6 - Recursive Functions

Introduction

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
```

Recursive Functions

In Haskell, functions can also be defined in terms of themselves. Such functions are called **recursive**.

```
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*.

Exercises

Write the multiplication function recursivly.

```
> 4 * 3
12
```

```
(*) :: Int -> Int -> Int
n * 0 = 0
n * m = n + (n * (m-1))
```

```
n * 0 = 0
n * m | m > 0 = n + (n * (m-1))
      | otherwise = -(n * (-m))
```

Exercises

Write **even**

```
> even 10
True
> even 1000000
True
> even 1000001
False
```

```
even n = n `mod` 2 == 0
```

```
even 0 = True
even 1 = False
even n = not (even (n-1))
```

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
```

Exercises

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

```
> length [1,2,3,4,5]
5
```

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

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

Exercises

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

```
> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

```
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]
```

Methodology

Define the type

```
product :: [Int] → Int
```

Enumerate the cases

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

Define the simple cases

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

Define the other cases

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

Generalise and simplify

```
product :: Num a ⇒ [a] → a
```

Methodology - drop

Define the type

```
drop :: Int → [a] → [a]
```

Enumerate the cases

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

Define the simple cases

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

Methodology - drop

Define the other cases

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

Generalise and simplify

```
drop :: Integral b => b → [a] → [a]
```

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

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

Methodology - `init`

Define the type

```
init :: [a] → [a]
```

Enumerate the cases

```
init (x:xs) =
```

Define the simple cases

```
init (x:xs) | null xs = []
            | otherwise =
```

Define the other cases

```
init (x:xs) | null xs = []
            | otherwise = x : init xs
```

Generalise and simplify

```
init [_] = []
init (x:xs) = x : init xs
```

Multiple arguments

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

- Zipping the elements of two lists:

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

Exercises

Appending two lists:

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

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

Select the n-th element of a list:

```
> [1,2,3,4,5] !! 2
3
```

```
(!!) (x:xs) 0 = x
(!!) (x:xs) n = (!!) xs (n-1)
```

Multiple recursion

Functions can be defined using multiple recursion.

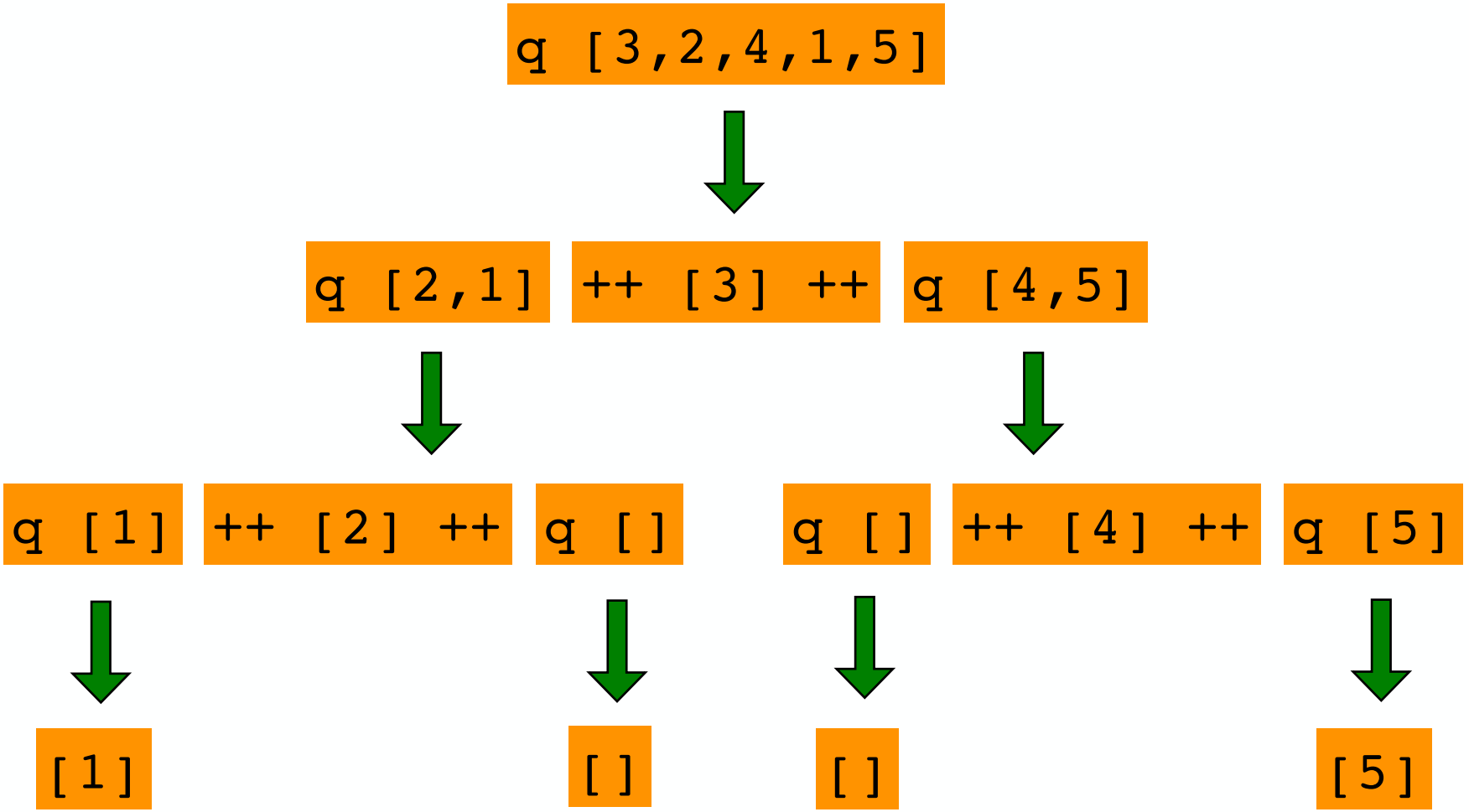
```
fib :: Int → Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

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 ≤ the head, sorting the tail values > the head, and then appending the resulting lists on either side of the head value.

For example (abbreviating qsort as q):



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 *simplest* implementation of quicksort in any programming language!

Mutual recursion

Several functions can be all defined recursively in terms of each other.

```
even :: Int -> Bool
even 0 = True
even n = odd (n-1)

odd :: Int -> Bool
odd 0 = False
odd n = even (n-1)
```

Exercises

- Compute the sum of all numbers from n to 0:

```
sumdown :: Int → Int
```

```
sumdown 0 = 0
sumdown n = n + (sumdown (n-1))
```

- Define exponentiation in terms of multiplication:

```
(^) :: Int → Int → Int
```

```
n ^ 0 = 1
n ^ m = n * (n ^ (m-1))
```

- Define the gcd of 2 integers:

```
gcd :: Int → Int → Int
```

```
gcd m n | m == n  =  m
        | m > n   = gcd (m-n) n
        | otherwise   = gcd m (n-m)
```

Exercises

Define the following functions using recursion:

- Decide if all logical values in a list are true:

```
and :: [Bool] → Bool
```

```
and [] = True
and (x:xs) = x && (and xs)
```

- Concatenate a list of lists:

```
concat :: [[a]] → [a]
```

```
concat [] = []
concat ((x:xs):ys) = x : (concat (xs:ys))
```

Exercises

- Produce a list with n identical elements:

```
replicate :: Int → a → [a]
```

```
replicate 0 _ = []
replicate n x = x : (replicate (n-1) x)
```

- Decide if a value is an element of a list:

```
elem :: Eq a ⇒ a → [a] → Bool
```

```
elem _ [] = False
elem x (y:ys) | x==y = True
              | otherwise = elem x ys
```

Exercises

Define a recursive function

```
insert :: Ord a ⇒ a → [a] → [a]
```

that inserts an element in a sorted list.

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

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

Exercises

Define a recursive function

```
isort :: Ord a ⇒ [a] → [a]
```

that implements *insert sort*:

- the empty list is sorted
- insert the head of the list in the sorted tail

```
isort [] = []
isort (x:xs) = insert x (isort xs)
```

Exercises

Define a recursive function

```
merge :: Ord a ⇒ [a] → [a] → [a]
```

that merges two sorted lists of values to give a single sorted list.

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

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

```
merge [] xs = xs
merge xs [] = xs
merge (x:xs) (y:ys) | x<y = x:merge xs (y:ys)
                    | otherwise = y:merge (x:xs) ys
```