

Higher order functions : take fns. as parameters and return functions as return values.

↓

They are very important to the way computations are done in Haskell , i.e . by defining things rather than changing states of objects .

Curried functions → Every function officially takes one parameter .

All functions that accepted several parameters have been curried functions .

↓

$\text{max } 4 \ 5$  : creates a fn. that takes a parameter and returns either 4 or that parameter , depending on which is bigger . Then 5 is applied to this function , and the desired result is produced

↓

$(\text{max } 4)5 \equiv \text{max } 4 \ 5$

:t max

Space between two things is function application .

$\text{max} :: (\text{Ord} a) \Rightarrow a \rightarrow a \rightarrow a$

$\text{max} :: (\text{Ord} a) \Rightarrow a \rightarrow (a \rightarrow a)$

↓

max takes an a and returns a function that takes an a and returns an a .

↓

If we call a function with too few parameters , we get back a partially applied function , i.e . a fn. that takes addn. parameters as many we left out to run .

↓  
more like an operator with the highest precedence .

This is a neat way to create functions on the fly, so we can pass them to another fn. or to seed them with some data.

↓

multThree :: (Num a)  $\Rightarrow a \rightarrow a \rightarrow a \rightarrow a$

multThree  $x \cdot y \cdot z = x * y * z$

↓

((multThree 3) 5) 9 : first 3 is applied to multThree, returns a partially applied fn, which now accepts two parameters, gets 5, returns a partially applied fn, which now accepts one parameter, and multiplies the parameter with 15; and then applies it with 9. The function is now completely applied and returns 135.

↓

multThree :: (Num a)  $\Rightarrow a \rightarrow (a \rightarrow (a \rightarrow a))$

↓                    ↓  
parameter        returns a fn. that

$(a \rightarrow a) \rightarrow a \rightarrow a$

takes a parameter  
and returns a fn. that  
takes a parameter and  
returns a value

this type of type declaration  
is necessary for higher order  
functions.

function parameter  
↑                    ↑

apply Twice ::  $(a \rightarrow a) \rightarrow a \rightarrow a \rightarrow$  return type

apply Twice  $f \cdot x = f(f(x))$

↓                    ←  
                    returns a normal parameter  
                    which is used by the function  
                    again.

↓

i.e. we use the parameter  $f$  as a function,  
applying  $x$  to it by separating them  
with a space.

↓

$\text{applyTwice } (3 :) [1] \rightarrow [3, 3, 1]$

↓      ↓  
function   argument

•) Implementing `zipWith` (part of Haskell standard library )  
↓

takes a fn. and two lists as parameters and then  
joins the lists by applying the fn. between  
corresponding elements.

$\text{zipWith}' :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$

$\text{zipWith}' - [] - = []$

$\text{zipWith}' - - [ ] = [ ]$

$\text{zipWith}' f (x:xs) (y:ys) = (f x y) : \text{zipWith}' f xs ys$

•) Implement `flip` (takes a fn. , and returns a fn. that is  
like our original fn, only the first two arguments  
are flipped)

$\text{flip}' :: (a \rightarrow b \rightarrow c) \rightarrow (b \rightarrow a \rightarrow c)$

$\text{flip}' f = g$

where  $g x y = f y x$

$$\text{flip}' f x y = f y x$$

→ Maps and filters : map takes a fn. and a list and applies that function to every element in the list, producing a new list

$$\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

$$\text{map } [] = []$$

$$\text{map } f (x : xs) = f x : \text{map } f xs$$

filter is a fn. that takes a predicate and a list , and then returns a list that satisfy that predicate

$$\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$$

$$\text{filter } [] = []$$

$$\text{filter } f (x : xs)$$

$$| f x = x : \text{filter } f xs$$

$$| \text{otherwise} = \text{filter } f xs$$

⇒ quicksort using filter

$$\text{quicksort} :: (\text{Ord } a) \Rightarrow [a] \rightarrow [a]$$

$$\text{quicksort } [] = []$$

$$\text{quicksort } (x : xs) =$$

$$\text{let smallsorted} = \text{quicksort } \$ \text{filter } (<= x) xs$$

$$\text{largesorted} = \text{quicksort } \$ \text{filter } (> x) xs$$

in smallsorted ++ [x] ++ largesorted.

Mapping & filtering is  
the bread and butter  
of every functional  
programmer's toolbox



doesn't matter how  
it is accomplished .

(\*) find the largest no. under 100,000 that's divisible by 3829

Largest Divisible  $\therefore$  (Integral a)  $\Rightarrow$  a

Largest Divisible =  $\text{head}(\text{filter } [100000, 99999, \dots])$

↓  
largest number

where  $P X = X \text{ `mod' } 3829 = 0$

↓  
laziness acts  
here, evaluation

stops once the  
suitable head is  
found.

(\*) takeWhile function : takes a predicate and a list and then goes from the beginning of the list and returns the elements while the predicate holds true.

↓  
once an element is found, for which the predicate doesn't hold, it stops.

takeWhile ( $/= 11$ ) "elephants know how to party"

(\*) sum of all odd squares that are smaller than 10000.

sum \$ takeWhile (< 10000) \$ filter odd \$ map (^2) [1..]

↓ can also be accomplished using list comprehension

sum \$ takeWhile (< 10000) [n^2 | n < [1..], odd (n^2)]