Compared to my other summaries, I got pretty lazy due to a lot of hw.
Also note the chapters are off by 1 since I'm going by the book.

Chapter 4: Hello Recursion:

Summary: Recursion is a function that calls itself; it keeps going until it reach its base case when it can't go further. The book showed examples of recursion through maximum, take, reverse, repeat, zip, replicate, and elem. The steps used to solve recursion are to first define a base case, which is a nonrecursive solution that holds when the input is trivial (e.g. the results of an empty list is the empty list). The next step(s) is to just keep breaking the problem down into sub-problems and recursively solve them (e.g. for the quicksort function, a list is broken into two lists plus a pivot and continually sort until finally they all combine together into one sorted list).

Notes:

TL:DR : recursion is a function that calls itself. (Keeps going until it reach its base case when it can't go further)

Really lazy so just giving short descriptions:
Examples on how to use recursion on the following functions: Maximum, take, reverse, repeat, zip, replicate, and elem.

They then implemented the quick sort algorithm using recursion.

Steps of Recursion:

1. Define base case: simple nonrecursive solution that holds when the input is trivial.
   E.g. the results of an empty list is the empty list
2. Just keep breaking up the problems into one or more subproblems and recursively solve them.
   E.g. In the quicksort, you break the list into two lists plus a pivot and continually sort. At the end, they get combined into one sorted list.

Chapter 5: Higher-Order Functions:

Summaries:

The main goal of this chapter is creating function that can take functions as parameters and returning those functions as return values, this is considered as a higher order function. Some of the most useful functions that use these are map, fold, and scan. The map function would take a function and a list, and apply that function to every element in the list, producing a new list. An example would be map (replicate 2) [2..5] with the output of [[2,2],[3,3],[4,4],[5,5]]. There's also the (.) and ($) function to help shorten the code written.

Notes:

Higher order function is a Haskell function that can take functions as parameters and return functions as return values.

The functions used previously in the book have been curried functions, a function that always takes exactly one parameter.
An example is the max function: max 4 5 = (max 4) 5

max is applied to 4 then returns a value from another function, which is then applied to 5

ghci> :t max
max :: (Ord a) => a -> a->a can also be written as max:: (Ord a) => a -> (a->a)

If you call a function with too few parameters, you get back a partially applied function (takes in as many as you left out).

The type (or type variable) before the -> is the type of the values that a function takes, and the type after is the type of the values it returns.

Shortcut example:
compareHundred :: int -> Ordering
compareHundred x = compare 100 x

is equivalent to
compareHundred :: Int -> Ordering
compareHundred = compare 100

since the type declaration stays the same because compare 100 returns a function.

Sections:
Infix functions can be partially supplied using sections by just surrounding with parenthesis.
Example:
divideByTen :: (Floating a) => a -> a
divideByTen :: (/10)

Some Higher-Orderism Is in Order
Functions can take other functions as parameters and can also return functions as return values.
ExaMPLE:
applyTwice :: (a->a) ->a->a
applyTwice f x = f (f x)
Parentheses here are now mandatory since they indicate the first parameter is a function that takes one parameter and returns a value of the same type (a->a). When using a, a can be of any type, but they all must be of the same type.

There's some examples implementing zipWith and flip

The Functional Programmer's Toolbox
Some really important functions used often

The map Function
It takes a function and a list, and applies that function to every element in the list, producing a new list.
map :: (a->b) ->[a] ->[b]
map –[] = []
map f (x:xs) = fx : map f xs
Example:
ghci> map (replicate 2) [2..5]

[[2,2],[3,3],[4,4],[5,5]]
map (+3) [1,5,3,1,6] is equivalent to [x+3 | x<- [1,5,3,1,6]]

The filter Function
It takes a predicate and a list, and returns the list of elements that satisfy that predicate. (predicate is a function that tells whether something is True or False)
filter :: (a->Bool) -> [a] ->[a]
filter _ [] = []
filter p (x:xs)
| p x = x : filter p xs
| otherwise = filter p xs

Example:
ghci> filter odd [1..10]
[1,3,5,7,9]
Here's an example of applying several predicates in a list comprehension
ghci> filter (<15) (filter even [1..20])
[2,4,6,8,10,12,14]

Then they provided more examples of map and filter.

There's also a neat example on the Collatz chain on page 69.

Lambdas
They are anonymous functions that are used when you need a function only once.
It uses the (\) as a replacement for the lambda symbol but with a missing leg.
Then write the function's parameters, separated by spaces, afterwards comes a -> and the function's body.
Example:
numLongChains :: Int
numLongChains = length (filter(\xs -> length xs >15) (map chain [1..100]))

equivalences: map (+3) [1,6,3,2] = (\x->x+3) [1,6,3,2] = [4,9,6,5]
lambdas can take any number of parameters
Example:
ghci> sipwith (\a b -> (a*30+3)/b)[5,4,3,2,1][1,2,3,4,5]
[153.0,61.5,31.0,15.75,6.6]

I Fold You So
Folds allow you to reduce a data structure (like a list) to a single value
A fold takes a binary function (one that takes two parameters, such as + or div), a starting value (often called accumulator), and a list to fold up (pg 73).

<u>Left Folds with foldl</u>
Fold the list up from the left side.
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
or you can write it as

```
sum' = foldl (+) 0

ghci> sum' [1..5]
15
```

Right Folds with foldr

The accumulator eats up values from the right meaning the right fold's binary is reversed: current list value is first parameter and the accumulator as the second.

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

or using left fold

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

recommended to use right folds for building up new lists from a list since : is faster than ++

The foldl and foldr1 functions

foldl1 and foldr1 works the same way except w/o the need of a starting accumulator since it assumes the first element or last element respectively.

Scans

scanl and scanr are similar to fold except they report all the intermediate accumulator stats in the form of a list.

example:

```
ghci> scanl (+) 0 [3,5,2]
[0,3,8,10]
ghci> scanr (+) 0 [3,5,2]
[10,7,2,0]
```

Function Application with $

It's is called the function application operator.

```
($) :: (a->b) ->a ->b
f $ x = f x
sum( map sqrt [1..130])  == sum $ map sqrt [1..130]
```

This is because $ has the lowest precedence so it is right associative.

Another example:

```
sum(filter (>10)(map(*2)[2..10])) == sum $filter (>10)(map(*2)[2..10])==
sum $ filter (>10) $ map(*2)[2..10]
```

Function Composition

We do function compositions with the . function

```
(.) :: (b->c)->(a->b) -> a->c
f.g = \x-> f(g x)
```

Example:

```
ghci> map (\xs -> negate (sum (tail xs))) [[1..5].[3..6],[1..7]]
[-14,-15,-27]
```

```
ghci> map (negate . sum . tail) [[1..5].[3..6],[1..7]]
[-14,-15,-27]
```

Function Composition with Multiple Parameters

Example:

sum (replicate 5 (max 6.7 8.9))

- ⇨ (sum . replicate 5) max 6.7 8.9
  - sum . replicates 5 $ max 6.7 8.9

Point-Free Style: feel free look in book for this, it's just a way of making the code looks shorter