Haskell programming from first princ

So you want to learn Haskell, huh? Well, you came to the right place. Maybe. This is the first in a series of posts in which I try to consolidate my understanding of Haskell programming language through the book Haskell Programming from First Principles by Christopher Allen and Julie Moronuki (ISBN: 9781945388033.) I will take care of the book and write a summary of the most important parts of each chapter while I go ahead. For a while, I was very interested in Haskell and use it whenever I can for my projects, but I was also very busy, and I didn't have much time to play with it as I would like. I read You are a Haskell for Great Good and too many monad tutorials when I started, but since then, it has been mostly a kind of right process in time when I look for answers to the problems I met. So, while I feel I have a basic understanding of the language, some of the most advanced features still escape me. This book was on my reading list for a while, but I didn't make a serious effort to get over it first. Now, however, I have time and I have motivation. Time to learn Haskell. Really. All right, here we go. Deep breaths. Deep breaths. And there's not one Haskell line in this chapter. I'm sorry to disappoint you, dear reader, but this chapter is entirely about Haskell's theoretical bases: the calculation of the lambda. But that doesn't mean we can't do anything! EnoughI do. So put on the math hat and make some stretches of light....Ready? While an in-depth introduction to the lambda calculation is outside the scope of this post (not to mention, not something that is qualified to do,) there are some concepts that will help our learning, so we try a basic introduction. In the summary of the first chapter, the calculation of the lambda is defined as "a formal system to express programs in terms of abstraction and application". Wow. That's clarity. Let's see if we can find something more digestible. Previously, the book defines a calculation as a method of calculation or reasoning. In other words, it is a system we can use to solve problems. Lambda calculation is only a process to formalize one of these methods or systems. In other words, it is a way of thinking and calculating some problems and gives you a set of tools you can use to solve these. Then why are we looking at the lambda calculation before we get into the code? Well, because functional programming consists of expressions (values, variables, functions) that shape the typical mathematical behavior, and understanding how they interact will help us understand how the language works later. Or something like that. It's almost as much as you're gonna have to start. We'll cover anything else like and when it comes. Let's talk about lamb terms, the three basic components of lamb calculation. They are:Variables A variable is just something you can assign a value. It has no meaning or value, butexists as potential input for a function. Simple. Abstracts An abstraction is a function. It is a lamb word that has a head (a lambda (λ) followed by a variable name) and a body (another expression) and is applied to a subject. A topic is an input value. Expressions An expression can be a variable name, an abstraction, or any combination of these two. A function in the calculation of the lambda is nothing special. It is only an expression that can be applied to another expression and that returns another expression. If this sounds confused, keep in mind that expressions can be both values and other functions. A lambda abstraction (i.e. a function) looks like this: $λx.x$ where everything from $λ$ to \(.\) is known as the head, and the body is what comes after \(.\) (x in this case). The head binds the variables to be used in the body (i.e. function parameters). Now, one interesting thing to note is that every lamb can only take a topic. Does this mean that in the calculation of lamb you can only have functions that take a single topic? Not really. See, the functions that take multiple arguments are actually only nested heads:$λx.λy.xy$ But this is quite verbose, so we will simply simplify it by writing $λxy.xy$ It may seem trivial but it is a very important property of lamb, known as curry (after Haskell Curry (no, really)). This is also what allows partial application. Since an expression can be evaluated in part, we can alsoit to a value and get a new function back. For a concrete example, we say we have a fine function that takes two arguments and returns their product:$mul=λxy.x*y$ If we want to create a function that doubles a value, we can do it by applying fines to $2:$double=mul(2)$or, written as a result of the previous expression:$double=λy.2*y$ The double function expects a subject and will return twice what it receives and is defined as a partially applied mul$. As the lambda terms interact is probably the most important part of this chapter, but let's take a look at some other terms mentioned in the chapter--- especially because you can use them to play smart on Internet message boards, but also because they help us understand what we are talking about when we are talking about lamb calculation. Alpha Two expressions are equivalent alpha if they are the same function, even if they are written with different symbols. The following functions are equivalent: Application How we valuate/reduce lamb. Beta normal shape When an expression cannot be reduced further, or because there are no more applications to do (no more lambs,) or because there are no free variables to apply the function. Beta reduction Evaluate expressions. This is what happens when you go from the expression $λa.a (2)$ and apply the lambda to the number, thus ending with 2$ Variables that are declared in the head of a function (yes, thetogether of possible outputs of a function. Combinations A lamb term without free variables. In other words, all variables are in the head. Divergence Any function that never ends is divergent (ointment: recurring functions that never go out, type loops (real)). DomainThe set of possible inputs to a function. Variables that exist in a function body that are not defined in the head, such as y in this expression:$λx.xy$ Lambda The Greek letter λ. Used to indicate abstractions (functions) and bind variables. Lambda Extraction An anonymous function or lamb term. You can think of the head as an abstraction for the body, that is, something we can put in place of the calculation. Transparency (aka purity) This means that, given the same input, a function will always produce the same output. While this may seem like a strange place to start, I can see logic in doing so, and getting a better understanding of lambda calculation is useful anyway, so I'm fine with starting like this. However, there should be more code in the next chapters. Until next time! haskell programming from first principles. haskell programming from first principles pdf. haskell programming from first principles amazon. haskell programming from first principles solutions. haskell programming from first principles pdf download. haskell programming from first principles exercises. haskell programming from first principles isbn. haskell programming from first principles reddit