We want to discover frequently occurring patterns of computation. These patterns are then made into (often higher-order) functions which can be specialized and combined. `map f L` and `filter f L` can be specialized and combined:

```haskell
double :: [Int] -> [Int]
double xs = map ((*) 2) xs

positive :: [Int] -> [Int]
positive xs = filter ((<) 0) xs

doublePos xs = map ((*) 2) (filter ((<) 0) xs)
```

`doublePos [2,3,0,-1,5]` gives `[4, 6, 10]`

Composing Functions

Functional composition is a kind of “glue” that is used to “stick” simple functions together to make more powerful ones.

In mathematics the ring symbol `(◦)` is used to compose functions:

\[(f \circ g)(x) = f(g(x))\]

In Haskell we use the dot (`"."`) symbol:

```haskell
infixr 9 .
(.) :: (b->c) -> (a->b) -> (a->c)
(f . g)(x) = f(g(x))
```

"." takes two functions `f` and `g` as arguments, and returns a new function `h` as result.

- `g` is a function of type `a->b`.
- `f` is a function of type `b->c`.
- `h` is a function of type `a->c`.
- `(f . g)(x)` is the same as `z=g(x)` followed by `f(z)`. 
Composing Functions...

- We use functional composition to write functions more concisely. These definitions are equivalent:

  \[
  \text{doit } x = f_1 \ (f_2 \ (f_3 \ (f_4 \ x))) \\
  \text{doit } x = (f_1 \ . \ f_2 \ . \ f_3 \ . \ f_4) \ x \\
  \text{doit } = f_1 \ . \ f_2 \ . \ f_3 \ . \ f_4
  \]

- The last form of \text{doit} is preferred. \text{doit}'s arguments are implicit; it has the same parameters as the composition.

- \text{doit} can be used in higher-order functions (the second form is preferred):

  ? map (doit) xs \\
  ? map (f_1 \ . \ f_2 \ . \ f_3 \ . \ f_4) \ xs

Example: Splitting Lines

- Assume that we have a function \text{fill} that splits a string into filled lines:

  \[
  \text{fill } :: \text{string} \to \text{[string]} \\
  \text{fill } s = \text{splitLines } (\text{splitWords } s)
  \]

  \text{fill} first splits the string into words (using \text{splitWords}) and then into lines:

  \[
  \text{splitWords } :: \text{string} \to \text{[word]} \\
  \text{splitLines } :: \text{[word]} \to \text{[line]}
  \]

- We can rewrite \text{fill} using function composition:

  \[
  \text{fill } = \text{splitLines } \ . \ \text{splitWords}
  \]

Precedence & Associativity

1. "." is right associative. I.e.
   \[
   f . g . h . i . j = f . (g . (h . (i . j)))
   \]

2. "." has higher precedence (binding power) than any other operator, except function application:
   \[
   5 + f . g \ 6 = 5 + (f . (g \ 6))
   \]

3. "." is associative:
   \[
   f . (g . h) = (f . g) . h
   \]

4. "id" is "."'s identity element, i.e id . f = f . id:
   \[
   \text{id } :: \ a \to a \\
   \text{id } x = x
   \]

The count Function

- Define a function \text{count} which counts the number of lists of length \(n\) in a list \(L\):

  \[
  \text{count } 2 \ [[1],[],[2,3],[4,5],[[]]] \Rightarrow 2
  \]

  Using recursion:

  \[
  \text{count } :: \text{Int} \to \text{[[a]]} \to \text{Int} \\
  \text{count } = 0 \\
  \text{count } n \ (x:xs) \\
  | \text{length } x = n = 1 + \text{count } n \ xs \\
  | \text{otherwise} = \text{count } n \ xs
  \]

  Using functional composition:

  \[
  \text{count'} \ n = \text{length } \ . \ \text{filter } (=n) \ . \ \text{map } \text{length}
  \]
The count Function...

`count' n = length . filter (==n) . map length`

• What does `count'` do?

```
[[1],[1],[2,3],[4,5],[1]]
  map length
[1,0,2,2,0]
  filter (==2)
[2,2]
  length
2
```

• Note that

```
count' n xs = length (filter (==n) (map length xs))
```

The init & last Functions

• `last` returns the last element of a list.
• `init` returns everything but the last element of a list.

```
--------------------- Definitions: ---------------------
last = head . reverse
init = reverse . tail . reverse
--------------------- Simulations: ---------------------
[1,2,3] reverse ➞ [3,2,1] head ➞ 3
[1,2,3] reverse ➞ [3,2,1] tail ➞ [2,1] reverse ➞ [1,2]
```

The any Function

• `any p xs` returns `True` if `p x == True` for some `x` in `xs`:

```
any (==0) [1,2,3,0,5] ➞ True
any (==0) [1,2,3,4] ➞ False
--------------------- Using recursion: ---------------------
any :: (a -> Bool) -> [a] -> Bool
any _ [] = False
any p (x:xs) = | p x = True
               | otherwise = any p xs
--------------------- Using composition: ---------------------
any p = or . map p
[1,0,3] map (==0) ➞ [False,True,False] or ➞ True
```

commaint Revisited...

• Let's have another look at one simple (!) function, `commaint`.
• `commaint` works on strings, which are simply lists of characters.
• You are NOT now supposed to understand this!

```
[commaint] takes a single string argument containing a sequence of digits, and outputs the same sequence with commas inserted after every group of three digits, · · ·
```
Sample interaction:

```
? commaint "1234567"
1,234,567
```

commaint in Haskell:

```
commaint = reverse . foldr1 (\x y->x++","++y) .
    group 3 . reverse
    where group n = takeWhile (not.null) .
        map (take n).iterate (drop n)
```

- \(\text{iterate (drop 3)} \ s\) returns the infinite list of strings
  
  \([s, \text{drop 3} s, \text{drop 3} (\text{drop 3} s), \text{drop 3} (\text{drop 3} (\text{drop 3} s)), \ldots]\)

- \(\text{map (take n) xss}\) shortens the lists in xss to n elements.

\(\text{takeWhile (not.null)}\) removes all empty strings from a list of strings.

\(\text{foldr1 (\x y->x++","++y)} \ s\) takes a list of strings s as input. It appends the strings together, inserting a comma in between each pair of strings.
**Lambda Expressions**

- \( (\lambda x \ y \to x++"","++y) \) is called a lambda expression.
- Lambda expressions are simply a way of writing (short) functions inline. Syntax:
  \[ \ \text{arguments} \to \text{expression} \]
- Thus, `commaint` could just as well have been written as

\[
\text{commaint} = \ldots \ . \ \text{foldr1} \ \text{insert} \ . \ \ldots \\
\text{where} \ \text{group} \ n = \ldots \\
\text{insert} \ x \ y = x++"","++y
\]

Examples:

- `squareAll xs = map (\ x \to x * x) xs`
- `length = foldl' (\ n \to n+1) 0`

**Summary**

- The built-in operator "." (pronounced “compose”) takes two functions \(f\) and \(g\) as argument, and returns a new function \(h\) as result.
- The new function \(h = f \ . \ g\) combines the behavior of \(f\) and \(g\): applying \(h\) to an argument \(a\) is the same as first applying \(g\) to \(a\), and then applying \(f\) to this result.
- Operators can, of course, also be composed: \(((+2) \ . \ (*3))\) 3 will return \(2 + (3 * 3) = 11\).

**Homework**

- Write a function \(\text{mid} \ \text{xs}\) which returns the list \(\text{xs}\) without its first and last element.
  - use recursion

\[
? \ \text{mid}[1,2,3,4,5] \Rightarrow [2,3,4] \\
? \ \text{mid}[] \Rightarrow \text{ERROR} \\
? \ \text{mid}[1] \Rightarrow \text{ERROR} \\
? \ \text{mid}[1,3] \Rightarrow []
\]