7 - Higher-Order Functions

Introduction

A function is called **higher-order** if it takes a function as an argument or returns a function as a result.

```
twice :: (a → a) → a → a
twice f x = f (f x)
```

twice is higher-order because it takes a function as its first argument.

```
> twice (*2) 4
16
```

```
> twice reverse [1..5]
[1,2,3,4,5]
```

Why Are They Useful?

- *Common programming idioms* can be encoded as functions within the language itself.
- *Domain specific languages* can be defined as collections of higher-order functions.
- *Algebraic properties* of higher-order functions can be used to reason about programs.

The Map Function

The higher-order library function called **map** applies a function to every element of a list.

```
map :: (a → b) → [a] → [b]
```

For example:

```
> map (+1) [1,3,5,7]
[2,4,6,8]
```

```
> map reverse ["abc", “def"]
["cba","fed"]
```

```
> map (map (+1)) [[1..3],[4..6]]
[[2,3,4],[5,6,7]]
```

The map function can be defined in a particularly simple manner using a list comprehension:

```
map f xs = [f x | x ← xs]
```

Alternatively, for the purposes of proofs, the map function can also be defined using recursion:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

The Filter Function

The higher-order library function **filter** selects every element from a list that satisfies a predicate.

```
filter :: (a → Bool) → [a] → [a]
```

For example:

```
> filter even [1..10]
[2,4,6,8,10]
```

```
> filter (/= ' ') "abc def ghi"
"abcdefghi"
```

Filter can be defined using a list comprehension:

```
filter p xs = [x | x ← xs, p x]
```

Alternatively, it can be defined using recursion:

```
filter p [] = []
filter p (x:xs)
   | p x       = x : filter p xs
   | otherwise = filter p xs
```

Decide if all elements in a list satisfy a predicate

```
> all even [4..8]
False
> all even [2,4..8]
True
```

Decide if any element in a list satisfies a predicate

```
> any even [4..8]
True
```

Select elements in a list while they satisfy a predicate

```
> takeWhile even [2,4,6,8,9,10,12]
[2,4,6,8]
```

Remove elements from a list while they satisfy a predicate

```
> dropWhile even [2,4,6,8,9,10,12]
[9,10,12]
```

Exercises

Define the function which calculates the sum of the squares of all even numbers in a list

```
sumsqreven :: [Int] -> Int
sumsqreven xs = sum (map (^2) (filter even xs))
```

```
sumsqreven > sumsqreven [1..5]
20
```

The Foldr function

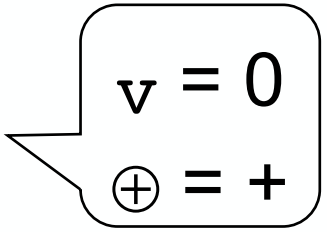
A number of functions on lists can be defined using the following simple pattern of recursion:

```
f []     = v
f (x:xs) = x ⊕ f xs
```

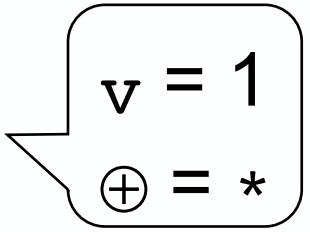
f maps the empty list to some value **v**, and any non-empty list to some function ⊕ applied to its head and f of its tail.

For example:

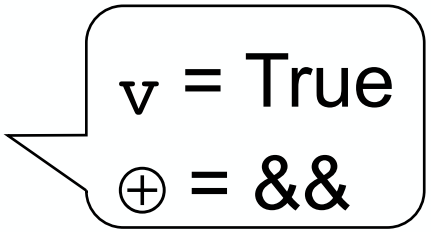
```
sum []     = 0
sum (x:xs) = x + sum xs
```




```
product []     = 1
product (x:xs) = x * product xs
```




```
and []     = True
and (x:xs) = x && and xs
```



The higher-order library function foldr (fold right) encapsulates this simple pattern of recursion, with the function $\oplus$ and the value v as arguments.

For example:

```
sum = foldr (+) 0

product = foldr (*) 1

or = foldr (||) False

and = foldr (&&) True
```

Foldr itself can be defined using recursion:

```
foldr :: (a → b → b) → b → [a] → b

foldr f v []     = v

foldr f v (x:xs) = f x (foldr f v xs)
```

However, it is best to think of foldr *non-recursively*, as simultaneously replacing each (:) in a list by a given function, and [] by a given value.

For example:

```
  sum [1,2,3]
=
  foldr (+) 0 [1,2,3]
=
  foldr (+) 0 (1:(2:(3:[])))
=
  1+(2+(3+0))
=
  6
```

Replace each (:)
by (+) and [] by 0.

For example:

```
  product [1,2,3]
=
  foldr (*) 1 [1,2,3]
=
  foldr (*) 1 (1:(2:(3:[])))
=
  1*(2*(3*1))
=
  6
```

Replace each (:) by (*) and [] by 1.

Other Foldr Examples

Even though **foldr** encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected.

Recall the length function:

```
length :: [a] → Int
length []     = 0
length (_:xs) = 1 + length xs
```

For example:

```
  length [1,2,3]
=
  length (1:(2:(3:[])))
=
  1+(1+(1+0))
=
  3
```

Replace each (:) by λ_ n → 1+n and [] by 0.

Hence, we have:

```
length = foldr (λ_ n → 1+n) 0
```

Now recall the reverse function:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

For example:

```
  reverse [1,2,3]
=
  reverse (1:(2:(3:[])))
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

Replace each (:) by λx xs → xs ++ [x] and [] by [].

Hence, we have:

```
reverse = foldr (λx xs → xs ++ [x]) []
```

Finally, we note that the append function (++) has a particularly compact definition using foldr:

```
(++ ys) = foldr (:) ys
```

Replace each (:) by (:) and [] by ys.

Why Is Foldr Useful?

Some recursive functions on lists, such as sum, are *simpler* to define using foldr.

Properties of functions defined using foldr can be proved using algebraic properties of foldr, such as *fusion* and the *banana split* rule.

Advanced program *optimisations* can be simpler if foldr is used in place of explicit recursion.

The Foldl function

foldr assumes an operator which associates to the right

```
foldr (#) v [x0,x1,…,xn] =
                x0 # (x1 # (… ( xn # v ) … ))
```

We can define recursive functions with an operator which associates to the left

```
sum :: Num a => t a -> a
sum = sum' 0
   where sum' v [] = v
         sum' v (x:xs) = sum' (v + x) xs
```

The Foldl function

A number of functions on lists can be defined using the following simple pattern of recursion:

```
f v []     = v
f v (x:xs) = f (v ⊕ x) xs
```

foldl itself can be defined using recursion:

```
foldl :: (a → b → a) → a → [b] → a

foldl f v []     = v

foldl f v (x:xs) = foldl f (f v x) xs
```

foldl can be summarised similarly to foldr

```
foldl (#) v [x0,x1,…,xn] =
                (… (( v # x0) # x1) …) # xn
```

Functions defined using this pattern of recursion:

```
sum = foldl (+) 0

product = foldl (*) 1

or = foldl (||) False

and = foldl (&&) True
```

```
length = foldl (λn _ → n+1) 0
```

```
reverse = foldl (λxs x → x:xs) []
```

Function composition

The library function (.) returns the **composition** of two functions as a single function.

```
(.) :: (b → c) → (a → b) → (a → c)
f . g = λx → f (g x)
```

For example:

```
odd :: Int → Bool
odd = not . even
```

```
sumsqreven :: [Int] -> Int
sumsqreven = sum . map (^2) . filter even
```

Exercises

- Express the comprehension [f x | x ← xs, p x] using the functions map and filter.

```
map f . filter p
```

Exercises

Define the function **all** which decides if every element of a list satisfies a given predicate.

```
> all even [2,4,6,8,10]
True
```

```
all :: (a → Bool) → [a] → Bool
```

```
all p xs = and [p x | x <- xs]
```

```
all p [] = True
all p (x:xs) = p x && all p xs
```

```
all p = and . map p
```

```
all p = null . filter (not.p)
```

```
all p = foldr (\x y -> p x && y) True
```

Exercises

Dually, the library function **any** decides if at least one element of a list satisfies a predicate.

```
> any (== ' ') "abc def"
True
```

```
any :: (a → Bool) → [a] → Bool
```

```
any p [] = True
any p (x:xs) = p x || any p xs
```

```
any p xs = or [p x | x <- xs]
```

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

```
any p = not . null . filter p
```

```
any p = foldr (\x y -> p x || y) True
```

Exercises

Redefine `map f` and `filter p` using `foldr`.

```
map' :: (a -> b) -> [a] -> [b]
map' f = foldr (\x xs -> f x : xs) []
```

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x xs -> if p x then x:xs
                                   else xs)
                  []
```

Exercises

Define **takeWhile** which selects elements from a list while a predicate holds of all the elements.

```
> takeWhile (/= ' ') "abc def"
"abc"
```

```
takeWhile :: (a → Bool) → [a] → [a]
```

```
takeWhile p [] = []
takeWhile p (x:xs)
    | p x       = x : takeWhile p xs
    | otherwise = []
```

```
takeWhile p = foldr
                  (\x xs -> if p x then x:xs
                                   else [])
                  []
```

Exercises

Dually, the function **dropWhile** removes elements while a predicate holds of all the elements.

```
> dropWhile (== ' ') "   abc"
"abc"
```

```
dropWhile :: (a → Bool) → [a] → [a]
```

```
dropWhile p [] = []
dropWhile p (x:xs)
    | p x       = dropWhile p xs
    | otherwise = x:xs
```

Exercises

Using `foldl` define a function that converts a decimal number into an integer:

```
> dec2int [2,3,4,5]
2345
```

```
dec2int :: [Int] -> Int
dec2int = foldl (\y x -> x + (10 * y)) 0
```

Exercises

Define `curry` and `uncurry` which curryfies, respectively decurryfies a function:

```
> uncurry (+) (1,2)
3
```

```
uncurry' :: (a -> b -> c) -> (a, b) -> c
uncurry' f = g
             where g (a,b) = f a b
uncurry'' f = \(a, b) -> f a b
```

```
curry' :: ((a, b) -> c) -> a -> b -> c
curry' f = g
           where g a b = f (a,b)
curry'' f = \a b -> f (a, b)
```

Exercises

Define `altMap` which applies alternatively 2 functions to the elements of a list:

```
> altMap (+1) (+2) [1,1,1,1,1,1]
[2,3,2,3,2,3]
```

```
altMap :: (a -> b) -> (a -> b) -> [a] -> [b]
altMap f h [] = []
altMap f h (x:xs) =  f x : altMap h f xs
```