

Understanding Recursive Definitions Through Interpretive Dance (and other techniques)

COMP1100 — Introduction to Programming and Algorithms

Clem Baker-Finch

Australian National University
Semester 1, 2006

Understanding recursion by ... *correspondence with the list structure*

(We did this last week.)

Compare the sum calculation with the structure of the list:

```
x1 + (x2 + (x3 + ... (xn + 0 )))  
x1 : (x2 : (x3 : ... (xn : [])))
```

Notice the direct correspondence:

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

Recursion

Recursion is ubiquitous in programming, including Haskell.

Recursion is **the fundamental technique** for implementing **repetition** in any computation, any programming language.

Any other techniques (loops) can always be transformed to recursion.

General recursion *cannot* be directly implemented using loops.

Recursive definitions can be confusing at first, so here are some ideas that might help you understand them.

The sum function traverses the list:

- replacing [] with 0
- replacing (:) with (+)

```
sum []           = 0  
sum (x:xs)      = x + sum xs
```

How about the length of a list?

```
1 + (1 + (1 + ... (1 + 0 )))  
x1 : (x2 : (x3 : ... (xn : [])))
```

- instead of [] we have 0
- instead of (:) we have (+)
- instead of each x_i we have 1

so:

```
length [] = 0  
length (x:xs) = 1 + length xs
```

Another one?

```
sum [] = 0  
sum (x:xs) = x + sum xs
```

Understanding recursion by ... *interpretive dance*

- Get some friends together
- Each play a role in the computation

```
length [] = 0  
length (x:xs) = 1 + length xs
```

(This is no fun by yourself.)

Accumulating Parameters

Forget programming for a moment.

How would **you** count the elements in a list?

like this?

```
a : (b : (c : (d : [])))  
4  3  2  1  0
```

or like this?

```
a : (b : (c : (d : [])))  
0  1  2  3  4
```

Length of a list — accumulating parameter

Use an extra parameter to keep a running total:

```
count :: [a] -> Int -> Int
count [] total = total
count (x:xs) total = count xs (total+1)
```

Start the count at zero:

```
length :: [a] -> Int
length xs = count xs 0
```

Another one?

```
sum [] = 0
sum (x:xs) = x + sum xs
```

Understanding recursion by ... *trusting the recursive call*

- Pick out the recursive call
- Suppose it works
- Think about the definition

```
length [] = 0
length (x:xs) = 1 + length xs
```

(This is like solo interpretive dance.)

Sum a list of numbers — accumulator

```
addUp :: Num a => [a] -> a -> a
addUp [] total = total
addUp (x:xs) total = addUp xs (total+x)
```

```
sum :: Num a => [a] -> a
sum xs = addUp xs 0
```