Programming In Haskell Chapter 7

CS 1JC3

Recap: Iterating through a List

Function's that iterate through a list using recursion generally use the basic pattern matching scheme

```
rec_func [] = ...                          -- Base Case
rec_func (x:xs) = ... rec_func xs          -- Recursive Step
```

Defining Product

Often, **Recursion** and **Pattern Matching** are combined to define functions working on **lists**

```
product        :: [Int] -> Int
product []      = 1
product (n:ns) = n * product ns
```

Defining Product

Example Evaluation:

- product [2,3,4]

Defining Product

Example Evaluation:

- product [2,3,4]
- $= 2 * \text{product } [3,4]$

Defining Product

Example Evaluation:

- product [2,3,4]
- $= 2 * \text{product } [3,4]$
- $= 2 * (3 * \text{product } [4])$

Defining Product

Example Evaluation:

- product [2,3,4]
- $= 2 * \text{product } [3,4]$
- $= 2 * (3 * \text{product } [4])$
- $= 2 * (3 * (4 * \text{product } []))$

Defining Product

Example Evaluation:

- product [2,3,4]
- $= 2 * \text{product } [3,4]$
- $= 2 * (3 * \text{product } [4])$
- $= 2 * (3 * (4 * \text{product } []))$
- $= 2 * (3 * (4 * 1))$

Defining Product

Example Evaluation:

- product [2,3,4]
- $= 2 * \text{product } [3,4]$
- $= 2 * (3 * \text{product } [4])$
- $= 2 * (3 * (4 * \text{product } []))$
- $= 2 * (3 * (4 * 1))$
- $= 24$

Defining Reverse

Using a similar pattern of recursion we can define the **reverse** function on lists.

Defining Reverse

Using a similar pattern of recursion we can define the **reverse** function on lists.

```
reverse          :: [a] -> [a]
reverse []       = []
reverse (x:xs) = reverse xs ++ [x]
```

Multiple Arguments

Functions with more than one argument can also be defined using recursion. For Example:

- Defining **zip**

Multiple Arguments

Functions with more than one argument can also be defined using recursion. For Example:

- Defining **zip**

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

Multiple Arguments

- Defining **drop**

Multiple Arguments

- Defining **drop**

```
drop                :: Int -> [a] -> [a]
drop 0      xs       = xs
drop  _     []       = []
drop  n (_:xs) = drop (n-1) xs
```

- Defining (++)

Multiple Arguments

- Defining **drop**

```
drop                :: Int -> [a] -> [a]
drop 0      xs       = xs
drop  _     []       = []
drop  n (_:xs) = drop (n-1) xs
```

- Defining (++)

```
(++)            :: [a] -> [a] -> [a]
[]      ++  ys = ys
(x:xs) ++  ys = x : (xs ++ ys)
```

Multiple Arguments

- Define filter (takes a Boolean function and uses it to *filter* elements of a list)

Multiple Arguments

- Define filter (takes a Boolean function and uses it to *filter* elements of a list)

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

- Define partition (takes a Boolean function and uses it to separate a list into two)

Multiple Arguments

- Define filter (takes a Boolean function and uses it to *filter* elements of a list)

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

- Define partition (takes a Boolean function and uses it to separate a list into two)

```
partition :: (a -> Bool) -> [a] -> ([a],[a])
partition include xs = let
     ys = filter include xs
     zs = filter (not . include) xs
  in (ys,zs)
```

BubbleSort

- The bubble sort works by iterating through a list and swapping adjacent elements when they are out of order

BubbleSort

- The bubble sort works by iterating through a list and swapping adjacent elements when they are out of order

```
bubble :: (Ord a) => [a] -> [a]
bubble (y0:y1:ys)
  | y0 > y1    = y1 : (bubble (y0:ys))
  | otherwise  = y0 : (bubble (y1:ys))
bubble ys      = ys
```

BubbleSort

- The bubble sort works by iterating through a list and swapping adjacent elements when they are out of order

```haskell
bubble :: (Ord a) => [a] -> [a]
bubble (y0:y1:ys)
   | y0 > y1    = y1 : (bubble (y0:ys))
   | otherwise = y0 : (bubble (y1:ys))
bubble ys       = ys
```

- If you repeat this length(xs) times (where xs is the list you're sorting), the list will be sorted

Useless Trivia: The algorithm is called bubble sort because of the way smaller bubbles gradually float on top of bigger ones

BubbleSort

Example Evaluation:

```
  bubble [5,1,4,2]
= bubble (5:1:[4,2])
= 1 : bubble (5:4:[2])
= 1 : 4 : bubble(5:2:[])
= 1 : 4 : 2 : bubble (5:[])
= 1 : 4 : 2 : [5]
= [1,4,2,5]
```

Note: the list still isn't sorted, the only guarantee is that the smallest element is up front

BubbleSort

The Bubble Sort can be implemented by repeating the bubble function until a fixed point

BubbleSort

The Bubble Sort can be implemented by repeating the bubble function until a fixed point

```
bubbleSort :: (Ord a, Eq a) => [a] -> [a]
bubbleSort [] = []
bubbleSort xs = let
     bubble (y0:y1:ys)
       | y0 > y1    = y1 : (bubble (y0:ys))
       | otherwise = y0 : (bubble (y1:ys))
     bubble ys      = ys

     xs' = bubble xs
  in if xs == xs' then xs' else bubbleSort xs'
```

Notes on BubbleSort

- Extremely Inefficient, never use the bubble sort in practice

Notes on BubbleSort

- Extremely Inefficient, never use the bubble sort in practice
- Considered the most *"intuitive"* sorting algorithm, usually taught as an intro to sorting algorithms

Notes on BubbleSort

- Extremely Inefficient, never use the bubble sort in practice
- Considered the most *"intuitive"* sorting algorithm, usually taught as an intro to sorting algorithms
- Perhaps less intuitive in functional languages, due to their nature of encouraging divide and conquer style code

Notes on BubbleSort

- Extremely Inefficient, never use the bubble sort in practice
- Considered the most *"intuitive"* sorting algorithm, usually taught as an intro to sorting algorithms
- Perhaps less intuitive in functional languages, due to their nature of encouraging divide and conquer style code
- Although the given implementation repeats until no more changes occur, one can prove bubble repeats at most length(xs) iterations

Insertion Sort

An improvement over the Bubble Sort, the first part of an insertion sort is to insert an element into an already sorted list

```
insert y0 []   = [y0]
insert y0 (y1:ys)
  | y0 >  y1   = y1 : insert y0 ys
  | otherwise = y0 : y1 : ys
```

Insertion Sort

Example Evaluation:

```
insert 4 [1,3,4,6]
= insert 4 (1:[3,4,6])
= 1 : insert 4 [3,4,6]
= 1 : 3 : insert 4 [4,6]
= 1 : 4 : 3 : 4 : [6]
= [1,3,4,4,6]
```

Insertion Sort

We define insertion sort by iterating through the list, rebuilding it element by element using insert

Insertion Sort

We define insertion sort by iterating through the list,rebuilding it element by element using insert

```
insertSort [] = []
insertSort (x:xs) = let
  insert y0 []  = [y0]
  insert y0 (y1:ys)
    | y0 >= y1  = y1 : insert y0 ys
    | otherwise = y0 : y1 : ys
  in insert x (insertSort xs)
```

Quicksort

The QuickSort algorithm works by recursively filtering elements in a list based on a pivot point

Informally, the general algorithm works like so.

- Choose any point in the list to be a pivot point: p_i
- Partition the list into two other lists:
  - lesser - all elements $x_i <= p_i$
  - greater - all elements $x_i > p_i$
- Repeat on lesser & greater until singleton/empty list and place p_i in-between

Quicksort

This algorithm can be defined on lists as follows:

```haskell
qsort          :: (Ord a) => [a] -> [a]
qsort []       = []
qsort (p:xs) = (qsort lesser) ++ [p] ++ (qsort greater)
     where
        lesser  = [x | x <- xs, x < p]
        greater = [x | x <- xs, x >= p]
```

Quicksort

Example: Quicksort Evaluation

```
                        qsort [3,2,4,1,5]
                                |
                         ++ [3] ++
                qsort [2,1]          qsort [4,5]
                  |                            |
            ++  [2]  ++                   ++ [4] ++
      qsort [1]      qsort []         qsort []     qsort [5]
        |                 |              |               |
       [1]               []             []              [5]
```

Exercise 1

Define a function splitHalf that takes a list and splits it in half (as evenly as possible), and returns a tuple of lists of the two halves.

```
split :: [a] -> ([a],[a])
```

```
Example
    split [1,2,3]
      = ([1],[2,3])
```

Solution 2

```haskell
split :: [a] -> ([a],[a])
split xs = let
     half = length xs `div` 2
  in (take half xs,drop half xs)
```

Exercise 2

Define a function merge that takes two already sorted lists and returns a combined sorted list

```
merge :: (Ord a) => [a] -> [a] -> [a]
```

```
Example
    merge [2,5,6] [1,3,4]
      =  [1,2,3,4,5,6]
```

Solution 2

```haskell
merge :: (Ord a) => [a] -> [a] -> [a]
merge xs     []   = xs
merge []     ys   = ys
merge (x:xs) (y:ys)
         | x <= y     = x: merge xs (y:ys)
         | otherwise = y: merge (x:xs) ys
```

Exercise 3

The Merge Sort works by recursively splitting a list apart and then merging sorted lists together starting from a singleton / empty list (which are already sorted).

Using the split and merge functions you already defined, implement the function

```
mergeSort :: (Ord a) => [a] -> [a]
```

Solution 3

```haskell
mergeSort :: (Ord a) => [a] -> [a]
mergeSort []  = []
mergeSort [x] = [x]
mergeSort xs  = let
     (ys,zs) = split xs
  in merge (mergeSort ys) (mergeSort zs)
```

Exercise 4

Implement a QuickCheck property

```haskell
sortProp :: (Ord a) => ([a] -> [a]) -> [a] -> Bool
```

That takes a sorting function and checks if it properly sorts its input.

For example, you can call

```haskell
quickCheck (sortProp mergeSort)
```

to test your implementation of mergeSort

Solution 4

```
sortProp :: (Ord a) => ([a] -> [a]) -> [a] -> Bool
sortProp sort xs = let
     inOrder (x0:x1:xs) = x0 <= x1 && inOrder (x1:xs)
     inOrder _          = True
  in inOrder (sort xs)
```

Exercise 5

- Define the prelude function replicate that takes a value and repeats that value n times.

```
replicate  :: Int -> a -> [a]
```

- Define the prelude function (!!) that selects the nth element of a list

```
(!!)  :: [a] -> Int -> a
```

- Define the prelude function elem that takes a value and a list and returns True if the value is an element of the list

```
elem  :: (Eq a) => a -> [a] -> Bool
```

Solution 5

```
replicate :: Int -> a -> [a]
replicate 0 _ = []
replicate n x = x : replicate (n-1)
```

```
(!!) :: [a] -> Int -> a
(x:_)  !!  0  = x
(_:xs) !!  n  = xs !! (n-1)
```

```
elem :: (Eq a) => a -> [a] -> Bool
elem y []     = False
elem y (x:xs) = (y == x) || elem y xs
```

Exercise 6

- Define the prelude function and which takes a list of Bool and only returns True if all the elements are True (use recursion)

```haskell
and :: [Bool] -> Bool
```

- Define the prelude function concat that takes a list of lists and concats the inner list (use recursion)

```haskell
concat :: [[a]] -> [a]
```

Solution 6

```haskell
and :: [Bool] -> Bool
and []       = True
and (x:xs) = x && and xs
```

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