Haskell
High-Order Functions

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

Stéphane Vialette

LIGM, Université Gustave Eiffel

November 22, 2022

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

```
λ: 1 + 2
3
λ: :type +
<interactive>:1:1: parse error on input '+'
λ: (+) 1 2
3
λ: :type (+)
(+) :: Num a => a -> a -> a
```

Curried functions

```
λ:  add = (+)
λ:  :type add
add :: Num a => a -> a -> a
λ:  add 1 2
3

λ:  (add 1) 2
3
λ:  add1 = add 1
λ:  :type add1
add1 :: Num a => a -> a
λ:  add1 2
3
```

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.

Curried functions

When we have something like `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`.

In other words `a -> a -> a` reads as `a -> (a -> a)`.

Curried functions

```
λ: multThree x y z = x * y * z
λ: :type multThree
multThree :: Num a => a -> a -> a -> a

λ: multTwoWithNine = multThree 9
λ: :type multTwoWithNine
multTwoWithNine :: Num a => a -> a -> a
λ: multTwoWithNine 2 3
54

λ: multWithNineAndFive = multTwoWithNine 5
λ: :type multWithNineAndFive
multWithNineAndFive :: Num a => a -> a
λ: multWithNineAndFive 2
90
λ: multThree 2 5 9
90
```

Curried functions

```
λ: :type compare
compare :: Ord a => a -> a -> Ordering
λ: :type (compare 100)
(compare 100) :: (Ord a, Num a) => a -> Ordering

λ: compareWithHundred x = compare 100 x
λ: compareWithHundred 99
GT
λ: :type compareWithHundred
compareWithHundred :: (Ord a, Num a) => a -> Ordering

λ: compareWithHundred' = compare 100
λ: :type compareWithHundred'
compareWithHundred' :: (Ord a, Num a) => a -> Ordering
λ: compareWithHundred' 99
GT
```

Curried functions

```
λ: divideByTen = (/10)
λ: :type divideByTen
divideByTen :: Fractional a => a -> a
λ: divideByTen 200
20.0
λ: (/ 10) 200
20.0

λ: isUpperAlphanum = (`elem` ['A'..'Z'])
λ: :type isUpperAlphanum
isUpperAlphanum :: Char -> Bool
λ: isUpperAlphanum 'k'
False
λ: isUpperAlphanum 'K'
True
```

curry

```
λ: :type curry
curry :: ((a, b) -> c) -> a -> b -> c

λ: f (xs, ys) = xs ++ ys
λ: :type f
f :: ([a], [a]) -> [a]
λ: f ("aa", "zz")
"aazz"
λ: curry f "aa" "zz"
"aazz"
```

uncurry

```
λ: :type uncurry
uncurry :: (a -> b -> c) -> (a, b) -> c

λ: f xs ys = xs ++ ys
λ: :type f
f :: [a] -> [a] -> [a]
λ: f "aa" "zz"
"aazz"
λ: uncurry f ("aa", "zz")
"aazz"
λ: uncurry (++) ("aa", "zz")
"aazz"
```

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.

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

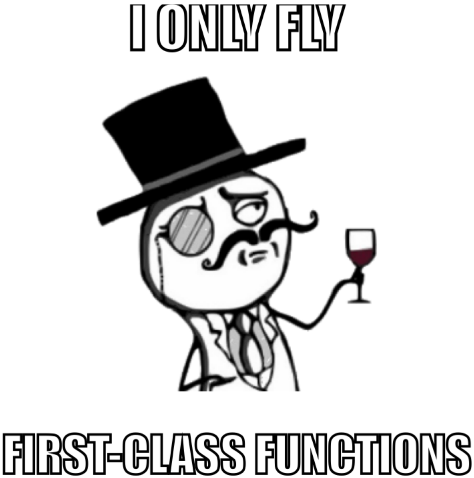
`->` is naturally right-associative. Therefore, here parentheses are mandatory as `a -> a -> a -> a` is interpreted by Haskell as `a -> (a -> (a -> a))`.

Some Higher-Orderism Is in Order

```
λ: applyTwice f x = f (f x)
λ: :type applyTwice
applyTwice :: (t -> t) -> t -> t

λ: applyTwice (+3) 10
16
λ: (+3) ((+3) 10)
16
λ: applyTwice (++ " HAHA") "HEY"
"HEY HAHA HAHA"
λ: applyTwice ("HAHA " ++) "HEY"
"HAHA HAHA HEY"
λ: let mult3 x y z = x * y * z in applyTwice (mult3 2 2) 9
144
λ: applyTwice (1:) [2]
[1,1,2]
```

First-class and higher-order functions



Some Higher-Orderism Is in Order

Implementing zipWith

zipWith takes a function and 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

```
λ:  :type zipWith'
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]

λ:  zipWith' (+) [1,2,3] [11,12,13]
[12,14,16]
λ:  zipWith' max [1,12,3] [11,2,13]
[11,12,13]
λ:  zipWith' (++) ["foo","bar"] ["fighther","hoppers"]
["foofighther","barhoppers"]
λ:  zipWith' (*) (replicate 5 2) [1..]
[2,4,6,8,10]
λ:  zipWith' (zipWith' (*)) [[1,2],[3,4]] [[5,6],[7,8]]
[[5,12],[21,32]]
```

Some Higher-Orderism Is in Order

Implementing `flip`

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

```
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)`.

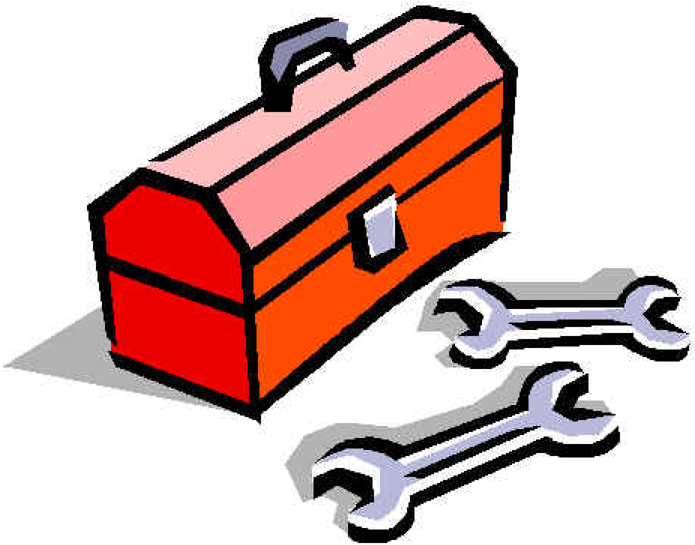
```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f x y = f y x
```

Some Higher-Orderism Is in Order

Implementing `flip`

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

The functionnal Programmer's Toolbox



The functionnal Programmer'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 Programmer's Toolbox

The map function

```
λ: map (+ 1) [1,2,3,4,5]
[2,3,4,5,6]
λ: map (++ "!") ["BIFF","BANG","POW"]
["BIFF!","BANG!","POW!"]
λ: map (replicate 3) [1,2,3]
[[1,1,1],[2,2,2],[3,3,3]]
λ: map (map (^2)) [[1,2],[3,4]]
[[1,4],[9,16]]
λ: map fst [(1,2),(3,4),(5,6)]
[1,3,5]
λ: map snd [(1,2),(3,4),(5,6)]
[2,4,6]
λ: map (map (+1)) [[1..4], [6..9]]
[[2,3,4,5],[7,8,9,10]]
```

The functionnal Programmer'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 Programmer's Toolbox

The filter function

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

The functionnal Programmer'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.

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

The functionnal Programmer'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 Programmer's Toolbox

More examples of `map` and `filter`

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

The functionnal Programmer's Toolbox

More examples of map and filter

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

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

The functionnal Programmer's Toolbox

More examples of `map` and `filter`

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

The functionnal Programmer's Toolbox

More examples of `map` and `filter`

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

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

The functionnal Programmer'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 is 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.

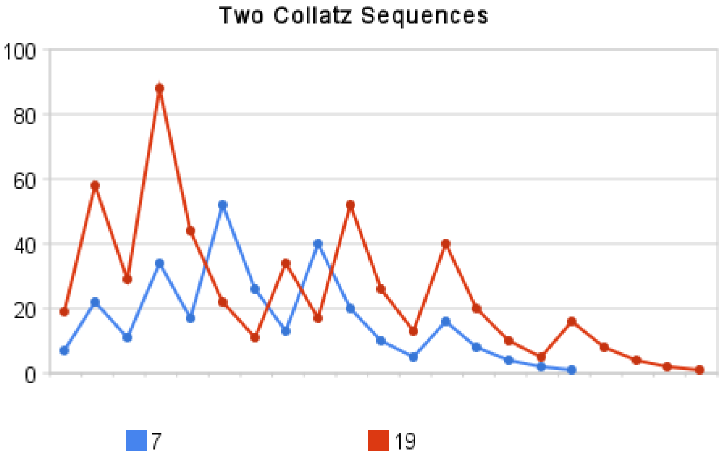
The functionnal Programmer's Toolbox

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 Programmer's Toolbox

More examples of `map` and `filter`



The functionnal Programmer's Toolbox

More examples of map and filter

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

λ: collatz 10
[10,5,16,8,4,2,1]
λ: collatz 20
[20,10,5,16,8,4,2,1]
λ: length $ collatz 100
26
λ: length $ collatz 1000
112
```

The functionnal Programmer's Toolbox

Mapping functions with Multiple Parameters

```
λ: listOfFuns = map (*) [0..]
λ: :type listOfFuns
listOfFuns :: (Num a, Enum a) => [a -> a]

λ: take 10 $ zipWith (\ f x -> f x) listOfFuns (cycle [5])
[0,5,10,15,20,25,30,35,40,45]
λ: take 10 $ zipWith ($) listOfFuns (cycle [5])
[0,5,10,15,20,25,30,35,40,45]
```

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

```
λ: (\ x -> x+1) 3
4
λ: (\ x y -> x + y) 3 5
8

λ: addOne = \ x -> x + 1
λ: addOne 3
4

λ: mapAddOne xs = map (\ x -> x + 1) xs
λ: :type mapAddOne
addOneL :: Num a => [a] -> [a]
λ: mapAddOne [1..4]
[2,3,4,5]
```

Lambdas

```
λ: map (+3) [1..5]
[4,5,6,7,8]
λ: map (\ x -> x + 3) [1..5]
[4,5,6,7,8]

λ: zipWith (+) [1..5] [101..105]
[102,104,106,108,110]
λ: zipWith (\ x y -> x + y) [1..5] [101..105]
[102,104,106,108,110]

λ: map (\ (x,y) -> x + y) [(1,2),(3,4),(5,6)]
[3,7,11]
```

Lambdas

Lambdas

The following functions are equivalent:

```
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 surrounded 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.

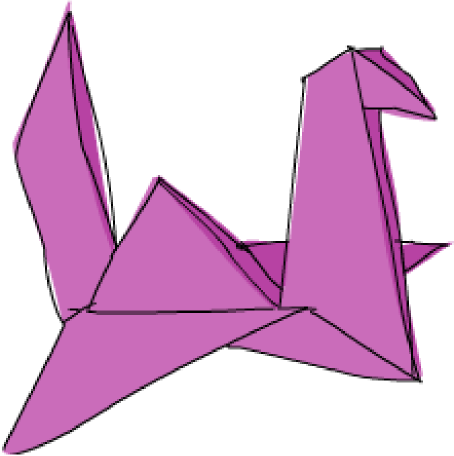
Lambdas

```
λ: fs = [\ x -> i * x | i <- [0,2..]]
λ: :type fs
fs :: (Num a, Enum a) => [a -> a]

λ: take 5 $ map (\ f -> f 10) fs
[0,20,40,60,80]

λ: take 5 $ map ($ 10) fs
[0,20,40,60,80]
```

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
```

Lambdas

I fold you so

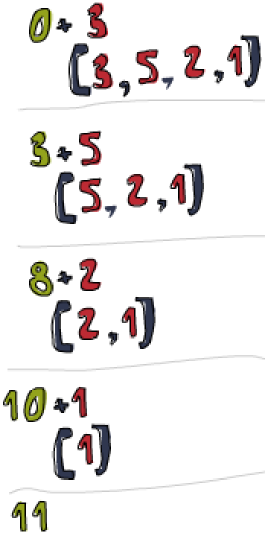
Left fold with foldl

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

λ: sum' []
0
λ: sum' [3,5,2,1]
11
```

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

λ: sum'' []
0
λ: sum'' [3,5,2,1]
11
```

I fold you so

Left fold with foldl

```
foldl (*) 1 [1..3]
  = foldl (*) (1 * 1)             (2::3::[])
  = foldl (*) ((1 * 1) * 2)       (3::[])
  = foldl (*) (((1 * 1) * 2) * 3) ([])
  = (((1 * 1) * 2) * 3)
  = 6
```

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

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

is usually rewritten as:

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

I fold you so

Left fold with foldl

```
elem' :: (Foldable t, Eq a) => a -> t a -> Bool
elem' x = foldl (\ acc y -> x == y || acc) False

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

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`

```
      :
     / \
   x1   :
       / \      xs
     x2
          \
           :
          / \
        xn    []
```

```
      f
     / \
   x1   f
       / \      foldr f z xs
     x2
          \
           f
          / \
        xn    z
```

I fold you so

Right fold with `foldr`

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

λ: map' (+ 10) []
[]
λ: map' (+ 10) [1..5]
[11,12,13,14,15]
```

I fold you so

Right fold with foldr

```
L.foldr (*) [1..3] 1
  = 1 *           L.foldr (*) (2::3::[]) 1
  = 1 * (2 *      L.foldr (*) (3::[]) 1)
  = 1 * (2 * (3 * L.foldr (*) ([]) 1))
  = 1 * (2 * (3 * 1))
  = 6
```

I fold you so

Right fold with `foldr`

```
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 L.

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' x = foldr (\ y acc -> x == y || acc) False

λ: :type elem'
elem' :: Eq a => a -> [a] -> Bool
λ: 5 `elem` [10..20]
False
λ: 15 `elem` [10..20]
True
```

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

```
λ: :type foldl1
foldl1 :: (a -> a -> a) -> [a] -> a
λ: :type foldr1
foldr1 :: (a -> a -> a) -> [a] -> a
```

I fold you so

The foldl1 and foldr1 functions

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

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

λ: :type minimum'
minimum' :: Ord a => [a] -> a
λ: minimum' []
*** Exception: Prelude.foldl1: empty list
λ: minimum' [1]
1
λ: minimum' ([10..20] ++ [1..10])
1
```

I fold you so

Some fold examples

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

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

```
λ: reverse' []
[]
λ: reverse' [1..5]
[5,4,3,2,1]

λ: reverse'' []
[]
λ: reverse'' [1..5]
[5,4,3,2,1]
```

I fold you so

Some fold examples

```haskell
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

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

λ: and' (repeat False)
False
```

`foldl` versus `foldr` behavior with infinite lists

How folds differ seems to be a frequent source of confusion, so here's a more general overview:

Consider folding a list of *n* values `[x1, x2, x3, x4 ... xn ]` with some function `f` and seed `z`.

`foldl` versus `foldr` behavior with infinite lists

How folds differ seems to be a frequent source of confusion, so here's a more general overview:

Consider folding a list of *n* values `[x1, x2, x3, x4 ... xn ]` with some function `f` and seed `z`.

`foldl` is:

- **Left associative**:
  `f ( ... (f (f (f (f z x1) x2) x3) x4) ... ) xn`.
- **Tail recursive**: It iterates through the list, producing the value afterwards.
- **Lazy**: Nothing is evaluated until the result is needed.
- **Backwards**: `foldl (flip (:)) []` reverses a list.

`foldl` versus `foldr` behavior with infinite lists

How folds differ seems to be a frequent source of confusion, so here's a more general overview:

Consider folding a list of *n* values `[x1, x2, x3, x4 ... xn ]` with some function `f` and seed `z`.

`foldr` is:

- **Right associative**:
  `f x1 (f x2 (f x3 (f x4 ...(f xn z) ... )))`.
- **Recursive into an argument**: Each iteration applies `f` to the next value and the result of folding the rest of the list.
- **Lazy**: Nothing is evaluated until the result is needed.
- **Forwards**: `foldr (:) []` returns a list unchanged.

- 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

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

I fold you so

Function application with $

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

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

I fold you so

Function application with $

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 $

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

I fold you so

Function application with $

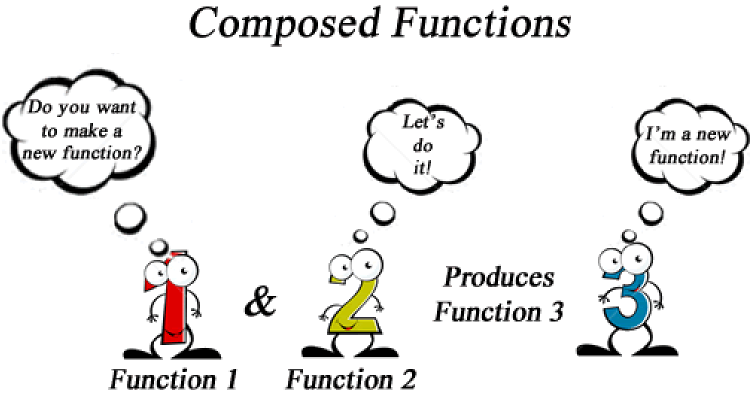
Apart of getting rid of parentheses, $ let us treat function application like just another function.

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

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

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

Function composition

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

`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.

```
λ: sum (replicate 5 (max 6.7 8.9))
44.5
λ: (sum . replicate 5) (max 6.7 8.9)
44.5
λ: sum . replicate 5 $ max 6.7 8.9
44.5
[180,180]
```

```
λ: replicate 2 (product (map (*3) (zipWith max [1,2] [4,5])))
[180,180]
λ: replicate 2 . product . map (*3) $ zipWith max [1,2] [4,5]
[180,180]
```

Function composition

Point-Free Style

Another common use of function composition 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

```
oddSSum :: Integer
oddSSum = sum (takeWhile (<100) (filter odd (map (^2) [1..])))

oddSSum' :: Integer
oddSSum' = sum . takeWhile (<100) . filter odd . map (^2) $ [1..

oddSSum'' :: Integer
oddSSum'' = sum belowLimit
  where
    oddSs      = filter odd $ map (^2) [1..]
    belowLimit = takeWhile (<100) oddSs
```

Origami programming

```haskell
sum :: (Foldable t, Num a) => t a -> a
```

```haskell
sum :: (Foldable t, Num a) => t a -> a
sum xs = foldl (\ acc x -> acc + x) 0 xs
```

```haskell
sum :: (Foldable t, Num a) => t a -> a
sum = foldl (+) 0
```

```haskell
sum :: (Foldable t, Num a) => t a -> a
sum xs = foldr (\ x acc -> acc + x) 0 xs
```

```haskell
sum :: (Foldable t, Num a) => t a -> a
sum = foldlr (+) 0
```

Origami programming

```
prod :: (Foldable t, Num a) => t a -> a

prod :: (Foldable t, Num a) => t a -> a
prod xs = foldl (\ acc x -> acc * x) 1 xs

prod :: (Foldable t, Num a) => t a -> a
prod = foldl (*) 1

prod :: (Foldable t, Num a) => t a -> a
prod xs = foldr (\ x acc -> acc * x) 1 xs

prod :: (Foldable t, Num a) => t a -> a
prod = foldlr (*) 1
```

Origami programming

```haskell
reverse :: Foldable t => t a -> [a]

reverse :: Foldable t => t a -> [a]
reverse = foldl (\ acc x -> x : acc) []

reverse :: Foldable t => t a -> [a]
reverse = foldl (flip (:)) []

reverse :: Foldable t => t a -> [a]
reverse = foldr (\ x acc -> acc ++ [x]) []

reverse :: Foldable t => t a -> [a]
reverse xs = foldr f id xs [] where f x acc = acc . (x :)
```

Origami programming

```haskell
append :: [a] -> [a] -> [a]

append :: [a] -> [a] -> [a]
append xs ys = foldl (flip (:)) ys $ reverse xs

append :: Foldable t => t a -> [a] -> [a]
append = flip (foldr (:))
```

Origami programming

```
concat :: Foldable t => t [a] -> [a]
```

```
concat :: Foldable t => t [a] -> [a]
concat = foldr (++) []
```

Origami programming

```
map :: (a -> b) -> [a] -> [b]

map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]

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