Haskell
Hello Recursion!

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

January 18, 2016

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 ith imperative languages, you do computation in Haskell by declaring *what* something is rather than specifying *how* to compute it.

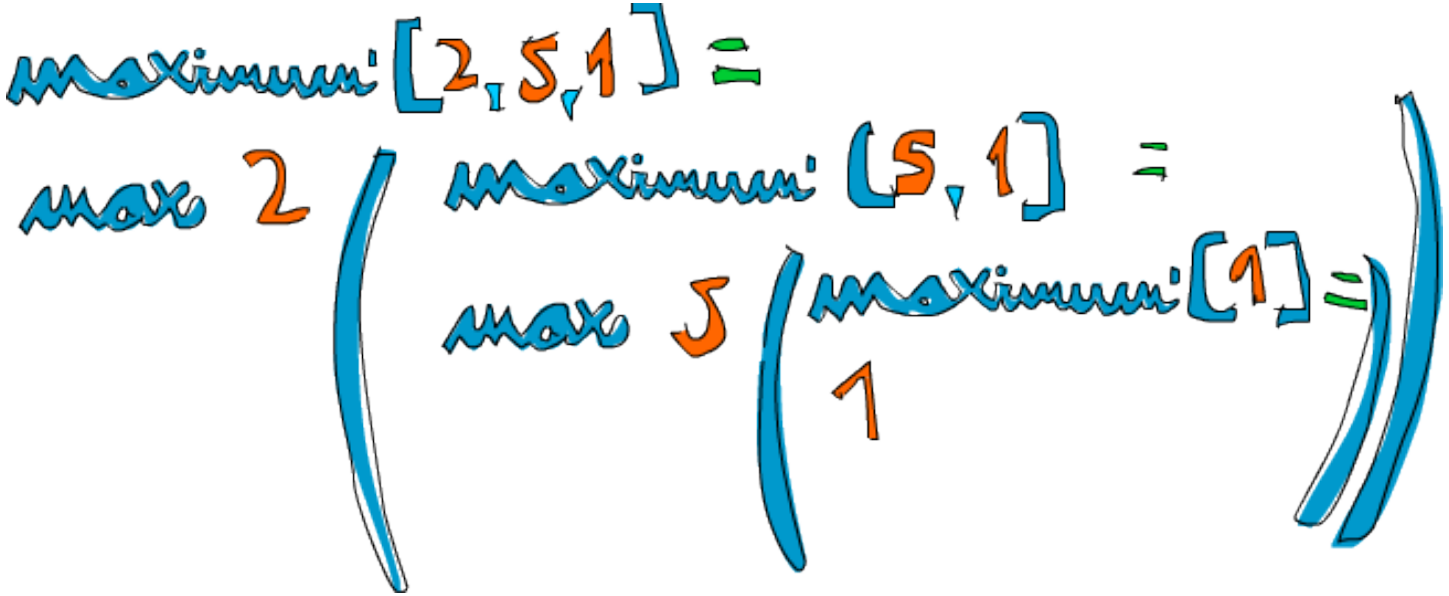
That's why Haskell isn't about issuing your computer a sequence of setps to execute, but rather about directly defining what the desired result, often in a recursive manner.

Maximum Awesome

```haskell
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

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



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
```

```
*Main> replicate' 0 5
[]
*Main> replicate' 1 5
[5]
*Main> replicate' 10 5
[5,5,5,5,5,5,5,5,5,5]
*Main>
```

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

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' n _
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

```
*Main> take' 0 ['a'..'z']
""
*Main> take' 1 ['a'..'z']
"a"
*Main> take' 5 ['a'..'z']
"abcde"
*Main> take' 5 []
[]
*Main>
```

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]
```

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

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
```

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

A Few More Recursive Functions

zip

`zip` takes two lists and zips them together.

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

`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
```

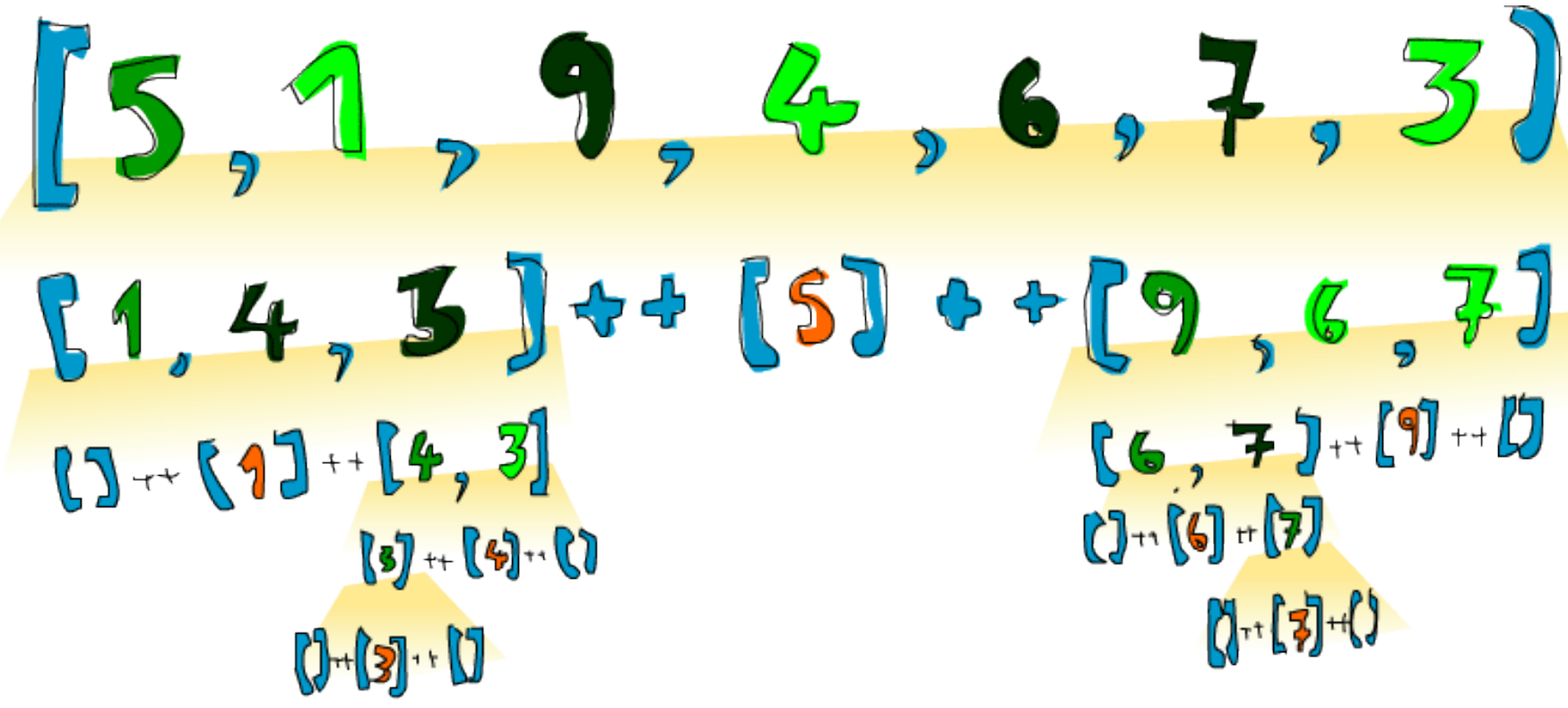
Quick, Sort!

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!



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!

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

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

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

```haskell
factorial' :: Integer -> Integer
factorial' = go 1
  where
    go acc n
      | n <= 1    = acc
      | otherwise = go (acc * n) (n - 1)
```

Lists

```
Prelude> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
Prelude> "Hello " ++ "world" -- Strings are lists of C
"Hello world"
```

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

Don't get TOO excited about recursion...

```haskell
evenSum :: [Integer] -> Integer
evenSum = go 0
  where
    go acc [] = acc
    go acc (x:xs)
      | even x    = go (acc + x) xs
      | otherwise = go acc xs


evenSum' :: [Integer] -> Integer
evenSum' = sum . filter even
```