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. \( \text{map } f \ L \) and \( \text{filter } f \ L \) can be specialized and combined:

\[
\text{double} :: [\text{Int}] \rightarrow [\text{Int}]
\]
\[
\text{double } xs = \text{map } ((*) 2) \ xs
\]

\[
\text{positive} :: [\text{Int}] \rightarrow [\text{Int}]
\]
\[
\text{positive } xs = \text{filter } ((<) 0) \ xs
\]

\[
\text{doublePos } xs = \text{map } ((*) 2) \ (\text{filter } ((<) 0) \ xs)
\]

? doublePos [2,3,0,-1,5]
[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 (\( \circ \)) is used to compose functions:

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

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

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

"." takes two functions \( f \) and \( g \) as arguments, and returns a new function \( h \) as result.
\( g \) is a function of type \( a \rightarrow b \).
\( f \) is a function of type \( b \rightarrow c \).
\( h \) is a function of type \( a \rightarrow c \).
\( (f \ g) \ (x) \) 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:

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

- 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):

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

Example: Splitting Lines

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

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

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

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

- We can rewrite 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. "." is "."'s identity element, i.e id . f = f = f . id:

\[
id :: \text{a} \to \text{a}
\]
\[
id \ x = x
\]

The count Function

- Define a function 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)

\mid \text{length} \ x == n \quad = 1 + \text{count} \ n \ xs
\mid \text{otherwise} \quad = \text{count} \ n \ xs
\]

Using functional composition:

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

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

- What does `count'` do?

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

- Note that

```haskell
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:**

```haskell
last = head . reverse
init = reverse . tail . reverse
```

**Simulations:**

```haskell
[1,2,3] reverse  \rightarrow [3,2,1] head \rightarrow 3
```

```haskell
[1,2,3] reverse  \rightarrow [3,2,1] tail  \rightarrow [2,1] reverse \rightarrow [1,2]
```

The any Function

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

```haskell
any (==)0 [1,2,3,0,5] \Rightarrow True
any (==)0 [1,2,3,4] \Rightarrow False
```

**Using recursion:**

```haskell
any :: (a -> Bool) -> [a] -> Bool
any _ [] = False
any p (x:xs) = | p x = True |
  otherwise = any p xs
```

**Using composition:**

```haskell
any p = or . map p
```

```haskell
[1,0,3] map (==0) [False,True,False] or \Rightarrow 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!

**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, ...
Sample interaction:

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

**commaint in Haskell:**

```haskell
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.

**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.
Lambda Expressions

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

Examples:

\[
\begin{align*}
\text{squareAll } xs & = \text{map } (\lambda x \to x^2) \ xs \\
\text{length} & = \text{foldl'} (\lambda n x \to n+1) \ 0
\end{align*}
\]

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 $2 + (3 \times 3) = 11$.

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.

\[
\begin{align*}
? \text{mid } [1,2,3,4,5] & \Rightarrow [2,3,4] \\
? \text{mid } [] & \Rightarrow \text{ERROR} \\
? \text{mid } [1] & \Rightarrow \text{ERROR} \\
? \text{mid } [1,3] & \Rightarrow []
\end{align*}
\]