

Haskell

Hello Recursion!

Stéphane Vialette

LIGM, Université Gustave Eiffel

September 28, 2020



Hello Recursion!



Hello Recursion!

Recursion is a way of defining functions in which a function is applied inside its own definition.

Recursion is important in Haskell because, unlike with imperative languages, you do computation in Haskell by declaring *what* something is rather than specifying *how* to compute it.

That's why Haskell isn't about issuing your computer a sequence of setps to execute, but rather about directly defining what the desired result, often in a recursive manner.



Maximum Awesome

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x : xs)
  | x > maxTail = x
  | otherwise   = maxTail
where
  maxTail = maximum' xs
```



Maximum Awesome

```
>>> maximum' [2,5,1]
```

```
5
```

$$\begin{aligned} \text{maximum}' [2, 5, 1] &= \\ \max 2 \left(\begin{aligned} \text{maximum}' [5, 1] &= \\ \max 5 \left(\begin{aligned} \text{maximum}' [1] &= \\ 1 \end{aligned} \end{aligned} \right) \end{aligned} \right) \end{aligned}$$



Maximum Awesome

```
maximum' :: (Ord a) => [a] -> a
maximum' []          = error "maximum of empty list"
maximum' [x]         = x
maximum' (x : xs)    = max x (maximum' xs)
```



A Few More Recursive Functions



A Few More Recursive Functions

`replicate`

`replicate` takes an `Int` and a value, and returns a list that has several repetitions of the same element.

```
replicate' :: (Num i, Ord i) => i -> a -> [a]
```

```
replicate' n x
```

```
  | n <= 0    = []
```

```
  | otherwise = x : replicate' (n-1) x
```

```
>>> replicate' 0 5
```

```
[]
```

```
>>> replicate' 1 5
```

```
[5]
```

```
>>> replicate' 10 5
```

```
[5,5,5,5,5,5,5,5,5,5]
```



In Passing

`Num` is not a subclass of `Ord`.

That means that what constitutes for a number doesn't really have to adhere to an ordering.

So that's why we have to specify both the `Num` and `Ord` class constraints when doing addition or subtraction and also comparison.



A Few More Recursive Functions

take

`take` returns a specified number of elements from a specified list.

```
take' :: (Num i, Ord i) => i -> [a] -> [a]
```

```
take' _ [] = []
```

```
take' n (x : xs) = x : take' (n-1) xs
```

```
>>> take' 0 ['a'..'z']
```

```
""
```

```
>>> take' 1 ['a'..'z']
```

```
"a"
```

```
>>> take' 5 ['a'..'z']
```

```
"abcde"
```

```
>>> take' 5 []
```

```
[]
```



A Few More Recursive Functions

reverse

`reverse` takes a list and return a list with the same elements, but in the reverse order.

```
reverse' :: [a] -> [a]
```

```
reverse' [] = []
```

```
reverse' (x : xs) = reverse' xs ++ [x]
```

```
>>> reverse' []
```

```
[]
```

```
>>> reverse' [1..10]
```

```
[10,9,8,7,6,5,4,3,2,1]
```

```
>>>
```



A Few More Recursive Functions

reverse

`reverse` takes a list and return a list with the same elements, but in the reverse order.

```
reverse' = aux []  
  where  
    aux rs [] = rs  
    aux rs (x : xs) = aux (x : rs) xs
```

```
>>> reverse' []  
[]  
>>> reverse' [1..10]  
[10,9,8,7,6,5,4,3,2,1]
```



A Few More Recursive Functions

repeat

`repeat` takes an element and returns an infinite list composed of that element.

```
repeat' :: a -> [a]
repeat' x = x : repeat' x
```

```
>>> take 10 (repeat' 5)
[5,5,5,5,5,5,5,5,5,5]
```



A Few More Recursive Functions

zip

`zip` takes two lists and zips them together.

```
>>> zip [1,2,3] [2,3]
[(1,2),(2,3)]
```

`zip` truncates the longer list to match the length of the shorter one.

How about if we zip something with an empty list? Well, we get an empty list back then.

```
zip' :: [a] -> [b] -> [(a,b)]
zip'  _      []      = []
zip' []      _      = []
zip' (x : xs) (y : ys) = (x,y) : zip' xs ys
```



A Few More Recursive Functions

`elem`

`elem` takes an element and a list and sees if that element is in the list.

The edge condition, as is most of the times with lists, is the empty list. We know that an empty list contains no elements, so it certainly doesn't have the droids we're looking for.

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' a [] = False
elem' a (x : xs)
  | a == x    = True
  | otherwise = a `elem'` xs
```



A Few More Recursive Functions

`elem`

`elem` takes an element and a list and sees if that element is in the list.

The edge condition, as is most of the times with lists, is the empty list. We know that an empty list contains no elements, so it certainly doesn't have the droids we're looking for.

```
elem' :: (Eq a) => a -> [a] -> Bool
```

```
elem' a [] = False
```

```
elem' a (x : xs) = a == x || a `elem'` xs
```



Quick, Sort!

There are many approaches to recursively sorting lists.

The quicksort algorithm works like this: You select the first element (called the *pivot*), put all the other list elements that are less than or equal to the first element on its left side, and put all the other list elements that are greater than the first element to its right side.

Now we recursively sort all the elements that are on the left and right sides of the pivot by calling the same function on them.



Quick, Sort!

[5, 1, 9, 4, 6, 7, 3]

[1, 4, 3] ++ [5] ++ [9, 6, 7]

[] ++ [1] ++ [4, 3]

[] ++ [4] ++ [3]

[] ++ [3] ++ []

[6, 7] ++ [9] ++ []

[] ++ [6] ++ [7]

[] ++ [7] ++ []



Quick, Sort!

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted  = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```



Quick, Sort!

```
>>> :t quicksort
quicksort :: Ord a => [a] -> [a]
>>> quicksort []
[]
>>> quicksort [1]
[1]
>>> quicksort [1,5,9,8,2,6,4,7,3]
[1,2,3,4,5,6,7,8,9]
>>> quicksort "to be or not to be"
"    bbeenoooorttt"
>>> quicksort [(5,6),(1,2),(3,4)]
[(1,2),(3,4),(5,6)]
```



Thinking recursively



Thinking recursively

Pattern

Start by defining a base case: simple non-recursive solution that holds when the input is trivial.

Then, break your problem down into one or many subproblems and recursively solve those by applying the same function to them.

Build up your final solution from those solved subproblems.



Use accumulators

Factorial

```
factorial :: (Eq a, Num a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

```
factorial' :: Integer -> Integer
factorial' = aux 1
```

where

```
aux acc n
  | n <= 1      = acc
  | otherwise = aux (acc * n) (n - 1)
```



Lists

```
>>> [1,2,3] ++ [4,5,6]  
[1,2,3,4,5,6]  
>>> "Hello " ++ "world"  
"Hello world"
```

```
(++) :: [a] -> [a] -> [a]  
[]      ++ ys = ys  
(x : xs) ++ ys = x : (xs ++ ys)
```



Merge sort

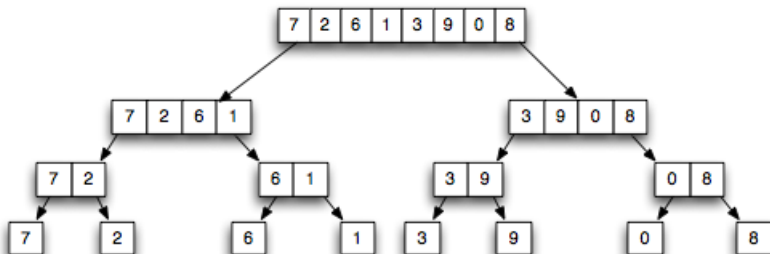
Mergesort is a little more complicated to implement.

The algorithm as follows:

1. List is split into two parts.
2. Two parts are sorted by the algorithm
3. The sorted parts are merged by a special merging procedure for sorted lists



Merge sort



Merge sort

Let's first define how we split a list into two parts:

```
mergeSortSplitInHalf :: [a] -> ([a], [a])
mergeSortSplitInHalf xs = (take n xs, drop n xs)
  where
    n = (length xs) `div` 2
```

```
>>> mergeSortSplitInHalf []
([], [])
>>> mergeSortSplitInHalf [1..5]
([1,2], [3,4,5])
>>> mergeSortSplitInHalf [1..6]
([1,2,3], [4,5,6])
```



Merge sort

Let's now define a function for merging two sorted arrays:

```
mergeSortMerge :: (Ord a) => [a] -> [a] -> [a]
mergeSortMerge [] xs = xs
mergeSortMerge xs [] = xs
mergeSortMerge (x : xs) (y : ys)
    | x < y      = x : mergeSortMerge xs      (y:ys)
    | otherwise  = y : mergeSortMerge (x : xs) ys
```

```
>>> mergeSortMerge [1..3] []
[1,2,3]
>>> mergeSortMerge [] [1..3]
[1,2,3]
>>> mergeSortMerge [1,3,4] [2,4,6]
[1,2,3,4,4,6]
```



Merge sort

```
mergeSort :: (Ord a) => [a] -> [a]
mergeSort xs = mergeSortMerge ls' rs'
  where
    (ls, rs) = mergeSortSplitInHalf xs
    ls'      = mergeSort ls
    rs'      = mergeSort rs
```

```
>>> mergeSort []
[]
>>> mergeSort [1]
[1]
>>> mergeSort [1,3,4,2,5,7,6]
[1,2,3,4,5,6,7]
```



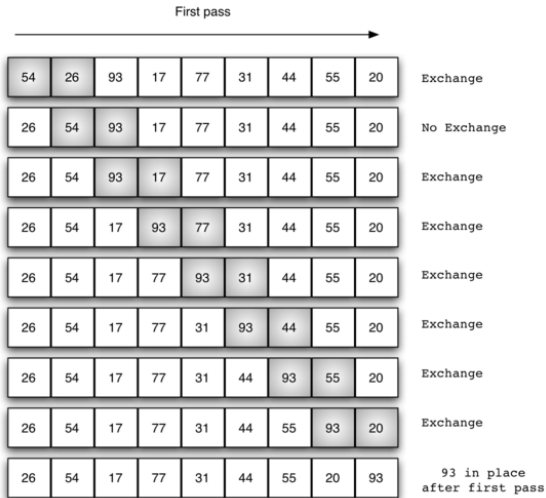
Bubble sort

Bubble sort is as follows:

```
procedure bubbleSort( A : list of sortable items )
  n = length(A)
  repeat
    swapped = false
    for i = 1 to n-1 inclusive do
      /* if this pair is out of order */
      if A[i-1] > A[i] then
        /* swap them and remember something changed */
        swap( A[i-1], A[i] )
        swapped = true
      end if
    end for
  until not swapped
end procedure
```



Bubble sort



Bubble sort

Let's first define the function that will go through all the elements in a list and exchange pairs of elements when it sees that the sorting order is wrong.

```
bubbleSortSwap :: (Ord a) => [a] -> [a]
bubbleSortSwap [] = []
bubbleSortSwap [x] = [x]
bubbleSortSwap (x : y : xs)
  | x > y      = y : bubbleSortSwap (x : xs)
  | otherwise = x : bubbleSortSwap (y : xs)
bubbleSortSwap (x) = (x)
```

```
>>> bubbleSortSwap []
[]
>>> bubbleSortSwap [1]
[1]
>>> bubbleSortSwap [4,3,2,1]
[3,2,1,4]
```



Bubble sort

Then we just need to apply this function n times – the length of the list that should be sorted.

```
bubbleSort' :: (Ord a) => [a] -> Int -> [a]
bubbleSort' xs i
  | i == (length xs) = xs
  | otherwise = bubbleSort' (bubbleSortSwap xs) (i + 1)
```

```
bubbleSort :: (Ord a) => [a] -> [a]
bubbleSort xs = bubbleSort' xs 0
```

```
>>> bubbleSort []
[]
>>> bubbleSort [2,4,1,6,5,3]
[1,2,3,4,5,6]
```



Thinking recursively



Don't get TOO excited about recursion...

Prefer

```
sumEven :: Integral a => [a] -> a
sumEven = sum . filter even
```

to

```
sumEven' :: Integral a => [a] -> a
sumEven' [] = 0
sumEven' (x : xs)
  | even x      = x + sumEven' xs
  | otherwise = sumEven' xs
```



Don't get TOO excited about recursion...

Prefer

```
sumEven :: Integral a => [a] -> a
sumEven = sum . filter even
```

to

```
sumEven'' :: Integral a => [a] -> a
sumEven'' = go 0
  where
    go acc [] = acc
    go acc (x : xs)
      | even x      = go (acc + x) xs
      | otherwise = go acc xs
```



Don't get TOO excited about recursion...

Prefer

```
sumEven :: Integral a => [a] -> a  
sumEven = sum . filter even
```

to

```
sumEven''' :: Integral a => [a] -> a  
sumEven''' xs = sum [x | x <- xs, even x]
```



Don't get TOO excited about recursion...

Prefer

```
pairs' :: [a] -> [(a, a)]  
pairs' xs = zip xs (tail xs)
```

to

```
pairs :: [a] -> [(a, a)]  
pairs []           = []  
pairs [_]          = []  
pairs (x : x' : xs) = (x, x') : pairs (x' : xs)
```



Don't get TOO excited about recursion...

Prefer

```
average :: Fractional t => [t] -> [t] -> [t]  
average = zipWith (\ x y -> (x + y) / 2.0)
```

or

```
average :: Fractional t => [t] -> [t] -> [t]  
average = zipWith f  
  where  
    f x y = (x + y) / 2.0
```

to

```
average' :: Fractional t => [t] -> [t] -> [t]  
average' []           = []  
average' _ []         = []  
average' (x : xs) (y : ys) = a : average' xs ys  
  where  
    a = (x + y) / 2.0
```

