.

After that comes a `->`, and then the function body.

If a lambda match fails in a lambda, a runtime error occurs, so be careful!

Lambdas

```
*Main> map (+3) [1..5]
[4,5,6,7,8]
*Main> map (\x -> x + 3) [1..5]
[4,5,6,7,8]
*Main> zipWith (+) [1..5] [101..105]
[102,104,106,108,110]
*Main> zipWith (\x y -> x + y) [1..5] [101..105]
[102,104,106,108,110]
*Main> map (\(x,y) -> x + y) [(1,2),(3,4),(5,6)]
[3,7,11]
*Main>
```

Lambdas

The following functions are equivalent:

```haskell
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z

addThree' :: Int -> Int -> Int -> Int
addThree' = \x -> \y -> \z -> x + y + z
```

In the second example, the lambdas are not surroounded with parentheses. When you write a lambda without parentheses, it assumes that everything to the right of the arrow `->` belongs to it.

Lambdas

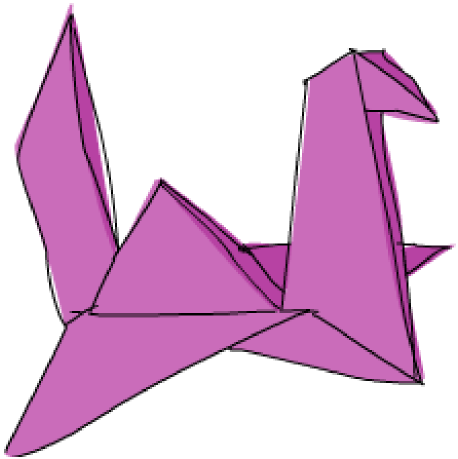
The following functions are equivalent:

```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f x y = f y x

flip'' :: (a -> b -> c) -> b -> a -> c
flip'' f = \x y -> f y x
```

In the second example, our new notation makes it obvious that this will often be used for producing a new function.

I fold you so



I fold you so

- Folds can be used to implement any function where you traverse a list once, element by element, and then return something based on that.
- A fold takes a *binary function* (one that takes two parameters, such as `+` or `div`), a starting value (often called the *accumulator*), and a list to fold up.
- Lists can folded up from the left or from the right.
- The fold function calls the given binary function, using the accumulator and the first (or last) element of the list as parameters. The resulting value is the new accumulator.
- The accumulator value (and hence the result) of a fold can be of any type.

Left fold with `foldl`

```
   :                     f
  / \                   / \
x1   :                     xn
    / \               /
  x2       xs        f   foldl f a xs
        \           / \
         :         f   x2
        / \       / \
      xn   []    a   x1
```

I fold you so

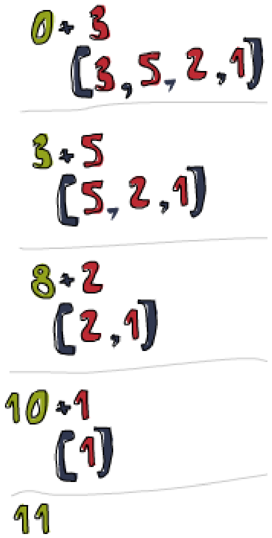
Left fold with `foldl`

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

```
*Main> sum' []
0
*Main> sum' [3,5,2,1]
11
*Main>
```

I fold you so

Left fold with `foldl`



I fold you so

Left fold with `foldl`

The lambda function `\acc x -> acc + x` is the same as `(+)`

```
sum'' :: (Num a) => [a] -> a
sum'' = foldl (+) 0
```

```
*Main> sum'' []
0
*Main> sum'' [3,5,2,1]
11
*Main>
```

A quick parenthesis

η-reduction

- An **eta conversion** (also written η-**conversion**) is adding or dropping of abstraction over a function.
- For example, the following two values are equivalent under η-conversion: `\x -> abs x` and `abs`.
- Converting from the first to the second would constitute an η-**reduction**, and moving from the second to the first would be an η-**abstraction**.
- The term η-**conversion** can refer to the process in either direction.
- Extensive use of η-reduction can lead to **Pointfree programming**.

A quick parenthesis

η-reduction

Therefore

```haskell
sum'' :: (Num a) => [a] -> a
sum'' xs = foldl (+) 0 xs
```

is usually rewritten as:

```haskell
sum'' :: (Num a) => [a] -> a
sum'' = foldl (+) 0
```

I fold you so

Left fold with `foldl`

```haskell
elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldl (\acc x -> if x == y then True else acc
```

```
Prelude> elem' 'a' ['a'..'l']
True
Prelude> elem' 'm' ['a'..'l']
False
Prelude> elem' (3, 9) [(i, i^2) | i <- [1..100]]
True
Prelude> elem' (4, 17) [(i, i^2) | i <- [1..100]]
False
Prelude>
```

I fold you so

Right fold with `foldr`

- The right fold function `foldr` is similar to the left fold, except that the accumulator eats up the values from the right.
- Also, the order of the parameters in the right fold's binary function is reversed: The current list value is the right parameter and the accumulator is the second.

Right fold with `foldr`

```
    :                      f
   / \                    / \
 x1    :                x1    f
      / \      xs            / \      foldr f z xs
    x2                     x2
          \                       \
           :                       f
          / \                     / \
        xn    []                xn    z
```

I fold you so

Right fold with `foldr`

```haskell
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (\x acc -> f x : acc) [] xs
```

```
*Main> map' (+ 10) []
[]
*Main> map' (+ 10) [1..5]
[11,12,13,14,15]
*Main>
```

I fold you so

Right fold with `foldr`

```haskell
map'  :: (a -> b) -> [a] -> [b]
map'  f = foldr (\x acc -> f x : acc) []

map'' :: (a -> b) -> [a] -> [b]
map'' f = foldl (\acc x -> acc ++ [f x]) []
```

Notice that the `++` function is much slower than `:`, so we usually use right fold when we are building up new lists from lists.

I fold you so

Right fold with foldr

The elem function checks chether a value is part of a list.

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' x = foldr (\y acc -> if x==y then True else acc) Fals
```

```
*Main> :t elem'
elem' :: Eq a => a -> [a] -> Bool
*Main> 5 `elem` [10..20]
False
*Main> 15 `elem` [10..20]
True
*Main>
```

I fold you so

The `foldl1` and `foldr1` functions

- The `foldl1` and `foldr1` functions work much like `foldl` and `foldr`, except that you don't need to provide them with an explicit starting accumulator.
- The `foldl1` and `foldr1` functions assume the first (or last) element of the list to be the starting accumulator, and then start the fold with the next element next to it.

I fold you so

The foldl1 and foldr1 functions

```
*Main> :t foldl1
foldl1 :: (a -> a -> a) -> [a] -> a
*Main> :t foldr1
foldr1 :: (a -> a -> a) -> [a] -> a
*Main>
```

I fold you so

The foldl1 and foldr1 functions

```haskell
minimum' :: (Ord a) => [a] -> a
minimum' = foldl1 min

maximum' :: (Ord a) => [a] -> a
maximum' = foldl1 max
```

```
*Main> :t minimum'
minimum' :: Ord a => [a] -> a
*Main> minimum' []
*** Exception: Prelude.foldl1: empty list
*Main> minimum' [1]
1
*Main> minimum' $ [10..20] ++ [1..10]
1
*Main>
```

I fold you so

Some fold examples

```haskell
reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

reverse'' :: [a] -> [a]
reverse'' = foldl (flip (:)) []
```

```
Main> reverse' []
[]
*Main> reverse' [1..5]
[5,4,3,2,1]
*Main> reverse'' []
[]
*Main> reverse'' [1..5]
[5,4,3,2,1]
*Main>
```

I fold you so

Some fold examples

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc)

last' :: [a] -> a
last' = foldl1 (\_ x -> x)

length' :: Num b => [a] -> b
length' = foldr (\_ -> (+1)) 0
```

I fold you so

Folding infinite lists

```haskell
and' :: [Bool] -> Bool
and' = foldr (&&) True
```

```
*Main> and' (repeat False)
False
*Main>
```

I fold you so

Scans

- The `scanl` and `scanr` functions are like `foldl` and `foldr`, except they report all the intermediate accumulator states in the form of a list.
- The `scanl1` and `scanr1` functions are analogous to `foldl1` and `foldr1`.

I fold you so

Scans

```
*Main> scanl (+) 0 [1,2,3,4]
[0,1,3,6,10]
*Main> scanr (+) 0 [1,2,3,4]
[10,9,7,4,0]
*Main> scanl1 (\acc x -> if x > acc then x else acc) [1..5]
[1,2,3,4,5]
*Main> scanl1 max [1..5]
[1,2,3,4,5]
*Main> scanl (flip (:)) [] [3,2,1]
[[],[3],[2,3],[1,2,3]]
*Main>
```

I fold you so

Function application with $

The *function application operator* $ is defined as follows:

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

I fold you so

Function application with $

What is this useless function? It is just function application! Well, that is almost true, but not quite.

Whereas normal function application (putting a space between two things) has a really high precedence, the $ function has the lowest precedence.

Function application with a space is left-associative (so `f a b c` is the same as `(((f a) b) c)`), while function application with $ is right-associative.

I fold you so

Function application with $

```
*Main> sum (filter (> 10) (map (*2) [2..10]))
80
*Main> sum $ filter (> 10) (map (*2) [2..10])
80
*Main> sum $ filter (> 10) $ map (*2) [2..10]
80
*Main>
```

I fold you so

Function application with $

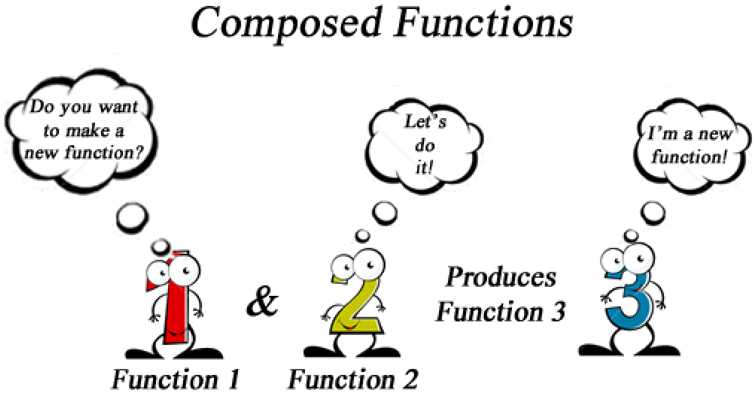
Apart of getting rid of parentheses, $ let us treat function application like just another function.

```
*Main> :t (4+)
(4+) :: Num a => a -> a
*Main> :t (^2)
(^2) :: Num a => a -> a
*Main> :t sqrt
sqrt :: Floating a => a -> a
*Main> :t [(4+),(^2),sqrt]
[(4+),(^2),sqrt] :: Floating a => [a -> a]
*Main> map ($ 3) [(4+),(^2),sqrt]
[7.0,9.0,1.7320508075688772]
*Main>
```

Function composition

In mathematics, *function composition* is defined as follows:

$$(f \circ g)(x) = f(g(x))$$



Function composition

In Haskell, function composition is pretty much the same thing.

We do function composition with the `.` function:

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

Function composition

```
*Main> :t negate
negate :: Num a => a -> a
*Main> :t abs
abs :: Num a => a -> a
*Main> map (\x -> negate(abs x)) [1,-2,3,-4,5,-6]
[-1,-2,-3,-4,-5,-6]
*Main> map (negate . abs) [1,-2,3,-4,5,-6]
[-1,-2,-3,-4,-5,-6]
*Main>
```

Function composition

```
*Main> map (\xs -> negate (sum (tail xs))) [[1..5],[3..6]]
[-14,-15]
*Main> map (negate . sum .tail) [[1..5],[3..6]]
[-14,-15]
*Main>
```

`negate . sum .tail` is a function that takes a list, applies the `tail` function to it, then applies the `sum` function to the result, and finally applies `negate` to the previous result.

Function composition

Function Composition with Multiple Parameters

But what about functions that take several parameters?

Well, if we want to use them in function composition, we usually must partially apply them so that each function takes just one parameter.

```
*Main> sum (replicate 5 (max 6.7 8.9))
44.5
*Main> (sum .replicate 5) (max 6.7 8.9)
44.5
*Main> sum .replicate 5 $ max 6.7 8.9
44.5
*Main> replicate 2 (product (map (*3) (zipWith max [1,2] [4,5]))
[180,180]
*Main> replicate 2 . product . map (*3) $ zipWith max [1,2] [4,
[180,180]
*Main>
```

Function composition

Point-Free Style

Another common use of function compisition is defining function in the *point-free style*.

```
f :: (RealFrac a, Integral b, Floating a) => a -> b
f x = ceiling (negate (tan (cos (max 50 x))))

f' :: (RealFrac a, Integral b, Floating a) => a -> b
f' = ceiling . negate . tan . cos . max 50
```

Function composition

Point-Free Style

```haskell
oddSquareSum :: Integer
oddSquareSum = sum (takeWhile (<10000) (filter odd (map (^2) [1..])))

oddSquareSum' :: Integer
oddSquareSum' = sum . takeWhile (<10000) . filter odd . map (^2) $ [1..]

oddSquareSum'' :: Integer
oddSquareSum'' =
    let oddSquares = filter odd $ map (^2) [1..]
        belowLimit = takeWhile (<10000) oddSquares
    in  sum belowLimit
```