Recursion and Higher-Order Functions

Stephen A. Edwards

Columbia University

Fall 2020

Recursion in Haskell

Pattern matching works nicely:

recfun <base case> = <base value>
recfun <part> <rest> = <some work> <part> <combined with> recfun <rest>

```haskell
maximum'          :: Ord a => [a] -> a
maximum' []       = error "empty list"
maximum' [x]      = x                   -- base case
maximum' (x:xs)
   | x > maxTail = x                    -- found a new maximum
   | otherwise   = maxTail
  where maxTail = maximum' xs           -- recurse
```

The list elements need to be ordered so we can perform > on them

maximum is part of the standard prelude; you do not need to write this

Maximum

Far better: build the solution out of helpful pieces, even if they are small. It is efficient; GHC aggressively inlines code to avoid function call overhead

```
max'             :: Ord a => a -> a -> a
max' a b
  | a > b       = a
  | otherwise = b

maximum'         :: Ord a => [a] -> a
maximum' []      = error "empty list"
maximum' [x]     = x
maximum' (x:xs) = x `max'` maximum' xs
```

This is still twice as complicated as it needs to be; we'll revisit this later

Replicate and Take

```
replicate'     :: (Num n, Ord n) => n -> a -> [a]
replicate' n x
   | n <= 0     = []
   | otherwise = x : replicate' (n-1) x
```

The Num typeclass (–) does not include Ord (for <=), so Ord is needed

Used a guard since we're testing a condition `n <= 0` rather than a constant.

```
take'                  :: (Num n, Ord n) => n -> [a] -> [a]
take' n _ | n <= 0 = []                       -- base case
take' _ []             = []                   -- base case
take' n (x:xs)         = x : take' (n-1) xs   -- recurse
```

Replicate and Take Revisited

The Standard Prelude implementation uses infinite lists

```
take'                  :: (Num n, Ord n) => n -> [a] -> [a]
take' n _ | n <= 0     = []
take' _ []             = []
take' n (x:xs)         = x : take' (n-1) xs

repeat'   :: a -> [a]
repeat' x = xs where xs = x : xs             -- Infinite list

replicate'     :: (Num n, Ord n) => n -> a -> [a]
replicate' n x = take' n (repeat' x)
```

Zip: Combine Two Lists Into a List of Pairs

```
zip'                 :: [a] -> [b] -> [(a,b)]
zip'    []      _    = []
zip'    _       []   = []
zip' (x:xs) (y:ys) = (x,y) : zip' xs ys
```

Works nicely with lists of mismatched lengths, including infinite:

```
*Main> zip' [0..3] [1..5] :: [(Int, Int)]
[(0,1),(1,2),(2,3),(3,4)]

*Main> zip' "abc" ([1..]  :: [Int])
[('a',1),('b',2),('c',3)]
```

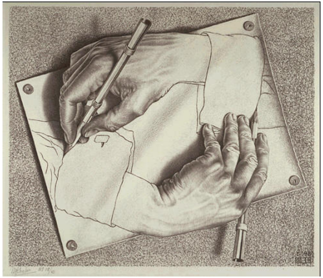
Quicksort in Haskell

- Pick and remove a pivot
- Partition into two lists: smaller or equal to and larger than pivot
- Recurse on both lists
- Concatenate smaller, pivot, then larger

```
quicksort          :: Ord a => [a] -> [a]
quicksort []       = []
quicksort (p:xs) = quicksort [x | x <- xs, x <= p] ++
                   [p] ++
                   quicksort [x | x <- xs, x > p]
```

Efficient enough: `++` associates to the right so `a ++ b ++ c` is `(a ++ (b ++ c))`

Using Recursion in Haskell



Haskell does not have classical *for* or *do* loops

Recursion can implement either of these plus much more. Tail-recursion is just as efficient as such loops

Most of the time, however, your loop or recursive function fits a well-known pattern that is already in a Standard Prelude function that you should use instead

A key advantage of functional languages, including Haskell, is that you can build new control constructs

Partially Applied Functions

The (+) syntax also permits a single argument to be applied on either side and returns a function that takes the “missing” argument:

```
Prelude> (++ ", hello") "Stephen"
"Stephen, hello"
Prelude> ("Hello, " ++) "Stephen"
"Hello, Stephen"
Prelude> (<= (5::Int)) 10
False
Prelude> (<= (5::Int)) 5
True
Prelude> (<= (5::Int)) 4
True
```

– is weird because (-4) means negative four. Use `subtract`:

```
Prelude> (subtract 4) 10
6
```

Higher-Order Functions

Passing functions as arguments is routine yet powerful

```
Prelude> :{
Prelude| applyTwice :: (a -> a) -> a -> a
Prelude| applyTwice f x = f (f x)
Prelude| :}

Prelude> applyTwice (+5) 1
11
Prelude> applyTwice (++ " is stupid") "Stephen"
"Stephen is stupid is stupid"
```

“applyTwice takes a function and return a function that takes a value and applies the function to the value twice”

Flip

Standard Prelude function that reverses the order of the first arguments

```
flip'    :: (a -> b -> c) -> (b -> a -> c)
flip' f = g where g x y = f y x
```

But since the “function type” operator –> associates right-to-left,

```
flip'        :: (a -> b -> c) -> b -> a -> c
flip' f x y = f y x
```

```
Prelude> zip [1..5] "Hello"
[(1,'H'),(2,'e'),(3,'l'),(4,'l'),(5,'o')]
Prelude> flip zip [1..5] "Hello"
[('H',1),('e',2),('l',3),('l',4),('o',5)]
Prelude> zipWith (flip div) [2,2..] [10,8..2]
[5,4,3,2,1]
```

Map: A Foundation of Functional Programming

A Standard Prelude function. Two equivalent ways to code it:

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

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

```
*Main> map (+5) ([1..5] :: [Int])
[6,7,8,9,10]
*Main> map (++ "!") ["BIFF","BAM","POW"]
["BIFF!","BAM!","POW!"]
```

You've written many loops that fit *map* in imperative languages

zipWith

Another Standard Prelude function *zipWith* takes a function and two lists and applies the function to the list elements, like a combination of *zip* and *map*:

```
zipWith'                 :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ []      _      = []
zipWith' _ _       []     = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

```
Prelude> zipWith (+) [1..5] [10,20..] :: [Int]
[11,22,33,44,55]
```

The Standard Prelude implements *zip* with *zipWith*

```
zip' :: [a] -> [b] -> [(a,b)]
zip' = zipWith (,)    -- the "make-a-pair" operator
```

Filter: Select each element of a list that satisfies a predicate

```
filter                        :: (a -> Bool) -> [a] -> [a]
filter _ []                   = []
filter p (x:xs) | p x         = x : filter p xs
                | otherwise = filter p xs
```

```
filter       :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]
```

```
Prelude> filter (>= 3) [1..10] :: [Int]
[3,4,5,6,7,8,9,10]
```

What's the largest number under 100,000 that's divisible by 3,829?

```
Prelude> x `divides` y = y `mod` x == 0
Prelude> head (filter (3829 `divides`) [100000,99999..])
99554
```

Quicksort Revisited

Using *filter* instead of list comprehensions:

```
quicksort          :: Ord a => [a] -> [a]
quicksort []        = []
quicksort (p:xs) = quicksort (filter (<= p) xs) ++ [p] ++
                     quicksort (filter (> p)  xs)
```

Similar performance; choose the one that's easier to understand

takeWhile: Select the first elements that satisfy a predicate

Same type signature as *filter*, but stop taking elements from the list once the predicate is false. Also part of the Standard Prelude

```
takeWhile'                          :: (a -> Bool) -> [a] -> [a]
takeWhile' _ []                     = []
takeWhile' p (x:xs) | p x           = x : takeWhile' p xs
                    | otherwise     = []
```

```
Prelude> takeWhile (/= ' ') "Word splitter function"
"Word"
```

What’s the sum of all odd squares under 10,000?

```
Prelude> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650
Prelude> sum (takeWhile (<10000) [ n^2 | n <- [1..], odd (n^2) ])
166650
```

Twin Primes

Twin Primes differ by two, e.g., 3 and 5, 11 and 13, etc.

```
Prelude> primes = f [2..] where
Prelude|   f (p:xs) = p : f [ x | x <- xs, x `mod` p /= 0 ]

Prelude> twinPrimes = filter twin (zip primes (tail primes)) where
Prelude|   twin (a,b) = a+2 == b

Prelude> take 7 twinPrimes
[(3,5),(5,7),(11,13),(17,19),(29,31),(41,43),(59,61)]

Prelude> length twinPrimes
```

(Left as an exercise for the reader)

Collatz sequences

For starting numbers between 1 and 100, how many Collatz sequences are longer than 15?

```
collatz                      :: Int -> [Int]
collatz 1                    = [1]
collatz n | even n           = n : collatz (n `div` 2)
          | otherwise        = n : collatz (n * 3 + 1)

numLongChains :: Int
numLongChains = length (filter isLong (map collatz [1..100]))
  where isLong xs = length xs > 15
```

```
*Main> collatz 30
[30,15,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1]
*Main> numLongChains
66
```

Lambda Expressions

A *lambda expression* is an unnamed function. \ is a λ missing a leg:

\ <args> -> <expr>

Things like `(+ 5)` and `max 5` are also unnamed functions, but the lambda syntax is more powerful

Without a Lambda expression:

```
numLongChains = length (filter isLong (map collatz [1..100]))
  where isLong xs = length xs > 15
```

Using Lambda:

```
numLongChains = length (filter (\xs -> length xs > 15)
                               (map collatz [1..100]))
```

Lambda Expressions

Multiple and pattern arguments:

```
Prelude> zipWith (\a b -> a * 100 + b) [5,4..1] [1..5]
[501,402,303,204,105]
Prelude> map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
[3,8,9,8,7]
```

Function definitions are just convenient shorthand for Lambda expressions:

```
addThree :: Num a => a->a->a->a
addThree x y z = x + y + z
```

```
addThree :: Num a => a->a->a->a
addThree = \x -> \y -> \z ->
                    x + y + z
```

Some Lambdas are unncessary:

```
Prelude> zipWith (\x y -> x + y) [1..5] [100,200..500]
[101,202,303,404,505]
Prelude> zipWith (+) [1..5] [100,200..500]
[101,202,303,404,505]
```

Fold: Another Foundational Function

Apply a function to each element to accumulate a result:

$$\texttt{foldl}\ f\ z\ [a_1, a_2, \ldots, a_n] = f\ (\cdots(f\ (f\ z\ a_1)\ a_2)\cdots)\ a_n$$

```
foldl             :: (a -> b -> a) -> a -> [b] -> a
foldl f z []      =  z
foldl f z (x:xs)  =  foldl f (f z x) xs
```

```
Prelude> 0 + 1 + 2 + 3 + 4 + 5
15
Prelude> foldl (\acc x -> acc + x) 0 [1..5]
15
Prelude> foldl (+) 0 [1..5]
15
```

```
sum :: Num a -> [a] -> a
sum = foldl (+) 0          -- Standard Prelude definition
```

Foldl† in action

```
foldl              :: (a -> b -> a) -> a -> [b] -> a
foldl f z []        =  z
foldl f z (x:xs)    =  foldl f (f z x) xs
```

```
foldl f 100 [1..3] where f = \z x -> z + x -- a.k.a. (+)
  = foldl f 100         [1,2,3] -- Evaluate foldl: apply f to z and x
  = foldl f (f 100 1)     [2,3] -- Evaluate f: add z and x
  = foldl f 101           [2,3]
  = foldl f (f 101 2)       [3]
  = foldl f 103             [3]
  = foldl f (f 103 3)        []
  = foldl f 106              [] -- Base case: return z
  = 106
```

† Technically, this is `foldl'` in action; this gives the same result.

foldl1: foldl starting from the first element

```
foldl              :: (a -> b -> a) -> a -> [b] -> a
foldl f z []       =  z
foldl f z (x:xs)   =  foldl f (f z x) xs

foldl1             :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)    =  foldl f x xs           -- Start with the list's head
foldl1 _ []        =  error "Prelude.foldl1: empty list"
```

foldl vs. foldr

`foldl` from the left; `foldr` from the right. Function's arguments reversed

$$\texttt{foldl}\ f\ z\ [a_1, a_2, \ldots, a_n] = f\ (\cdots(f\ (f\ z\ a_1)\ a_2)\cdots)\ a_n$$

$$\texttt{foldr}\ f\ z\ [a_1, a_2, \ldots, a_n] = f\ a_1\ (f\ a_2\ (\cdots(f\ a_n\ z))\cdots)$$

```
foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     =  z
foldl f z (x:xs) =  foldl f (f z x) xs    -- f = \acc x -> ...
```

```
foldr            :: (b -> a -> a) -> a -> [b] -> a
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)    -- f = \x acc -> ...
```

Folds Are Extremely Powerful: They're Everywhere

```
concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss

reverse :: [a] -> [a]
reverse = foldl (\a x -> x : a) []  -- Lambda expression version
reverse = foldl (flip (:)) []       -- Prelude definition

and, or :: [Bool] -> Bool
and     =  foldr (&&) True
or      =  foldr (||) False

sum, product :: (Num a) => [a] -> a
sum          =  foldl (+) 0
product      =  foldl (*) 1

maximum, minimum :: Ord a => [a] -> a
maximum []       =  error "Prelude.maximum: empty list"
maximum xs       =  foldl1 max xs

minimum []       =  error "Prelude.minimum: empty list"
minimum xs       =  foldl1 min xs
```

Folds Subsume *map* and *filter*

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

A left fold also works, but is less efficient because of ++:

```haskell
map' f xs = foldl (\acc x -> acc ++ [f x]) [] xs
```

Filter is like a conditional *map*

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

The Standard Prelude uses the recursive definitions of *map* and *filter*

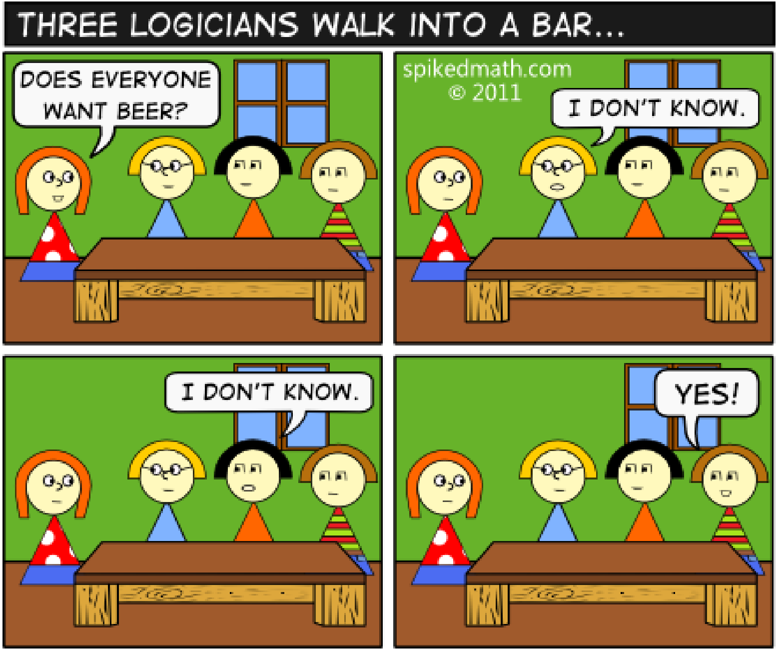
Foldr Evaluates Left-to-Right Because Haskell is Lazy

Haskell's *undefined* throws an exception only when it is evaluated

```
undefined :: a
undefined = error "Prelude.undefined"
```

$$\text{foldr}\ f\ z\ [a_1, a_2, \ldots, a_n] = f\ a_1\ (f\ a_2 (\cdots (f\ a_n\ z)) \cdots)$$

```
Prelude> quitZero x acc = if x == 0 then 0 else x + acc
Prelude> foldr quitZero 0 [3,2,1,0]
6
Prelude> foldr quitZero 0 [3,2,1,0,100]
6
Prelude> foldr quitZero 0 [3,2,1,undefined]
*** Exception: Prelude.undefined
Prelude> foldr quitZero 0 [3,2,1,0,undefined]
6
```


THREE LOGICIANS WALK INTO A BAR...
DOES EVERYONE WANT BEER?
spikedmath.com © 2011
I DON'T KNOW.
I DON'T KNOW.
YES!

&& and || are Short-Circuit Operators

```
(&&), (||) :: Bool -> Bool -> Bool
True  && x = x
False && _ = False
True  || _ = True
False || x = x
```

```
and, or :: [Bool] -> Bool
and     = foldr (&&) True
or      = foldr (||) False
```

```
Prelude> or [True, True, undefined]
True
Prelude> and [True, True, undefined]
*** Exception: Prelude.undefined
Prelude> and [True, False, undefined]
False
Prelude> or [False, True, undefined]
True
Prelude> or [False, False, undefined]
*** Exception: Prelude.undefined
```

Foldl Evaluates Left-to-Right Because of Laziness

```
foldl              :: (a -> b -> a) -> a -> [b] -> a
foldl f z []       =  z                       -- (base)
foldl f z (x:xs)   =  foldl f (f z x) xs      -- (recurse)
```

```
foldl f 100 [1..3]
  where f = \z x -> z + x                   -- (f)
  = foldl f           100      [1,2,3]      -- expand range
  = foldl f        (f 100 1)     [2,3]      -- (recurse)
  = foldl f     (f (f 100 1) 2)    [3]      -- (recurse)
  = foldl f (f (f (f 100 1) 2) 3) []        -- (recurse)
  =           f (f (f 100 1) 2) 3           -- (base)
  =           (f (f 100 1) 2) + 3           -- (f)
  =              (f 100 1) + 2 + 3          -- (f)
  =                100 + 1 + 2 + 3          -- (+)
  =                    101 + 2 + 3          -- (+)
  =                        103 + 3          -- (+)
  =                            106          -- (+)
```

Scanl and Scanr: Fold Remembering Accumulator Values

```haskell
scanl             :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q xs      = q : (case xs of []   -> []
                                    x:xs -> scanl f (f q x) xs)

scanr             :: (b -> a -> a) -> a -> [b] -> [a]
scanr f q0 []     =  [q0]
scanr f q0 (x:xs) =  f x q : qs where qs@(q:_) = scanr f q0 xs
```

```
Prelude> foldl (+) 0 [1..5]
15
Prelude> scanl (+) 0 [1..5]
[0,1,3,6,10,15]
Prelude> scanr (+) 0 [1..5]
[15,14,12,9,5,0]
```

Scanl and takeWhile Can Mimic a Do Loop

How many square roots added together just exceed 1000?

```
Prelude> length (takeWhile (<1000) (scanl1 (+) (map sqrt [1..])))
130
Prelude> sum (map sqrt [1..130])
993.6486803921487
Prelude> sum (map sqrt [1..131])
1005.0942035344083
```

Avoiding LISP† with $

Many functions put their complex-to-compute arguments at the end; applying these in sequence give expressions of the form f ... (g .... (h ...))

Use $ to eliminate the ending parentheses. It is right-associative at the lowest precedence so `f $ g $ h x` is `f (g (h x))`

Normal argument application (juxtaposition) is at the highest precedence

```
infixr 0 $     -- Right-associative, lowest precedence
($)   :: (a -> b) -> a -> b
f $ x = f x
```

```
Prelude> length (takeWhile (<1000) (scanl1 (+) (map sqrt [1..])))
130
Prelude> length $ takeWhile (<1000) $ scanl1 (+) $ map sqrt [1..]
130
```

† Lots of Irritating, Silly Parentheses

Applying an Argument as a Function

$ is the *function application* operator: it applies the function on its left to the argument on its right

Juxataposition does the same thing without an explicit operator

```
Prelude> map ($ 3) [ (4+), (10*), (^2), sqrt ]
[7.0,30.0,9.0,1.7320508075688772]
```

`($ 3)` is the "apply 3 as an argument to the function" function, equivalent to `\f -> f 3`.

Function Composition

In math notation, $(f \circ g)(x) = f(g(x))$; in Haskell,

```
infixr 9 .    -- Right-associative, highest precedence
(.)     :: (b -> c) -> (a -> b) -> a -> c
f . g   =  \ x -> f (g x)
```

So `(f . g . h) x` is `(f (g (h x)))`

```
Prelude> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
Prelude> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Best used when constructing functions to pass as an argument