

# Repetition, Recursion, Induction

Reading: Thompson Ch.7

**COMP1100 — Introduction to Programming and Algorithms**

Clem Baker-Finch

Australian National University

Semester 1, 2006

## Repetition

So far all our functions have used a **fixed number of operations**.

There are some functions that require an **different** number of operations, depending on the input.

An obvious example is the factorial function:

$$\text{factorial } n = 1 \times 2 \times \cdots \times n$$

Another is the standard `sum` function that sums the elements of a list of numbers:

$$\text{sum } [x_1, x_2 \dots x_n] \implies x_1 + x_2 + \cdots + x_n$$

How can we define such functions in Haskell?

## Observations:

- The number of operations to calculate both factorial and sum **depends on the input** ( $n$  in this case);
- A certain computation ( $\times$  in factorial,  $+$  in sum) is **repeatedly used**

## Recursion

The fundamental way to implement this kind of repetition is by a programming technique called **recursion**.

It is intimately related to the mathematical idea of **induction**.

## Calculating *factorial n*

The elipsis (...) in the description of *factorial* gives us the general idea, but how to write a precise definition?

Looking for clues:

$$\text{fact } 0 = 1$$

$$\text{fact } 1 = 1$$

$$\text{fact } 2 = 1 * 2$$

$$\text{fact } 3 = 1 * 2 * 3$$

$$\text{fact } 4 = 1 * 2 * 3 * 4$$

$$\text{fact } 5 = 1 * 2 * 3 * 4 * 5$$

...

## See the pattern?

fact 5 = 5 \* fact 4

fact 4 = 4 \* fact 3

fact 3 = 3 \* fact 2

fact 2 = 2 \* fact 1

fact 1 = 1 \* fact 0

fact 0 = 1

...

Generalising all except the fact 0 case, they are of the form:

fact n = n \* fact (n-1)

Combining the special case and the general case:

```
fact 0 = 1
```

```
fact n = n * fact (n-1)
```

Which is a correct Haskell definition — **Too easy!**

Notice the correspondence to mathematical induction:

- The definition has a base case, `fact 0`, and
- a step case, `fact n` **defined in term of** `fact (n-1)`

## Another example

The standard prelude has a function

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

where `replicate n x` gives us a list of `n` occurrences of `x`:

```
replicate 0 x = []           = []
replicate 1 x = [x]         = x : []
replicate 2 x = [x,x]       = x : x : []
replicate 3 x = [x,x,x]     = x : x : x : []
replicate 4 x = [x,x,x,x]   = x : x : x : x : []
...
```

## See the pattern?

```
replicate 4 x = x : replicate 3 x
replicate 3 x = x : replicate 2 x
replicate 2 x = x : replicate 1 x
replicate 1 x = x : replicate 0 x
replicate 0 x = []
```

So the definition is

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

( Again, notice the induction on  $n$ . )

## Lists are Recursive *Structures*

Every lists is constructed by a sequence of elements *cons*-ed onto the empty list.

$$[x_1, x_2, x_3, \dots, x_n] = x_1 : (x_2 : (x_3 : \dots : (x_n : [])))$$

A list is either:

1. the empty list []
2. constructed by cons-ing an element onto the front of **another list**

## List Patterns

Every list has either of the forms:

- `[]`
- `(x:xs)` where `x` is an element and `xs` is a list

Last week we used *pattern matching* to define functions on tuples:

```
fst :: (a, b) -> a
```

```
fst (x, y) = x
```

When we evaluate an expression like:

```
fst (10, 66)
```

10 is bound to `x`, and 66 is bound to `y` in the definition of `fst`.

We can similarly use **list patterns** to define functions on tuples:

```
head :: [a] -> a
```

```
head (x:xs) = x
```

```
tail :: [a] -> [a]
```

```
tail (x:xs) = xs
```

When we evaluate

```
tail [11, 12, 13, 4, 5]
```

11 is bound to x, and [12, 12, 4, 5] is bound to xs,  
so the result (xs) is [12, 12, 4, 5]

(Notice that neither `head` nor `tail` has a case for []. They are *partial* functions.)

**There are two basic list patterns, so in general functions over lists will use both of them.**

```
null :: [a] -> Bool
null []      = True
null (x:xs)  = False
```

If we evaluate

```
null [11, 12, 13, 4, 5]
```

Haskell will first attempt to match `[11, 12, 13, 4, 5]` to the pattern `[]` in the first clause of the definition and fail. Haskell will then attempt it match it to the pattern `(x:xs)` in the second clause and succeed (as in the previous slide). Hence the result will be the right hand side of the successful clause — `False`.

## Recursive List Functions

Return to our initial question: how to define a function to sum the elements of a list of numbers.

Compare the `sum` calculation with the structure of the list:

$$x_1 + (x_2 + (x_3 + \dots (x_n + 0)))$$
$$x_1 : (x_2 : (x_3 : \dots (x_n : [])))$$

(I admit choosing the parentheses and including the “+ 0” to make this look right.)

Notice the direct correspondence:

- instead of `[]` we have `0`
- instead of `(:)` we have `(+)`

The list patterns immediately suggest a starting point:

`sum []` = ...

`sum (x:xs)` = ...

The correspondence on the previous slide suggests the right hand sides.

The `sum` function substitutes 0 for `[]` and `(+)` for `(:)` to give

`sum :: Num a => [a] -> a`

`sum []` = 0

`sum (x:xs)` = `x + sum xs`

The evaluation of a call of `sum` proceeds as follows:

$$\begin{aligned} \text{sum } [2, 5, 7, 1] &\implies 2 + \text{sum } [5, 7, 1] \\ &\implies 2 + (5 + \text{sum } [7, 1]) \\ &\implies 2 + (5 + (7 + \text{sum } [1])) \\ &\implies 2 + (5 + (7 + (1 + \text{sum } []))) \\ &\implies 2 + (5 + (7 + (1 + 0))) \\ &\implies 15 \end{aligned}$$

## Another Example

Define a function to multiply every element of a list of numbers by 2.

Compare the `double` calculation with the structure of the list:

$$\begin{array}{cccccccc} x_1 * 2 & : & x_2 * 2 & : & x_3 * 2 & : & \dots & : & x_n * 2 & : & [] \\ x_1 & : & x_2 & : & x_3 & : & \dots & : & x_n & : & [] \end{array}$$

Notice the direct correspondence:

- instead of each  $x_i$  we have  $x_i * 2$
- where we had `[]` we still have `[]`

The list patterns suggest a starting point:

```
double []      = ...
double (x:xs) = ...
```

The correspondence gives us the right hand sides:

```
double []      = []
double (x:xs) = x*2 : double xs
```

**Applying the same function to every element of a list is a very common activity.**

**This pattern of recursion is coded up as the `map` function.** (More later.)