Since I own the book, everything is typed

**Introduction:**
**Summary:**
Since Haskell is a purely functional programming language, you tell the computer what everything is through functions. These functions have no side effect meaning they will only calculate something and return a result and allows programmers to deduce that it works. More complicated functions can be created by putting together a bunch of simple functions. Haskell is generally lazy and won't execute functions until it needs to show a result. It is also statically typed, meaning it knows which is a string, number, etc…, allowing you to catch errors at compile time.

# Chapter 1:
**Summary:**
Most functions used in Haskell are prefix functions where you call a function, a space in between then its parameters. In other cases, where it's hard to determine what the line of code does, you can use an infix function where the function is surrounded by backticks and the function would still work the same way. (e.g. div 101 10 = 101 'div' 10 = 10). Haskell if statements makes the else statements mandatory since Haskell program is a collection of functions. And functions always need to return some sort of value so that the other functions can use it. Lists (homogeneous: no different types) and tuples (heterogeneous: can have different types) was explained in great detail and mainly used to store information.

**Notes:**
Chapter one there's a lot of similarities to some of the other programming languages such as True and False as Boolean variables and precedence of operations.

There is something new called infix function where the function is surrounded by backticks where the function would still work the same way. (e.g. div 101 10 = 101 'div' 10 = 10).

The functions do not have to be defined in any particular order and they can be called from one another. So a whole bunch of simple functions can be combined or used together to make more complex functions.

Haskell if statements makes the else statements mandatory since Haskell program is a collection of functions. And functions always need to return some sort of value so that the other functions can use it.

Lists in Haskell are homogeneous data structures meaning can store a bunch of ints, strings or anything of the same type but no mixing between the others.

The "let" keyword allows you to define a name in GHCi. (E.g. let a =1 similar to a=1 in script and loading with :l)

Strings, as usual, are a list of characters and you can concatenate by using "++" (e.g. ['l','o'] ++ ['v','e'] ++ " " ++ "homework" = "love homework")

However, with long strings, using this method would take a while and if you use ":" (cons operator) it would instantaneously add the string to the beginning of the string or list. (e.g. 'A' : " banana = "A banana" or 'z' : ['e','b','r','a'] = "zebra")

You grab list values using "!!" where the indices starts at 0. (e.g. "My name is Bob" !! 6 = 'e')
Lists can be compared in lexicographical order as long as they both contain the same type of items. Other useful functions are head, tail (chops off head and return), last, init (chops off tail and return), length, reverse, null (checks if empty), take (e.g. take 2 [1,2,3,4] = [1,2]), maximum, minimum, sum, product, elem takes an item and a list of items and tells if it is an element of that list (e.g. 5 'elem' [2,3,7,7,5] = True)

Ranges makes your life considerably easier than typing a list of 1-20 you can just type [1..20] example with characters is ['a'..'z'] from a-z. To go from 20 to 1, you need to type in [20, 19 ..1]. You can use ranges to make infinite lists by not specifying an upper limit. (e.g. take 5 [2,4,..] = [2,4,6,8,10].

Cycle takes a list and replicates its elements until you stop it somewhere. (e.g. take 11(cycle "Bah ") = "Bah Bah Ba")
Repeat is similar where it takes an element and produces an infinite list of that element.

Replicate easier way to create a list composed of a single item (e.g. replicate 3 10 = [10,10,10])

List comprehensions is a way to filter, transform and combine lists one example is [x*3 | x<- [1..5]] = [3,6,9,12,15]. We **draw** the elements from [1..5], **bind** those elements to x, to the left of the pipe (|) is the **output**. You can add conditions at the end called **predicates**. (e.g. [x*3 | x<- [1..5], x*3 <10] = [3,6,9]. There can be as many predicates as you want and can use values from multiple lists. You can use use list comprehensions to process and produce strings.
(e.g if I did this right should remove all uppercase
removeUppercase st = [c | c <- st, c 'elem' ['a'..'z']]

Tuples are similar to lists but are heterogeneous meaning allowing to store elements of different types, but there's a fixed size. They are surrounded by parentheses and components by commas: (e.g. (25, 'I' , "tired"))
Tuples are more useful in representing vectors since a tuple of size two (pairs) and another of size three (triple) are treated as two distinct types. They are used to represent a variety of data, for example, a triple can be used to represent a person's first and last name and age ("Bob", "Schnieder", 120).

When using pairs there's fst returns first component and snd returns second component.
Zip function produces a list of pairs by taking two lists and joining them together. (e.g. zip [1,2,3] [3,3,3] = [(1,3), (2,3), (3,3)].

Haskell uses lazy evaluation allowing to zip finite lists with infinite lists.