Chapter 2: "Believe the Type" Summary:
What we know about Haskell is that compared to a lot of other programming languages, Haskell has type inference where it's able to determine the type of the number, character, . . . without the user telling it what it is. This also means that Haskell is able to catch a lot of errors during compile time since everything in Haskell has a type. One type can be an instance of many type classes, for example, Char is instance of Eq, Ord, and many more. Most of the common types Haskell has like in other languages are the Int, Float, Double, Bool, Char, and etc. Haskell also contains a lot of polymorphic functions that are able to operate on values of a variety of types in a safe manner. An example would be the equality symbol (==) in how it is able to compare between, for example, two ints, two doubles, two strings, and etc which returns a bool of True or False.

Chapter 2 Notes:
Haskell leads to safer code since all of the expression type is known at compile time; this is because everything in Haskell has a type. All errors would be caught on compile time. Unlike some of the other programs, Haskell has type inference (e.g. if you wrote in a number, Haskell can infer that it's a number without the user telling it).

Explicit Type Declaration: in GHCi you can use :t to examine the types of some expression.
Example:
ghci> :t 'g'
'g' :: Char
ghci> :t "APPLE"
'g' :: [Char]
The " : : " operator represents "has type of" and all tuple length has its own length.
Since functions have types, you are able to give them an explicit type declaration.
Here's an example from my last notes that has been modified:
removeUppercase :: [Char] -> [Char]
removeUppercase st = [c | c <- st, c 'elem' ['a'..'z']]
The [Char] -> [Char] means it takes one string as a parameter and returns another as a result
Example:
Taking four ints and subtracts them
subtractFour :: Int ->Int -> Int -> Int -> Int (the return type separated by -> and is the last one)
subtractFour a b c d = a – b – c - d

Common Haskell Types:
The Haskell types are used for representing things like numbers, characters, and Boolean values.
Int: integer, used for whole numbers and is bounded, meaning it has a minimum and maximum value.
Integer is also used to store integers and is not bounded, so can be used to represent really big numbers.
Float: real floating point number with single precision
Double: real floating point number with double the precision (meaning uses twice as many bits to represent the numbers)
Bool: is Boolean type that can have only two values: True and False
Char: represents Unicode character and is denoted by single quotes. (list of chars = a string)
Tuples: are types but dependent on their length and types of componenets.

Type Variables:
Some functions are able to operate on various types. An example given in the book is "head."
ghci> :t head

head :: [a] -> a
The a is an example of a type variable, which means that a can be any type.
Functions that use these type variables are called polymorphic functions and they allow functions to operate on values of various types in a type-safe manner.

Type Classes 101:
It is an interface that defines some behavior. If a type is an instance of a type class, then it supports and implements the behavior the type class describes.
Another example from book:
ghci> :t (==)
(==) :: (Eq a) => a->a -> Bool
Everything before the => symbol is called a class restraint. So using the example, the equality function takes any two values that are of the same type and returns a Bool (the type of those two values must be instance of Eq class).

The Eq Type Class:
Eq is used for types that support equality testing (== or /=)
Example:
5 == 5 ➔ True
"Ho Ho" /= "Ho Oh" ➔ True

The Ord Type Class:
It is a type class for types whose values can be put in some order.
Ord covers all standard comparisons such as >, <, >=, and <=
The compare function takes two values whose type is an ord instance and returns an Ordering (GT = greater than, LT = Lest than, or EQ=equal)

The Show Type Class:
"Values whose types are instances of the Show type class can be represented as strings" (pg. 29 got lazy)
The function that uses this is show, which prints the given value as a string.

The Read Type Class:
The opposite of show would be read, and the read function takes a string and returns a value whose type is an instance of Read.
examples:
ghci> read "25" – 5 ghci> read "True" || False
20 True
ghci> read "5" (an error will print out since GHCi does not know which type to return)
ghci> :t read
read :: (Read a) => String -> a
A way we can solve this problem is by using *type annotations.* This is a way to tell Haskell what the type of an expression should be. Do this by adding a :: then specifying a type at the end of expression
example:
ghci> (read "5.3" :: Float) * 4
21.2

The Enum Type Class:

They are sequentially ordered types (their values can be enumerated). Advantage is being able to use its values in list ranges and have defined successors and predecessors, which can be obtained with succ and pred functions. Types used are: (), Bool, Char, Ordering, Int, integer, Float, and Double.
Examples:

```
ghci> ['a' .. 'e']          ghci> [LT .. GT]
"abcde"                     [LT, EQ, GT]
```

The Bounded Type Class:
Instances in this have an upper and a lower bound which are checked using minBound and maxBound functions. (e.g. maxbound :: Char => '\1114111'
Tuples are also considered to be instances of Bounded themselves

The Num Type Class:
It is a numeric type class where its instances can act like numbers. Whole numbers are polymorphic constants and can act like any type that's an instance of Num type class (Int, Integer, Float or Double)

The Floating Type Class:
it includes the Float and Double types, which are used to store floating point numbers.

The Integral Type Class:
It is another numeric type class which includes only integral whole numbers. One useful function is fromIntegral that takes an integral number and turns into a more general number.
Example:

```
ghci> fromIntegral (length ['s','l','e','e','p','y']) + 2.5
8.5
```

Chapter 3: "Syntax in Functions" Summary (I've gotten lazier if you can tell)
In this chapter, they explain how to write Haskell functions to be more readable by utilizing patterns and other forms of syntax. You can pattern match on any data type, which will specify patterns to which some data should conform and deconstruct the data according to those patterns. Guards are very similar to if statements and are used to check if the properties of the values being passed true is true or false. Where and let both in their own way is used to store values or expressions into variables so there wouldn't be the constant need to retype everything over and over.

Notes:
Pattern Matching:
Used to specify patterns to which some data should conform and to deconstruct the data according to those patters. So you can create separate function bodies for different patterns (can pattern match any data types) leading to simple readable code. You can also call a function recursively, which is by calling itself inside its own definition. Sometimes patterns will fail to work and complain that you have "non-exhaustive patterns," so you should include a catchall pattern at the end of the program.

Pattern matching with Tuples:
Well let's skip straight to an example:

```
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

If you use a _ variable, it means the same thing it does in list comprehensions. It just mean you don't really care about that part.

Pattern Matching with Lists and List Comprehensions:
example of pattern matching in list comprehension
ghci> let xs = [(1,2),(2,3),(9,8)]
ghci> [a+b | (a,b) <- xs
[3,5,17]
If a pattern match fails, it will move onto the next element in the list (it won't add in the failed element). Regular lists can also be used in pattern matching (match with empty list [] or any pattern that involves ":" and the empty list. (e.g. a pattern x:xs would bind the head to x and everything else to xs.
The tell function is safe to use since it can match to the empty list, a singleton list, a list with two elements or more.

As-patterns:
It allows you to break up an item according to a pattern, while still keeping a reference to the entire original item. You can create an as-pattern by preceding a regular pattern with a name and an @ character.
Example:
using the following as-pattern: xs@(x:y:ys).
This allows you to easily access the entire whole list using xs instead of needing to type out x:y:ys

Guards, Guards!:
It is used to check if some property of those passed values is true or false.
It is indicated by a pipe (|) character, followed by a Boolean expression, followed by the function body that would be executed if it is True. Warning! Guards should be indented by at least 1 space. The last guard is usually otherwise which catches everything if all else is False.

Where?!:
You can store the result of a computation into a variable by using the "where" keyword. This allows you to bind a value to a variable and then using that variable in place of the calculation. In the book's example (check it out, I quite like it pg. 43) you would put the where keyword after the guards. This way the names would be visible across all the guards, and allows quick changes without too much hassle.

Where's Scope:
The variables you define in the where section of that function is visible only to that function, otherwise if you can declare the variables globally. Example:
noHomework :: String
noHomework = "Time to partay!"

friday :: String -> String
Friday "FinalsOver" = noHomework ++ "YEAHHHH!"

Pattern Matching with Where:
You can also use where bindings to pattern match, but I'm too lazy to write an example so look it up in the book.

Let it goo . . .I mean *ahem* . . . Let it Be:
let expressions are similar to where, but allow you to bind to variables anywhere and are expression themselves. The problem is, they are local and cannot span across guards.
Example:

```
pie :: Double -> Double -> Double
pie j k =
        let joke = j*j
            king = k^2
        in  joke + king
```

let expressions take the form of let <bindings> in <expression>.
Pattern matching with let expressions is quick to dismantle tuples into components by binding those components to names, example:

```
ghci> (let (x,y,z) = (3,2,1) in x-y+z) * 50
100
```

Let in List Comprehensions and let in GHCi (Too lazy)

Case Expressions:
This allows you to execute blocks of code for specific values of a particular variable.
Syntax:

```
Case expression of      pattern -> result
                        pattern -> result
                        pattern -> result
                        . . .
```

The first pattern that matches the expression is used; if all fails a runtime error occurs.
These case expressions can be used anywhere.