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 (\circ) is used to compose functions:

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

- In Gofer we use the dot ("." ) symbol:

\[
\text{infixr 9 .} \\
( .) :: (b \to c) \to (a \to b) \to (a \to c) \\
(f . g)(x) = f(g(x))
\]

- "." takes two functions \( f \) and \( g \) as arguments, and returns a new function \( h \) as result.
- \( g \) is a function of type \( a \to b \).
- \( f \) is a function of type \( b \to c \).
- \( h \) is a function of type \( a \to c \).
- \((f . g)(x)\) is the same as \( z=g(x) \) followed by \( f(z) \).

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:

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]  
[4, 6, 10]
Precedence & Associativity

1. "." is right associative. I.e.
   \[ f \circ g \circ h \circ i \circ j = f \circ (g \circ (h \circ (i \circ j))) \]
2. "." has higher precedence (binding power) than any other operator, except function application:
   \[ 5 + f \circ g \ 6 = 5 + (f \ (g \ 6)) \]
3. "." is associative:
   \[ f \ (g \ h) = (f \ g) \ h \]
4. "id" is "."’s identity element, i.e \[ id \circ f = f = f \circ id \]

The count Function

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

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

Using recursion:

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

Using functional composition:

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

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):

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

Example: Splitting Lines

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

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

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

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

• We can rewrite \( \text{fill} \) using function composition:

\[
\text{fill} = \text{splitLines} \ . \ \text{splitWords}
\]
The any Function

- **any** \( p \) \( xs \) returns **True** if \( p \ x == \text{True} \) for some \( x \) in \( xs \):

\[
\text{any } ((==)0) \ [1,2,3,0,5] \Rightarrow \text{True}
\]
\[
\text{any } ((==)0) \ [1,2,3,4] \Rightarrow \text{False}
\]

Using recursion: ____________

\[
\text{any :: (a -> Bool) -> [a] -> Bool}
\]
\[
\text{any } [] \Rightarrow \text{False}
\]
\[
\text{any } p \ (x:xs) = | p x = \text{True} \:
\]
\[
| \text{otherwise} = \text{any } p \ xs
\]

Using composition: ____________  

\[
\text{any } p \Rightarrow \text{map } ((==)0) \Rightarrow \text{or} \Rightarrow \text{True}
\]

**note that**

\[
\text{count'} \ n \ xs = \text{length} \ (\text{filter } ((==)n) \ (\text{map} \ \text{length} \ xs))
\]

The init & last Functions

- **last** returns the last element of a list.

- **init** returns everything but the last element of a list.

Definitions: ____________

\[
\text{last} \Rightarrow \text{head} \Rightarrow \text{reverse}
\]
\[
\text{init} \Rightarrow \text{reverse} \Rightarrow \text{tail} \Rightarrow \text{reverse}
\]

Simulations: ____________

\[
[1,2,3] \Rightarrow [3,2,1] \Rightarrow [1,2]
\]

\[
[1,2,3] \Rightarrow [3,2,1] \Rightarrow [2,1]
\]

**commaint Revisited...**

- Let’s have another look at one simple (!) function, **commaint**.

- **commaint** works on strings, which are simply lists of characters.

- You are \( \neq \) now supposed to understand this!

From the **commaint** documentation: ____________

[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, \( \cdot \)
**commaint Revisited...**

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

*iterate (drop 3) s* returns the infinite list of strings
[s, drop 3 s, drop 3 (drop 3 s),
 drop 3 (drop 3 (drop 3 s)), ...]

*map (take n) xss* shortens the lists in xss to n elements.

**commaint in Gofer:**

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

• *takeWhile (not.null) *removes all empty strings from a list of strings.

• *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.

---

**Sample interaction:**

? commaint "1234567"
1,234,567

---

**Slide 10–13**

```
"1234567"
  reverse
  "7654321"
  iterate (drop 3)
  ["7654321","4321","1","","",...]
  map (take 3)
  ["765","432","1","","",...
  takeWhile (not.null)
  ["765","432","1"]
  foldr1 (\x y->x++","++y)
  "765,432,1"
  reverse
  "1,234,567"
```
Lambda Expressions

- \( (\lambda \, y \to x++,"++y) \) is called a lambda expression.
- Lambda expressions are simply a way of writing (short) functions inline. Syntax:
  \( \lambda \ arguments \to expression \)
- Thus, `commaint` could just as well have been written as
  \[ \text{commaint} = \ldots \ . \ \text{foldr1 insert} \ . \ \ldots \]
  \[ \text{where group n} = \ldots \)
  \[ \text{insert x y} = x++","++y \]

Examples:

- `squareAll xs = map (\ x -> x * x) xs`
- `length = foldl' (\n _ -> 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 \( 2 \ + \ (3 * 3) = 11\).