Lecture 3 – Higher-order functions

- Functions are first-class values that can be saved in local variables, passed as arguments, or returned as results.
- Higher-order functions take functions as arguments and/or return functions as results.
- Higher-order functions promote polymorphism and reuse.
- Higher-order functions are also combinators: they combine functions to create other functions.

Curried function

A function that takes multiple arguments one-at-a-time is a curried function.

```
f :: Int -> Int -> Int
f a b = a / b

g :: (Int, Int) -> Int
g (a, b) = a / b
```

f and g are equivalent except that f is a 2nd-order function that takes one argument at a time.

Curried function

Curried/uncurried functions can be converted using ‘curry’/‘uncurry’.

```
f a b = a / b

g (a, b) = a / b

f = curry g

g = uncurry f
```

Flipped function

'flip' can change the ordering of arguments.

```
f a b = a / b

h b a = a / b

h = flip f

f = flip h
```

Ordering of function application

Function application has higher precedence than other operators.
Function application is left associative

```
f a b = sqrt(a * a + b * b)

f a b = sqrt $ a * a + b * b

f 3 4 -- evaluates to 5
```

$ function allows function application to be right associative.

Function composition

Functions can be composed using compositon operator.

```
add1 x = x + 1
times2 x = x * 2

f = times2 . add1    -- equals to f x = times2 (add1 x)

f 10                 -- evaluates 22
```

Anonymous function

An anonymous function can take one or more argument and return a value.

```
f a b = let sq2 = \x y -> x * x + y * y
        in sqrt $ sq2 a b

f 3 4 -- evaluates to 5
```

The ‘sq2’ variable is given an anonymous function value.

Map function

We can map a function to each element of a list.

```
map _ [] = []
map f (a:b) = f a : map f b
```

map function applies a function f to each element of a list and returns the resulting list. *map* is a builtin function.

```
map (\x -> x + 1) [1,2,3] -- evaluates to [2,3,4]

incList = map (\x -> x + 1)

incList [1,2,3]  -- [2,3,4]

addList = map (\x y -> x + y)

addList [(1,2),(3,4),(5,6)] -- [3,7,11]
```

Fold function

We can fold a list by applying an accumulation function to each element of a list.

```
foldl _ c [] = c
foldl f c (a:b) = foldl f (f c a) b
```

foldl function applies a function f to the accumulative result and each element of a list from left to right and returns final result. *foldl* is a builtin function.

```
foldl (\c e -> c + e) 0 [1,2,3] -- evaluates to 6
foldl (+) 0 [1,2,3]             -- evaluates to 6

filter f = foldl (\c e -> if f e then c++[e] else c) []
filter (\x -> x > 2) [1,2,3,4]  -- evaluates to [3,4]
```

filter is also a builtin function.

Fold function

We can find the max element from a list using fold function.

```
maxList (a:b) = foldl max a b
```

We can test the membership of an element in a list recursively.

```
elem _ [] = False
elem x (a:b) = x == a || elem x b
```

Or we can use fold function.

```
elem x l = foldl (\c e -> c || e == x) False l
```

Element function

We can use *elem* to define other membership functions.

```
isUpperCase = flip elem ['A'..'Z']
isLowerCase = flip elem ['a'..'z']
```

We can use convert letters to upper/lower cases.

```
lookup' x ((a,b):c) = if x == a then b else lookup' x c

toLowerCase x
 | isUpperCase x = lookup' x pairs
 | otherwise = x
 where pairs = zip ['A'..'Z'] ['a'..'z']

toUpperCase x
 | isLowerCase x = lookup' x pairs
 | otherwise = x
 where pairs = zip ['a'..'z'] ['A'..'Z']

flipCase x
 | isLowerCase x = toUpperCase x
 | isUpperCase x = toLowerCase x
 | otherwise = x
```

Upper/lowercases function

We can use convert letters to upper/lower cases.

```
allCaps = map toUpperCase

allCaps "This is shouting!"       -- "THIS IS SHOUTING!"

noCaps = map toLowerCase

noCaps "This Is Unreadable!"      -- "this is unreadable!"

weird = map flipCase

weird "How Can You Read This?"    -- "hOW cAN yOU rEAD tHIS?"
```

Fold function

The *foldr* function applies an accumulative function from *right to left* and an initial value to a list.

```
foldr _ c [] = c
foldr f c (a:b) = f a $ foldr f c b
```

We can use foldr to concatenate two lists.

```
concat l1 l2 = foldr (:) l2 l1

[1,2,3] `concat` [4,5,6] -- evaluates to [1,2,3,4,5,6]
```

It is more efficient to implement filter using foldr.

```
filter f = foldr (\e c -> if f e then e:c else c) []
```

Quicksort revisited

The split function (in quicksort) can be replaced by calling filter.

```
quicksort [] = []
quicksort [x] = [x]
quicksort (pivot:x) =
  let split [] = ([], [])
      split (a:b) = if a < pivot
                      then (a:left, right)
                      else (left, a:right)
         where (left, right) = split b

      (lower, upper) = split x

  in quicksort lower ++ (pivot : quicksort upper)
```

New quicksort with filter.

```
quicksort [] = []
quicksort [x] = [x]
quicksort (pivot:x) =
  let (lower, upper) = (filter (\e -> e < pivot) x,
                        filter (\e -> e >= pivot) x)

  in quicksort lower ++ (pivot : quicksort upper)
```

Quicksort revisited

Anonymous function can be simplified if it's just partial application of another function.

```
quicksort [] = []
quicksort [x] = [x]
quicksort (pivot:x) =
  let (lower, upper) = (filter (\e -> e < pivot) x,
                        filter (\e -> e >= pivot) x)

  in quicksort lower ++ (pivot : quicksort upper)
```

No need to explicitly define lambdas.

```
quicksort [] = []
quicksort [x] = [x]
quicksort (pivot:x) =
  let (lower, upper) = (filter (< pivot) x,
                        filter (>= pivot) x)

  in quicksort lower ++ (pivot : quicksort upper)
```

Quicksort revisited

Anonymous function can be simplified if it's just partial application of another function.

```
quicksort [] = []
quicksort [x] = [x]
quicksort (pivot:x) =
  let (lower, upper) = (filter (< pivot) x,
                        filter (>= pivot) x)

  in quicksort lower ++ (pivot : quicksort upper)
```

Elements equal to pivot do not need to be sorted.

```
quicksort [] = []
quicksort [x] = [x]
quicksort y@(pivot:x) =
  let (lower, middle, upper) = (filter (< pivot) y,
                                filter (== pivot) y,
                                filter (> pivot) y)

  in quicksort lower ++ middle ++ quicksort upper
```

Quicksort revisited

The quicksort function can be made polymorphic by taking a comparison function.

```
quicksort _ [] = []
quicksort _ [x] = [x]
quicksort f (pivot:x) =
  let (lower, upper) = (filter (\e -> f e pivot) x,
                        filter (\e -> not (f e pivot)) x)

  in quicksort f lower ++ (pivot : quicksort f upper)
```

quicksort can sort in any order.

```
increasing_sort = quicksort (<)
decreasing_sort = quicksort (>)
```

Tail recursion

Factorial function performs multiplication after recursive call returns.

```
fact 0 = 1
fact n = n * fact(n-1)
```

Tail recursive function does not perform additional computation after recursive call completes.

```
fact' n = f n sofar
  where f 0 sofar = sofar
        f n sofar = f (n-1) (n*sofar)
```

Tail recursion

Straightforward implementation of reverse is not efficient (quadratic time).

```
reverse [] = []
reverse (a:b) = reverse b ++ [a]
```

Tail recursive reverse avoids list concatenation in list reversal (linear time).

```
reverse' x = rev x []
  where rev [] sofar = sofar
        rev (a:b) sofar = rev b (a:sofar)
```

Zip and unzip

We can zip two lists to form a list of pairs.

```
zip x [] = []
zip [] x = []
zip (a:b) (x:y) = (a,x) : zip' b y
```

We can split a list of pairs into two lists using unzip.

```
unzip [] = ([], [])
unzip ((a,b):c) = (a:x, b:y)
    where (x,y) = unzip' c
```

zip and *unzip* are builtin functions.
For example,

```
maxTwoLists l1 l2 = map (uncurry max) $ zip l1 l2

maxTwoLists [1,3,9] [2,4,6] -- evaluates to [2,4,9]
```

Prime number testing

An integer n is prime if $n \mod x \neq 0, \forall x \ni 2 \leq x \leq \lfloor\sqrt{n}\rfloor$.

```
isPrime n = foldl f True [2..limit]
            where f c e = c && n `mod` e /= 0
                  limit = floor $ sqrt n
```

Unfortunately, this doesn't work even though it compiles – n is inferred to be both an integer and a float. A simple fix is to use `fromIntegral` to cast n to a fraction before applying `sqrt`.

```
isPrime n = foldl f True [2..limit]
            where f c e = c && n `mod` e /= 0
                  limit = floor $ sqrt $ fromIntegral n
```

Generate prime numbers

We can generate a list of prime numbers by filtering an infinite list using `isPrime` function.

```
isPrime n = foldl f True [2..limit]
            where f c e = c && n `mod` e /= 0
                  limit = floor $ sqrt $ fromIntegral n

makePrime = filter isPrime [2..]
```

We cannot run `makePrime` directly because it will not terminate. However, we can take a range of the lists – first 10 prime numbers.

```
take 10 $ makePrime                 -- [2,3,5,7,11,13,17,19,23,29]
```

Or the 1001st to 1010th prime numbers.

```
take 10 $ drop 1000 makePrime
        -- [7927,7933,7937,7949,7951,7963,7993,8009,8011,8017]
```

Vector and matrix

We can represent vector with list and matrix with list of lists.

```
v1 = [1,2,3]
v2 = [1,1,1]
m1 = [v1,v2]
```

We can find the max element of a vector and matrix using fold.

```
maxVector (a:b) = foldl max a b
maxMatrix m =  maxVector $ map maxVector m

maxMatrix m1 -- evaluates to 3
```

Vector and matrix

We can print a vector with specified spacing using fold function.

```
showVector n v = foldr (++) [] $ map (pad . show)  v
  where pad t = t ++ (take (n - length t) $ repeat ' ')

putStrLn $ showVector 2 [1, 2, 3] -- print 1 2 3
```

We can print a matrix by printing each row in a new line.

```
showMatrix m = join $ map (showVector n) m
  where n = (length $ show $ maxMatrix m) + 1
        join (a:b) = a ++ (foldr (++) [] $ map ('\n':) b)

putStrLn $ showMatrix [[1,2,3],[1,1,1]] -- print 1 2 3
                                        --       1 1 1
```

Vector and matrix

Element-wise product of two vectors can be implemented using *zip* and *map*.

```
elementwiseProduct x y = map (uncurry (*)) $ zip x y

elementwiseProduct [1,2,3] [2,2,2]    -- evalutes to [2,4,6]
```

Inner product can be implemented using *foldl*.

```
innerProduct x y = foldl (+) 0 $ elementwiseProduct x y

innerProduct [1,2,3] [2,2,2]          -- evaluates to 12
```

Vector and matrix

Matrix times vector can be implemented using *map*.

```
matrixTimesVector matrix columnVector =
                     map (`innerProduct` columnVector) matrix

matrixTimesVector [[1,2,3],[1,1,1]] [2,2,2]       -- [12,6]
```

Matrix transposition is also implemented using *map*.

```
transpose matrix = map (getColumn matrix) [0..len]
  where getColumn matrix i = map (!! i) matrix
        len = length (matrix !! 0) - 1

transpose [[1,10],[2,20],[3,30]] --- [[1,2,3],[10,20,30]]
```

Vector and matrix

Matrix times matrix can be implemented using *map*.

```
matrixTimesMatrix m1 m2 =
    map (\row -> let f = innerProduct row in map f m) m1
  where m = transpose m2

m1 = [[1,2,3],[1,1,1]]
m2 = [[1,10],[2,20],[3,30]]

putStrLn $ showMatrix $ m1     -- 1 2 3
                               -- 1 1 1

putStrLn $ showMatrix $ m2     -- 1  10
                               -- 2  20
                               -- 3  30

putStrLn $ showMatrix $ m1 `matrixTimesMatrix` m2
                               -- 14  140
                               -- 6   60
```