1 Composing Functions

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. \texttt{map f L} and \texttt{filter f L} can be specialized and combined:

\begin{verbatim}
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]
\end{verbatim}

2 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 Haskell we use the dot ("."\) symbol:

\begin{verbatim}
infixr 9 .
(.) :: (b->c) -> (a->b) -> (a->c)
(f . g)(x) = f(g(x))
\end{verbatim}
3 Composing Functions... 

(\cdot) :: (b->c) -> (a->b) -> (a->c)
(f . g)(x) = f(g(x))

- "\cdot" 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).

4 Composing Functions... 

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

doit x = f1 (f2 (f3 (f4 x)))
doit x = (f1 . f2 . f3 . f4) x
doit = f1 . f2 . f3 . f4

  - The last form of doit is preferred. doit’s arguments are implicit; it has the same parameters as the composition.
  - doit can be used in higher-order functions (the second form is preferred):

? map (doit) xs
? map (f1 . f2 . f3 . f4) xs

5 Example: Splitting Lines

- Assume that we have a function fill that splits a string into filled lines:

fill :: string -> [string]
fill s = splitLines (splitWords s)

  - fill first splits the string into words (using splitWords) and then into lines:

splitWords :: string -> [word]
splitLines :: [word] -> [line]

  - We can rewrite fill using function composition:

fill = splitLines . splitWords
6 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 \( \text{id} . f = f = f . \text{id} \):
   \[
   \begin{align*}
   \text{id} & : \ a \rightarrow a \\
   \text{id} \ x & = x
   \end{align*}
   \]

7 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} :: \ Int \rightarrow [\[a\]] \rightarrow \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:**
  \[
  \text{count’} \ n = \text{length} \ . \ \text{filter} \ (==n) \ . \ \text{map} \ \text{length}
  \]

8 The count Function...

\text{count’} \ n = \text{length} \ . \ \text{filter} \ (==n) \ . \ \text{map} \ \text{length}

- What does \text{count’} do?

\[
\begin{array}{c}
[[1],[],[2,3],[4,5],[]] \\
\downarrow \text{map length}\rightarrow [1,0,2,2,0] \\
\downarrow \text{filter} \ (==2)\rightarrow [2,2] \\
\downarrow \text{length}\rightarrow 2
\end{array}
\]

- Note that

\[
\text{count’} \ n \ xs = \text{length} \ (\text{filter} \ (==n) \ (\text{map} \ \text{length} \ xs))
\]
9 The init & last Functions

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

**Definitions:**

\[
\text{last} = \text{head} \cdot \text{reverse} \\
\text{init} = \text{reverse} \cdot \text{tail} \cdot \text{reverse}
\]

**Simulations:**

\[
\begin{align*}
[1,2,3] & \xrightarrow{\text{reverse}} [3,2,1] & \xrightarrow{\text{head}} 3 \\
[1,2,3] & \xrightarrow{\text{reverse}} [3,2,1] & \xrightarrow{\text{tail}} [2,1] & \xrightarrow{\text{reverse}} [1,2]
\end{align*}
\]

10 The any Function

- **any** \( p \ AX \) returns \( True \) if \( p \ AX \ \Rightarrow \ True \) for some \( x \) in \( AX \):

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

**Using recursion:**

\[
\text{any} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool} \\
\text{any} _\cdot [\ ] = \text{False} \\
\text{any} p (x:xs) = \mid p x \Rightarrow \text{True} \\
\mid \text{otherwise} = \text{any} p xs
\]

**Using composition:**

\[
\begin{align*}
\text{any} p &= \text{or} \cdot \text{map} p \\
[1,0,3] & \xrightarrow{\text{map} ((==)0)} [\text{False,True,False}] & \xrightarrow{\text{or}} \text{True}
\end{align*}
\]

11 **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!

**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, \( \cdots \).
12 commaint Revisited...

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)

13 commaint Revisited...

iterate (drop 3) s returns the infinite list of strings

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

14 commaint Revisited...

iterate (drop 3) s returns the infinite list of strings

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

15 commaint Revisited...

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.

16 Lambda Expressions

• `(\x y->x++",++y)` is called a lambda expression.
• Lambda expressions are simply a way of writing (short) functions inline. Syntax:

  \ arguments -> expression

• Thus, `commaint` could just as well have been written as

  `commaint = ⋯ . foldr1 insert . ⋯`
  where `group n = ⋯`
  `insert x y = x++",++y`

  **Examples:**

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

17 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`.

18 Homework

• Write a function `mid xs` which returns the list `xs` without its first and last element.
  1. use recursion
  2. use `init`, `tail`, and functional composition.
  3. use `reverse`, `tail`, and functional composition.

  ? `mid [1,2,3,4,5]` ⇒ `[2,3,4]`
  ? `mid []` ⇒ ERROR
  ? `mid [1]` ⇒ ERROR
  ? `mid [1,3]` ⇒ []