Function programming in Haskell

Prof. Madhavan Mukund and S. P. Suresh

Chennai Mathematical Institute

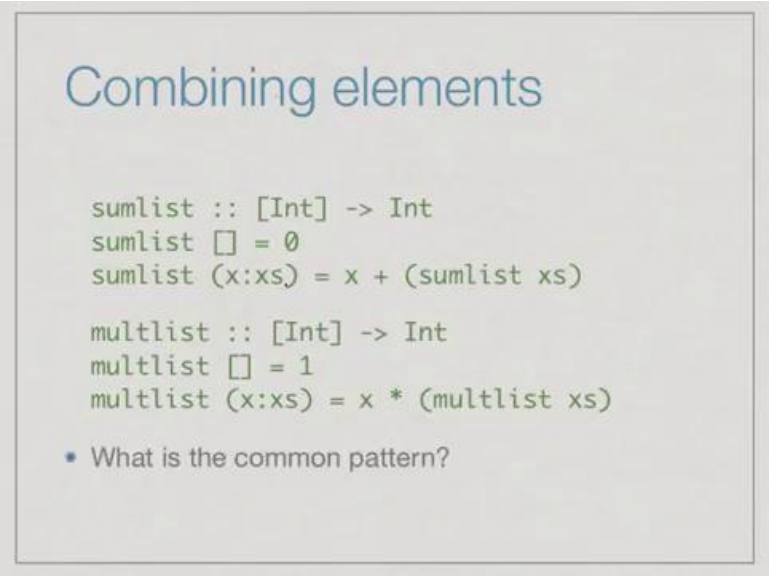
Module # 03

Lecture – 05

Folding through a List

Map and filter at important higher order functions on list that allows to manipulate entire list at a time. This today, we will look at another type of function, which involves folding a function through a list or combining the lists.

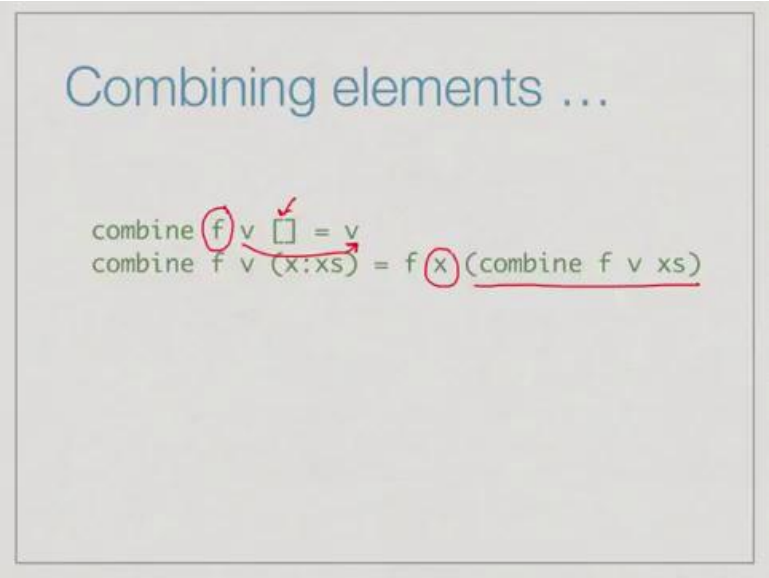
(Refer Slide Time: 00:17)



So, there is an example of combining the elements of list, when we want to take all the values in the list and add them up. So, this is the built in functions sum. So, let us call it some list. So, that we do not confuse that build in function sum. So, we have an empty list, this sum is 0, if we are the non-empty list, we add the first element, the sum of the rest.

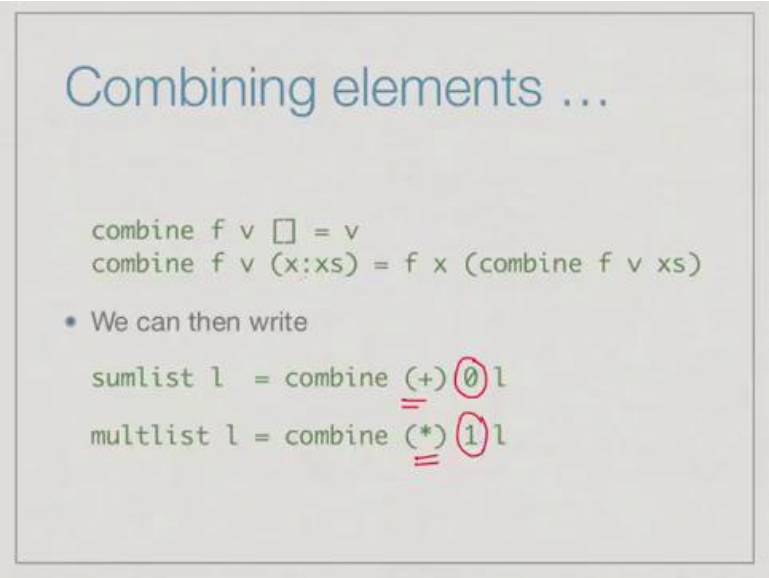
So, this can be also generalized to multiplying the values in a list. So, let us decide that the value of an empty list is 1. And now, if we want to multiply the values of list, we basically want to multiply X 1 by X 2 and so on. So, we take the head of the list and multiply it with the product of the remaining elements list.

(Refer Slide Time: 01:03)



So, the common pattern is that we are combining a function across a list with an initial value. So, if we have a function, so this case the functions were hardwired someone multiplied, but in general, we have function f. So, if we have the empty list, then we have this initial value v and if not, then what we do is, we applied the function to the current value and the recursively the compute value that we got from the rest of the list.

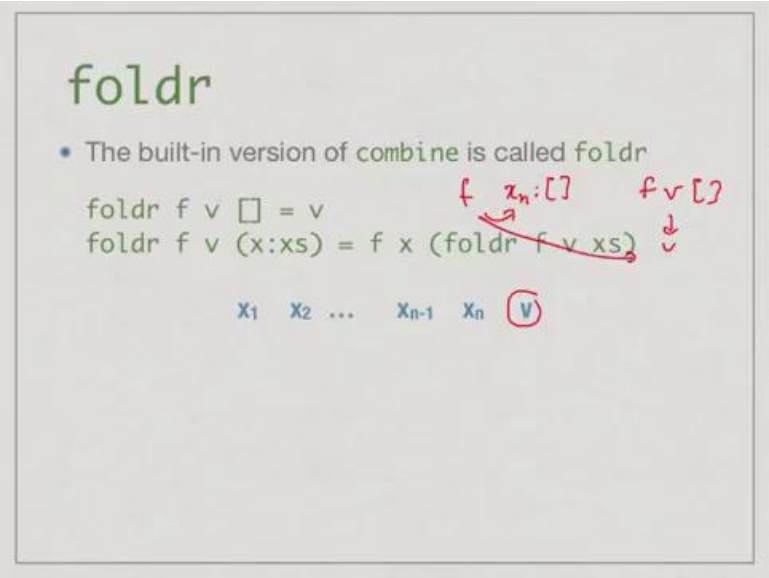
(Refer Slide Time: 01:35)



So, this then the way, we can write earlier functions. So, when we are deal sumlist, the function is plus and then, we take multiplying this, the function is times. So, the initial value for sumlist to 0, the initial value for multiply list is 1. So, when we combine 0 is an

initial value with the function plus, we get the sum of the numbers will be combine multiply with the initial value is 1; we get the product of the list.

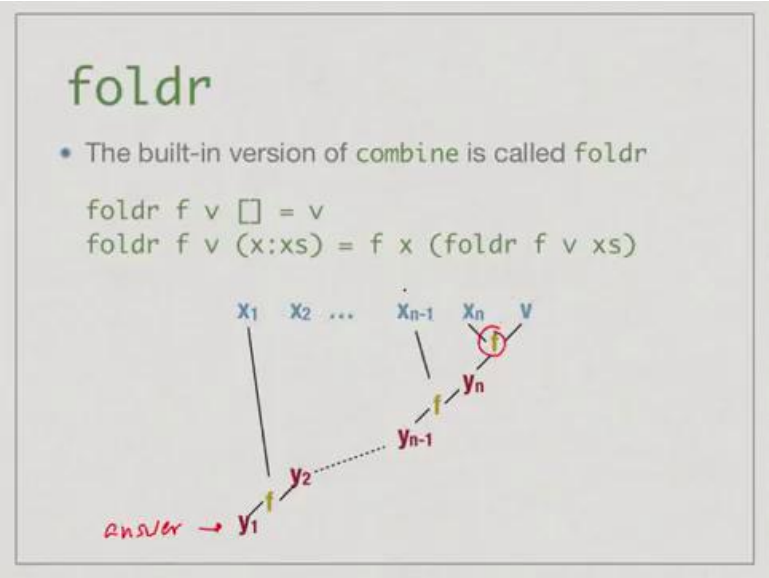
(Refer Slide Time: 02:02).



So, the function in a Haskell, it is does this is called foldr; in this process is called folding the function through list. So, foldr is basically just the redefinition of what we wrote for combine. So, we take a function and initial value for the empty list, we written the initial value and otherwise, we inductively apply the function to the current value, the head of the list and the value, we are compute in the tail of list.

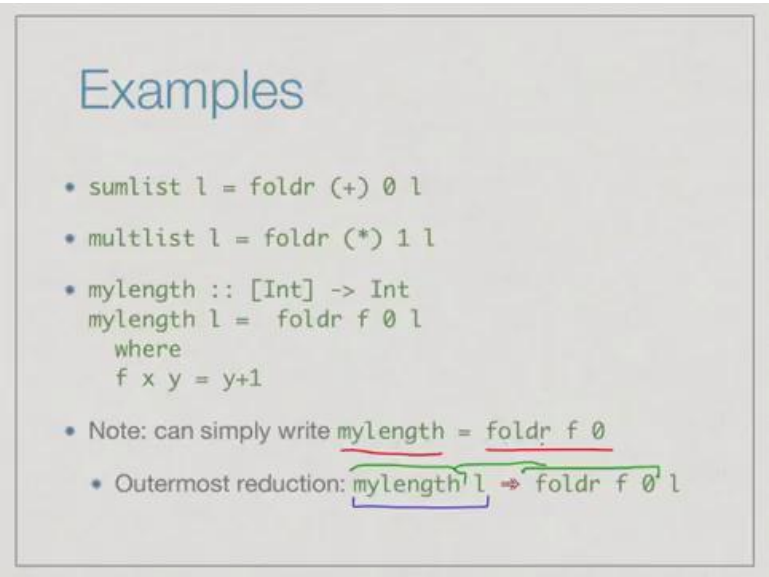
So, we can visualize this is follows, we start with the list X 1 to X n and the value v, which is our initial value. So, now, initially if we just look at the list, then as we inductively go through this, the thing that we do is, we operates on the last value. So, if we have x n colon empty list, then this will apply f to v and the empty list, it gives as v and then, apply f to this value and this value.

(Refer Slide Time: 02:55)



So, the first thing is we get is that, we combine X n with the initial value v and we get a value Y. Then, in the previous step, we apply X n minus 1 to the result of this value and we get a value Y n minus 1. So, in this way working backward from the right, so this r refers to the right. So, working right to left, we keep going. So, at some point, we will combine X 2 with the value X 3 onwards to get Y 2 and then, the final step, we combine X 1 with that value to get Y 1 and this is our final answer. So, this is the pictorial representation of what it means, if fold this function f from the right through this list with an initial value v.

(Refer Slide Time: 03:40)



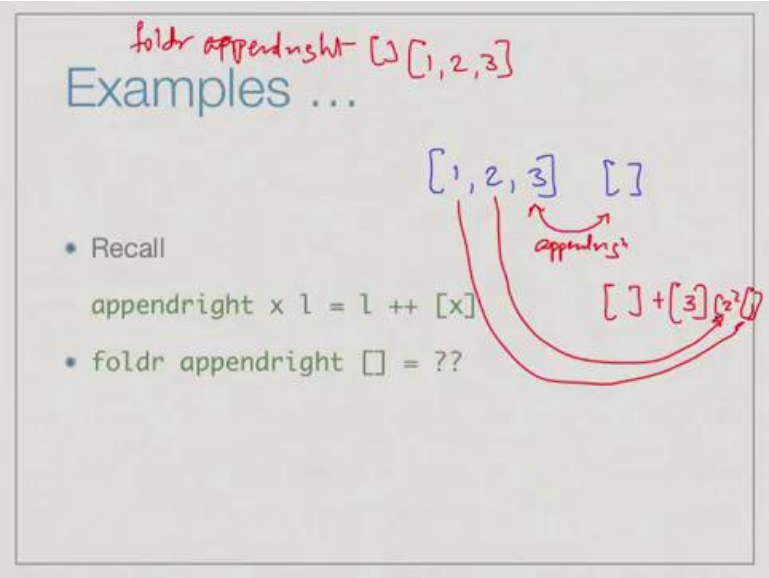
So, therefore, sum list the function that we wrote, this is same as folding that are function plus from the right, using the initial value 0. Multlist is the result of folding r from the right, the function times with an initial value 1. We can even this find the length of a list, if there in a total function, so if we just take the function with starts off with 0 and adds 1 for every value it sees, then we can take that function is folded.

So, for the empty list, we get 0, for any non-empty list, we get 1 plus result of the function on the remaining thing, because f of X Y is Y plus 1. So, in this case, why will be the result, let us come to the remaining list and we will get expression 1. So, my length, the length of the list can also be express using 4. So, this point, let us introduce the conventional, which is often used by Haskell programs.

Notice that, we can write my length directly to be this expression fold f 0; that is because, when we apply my length to any list, we apply this expression to the list. Then, if I have an expression of this form, remember that Haskell words outer most reduction, to rewrite the outer most definition. So, it will not try to expand l, it will try to expand my length.

So, it will take this expression my length and it will be expanded as foldr of f 0. So, by using this succinct definition as long as the arguments in the rewritten expression common the same position, they do not have to be insert it inside. We can omit the argument, when you write these functions. So, this is the usual piece of notation that when we have such situation, where the arguments are going to be apply with the expression in the same order with no reshuffling. Then, we can just use the expression itself as the definition functions without preferring to graph.

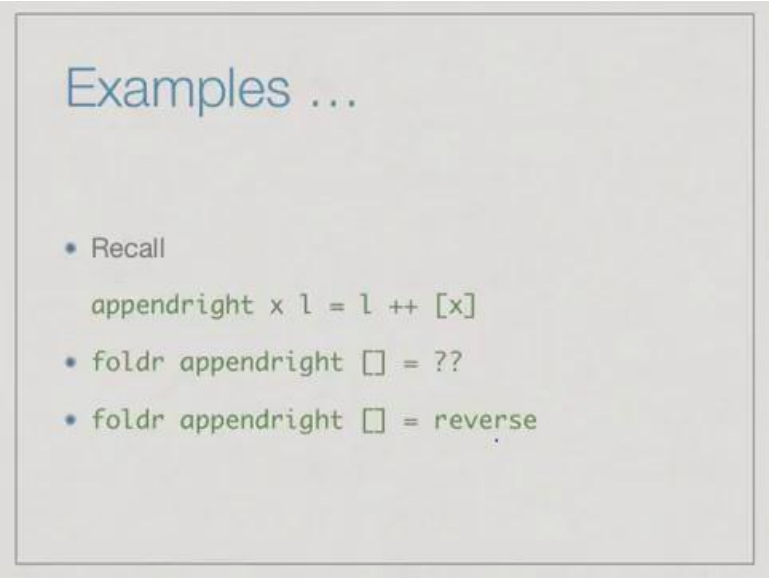
(Refer Slide Time: 05:42)



So, remember that one of the early functions we wrote has been verse to add a value to the end of the list, instead of instead of appending at the beginning as is done in the x colon l, we want it to take l as stick and x of the n. So, we can write this for instance as l plus, plus single term is X or you can write it in a definition, but this is what we need to append. So, what would happen if we used append right on the list with initial value empty.

So, let us write to executed that as supposing I apply this foldr append r, append, empty list of this. So, we have the empty list and we have 1, 2, 3. So, the first thing is that, I will take this element and combine into this using append. So, this will be me element on the right. Now, will take this element, stick it to the right of this and you will take this element, stick it to the right. So, we can see is that append r can write its shifting each element to the end of the list already created to it is right.

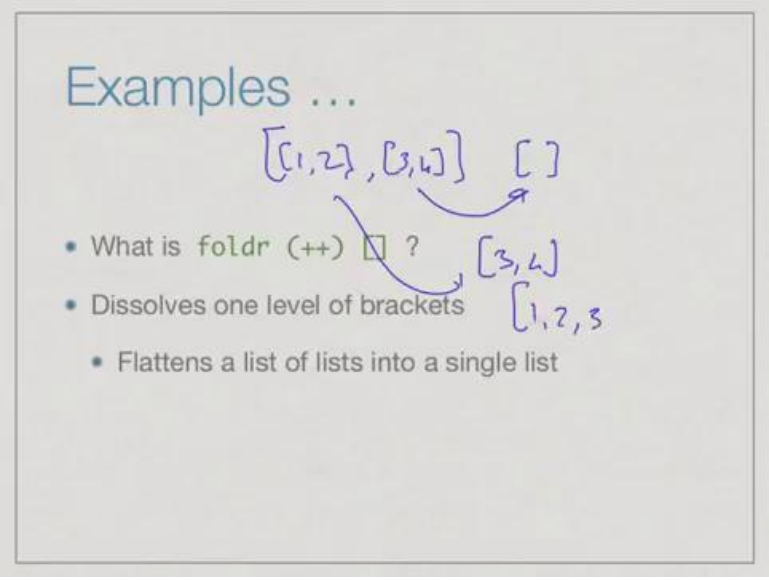
(Refer Slide Time: 07:03)



So, in other words foldr append right to the empty list is the same as reverse.

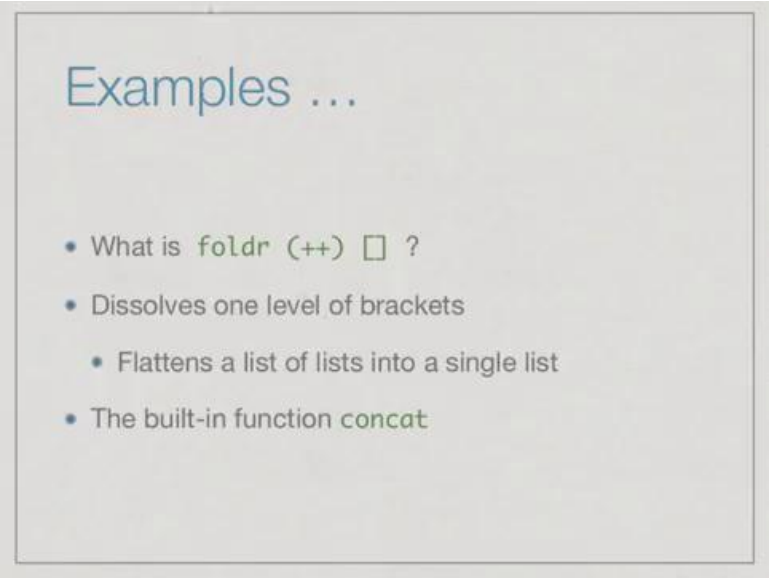
Is just reverse [07:08]

(Refer Slide Time: 07:12)



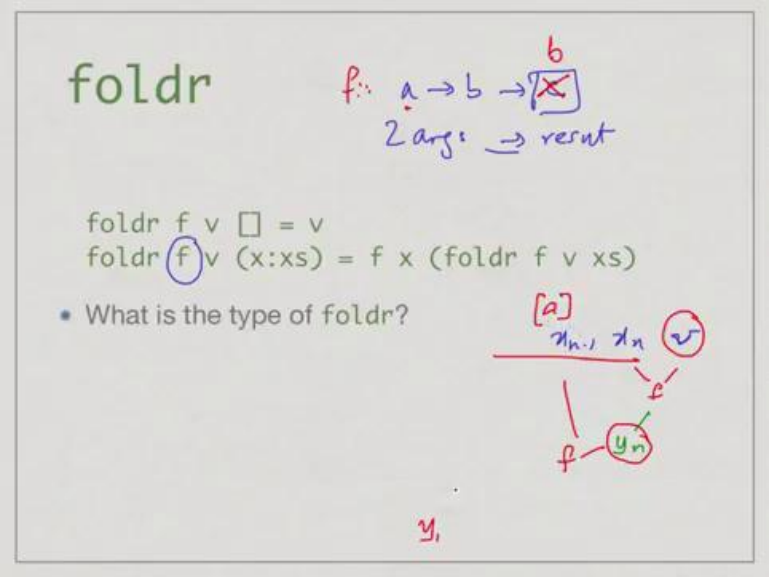
What would be fold the function plus, plus, the function that joins to list together using again the empty list. So, then we will see that this actually what is like concat, because dissolves one level of brackets. So, supposing have 1, 2 and 3, 4 and I take this, then if we take this plus, plus, this give me 3, 4 and we take this plus, plus is 1, 2, 3, 4.

(Refer Slide Time: 07:39)



So, this is just to same as the built in function concat. So, we can right familiar functions in different ways using foldr.

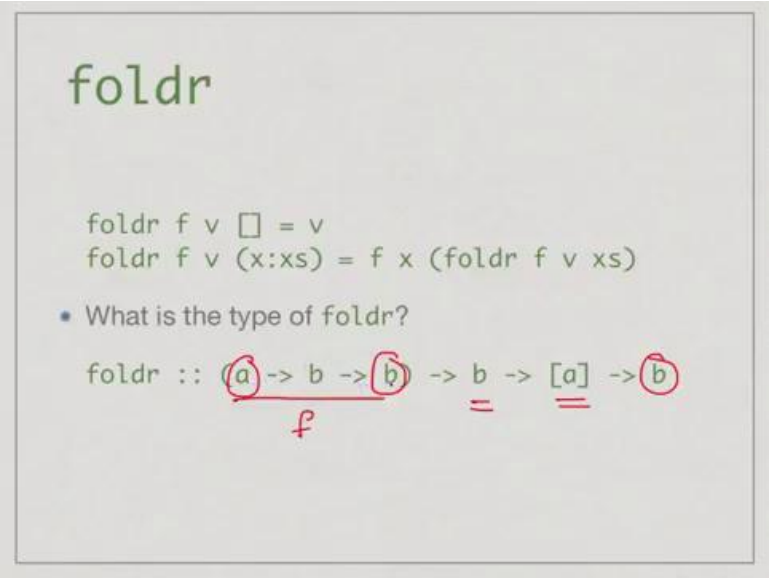
(Refer Slide Time: 07:49)



So, foldr as we saw before is depend like this. So, the question is, what is the type of this function? So, the type of this function involves first looking at the function itself. So, what is f 2, it takes two arguments and it produces the result. So, that three possible things could be in general if a to b to c. From the other hand, what we have seen is that the result, it produces is the right hand side is the next. So, we have, say the value v, then we have x n, x n minus 1, it is 1.

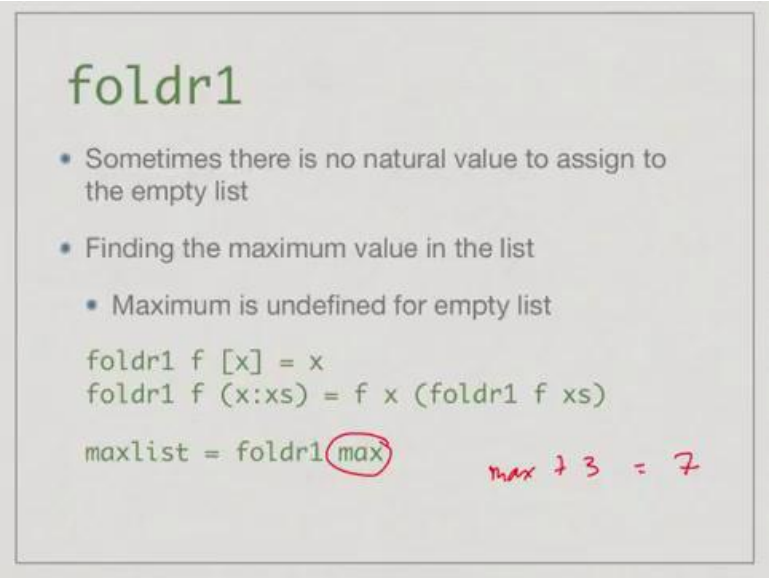
So, when I apply f to this and I produce a y n, then I am going to apply f to this again. So, therefore, the output of this function must be compare to them with the second argument of f. So, this has actually b and other b and then at the end on eventually going to produce y 1, which is the output of the function. So, that will be f b. So, it takes a function of the form f is takes two arguments of type a and b and producing output of b. So, I can keep repeating this process, we takes an input list, this whole input list in the type a and it produce an output of type b.

(Refer Slide Time: 09:06)



So, the type of foldr is the follow. So, this is my function, we take two arguments. So, this is an output of the same type is second argument, we takes an initial value which is a same type as finial output. It takes a list of inputs which are combatable with this argument in the f and produce an output, which is the output of the last application. So, foldr takes a function a to b to b and initial value b and input list of type a and produce an output value of type b.

(Refer Slide Time: 09:37)

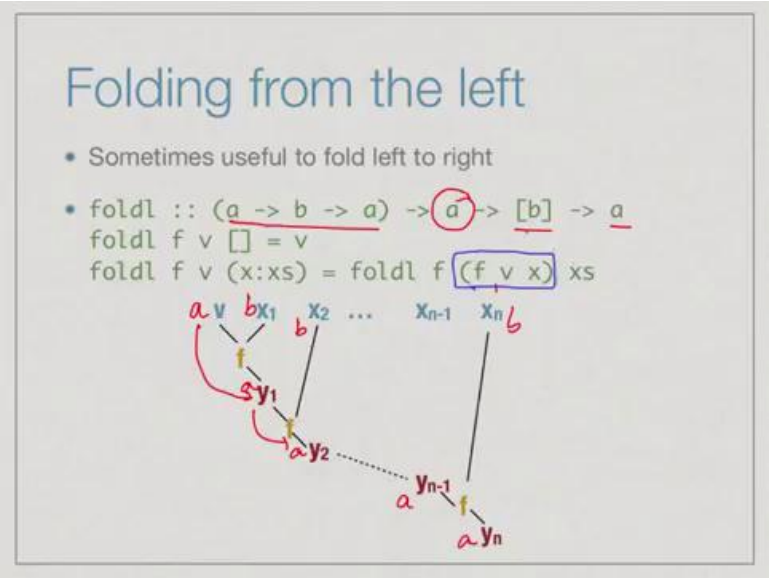


So, now in some situation there is no natural value to assign to the base case to the empty list. So, foldr basically takes two arguments initial value and then, the function. So, the function is sometimes may not be sensible for an empty list, for example, supposing we want to find the maximum value on the list, then there is no sensible maximum that you can assign to the empty list.

So, it makes sense to have function, which leaves is non-empty list, this in Haskell is called foldr 1. So, foldr 1 on a non-empty list, for it singleton will just return value itself. So, the base case for this is a non-empty list of length 1, because 0 is not allowed mod cannot have the empty list and otherwise, it behaves exactly like foldr. So, it takes the first argument of the non-empty list and applies f along with the result to bring to this.

So, in this case for instance the list, max list the maximum value of the list can be done by folding this is the built in function max, we have max of say 7, 3 is equal 7. So, there is a built in function in Haskell called max, which takes two numbers and compares and figure out 2. So, we start with max list of a singleton this being the value itself, we fold it through well is we get max list.

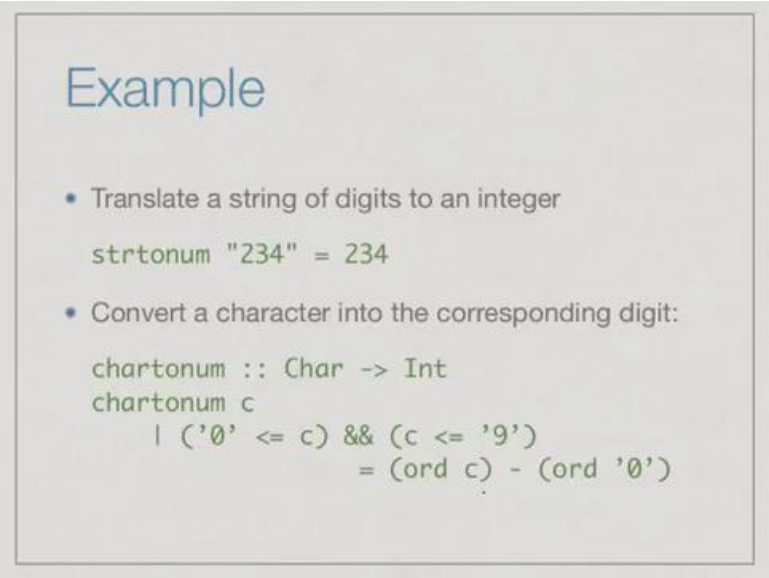
(Refer Slide Time: 11:01)



So, that is metric thing which you can do is to start from the left from fold from the left to the right. So, we start with the b at the beginning then we apply f to v an X 1 to Y 1 then we take Y 1 and apply f 2 Y 1 and X 2 to get Y 2 and so on. And then, eventually we get applying the last one is Y n. So, now, we can see that, this will have a slightly different type, because now I take a function to inputs are of type a and b, now, the output must be same as the first.

So, this much again we have type a, because again on the b, this will again b of type a and again b and finally, this is of type a type b. So, the function is from a to b to a and then, we have a initial value of type a, we have a list of b's and it produces an output type b and when we recursively applied, notice that, the first argument is the inductive thing of the left segment and we applied this value to out of this. So, slightly different from foldr, because they are going left to right.

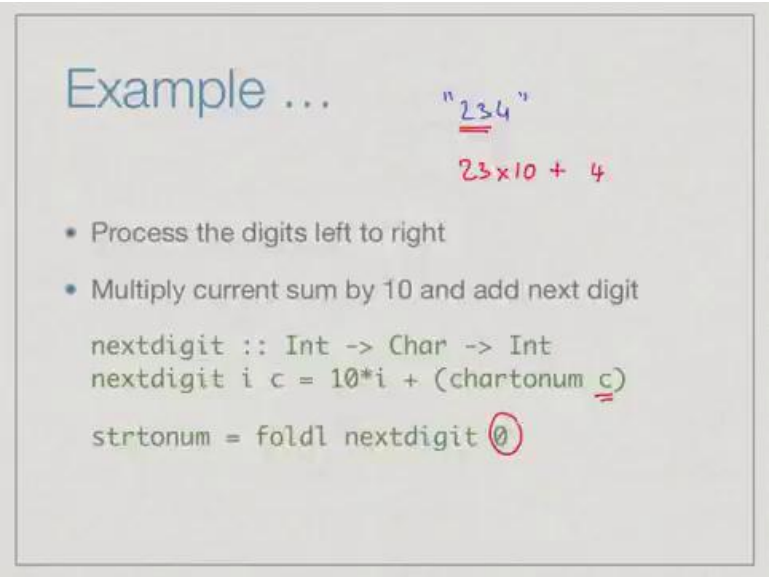
(Refer Slide Time: 12:16)



So, here is an example of using this fold l, if we want to take a string of digits written using a character and represented as a number. When, is the natural move from left to right, because as we see each digit, the number increases in it. So, base function can be 1, which takes a single character and converts to a digit. So, we have same function character to number char to number, which checks whether C lies in the range 0 to 9 characters.

Remember, we are soon that the characters are represent the tables on the digits are all introducing in this table. So, if it lies on this range, then we just compute the number by taking its offset to the table value of 0. So, 0 will be map to 0 and map to 1 itself.

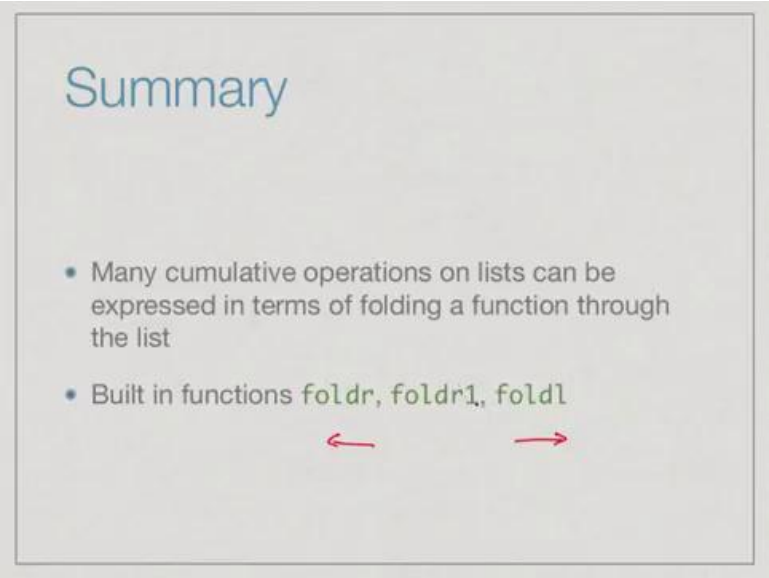
(Refer Slide Time: 13:12)



And now what we want to do is folds function from left to right. So, what we do is, we assume that we are say processed 2 and 3 and that going to 23 as a number. Now, I look at 4, so I will multiply this by 10 and add 4. So, this is we normally do it in place value thing. So, we multiply the current sum by 10 and add to next digit. So, the next digit will be 10 times for current digit plus the conversion of the next character.

Now, if you start with the base value 0 and go from left to right in the first step is to compare the character 2 to number 2, so I will get 2. Then, the next time I will take 2 times 20 and add 3, so we are get 23. So, I will get 23 and I will take 23 times 10 and add 4 and I will get 24. So, this is an example by where fold the function from the left to the right.

(Refer Slide Time: 14:12)



So, Mach and filter are functions which transform elements to elements individually, but if you want to combine the elements in the list into a single value as an output. Cumulative operation typically involves folding a functions with rest, applying it from one in to other end. So, the basic functions in Haskell is through this are foldr and foldl. So, foldr goes in right to left, foldl was left to right and there are versions of this functions, which do not have a sensible value for the initial empty list all foldr 1.