Class test

2:10–3:00pm Monday 22 October 2012
George Square Lecture Theatre

Past exams available on website
http://www.inf.ed.ac.uk/teaching/courses/inf1/fp/
Tutorials—extra tutorial

??–??pm Wednesday 17 October
Appleton Tower TBA

See course web page for Doodle poll to decide time.

Attempt the 2011 class test *in advance.*
*Print out* and bring your solutions.
Required text and reading

*Haskell: The Craft of Functional Programming (Third Edition),*
Simon Thompson, Addison-Wesley, 2011.

Reading assignment

<table>
<thead>
<tr>
<th>Date</th>
<th>Chapters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monday 24 September 2012</td>
<td>Chapters 1–3 (pp. 1–66)</td>
</tr>
<tr>
<td>Monday 1 October 2012</td>
<td>Chapters 4–7 (pp. 67–176)</td>
</tr>
<tr>
<td>Monday 8 October 2012</td>
<td>Chapters 8–9 (pp. 177–212)</td>
</tr>
<tr>
<td>Monday 15 October 2012</td>
<td>Chapters 10–12 (pp. 213–286)</td>
</tr>
<tr>
<td>Monday 22 October 2012</td>
<td><em>Class test</em></td>
</tr>
<tr>
<td>Monday 29 October 2012</td>
<td>Chapters 13–14 (pp. 287–356)</td>
</tr>
<tr>
<td>Monday 5 November 2012</td>
<td>Chapters 15–16 (pp. 357–414)</td>
</tr>
<tr>
<td>Monday 12 November 2012</td>
<td>Chapters 17–21 (pp. 415–534)</td>
</tr>
</tbody>
</table>
Part I

List comprehensions, revisited
Evaluating a list comprehension: generator

\[
[ x \times x \mid x \leftarrow [1..3] ]
\]

= 
\[
[ 1 \times 1 ] ++ [ 2 \times 2 ] ++ [ 3 \times 3 ]
\]

= 
\[
[ 1 ] ++ [ 4 ] ++ [ 9 ]
\]

= 
\[1, 4, 9\]
Evaluating a list comprehension: generator and filter

\[
\begin{align*}
[ x^2 | x \leftarrow [1..3], \text{odd } x ] &= \\
&= [ 1^2 | \text{odd 1} ] ++ [ 2^2 | \text{odd 2} ] ++ [ 3^2 | \text{odd 3} ] \\
&= [ 1 | \text{True} ] ++ [ 4 | \text{False} ] ++ [ 9 | \text{True} ] \\
&= [ 1 ] ++ [ ] ++ [ 9 ] \\
&= [1, 9]
\end{align*}
\]
Evaluating a list comprehension: two generators

\[
\begin{align*}
\{ (i,j) \mid i &\leftarrow [1..3], \ j \leftarrow [i..3] \} \\
&= \\
\{ (1,j) \mid j \leftarrow [1..3] \} ++ \\
\{ (2,j) \mid j \leftarrow [2..3] \} ++ \\
\{ (3,j) \mid j \leftarrow [3..3] \} \\
&= \\
\{ (1,1) \} ++ \{ (1,2) \} ++ \{ (1,3) \} ++ \\
\{ (2,2) \} ++ \{ (2,3) \} ++ \\
\{ (3,3) \} \\
&= \\
\{ (1,1), (1,2), (1,3), (2,2), (2,3), (3,3) \} 
\end{align*}
\]
Another example

\[
\begin{array}{c}
[ (i, j) \mid i \leftarrow [1..3], j \leftarrow [1..3], i \leq j ] \\
= \\
[ (1, j) \mid j \leftarrow [1..3], 1 \leq j ] ++ \n[ (2, j) \mid j \leftarrow [1..3], 2 \leq j ] ++ \n[ (3, j) \mid j \leftarrow [1..3], 3 \leq j ] \\
= \\
[(1,1)\mid1\leq1] ++ [(1,2)\mid1\leq2] ++ [(1,3)\mid1\leq3] ++ \n[(2,1)\mid2\leq1] ++ [(2,2)\mid2\leq2] ++ [(2,3)\mid2\leq3] ++ \n[(3,1)\mid3\leq1] ++ [(3,2)\mid3\leq2] ++ [(3,3)\mid3\leq3] \\
= \\
[(1,1)] ++ [(1,2)] ++ [(1,3)] ++ \n[] ++ [(2,2)] ++ [(2,3)] ++ \n[] ++ [] ++ [(3,3)] \\
= \\
[(1,1), (1,2), (1,3), (2,2), (2,3), (3,3)]
\end{array}
\]
Defining list comprehensions

\[ q ::= x \leftarrow l, \; q \mid b, \; q \mid * \]

\[ [ e \mid * ] \]
\[ = [ e ] \]
\[ [ e \mid x \leftarrow [ l_1, \ldots, l_n ], \; q ] \]
\[ = (\text{let } x = l_1 \text{ in } [ e \mid q ]) +++ \cdots +++ (\text{let } x = l_n \text{ in } [ e \mid q ]) \]
\[ [ e \mid b, \; q ] \]
\[ = \text{if } b \text{ then } [ e \mid q ] \text{ else } [] \]
Another example, revisited

\[
\begin{align*}
[ (i, j) &| i \leftarrow [1..3], j \leftarrow [1..3], i \leq j, \ast ] \\
= & \left[ (1, j) | j \leftarrow [1..3], 1 \leq j, \ast \right] ++ \\
& \left[ (2, j) | j \leftarrow [1..3], 2 \leq j, \ast \right] ++ \\
& \left[ (3, j) | j \leftarrow [1..3], 3 \leq j, \ast \right] \\
= & \left[ (1,1)|1\leq1, \ast \right] ++ \left[ (1,2)|1\leq2, \ast \right] ++ \left[ (1,3)|1\leq3, \ast \right] ++ \\
& \left[ (2,1)|2\leq1, \ast \right] ++ \left[ (2,2)|2\leq2, \ast \right] ++ \left[ (2,3)|2\leq3, \ast \right] ++ \\
& \left[ (3,1)|3\leq1, \ast \right] ++ \left[ (3,2)|3\leq2, \ast \right] ++ \left[ (3,3)|3\leq3, \ast \right] \\
= & \left[ (1,1)|\ast \right] ++ \left[ (1,2)|\ast \right] ++ \left[ (1,3)|\ast \right] ++ \\
& \left[ \right] ++ \left[ (2,2)|\ast \right] ++ \left[ (2,3)|\ast \right] ++ \\
& \left[ \right] ++ \left[ \right] ++ \left[ (3,3)|\ast \right] \\
= & \left[ (1,1) \right] ++ \left[ (1,2) \right] ++ \left[ (1,3) \right] ++ \\
& \left[ \right] ++ \left[ (2,2) \right] ++ \left[ (2,3) \right] ++ \\
& \left[ \right] ++ \left[ \right] ++ \left[ (3,3) \right] \\
= & \left[ (1,1), (1,2), (1,3), (2,2), (2,3), (3,3) \right]
\end{align*}
\]
Part II

Map
Squares

*Main> squares [1,-2,3]
[1,4,9]

squares :: [Int] -> [Int]
squares xs = [ x*x | x <- xs ]

squares :: [Int] -> [Int]
squares [] = []
squares (x:xs) = x*x : squares xs
Ords

```haskell
*Main> ords "a2c3"
[97,50,99,51]

ords :: [Char] -> [Int]
ords xs = [ ord x | x <- xs ]

ords :: [Char] -> [Int]
ords [] = []
ords (x:xs) = ord x : ords xs
```
Map

map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
Squares, revisited

*Main> squares [1,-2,3]
[1,4,9]

squares :: [Int] -> [Int]
squares xs = [ x*x | x <- xs ]

squares :: [Int] -> [Int]
squares [] = []
squares (x:xs) = x*x : squares xs

squares :: [Int] -> [Int]
squares xs = map sqr xs
    where
        sqr x = x*x
Map—how it works

map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]

map sqr [1,2,3]
=
  [ sqr x | x <- [1,2,3] ]
=
  [ sqr 1 ] ++ [ sqr 2 ] ++ [ sqr 3]
=
  [1, 4, 9]
Map—how it works

\[
\begin{align*}
\text{map} &:: (a \to b) \to [a] \to [b] \\
\text{map } f \ [\] &\quad = \ [\] \\
\text{map } f \ (x:xs) &\quad = \ f \ x \ : \ \text{map } f \ xs
\end{align*}
\]

\[
\begin{align*}
\text{map } \text{sqr} \ [1,2,3] &\quad = \\
&\quad = \ \text{map } \text{sqr} \ (1 \ : \ (2 \ : \ (3 \ : \ []))) \\
&\quad = \ \text{sqr } 1 \ : \ \text{map } \text{sqr} \ (2 \ : \ (3 \ : \ [])) \\
&\quad = \ \text{sqr } 1 \ : \ (\text{sqr } 2 \ : \ \text{map } \text{sqr} \ (3 \ : \ [])) \\
&\quad = \ \text{sqr } 1 \ : \ (\text{sqr } 2 \ : \ (\text{sqr } 3 \ : \ \text{map } \text{sqr} \ [])) \\
&\quad = \ \text{sqr } 1 \ : \ (\text{sqr } 2 \ : \ (\text{sqr } 3 \ : \ [])) \\
&\quad = \ 1 \ : \ (4 \ : \ (9 \ : \ [])) \\
&\quad = \ [1, \ 4, \ 9]
\end{align*}
\]
Ords, revisited

*Main> ords "a2c3"
[97,50,99,51]

ords :: [Char] -> [Int]
ords xs = [ ord x | x <- xs ]

ords :: [Char] -> [Int]
ords [] = []
ords (x:xs) = ord x : ords xs

ords :: [Char] -> [Int]
ords xs = map ord xs
Part III

Filter
Positives

*Main> positives [1,-2,3]
[1,3]

positives :: [Int] -> [Int]
positives xs = [ x | x <- xs, x > 0 ]

positives :: [Int] -> [Int]
positives [] = []
positives (x:xs) | x > 0 = x : positives xs
| otherwise = positives xs
Digits

*Main> digits "a2c3"
"23"

digits :: [Char] -> [Char]
digits xs = [ x | x <- xs, isDigit x ]

digits :: [Char] -> [Char]
digits [] = []
digits (x:xs) | isDigit x = x : digits xs
| otherwise = digits xs
Filter

\[ \text{filter} :: (a \to \text{Bool}) \to [a] \to [a] \]
\[ \text{filter} \ p \ \text{xs} \ = \ [ \ x \mid x \leftarrow \text{xs}, \ p \ x \] \]

\[ \text{filter} :: (a \to \text{Bool}) \to [a] \to [a] \]
\[ \text{filter} \ p \ [] \ = \ [] \]
\[ \text{filter} \ p \ (x:xs) \mid p \ x \ = \ x \ : \ \text{filter} \ p \ xs \]
\[ \mid \text{otherwise} \ = \ \text{filter} \ p \ xs \]
Positives, revisited

*Main> positives [1,-2,3]
[1,3]

positives :: [Int] -> [Int]
positives xs = [ x | x <- xs, x > 0 ]

positives :: [Int] -> [Int]
positives [] = []
positives (x:xs) | x > 0 = x : positives xs
                | otherwise = positives xs

positives :: [Int] -> [Int]
positives xs = filter pos xs
    where
        pos x = x > 0
Digits, revisited

*Main> digits "a2c3"
"23"

digits :: [Char] -> [Char]
digits xs = [ x | x <- xs, isDigit x ]

digits :: [Char] -> [Char]
digits [] = []
digits (x:xs) | isDigit x = x : digits xs
           | otherwise = digits xs

digits :: [Char] -> [Char]
digits xs = filter isDigit xs
Part IV

Fold
Sum

*Main> sum [1,2,3,4]
10

sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs
Product

*Main> product [1,2,3,4]
24

product :: [Int] -> Int
product [] = 1
product (x:xs) = x * product xs
Concatenate

*Main> concat [[1,2,3],[4,5]]
[1,2,3,4,5]

*Main> concat ["con","cat","en","ate"]
"concatenate"

concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ concat xss
Foldr

\[
\text{foldr :: (a -> a -> a) -> a -> [a] -> a}
\]
\[
\text{foldr } f \ a \ [] \ = \ a
\]
\[
\text{foldr } f \ a \ (x:x:s) \ = \ f \ x \ (\text{foldr } f \ a \ xs)
\]
Foldr, with infix notation

\[
\text{foldr} :: (a \to a \to a) \to a \to [a] \to a \\
\text{foldr } f \ a \ [\ ] \ = \ a \\
\text{foldr } f \ a \ (x:xs) \ = \ x \ 'f' \ (\text{foldr } f \ a \ xs)
\]
Sum, revisited

*Main> sum [1,2,3,4]
10

sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs

sum :: [Int] -> Int
sum xs = foldr (+) 0 xs

Recall that (+) is the name of the addition function, so x + y and (+) x y are equivalent.
Sum, Product, Concat

\[ \text{sum} :: [\text{Int}] \rightarrow \text{Int} \]
\[ \text{sum} \; \text{xs} \; = \; \text{foldr} \; (+) \; 0 \; \text{xs} \]

\[ \text{product} :: [\text{Int}] \rightarrow \text{Int} \]
\[ \text{product} \; \text{xs} \; = \; \text{foldr} \; (*) \; 1 \; \text{xs} \]

\[ \text{concat} :: [[\text{a}]] \rightarrow [\text{a}] \]
\[ \text{concat} \; \text{xs} \; = \; \text{foldr} \; (++ \text{r}) \; [] \; \text{xs} \]
Sum—how it works

```haskell
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs
```

```haskell
sum [1,2] =
  sum (1 : (2 : [])) =
    1 + sum (2 : []) =
      1 + (2 + sum []) =
        1 + (2 + 0) =
          3
```
Sum—how it works, revisited

\[
\text{foldr} :: (a \to a \to a) \to a \to [a] \to a
\]
\[
\text{foldr} f a [] = a
\]
\[
\text{foldr} f a (x:xs) = x \ 'f' \ (\text{foldr} f a xs)
\]

\[
\text{sum} :: [\text{Int}] \to \text{Int}
\]
\[
\text{sum} \ xs = \text{foldr} (\+) 0 \ xs
\]

\[
\text{sum} [1,2]
\]
\[
= \text{foldr} (\+) 0 [1,2]
\]
\[
= \text{foldr} (\+) 0 (1 : (2 : []))
\]
\[
= 1 + (\text{foldr} (\+) 0 (2 : []))
\]
\[
= 1 + (2 + (\text{foldr} (\+) 0 []))
\]
\[
= 1 + (2 + 0)
\]
\[
= 3
\]
Part V

Map, Filter, and Fold

All together now!
Sum of Squares of Positives

\[ f :: \text{[Int]} \rightarrow \text{Int} \]
\[ f \text{ } \text{xs} \text{ } = \text{ sum (squares (positives xs))} \]

\[ f :: \text{[Int]} \rightarrow \text{Int} \]
\[ f \text{ } \text{xs} \text{ } = \text{ sum [ } \text{x} \times \text{x} \text{ | } \text{x} \text{ } <- \text{ xs, } \text{x} \text{ } > \text{ 0 } \text{ ]} \]

\[ f :: \text{[Int]} \rightarrow \text{Int} \]
\[ f \text{ } [] \text{ } = \text{ [] } \]
\[ f \text{ } (\text{x} : \text{xs}) \]
\[ \text{ | x > 0 } \text{ } = \text{ (x} \times \text{x}) + f \text{ } \text{xs} \]
\[ \text{ | otherwise } \text{ } = \text{ f } \text{ } \text{xs} \]

\[ f :: \text{[Int]} \rightarrow \text{Int} \]
\[ f \text{ } \text{xs} \text{ } = \text{ foldr (+) 0 (map sqr (filter pos xs))} \]

where
\[ \text{sqr x } = \text{ x } \times \text{ x} \]
\[ \text{pos x } = \text{ x } > \text{ 0} \]
Part VI
Currying
How to add two numbers

```haskell
add :: Int -> Int -> Int
add x y = x + y

add 3 4
= 3 + 4
= 7
```
How to add two numbers

add :: Int -> (Int -> Int)
(add x) y = x + y

(add 3) 4
= 3 + 4
= 7

A function of two numbers
is the same as
a function of the first number that returns
a function of the second number.
Currying

```haskell
add :: Int -> (Int -> Int)
add x = g
    where
g y = x + y

(add 3) 4
= g 4
    where
g y = 3 + y
= 3 + 4
= 7
```

A function of two numbers
is the same as
a function of the first number that returns
a function of the second number.
Currying

\[
\text{add :: Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{add } x \ y \ = \ x \ + \ y
\]

means the same as

\[
\text{add :: Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \\
\text{add } x \ = \ g \\
\text{\hspace{1cm} where} \\
\text{\hspace{2cm} g } y \ = \ x \ + \ y
\]

and

\[
\text{add } 3 \ 4
\]

means the same as

\[
(\text{add } 3) \ 4
\]

This idea is named for \textit{Haskell Curry} (1900–1982).
It also appears in the work of \textit{Moses Schönfinkel} (1889–1942),
and \textit{Gottlob Frege} (1848–1925).
Putting currying to work

foldr :: (a -> a -> a) -> a -> [a] -> a
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)

sum :: [Int] -> Int
sum xs = foldr (+) 0 xs

is equivalent to

foldr :: (a -> a -> a) -> a -> ([a] -> a)
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)

sum :: [Int] -> Int
sum = foldr (+) 0
Compare and contrast

```
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs

sum [1,2,3,4]
= foldr (+) 0 [1,2,3,4]
```

```
sum :: [Int] -> Int
sum = foldr (+) 0

sum [1,2,3,4]
= (foldr (+) 0) [1,2,3,4]
```
Sum, Product, Concat

\[
\begin{align*}
\text{sum} & ::= [\text{Int}] \rightarrow \text{Int} \\
\text{sum} &= \text{foldr} \ (+) \ 0 \\
\text{product} & ::= [\text{Int}] \rightarrow \text{Int} \\
\text{product} &= \text{foldr} \ (\ast) \ 1 \\
\text{concat} & ::= [[\text{a}]] \rightarrow [\text{a}] \\
\text{concat} &= \text{foldr} \ (++) \ []
\end{align*}
\]