All You Need is Lambda

This is the first proper chapter of the book, theoretical background into Haskell.

Things learnt

- Definition of a function (One to One, One to many) -> Referential transparency
- Abstraction are functions! Variables have no inherent meaning/value, it is the idea of a value that we are concerned with. Expressions are a general term for variables and abstractions.
- An abstraction/function consists of two parts: The head (consisting of the λ term) and the tail (function expression) seperated by a single dot (.)
- lambda abstractions are *anonymous functions* and cannot be called by name
- Alpha equivalence occurs when two functions are essentially the same (i.e. they give the same output given the same input, or the expressions within the function definitions can be simplified/distilled to be the same)
- Beta Reduction is the act of applying a function to an argument. Applications in lambda calculus are *left associative* i.e. they group to the left. When an expression cannot be further beta-reduced, it is in its *beta normal form*. All references to normal forms from this point onwards will refer to *beta normal form*.
- Free vs. Bound Variables.... **NOTE** alpha equivalence does not apply to free variables. $\lambda x.\, xz$ and $\lambda x.\, xy$ are not alpha equivalent because z and y might be different (if not why u use different letter leh?). But $\lambda xy.\, yx$ and $\lambda ab.\, ba$ are, and so are $\lambda x.\, xz$ and $\lambda y.\, yz$
- What to do with mulitple arguments a.k.a. *Currying* : Functions that require multiple arguments have multiple nested heads. To *Curry* is to apply and remove the heads in order from the left.

  $\textbf{Example:}$

$$\begin{aligned}(\lambda xy.\, xy)\ 1\ 2 &= (\lambda x.\, (\lambda y.\, xy))\ 1\ 2\\ &= [x := 1]\ (\lambda y.\, xy)\ 2\\ &= (\lambda y.\, 1y)\ 2\\ &= [y := 2]\ (1y)\\ &= 1\ 2\end{aligned}$$

- An expression that has no free variables are known as *combinators*. They serve only to combine the arguments they are given. There won't be much attention to this special class of lambda expressions, they are only mentioned here for completeness sake.
- There also exists lambda expressions that do not reduce to a beta normal form. These are known as *diverging* lambda expressions. *Divergence* here means that the reduction process never terminates or ends. An example of a diverging lambda expression is the *omega* function, $(\lambda x.\, xx)(\lambda x.\, xx)$. Diverging functions are analogous to recursive functions in traditional programming paradigms, but on its own, the *omega* function doesn't produce any meaningful result. More attention will be paid to these types of expressions in later chapters, specifically into what will terminate so that we can understand what programs can

do meaningful work.