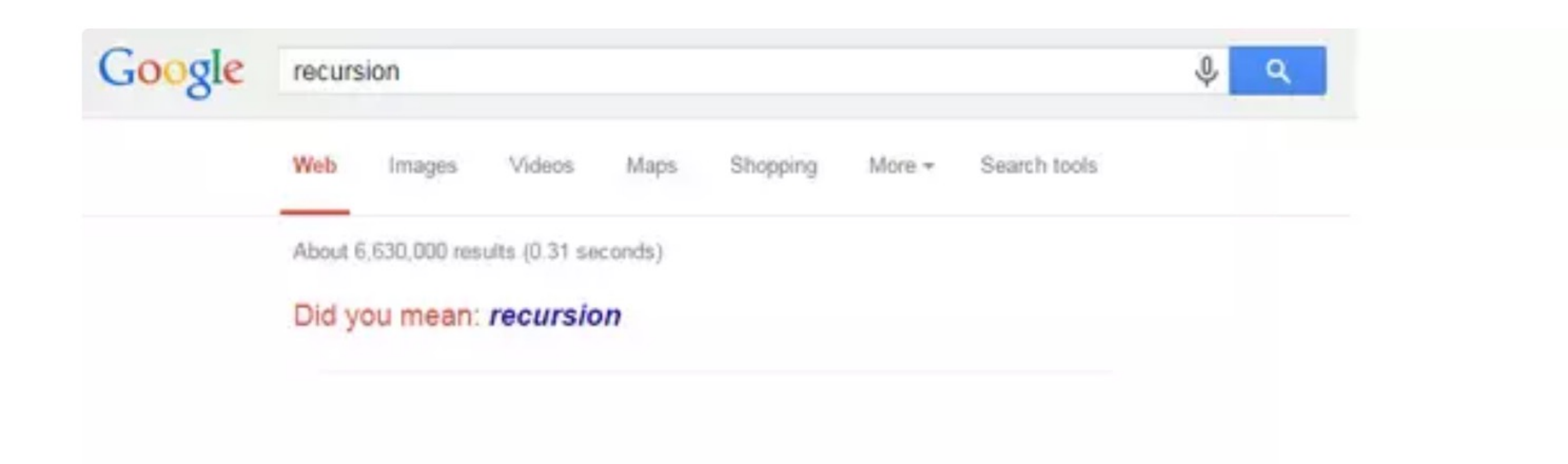
PROGRAMMING IN HASKELL

Chapter 6.1 - Recursive Functions - Introduction

What do we mean by recursion ?



Something is recursive if it is defined in terms of itself.

Recursion v iteration

In both cases we repeat a task.

- Iteration – repeat a task while a given condition holds

- Recursion – repeat a task on (a) smaller part(s) of the problem

Recursion example 'take one, pass them on'

The teacher comes into the classroom with a pile of pages. Each of the 100 students need to be handed a page. The instructions that the teacher gives to the students are :

1. Take the pile of pages from the previous student
2. Take one for yourself
3. Pass on the (smaller) pile to the next student if there are still some pages left.
4. If there is no page left, inform the teacher.

Recursion example ‘take one, pass them on’

We need to slightly reword the instructions to the students to be clearer:

1. Take the pile of pages from the previous student
2. Take a page from the pile (your page)
3. Check the number of pages left in the pile
4. If the number is zero -> inform the teacher ‘Job is done’
   Otherwise -> pass on the (smaller) pile to the next student.

Recursion and Iteration

Many algorithms naturally fall into the recursive family, e.g., binary search.

‘Look for an element in a sorted list’

Check in the ‘middle’. Now what you’re looking for is one of

-there (found it!!)

-< current element (in this case repeat on the left hand side, ignoring the right-hand side)

- > current element (in this case, repeat on the right hand, ignoring the left-hand side)

- not there at all (when the remaining list is empty)

Recursion and Iteration

- Recursive definitions are elegant and easy to understand

- THIS DOES NOT MEAN THAT THEY ARE EASY TO WRITE

- Iterative definitions (e.g., loops) are usually more efficient.

- We can mechanically derive an iterative definition from a recursive one.

Recursion and Haskell

- In this course, we use recursion a lot, starting with lists

- But first… try to think of something you do in your everyday life that you could describe recursively

So, show me some Haskell

As we have seen, many functions can naturally be defined in terms of other functions.

```
fac :: Int → Int
fac n = product [1..n]
```

fac maps any integer n to the product of the integers between 1 and n.

Expressions are evaluated by a stepwise process of applying functions to their arguments.

For example:

```
  fac 4
=
  product [1..4]
=
  product [1,2,3,4]
=
  1*2*3*4
=
  24
```

And now, recursive definition

Let us look at the factorial function using recursion

```
fac 0 = 1
fac n = n * fac (n-1)
```

fac maps 0 to 1, and any other integer to the product of itself and the factorial of its predecessor.

For example:

```
  fac 3
=
  3 * fac 2
=
  3 * (2 * fac 1)
=
  3 * (2 * (1 * fac 0))
=
  3 * (2 * (1 * 1))
=
  3 * (2 * 1)
=
  3 * 2
=
  6
```

Note:

- fac 0 = 1 is appropriate because 1 is the identity for multiplication: 1*x = x = x*1.

- The recursive definition diverges on integers < 0 because the base case is never reached:

```
> fac (-1)

*** Exception: stack overflow
```

Why is Recursion Useful?

- Some functions, such as factorial, are simpler to define in terms of other functions.

- As we shall see, however, many functions can naturally be defined in terms of themselves.

- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of induction.

Recursion on Lists

Recursion is not restricted to numbers, but can also be used to define functions on lists.

```
product :: Num a ⇒ [a] → a
product []     = 1
product (n:ns) = n * product ns
```

product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.

For example:

```
  product [2,3,4]
=
  2 * product [3,4]
=
  2 * (3 * product [4])
=
  2 * (3 * (4 * product []))
=
  2 * (3 * (4 * 1))
=
  24
```

Using the same pattern of recursion as in product we can define the <u>length</u> function on lists.

```
length :: [a] → Int
length []     = 0
length (_:xs) = 1 + length xs
```

length maps the empty list to 0, and any non-empty list to the successor of the length of its tail.

For example:

```
  length [1,2,3]
=
  1 + length [2,3]
=
  1 + (1 + length [3])
=
  1 + (1 + (1 + length []))
=
  1 + (1 + (1 + 0))
=
  3
```

Using a similar pattern of recursion we can define the reverse function on lists.

```
reverse :: [a] → [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

reverse maps the empty list to the empty list, and any non-empty list to the reverse of its tail appended to its head.

For example:

```
  reverse [1,2,3]
=
  reverse [2,3] ++ [1]
=
  (reverse [3] ++ [2]) ++ [1]
=
  ((reverse [] ++ [3]) ++ [2]) ++ [1]
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

In General..

When applying recursion to lists, it usually takes on the following structure:

```
recursiveFunction [] = []

recursiveFunction (x : xs) =
                    doSomethingWith x : recursiveFunction xs
```

Looking at insertion sort, we first look at a function that inserts a new element of any Ordered type into a sorted list to give another sorted list.

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

Now, using insert, we can define a function that implements

Insertion sort

in which the empty list is already sorted and any non-empty list is sorted by inserting its head into the list that results from sorting its tail.

```
isort :: Ord a => [a] → [a]
isort []     = []
isort (x:xs) = insert x (isort xs)
```

Multiple Arguments

Functions with more than one argument can also be defined using recursion. For example:

- Zipping the elements of two lists:

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

- Remove the first n elements from a list:

```
drop :: Int → [a] → [a]
drop 0 xs       = xs
drop _ []       = []
drop n (_:xs) = drop (n-1) xs
```

- Appending two lists:

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

Quicksort

The quicksort algorithm for sorting a list of values can be specified by the following two rules:

- The empty list is already sorted;
- Non-empty lists can be sorted by sorting the tail values $\leq$ the head, sorting the tail values $>$ the head, and then appending the resulting lists on either side of the head value.

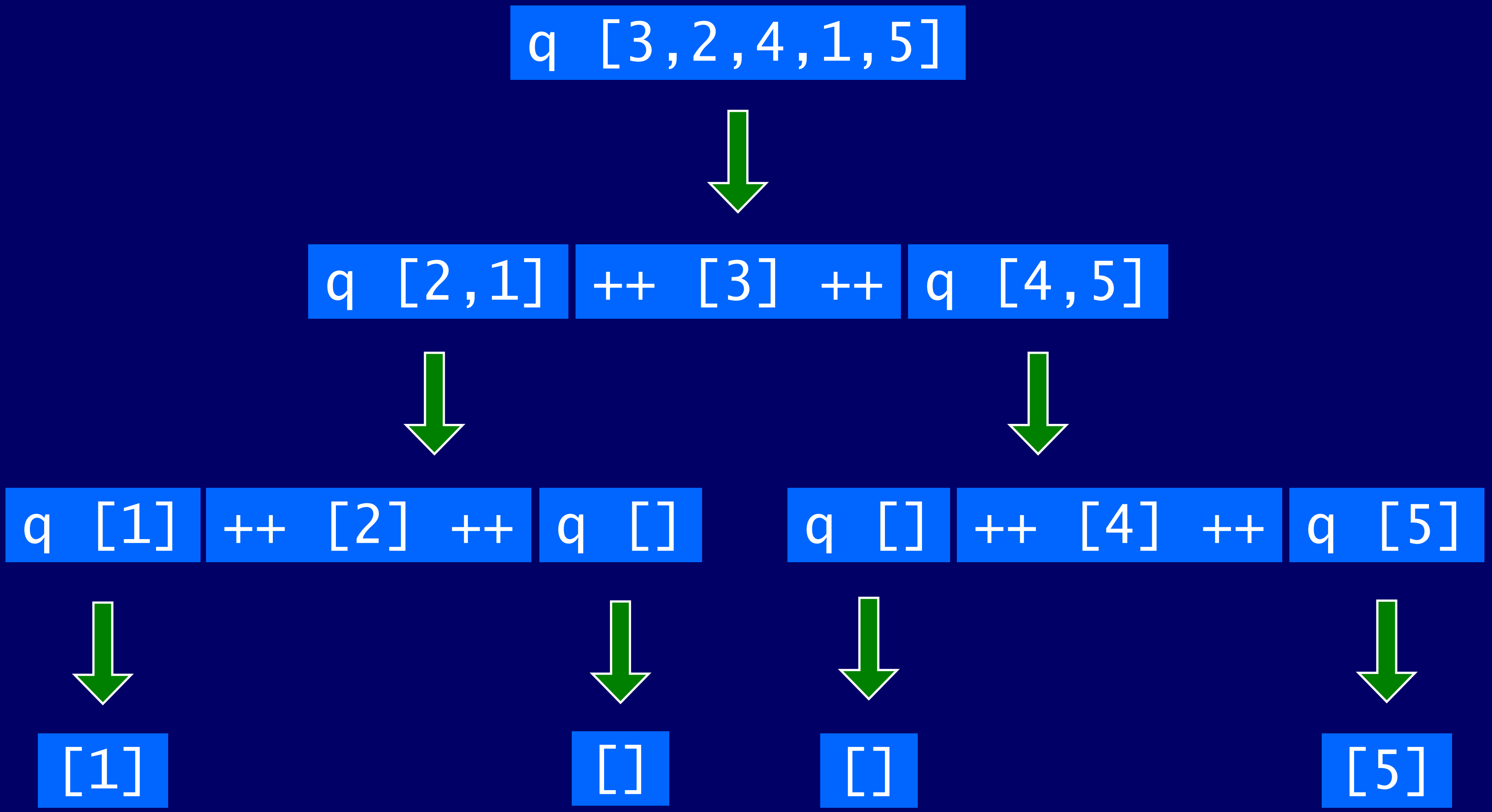
Using recursion, this specification can be translated directly into an implementation:

```
qsort :: Ord a ⇒ [a] → [a]
qsort []     = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a ← xs, a ≤ x]
      larger  = [b | b ← xs, b > x]
```

Note:

- This is probably the <u>simplest</u> implementation of quicksort in any programming language!

For example (abbreviating qsort as q):



Advice on Recursion

- Step 1 – Define the type,
- Step 2 – Enumerate the cases,
- Step 3 – Define the simple case,
- Step 4 – Define the other cases,
- Step 5 – Generalise and simplify.

Example 1 – product using steps

product – the function that takes a list and returns the product of the numbers.

Step 1 – Define the type

```
product :: [Int]  -> Int
```

(Use the simplest type possible)

Step 2 – Enumerate the cases

For most types, there are a number of standard cases to consider.

For lists, these are

- the empty list and
- the non-empty list

```
product []        =
product  (n:ns)   =
```

Step 3 – Define the simple case

By definition, the product of zero integers is one, because one is the identity for multiplication.

So:

```
product []        = 1
product (n:ns)    =
```

Step 4 – Define other cases

Consider the ingredients that can be used, e.g.

- the function itself (product) and
- the arguments (n and ns) and
- library functions of the relevant types (+, -, *, /)

```
product []        = 1
product  (n:ns)   = n * product ns
```

The other cases are often recursive cases.

Step 5 – Generalise and Simplify

```
product :: [Int] -> Int
product []          = 1
product  (n:ns)     = n * product ns
```

We can generalise from integers to any numeric type

```
product :: Num a => [a] -> a
product []          = 1
product  (n:ns)     = n * product ns
```

We will see how to simplify the definition later

Example 2 – drop using steps

drop – the function that takes an integer (n) and a list of values of some type a and returns a list with the first n elements dropped.

Step 1 – Define the type

```
drop  :: Int ->  [a]  -> [a]
```

(Use the simplest type possible)

Step 2 – Enumerate the cases

For integer type, the two standard cases are:

- 0 and
- n

For lists, these are

- the empty list and
- the non-empty list

```
drop  0  []        =
drop  0  (x:xs)    =
drop  n  []        =
drop  n  (x:xs)    =
```

Step 3 – Define the simple case

- By definition, removing zero elements from the start of a list gives the same list gives the same list.
- Removing n elements from an empty list can return an empty list (for safety).

So:

```
drop  0  []        =  []
drop  0  (x:xs)    =  x:xs
drop  n  []        =  []
drop  n  (x:xs)    =
```

Step 4 – Define other cases

Consider the ingredients that can be used, e.g.

- the function itself (drop) and
- the arguments (x and xs) and
- library functions of the relevant types (+, -, *, /)

```
drop  0  []        =   []
drop  0  (x:xs)    =   x:xs
drop  n  []        =   []
drop  n  (x:xs)    =   drop (n-1) xs
```

Step 5 – Generalise and Simplify

```
drop  :: Int ->  [a]  -> [a]
drop  0  []          =   []
drop  0  (x:xs)      =   x:xs
drop  n  []          =   []
drop  n  (x:xs)      =   drop (n-1) xs
```

We can generalise from integers to any integral type and simplify otherwise:

```
drop :: Integral b => b -> [a] -> [a]
drop  _  []          = []
drop  0  xs          = xs
drop  n (_:xs)       = drop (n-1) xs
```


ANY
QUESTIONS?