

## Comp 205: Comparative Programming Languages

Functional Programming Languages:

### More Lists

- Recursive definitions
- List comprehensions

Lecture notes, exercises, etc., can be found at:  
[www.csc.liv.ac.uk/~grant/Teaching/COMP205/](http://www.csc.liv.ac.uk/~grant/Teaching/COMP205/)

## Recursion

The type of lists is “recursively defined”:

A list is either:

- empty, or
- is an element followed by a list

Many function definitions follow this pattern:

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

## Recursion

Compute the sum of all the numbers in a list:

```
sumAll :: [Integer] -> Integer
```

The sum of all the numbers in the empty list is 0:

```
sumAll [] = 0
```

The sum of all the numbers in a list  $x : xs$  is  $x$  + the sum of all the numbers in  $xs$ :

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

## Recursion

Concatenate all the strings in a list of strings:

```
concat :: [[Char]] -> [Char]
```

Concatenating all the strings in the empty list gives the empty string:

```
concat [] = []
```

Concatenating all the strings in a list  $s : ss$  is  $s$  appended to the concatenation of all the strings in  $ss$ :

```
concat (s : ss) = s ++ concat ss
```

## Patterns

Patterns can be combined:

```
zip :: [Int] -> [String] -> [(Int,String)]
zip [] ss = []
zip is [] = []
zip (i:is) (s:ss) = (i,s) : zip is ss
```

## More Patterns

Patterns can be complex:

```
unzip [] = []
unzip ((x,y) : ps) = (x:xs, y:ys)
  where
    (xs,ys) = unzip ps
```



## Summary

Key topics:

- Recursion
- List Comprehensions

*Next: Higher-Order Functions*