Function Programming in Haskell

Prof. Madhavan Mukund and S. P. Suresh

Chennai Mathematical Institute

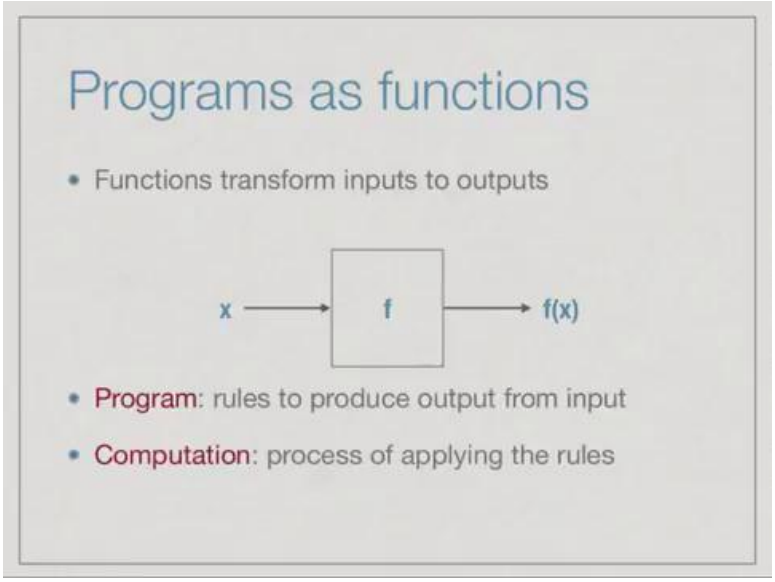
Module # 01

Lecture – 01

Functions

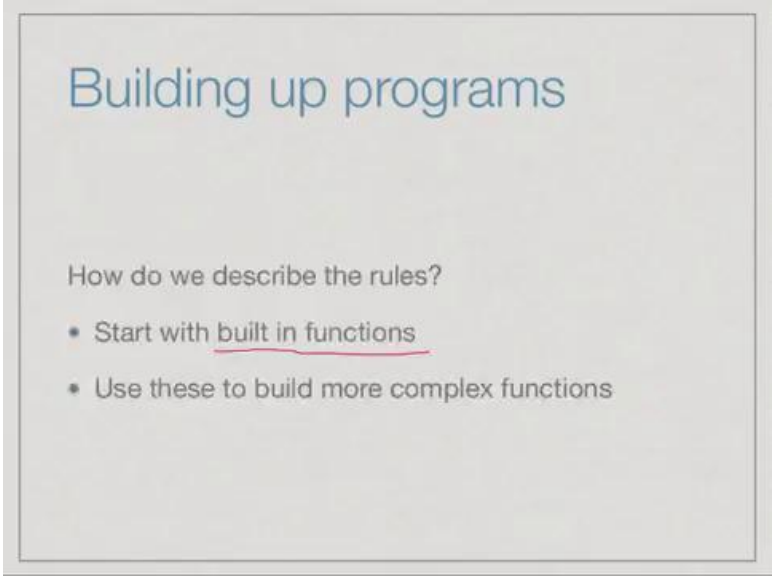
So, welcome to the first lecture of the course Functional Programming in Haskell.

(Refer Slide Time: 00:06)



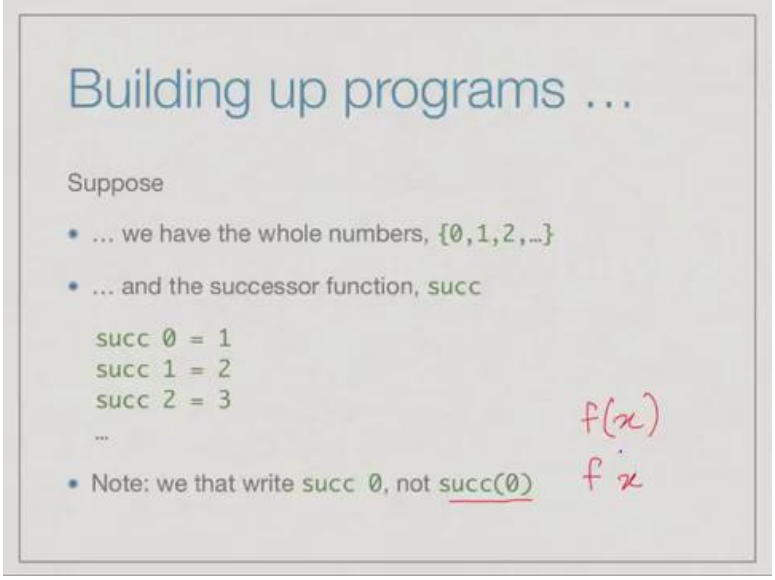
So, functional programming starts with the point of view that the program is a function; abstractly a function can be thought of as a black box; that takes inputs and produces outputs. In other words, a program is a function that transforms inputs to outputs. So, when we write a program in a function programming language, what we are doing is specifying the rules that describe how to generate a given output from a given input. And when we compute in a functional programming language, we apply these rules to the input that is given to us in order to actually produce the output; that is expected.

(Refer Slide Time: 00:50)



So, the first thing we have to ask ourselves is, how do we buildup these programs. Now, it is impossible to start from nothing, so we have to assume that we have some built in functions and values. So, we assume that we begin with something that is given to us and then, we use these built in functions and values to build more complex functions.

(Refer Slide Time: 01:17)

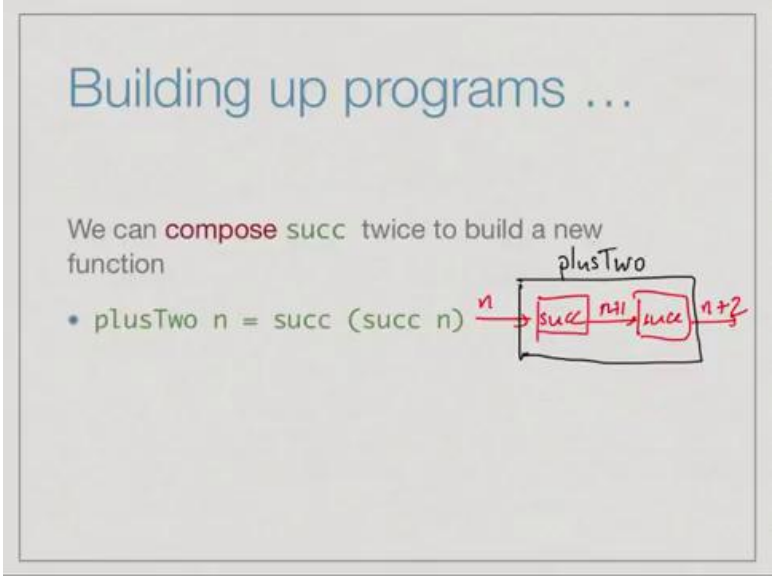


So, let us look at a concrete example, suppose we start with the whole numbers; that is the integer 0, 1, 2 and so on, so the non-negative integers and suppose the only thing that we know, how to do with these numbers is to add 1. So, we have a successor function,

which you can think of us plus 1 function. So, it says that 0 plus 1 is 1, the successor of 1 is 2, so 1 plus 1 is 2 and so on.

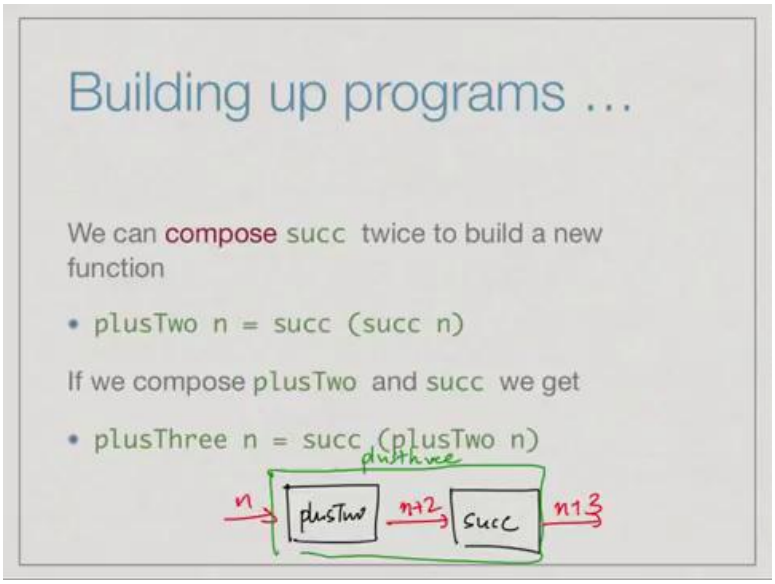
So, this is what is given to us, we have the whole numbers and the successor function. Note by the way; that we have uses slightly non-standard notation for the functions, normally one would successor of 0 with brackets like this. So, we write f of x normally, but in functional programming, we will see that, we will normally write f x eliminating the brackets. So, this has two advantages, one is you have fewer brackets to worry about, but it also have some interesting technical advantages, which you will describe in a later lecture in this week.

(Refer Slide Time: 02:18)



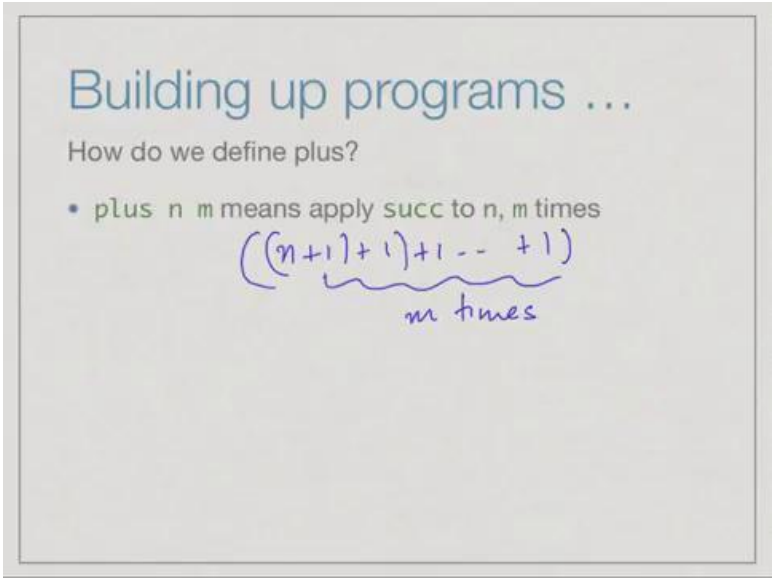
So, now, that we have a successor function, we can apply twice to an input for example, and I get a new function which adds 2. So, plus 2 of n takes n apply successor and then, apply successor to that, so it as though we had two boxes with us called successor. So, we feed n here, then we get n plus 1, if feeder to the second box and we get n plus 1 and what we are saying is that, now we can take this two boxes and call this outer box plus 2. So, this is what it means to compose to functions, you take the output of the first function and feed it to the second function. So, in this case, we have composed the same function twice, we have taken the successor, half successor.

(Refer Slide Time: 03:05)



However, we can also combine two different functions, for instance we can take the plus 2 which we are defined and feed it output successor. So, we have plus 2 and now, we have successor. So, we already know that, if we take a number n and feed it plus 2, we get n plus 2 and then, we feed it to success and we get n plus 3 and now, this gives as a new box, which we called plus 3. So, in this way we can combine functions by function composition, which is well known to us to mathematics.

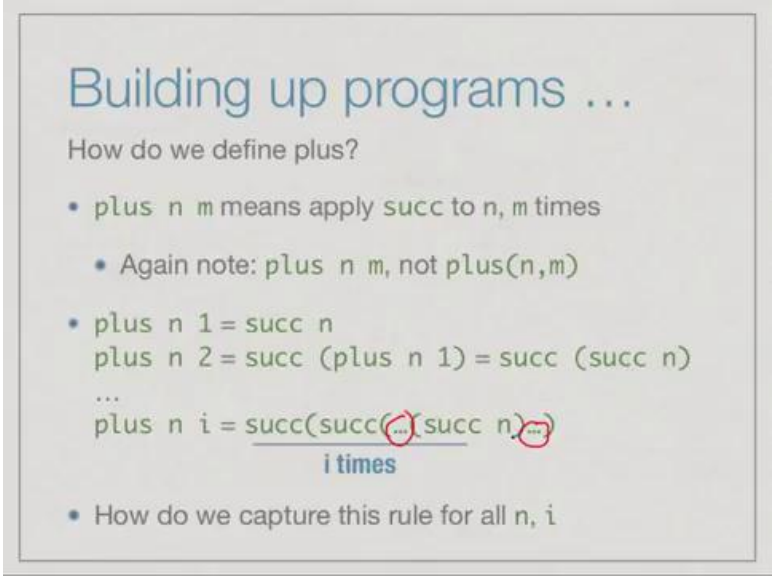
(Refer Slide Time: 03:42)



But, now suppose we want to extend or definition of plus 2 and plus 3, plus 4, plus 5 plus 6. In general, we want to define plus with two arguments n and m, where we mean that when we say plus n, m; we apply successor to n, m times. In other words, we start with n and then, we do plus 1 and then, we do plus 1 and then, we do plus 1. So, totally we add n, m times 1 m times.

So, this is what plus means and if you remember, this is how we were taught the definition of plus when you were in kinder garden, you just keep adding 1 by 1 by 1 m times. So, how would be describe this in our setting, because we need to describe this family of plus 1's.

(Refer Slide Time: 04:29)

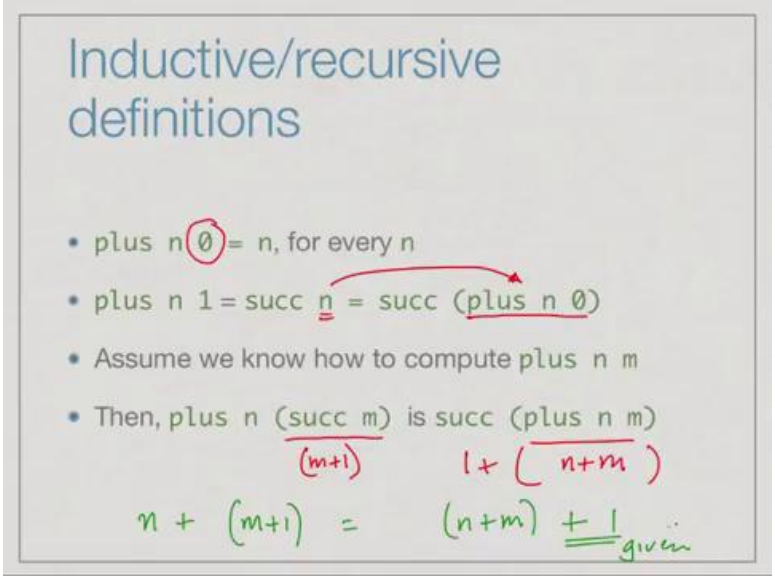


So, by the way again note this notation, we do not write a function of two arguments the bracket as usual, but we just write the arguments one after the other. So, plus followed by the first argument followed the second argument, you can just thing of it right now as a peculiar syntax which eliminates arguments brackets and commas. But, as we will see this is a very interesting way of thinking about functions from a computational stack.

So, what are the rules for plus will be know that plus m 1 is a same as successor the built in function adds 1, plus n 2 is, we have seen the successor of successive, but we can thing of successor of n as plus n 1. So, we are taking plus n 1 and then, applying successive to that. So, in the same way plus n i as we saw before will apply the successor i times and the question is, how do we write a rule which captures this mysterious dot,

dot, dot which says do something a fix number of times, but that fix number of times depends on the value of the argument.

(Refer Slide Time: 05:33)

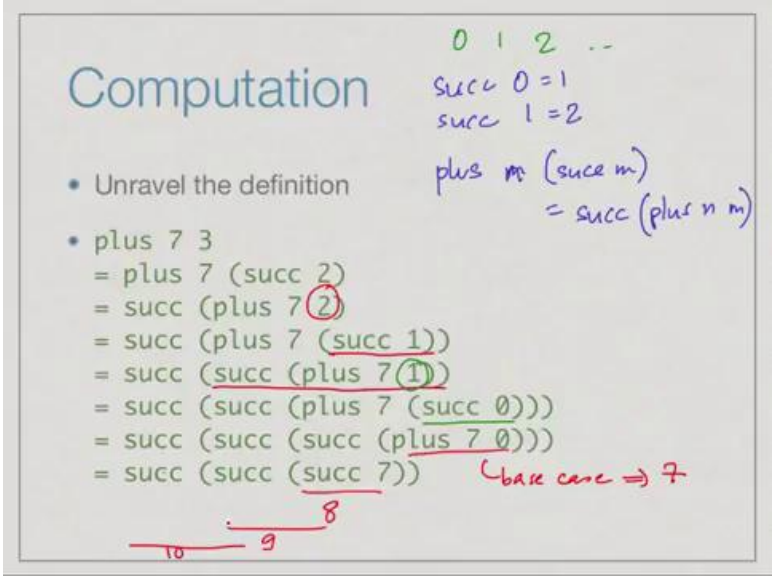


So, this brings us to the rime of inductive or recursive definitions. So, an inductive definition is 1, where we specify a base case and then, we specify the value for larger arguments in terms of smaller arguments. So, for instance, we know that, if we adds 0 when we do nothing, so plus n 0 is always n. So, this is the base case, we do not have to any computation, we just written the first argument.

Now, plus n 1 consist of adding 1 to n, but we can also thing of applying the previous value 0, so we get plus n 0. So, we just do not think of it is n, but we think of it as the base case and now, we apply successor to the base case. Similarly, in general, if I want to add something to m plus 1, then assuming that I know how to do n plus m, then I can add 1 to 11. So, this is just say that the value of n plus m plus 1 is the same as the value of n plus m, which I know how to do inductively, plus 1 which is given to be. So, this is my given function.

So, this is the basis of inductive for recursive definitions that you take the base case, whose values is obvious and for larger values to describe it in terms of operations that you know how to do plus the operation you trying to define one smaller values, which is inductively known to be true.

(Refer Slide Time: 07:07)

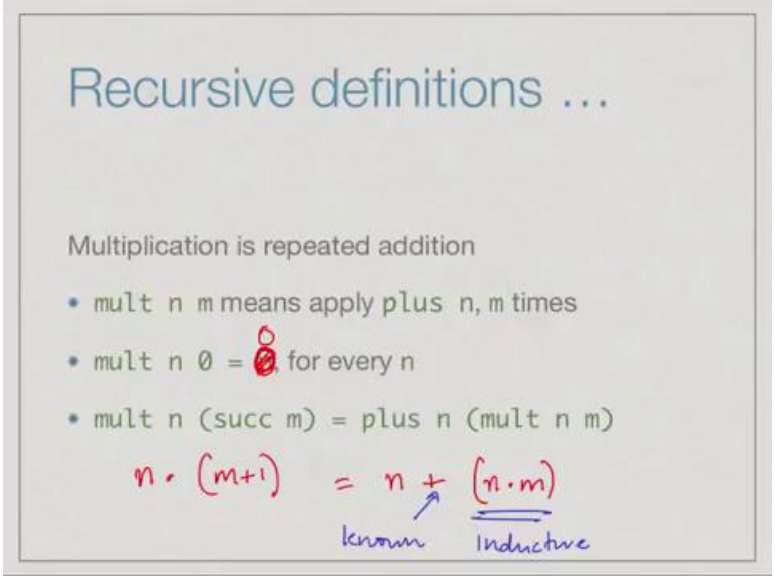


So, how does computation work, well it just unravels the definition. So, supposing I want to add 7 to 3, now in our universe, we have 0, 1, 2, etcetera and we know that successor of 0 is 1; successor of 1 is 2 and so on. So, I can think of 3 not as 3, but a successor of 2 and now, I have a rule with says plus, if we remember the rule is just plus n successor of m is equal to successor of plus n, m; this is the rule that we wrote. So, if I have plus 7 successor of 2, this is successor of plus 7, 2; when I plug in 7 for n and m for 2.

Now, once again I can expand this 2 as successor of 1 and apply that rule again and I get success of successor of plus 7, 1 and then, once again I can replace the number 1 by the expression successor of 0 and then, I get successor of successive 1 plus 7, 0. Now, this is the base case which is 7. So, I get successor of success of 7 and then, if I continue with this computation, this will give me 8 by the built in rule, this will give me 9 by the built in rule and this will give me 10 by the built in rule.

So, this is how I will get plus 7, 3 equal to 10, the important thing to note in this is that computing the value 10 does not involve understanding anything about numbers, it is just syntactically replacing expressions by expressions and this is something that we will see more formally later. So, this is how computation works in a functional programming language, you have rules which tell you how you can replace one expression by other expression and you keep replacing expressions, until you reach the value which cannot be simplify.

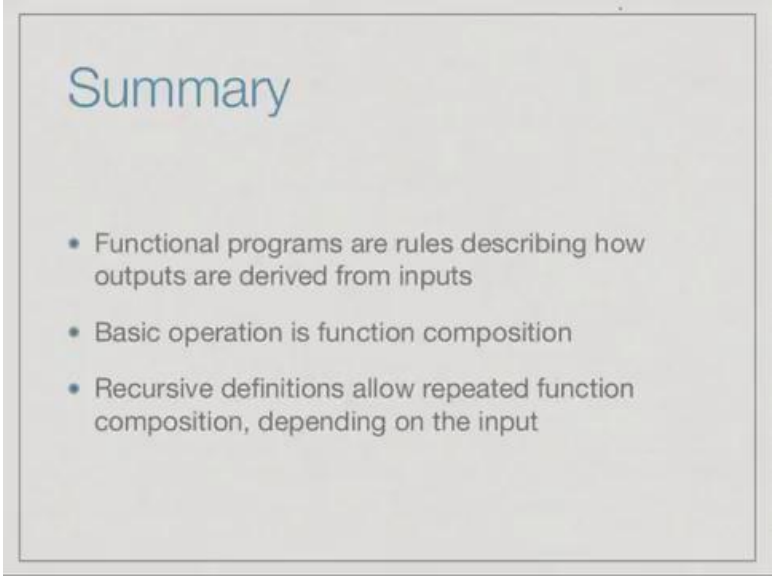
(Refer Slide Time: 09:06)



So, let us look at another recursively defined function. So, just as addition is repeated application of the successor function, multiplication as you remember from school is repeated addition, when I say multiply m by n, it means add n to itself m times. So, we have to apply plus n, m times starting from 0. So, the base case says that if I multiply any number base 0, then I will get 0.

So, multiplying, so this is not n, but 0, so multiply m by 0 is 0 for every n and now, n times m plus 1 is just m plus n times n. So, this is the inductive case, I know how to do this, because this is smaller and this is now unknown function, because we already define plus. So, plus was not a given function, successive was the given function, but we already define plus in terms of successive. So, now, we can use plus to define multiplication.

(Refer Slide Time: 10:13)



So, to summaries a functional program describes rules to tells us how to compute outputs from inputs, what we have seen is that the basic operations that we use is to combine functions. So, we use function composition, we feed the output of one function as the input of another function and now, often we have to apply this function composition more than once, but not number of times, we know an advance.

So, we did plus 2 and plus 3, we knew exactly how many times, we have to compose the function, but may wrote the general plus and the general multiply, we have to apply these function depending on the value of the argument. So, for such functions, we saw that recursive definitions or a good way to capture the dependence of the number of times the function has we composed based on the input value.