Functional Programming in Haskell

Prof. Madhavan Mukund and S. P. Suresh

Chennai Mathematical Institute

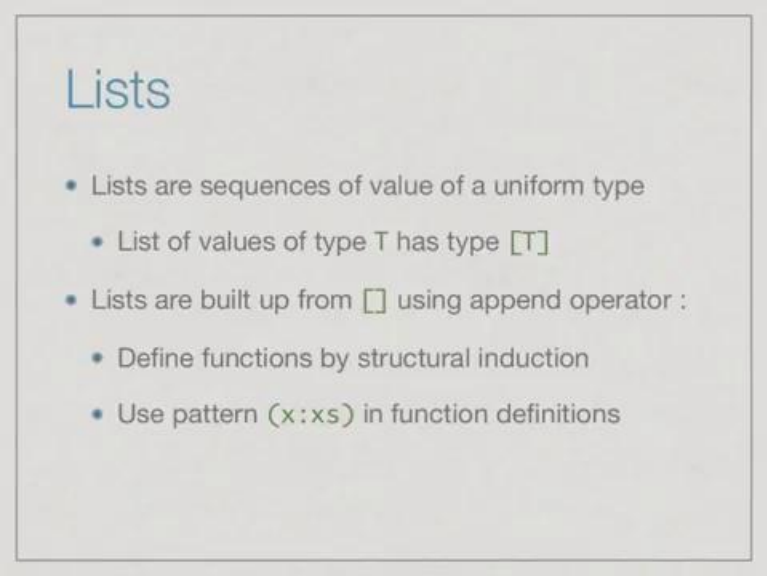
Module # 02

Lecture - 02

Functions on Lists

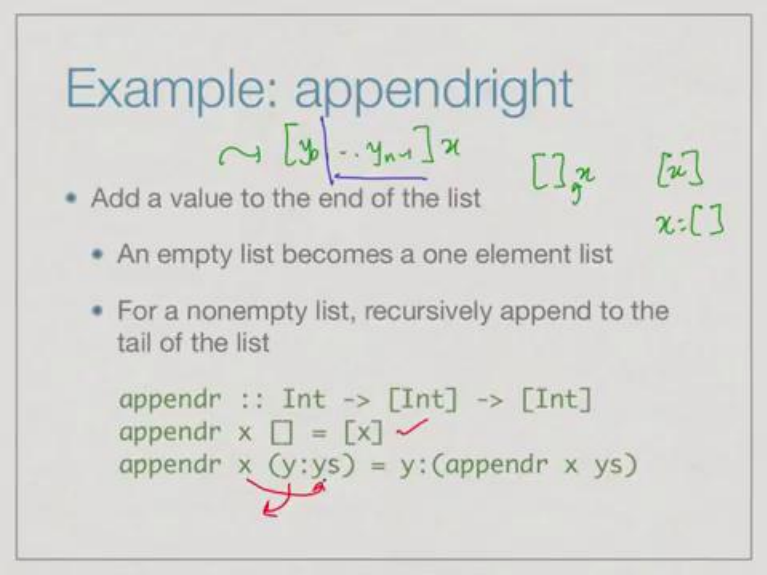
Last time we looked at list and so some simple functions on lists. So, let us gets some more practice and defining functions on lists.

(Refer Slide Time: 00:10)



So, recall that a list is a sequence of values of a uniform type and a type of a list whose underline values have type T is given as list of T between with square bracket on the bit. And all list and Haskell are built up in a canonical way using the append operator colon starting with empty list. And because of this we have a natural way to define functions on list using structural induction and we can use this pattern x colon and x s to decompose a non empty list into it is heads and it is tail in our inductive definitions. So, now, let us look at inductive definition or some interesting functions on lists.

(Refer Slide Time: 00:51)

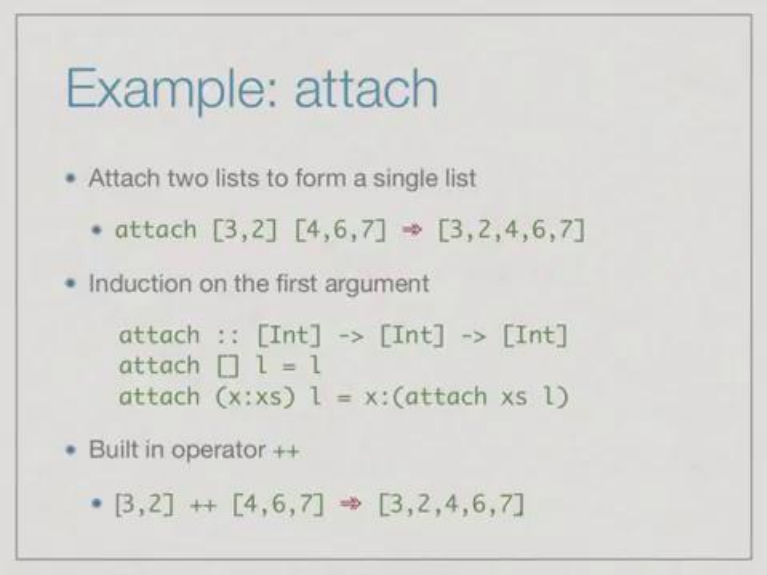


So, the built in append operator denoted colon attaches a value to a list on it is left. So, what if we want instead to attend attach the value on the right, so what if we wanted to append right. So, if we append right to an empty list if I attach a value here, then I will get a list consisting of the single value, which of course, the same of x colon empty list. So, append right to the base case just gives me a single term list and otherwise, it is inductive.

So, if I have a list y 0 to y n minus 1 and if I try to attach x on the right, then I can return that y 0 is the head of the new list and attach it to the tail of this list. So, here is the definition of the function we call it appendr, so appendr takes a value, so the first argument is an integer takes a list and returns a list. So, it is start Int to list of Int to list of Int, so as we saw the base cases if I want to append right a value x to an empty list I just get the singleton list containing x, so this is the base case.

And if I want to append right to a non empty list, I can pool this out and try to insert x into the y s. So, I have the first element of the new list will be the original y and then, what follows will be the result of inductively or recursively appending x to the right of x y s. So, this is again the same style that we saw before, we have a base case and an inductive style.

(Refer Slide Time: 02:30)

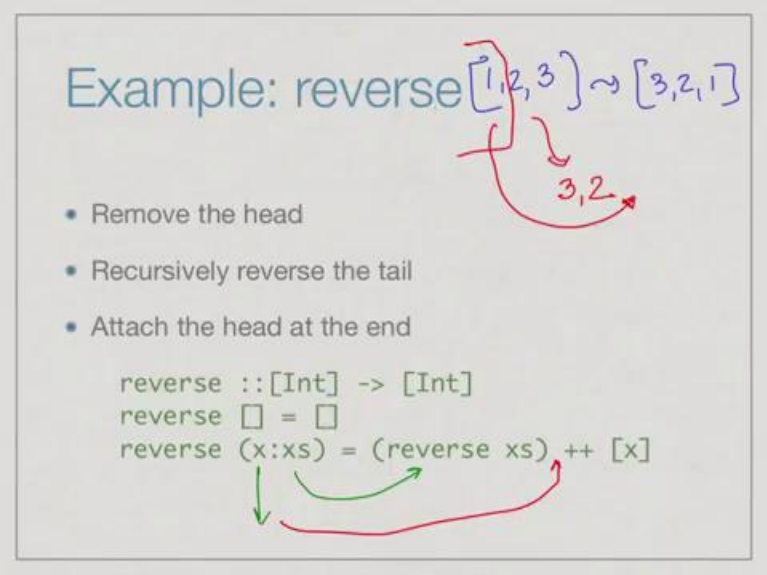


So, a natural operation on list is to take two lists and fuse them together, so let us call this function attach. So, we want to take a function, which takes two lists supposing we take 3 2 and 4 6 7 and we want to combine these in to a single list 3 2 4 6 7. So, we have to now in such a function we have to apply induction to one of the arguments, so let us just choose to apply induction to the first argument.

So, the first argument to the attach, so attach now takes two lists, so it take a list of integer, it takes a second list of integers and final two gives as the attached list, where both lists are combined. So, if the first argument is empty, then there is nothing to attach I just get back to this second arguments. The base case says that attaching the empty list to any other list l, this returns l. On the other hand, if I want to attach a non empty list to l, I can pullout the head of x, then I can attach the x s to l this in induction, then I can stick back the x.

So, that is precisely what we says is this inductively attach x s to l, now x is a smaller than x colon x s by 1. So, it is a smaller list, so this function is inductively defined and now, I can use the colon operated to stick the x. So, of course, this is an extremely useful function. So, it turns out the Haskell has a built in operator must plus, which does precisely this, so 3 comma 2 plus, plus 4 6 7 gives us 3 2 4 6 7.

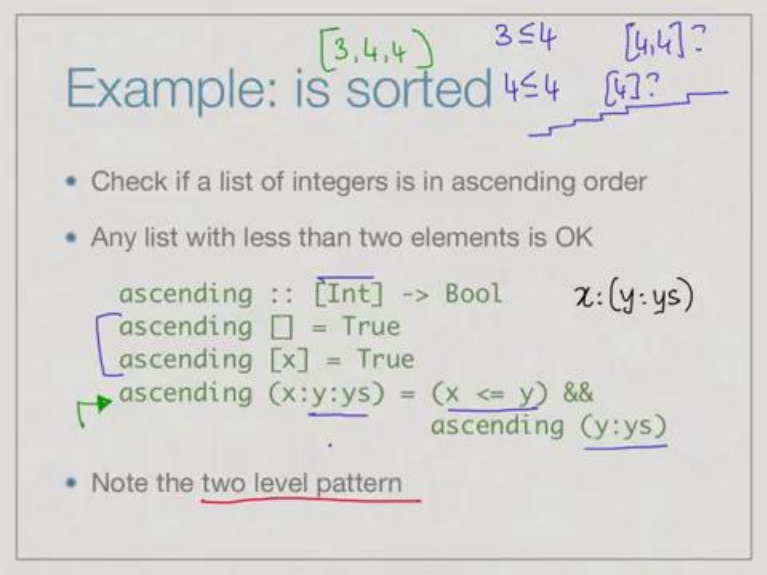
(Refer Slide Time: 04:09)



So, what if we want to reverse a list, so we want to take a list of this 1 2 3 and we want to produce 3 2 1. So, a natural way to do this by induction is to first, since we know how to extract the head extract this, then we take, what remains and reverse it, then we put this back on the right hand side using appendr or using plus plus. So, reverse takes a list of integers and returns a list of integers. If I have an empty list, reverse has nothing to do, so reverse of the empty list is just the empty list.

If I want to reverse a non empty list, then what I do is I decompose this x and reverse the tail. And then, I stick this x at the end using either plus plus or we could use appendr if you wrote last time go to ((Refer Time: 05:05)), but since plus plus is more compact we just use plus plus square bracket x. So, this is saying that adding the value x of the end is same as attaching the list containing the single element x.

(Refer Slide Time: 05:18)



So, our next function is one that checks whether a list is an ascending order, whether a list is sorted. So, we want if we think of the values in the list, then we, it should be going on up strict from, so need not strictly go up. So, we could have a section, where the values are equal and then we keeps going up, but when we go from left to right the values must been in ascending order, we could never go down.

So, since we are taking about ascending order this is like checking of an array is sorted. So, anything which has zero or one elements by definition sorted, because there is no comparison to make. Now, if I have two elements then I have to do something inductive, so it is sufficient to check that the first element is correctly ordered with respect to the second element and then, the rest of it is sorted.

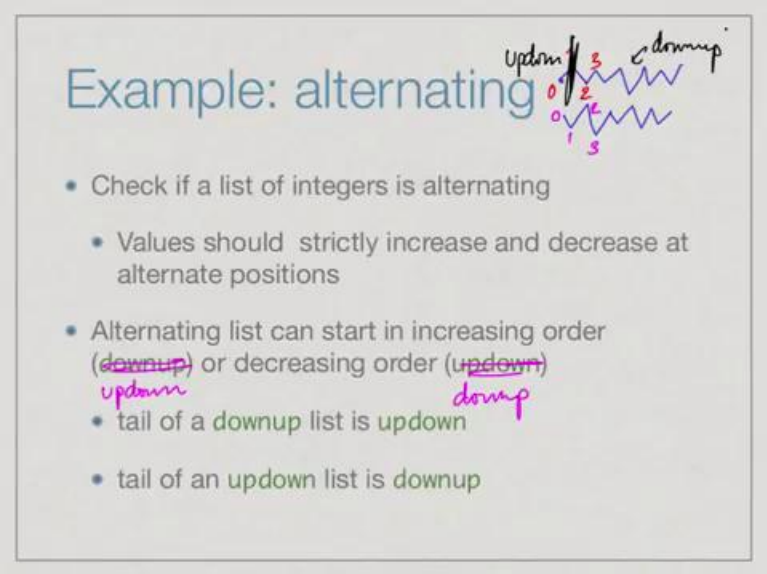
So, this is precisely, what this function says, it says that if we have zero or one elements, then ascending is definitely two, so ascending takes a list of integers and produces true or false. So, it is list of Int to Bool, so if it has no values or only one value, then ascending is trivially true. Otherwise, now notice that if it has reach this point, then the pattern x colon y colon y s make sense, because we have definitely at least two values in the list and remember that this is a short form for x colon y colon y s, which internal go further and so on.

So, I can decompose any number of levels provided that number of levels exists. So, this pattern will not be matched unless there are two values put it they are that there are two values from the first value will come to x, second value will come to y and whatever

remains it may be empty will go to the y s. So, now the inductive definition ascending says check that the first two values are correctly ordered that x is less than equal to y and that, what remains after that is also sorted.

So, this will walked on for instance, if we want to check it we says 3 4 4 for instance the first thing we check 3 is less than or equal to 4 and now is 4 4 7. So, this will say 4 is less than equal to 4 and is the single list fours add it now the singleton list will care match the base case and it will say. For the interesting thing about this particular definition is that we can have a two level pattern like this which are access not just the first element., but the second element or even the third element and we have a 3 level pattern provided a list as that new elements we can individually name all those elements directly in the pattern.

(Refer Slide Time: 07:58)



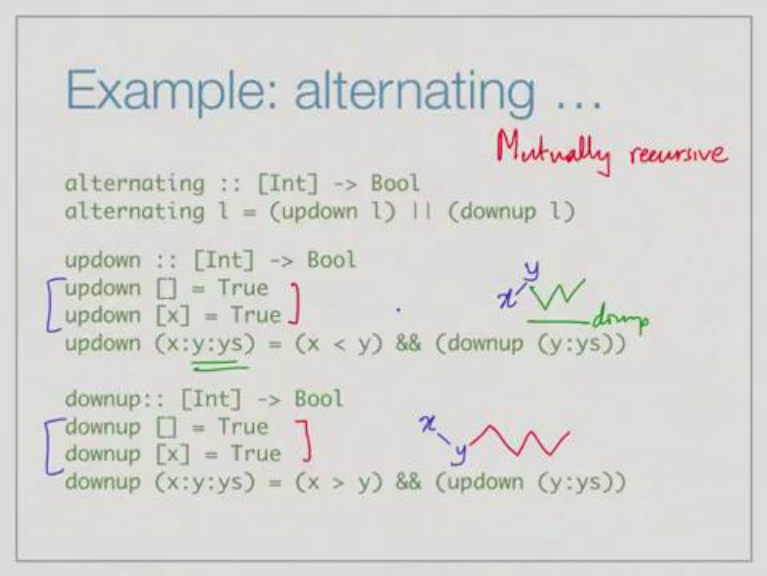
So, here is a more interesting version of the previous thing so, supposing I want to check that the list either looks like this that is the values keep going up and down or it looks like this the value skip going down arrow. So, we want to check, if I list of integers alternating, so the value should strictly increase and decrease of alternate positions, that we have seen, that there are two possibilities it could starts strictly increasing.

So, this is position 0 and in position 1 is bigger and position 2 is smaller 3 is bigger than and so on are one could be other where on and position 2 and it goes down and it is 2 goes on and 3 goes on and so on. So, let us look at the second case so should be list, so this starts an increasing order I will say it is an up down list it goes up first, then its goes down and if it starts from decreasing order I will says the down up this goes down and

goes up.

Now, if I cut of this list at this point, so it started in up down list, then this position this the remaining part must be down up. So, that is the observation that up down and down up is connected. So, if I start in up down list and I move the head the tail must be of the opposite type similarly if I have the down up list on move the head the tail must be opposite line. So, this is an interesting definition, because we will define up down, down up in terms of each other, so just let us look at the definition.

(Refer Slide Time: 09:31)



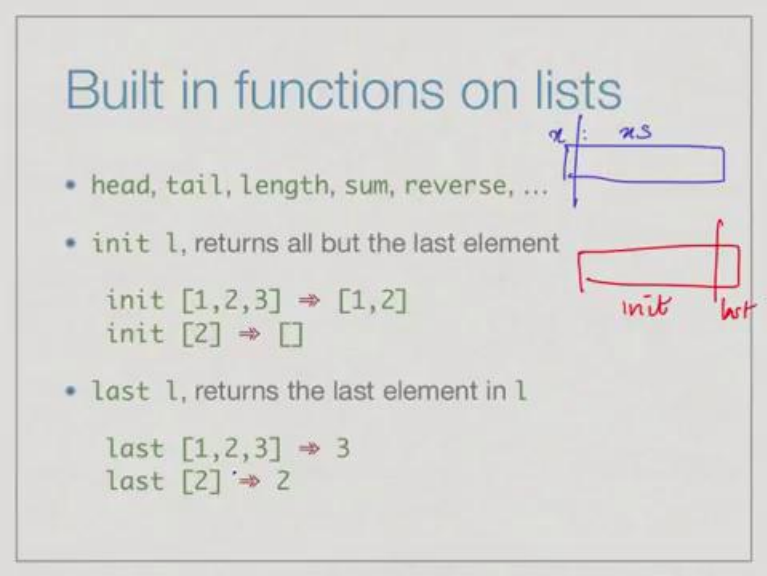
So, first of all we say that a list is alternating if it is either of the form up down or it is of the form down up, when is it up down when it goes up and then comes down. So, up down and down up just like sorted at trivial for list, which do not have at least two values. So, if I have one value a zero value, then both up down and down up will return two.

And the other hand, if I have two values then up down must start by going up. So, I want x and then y I have wanted to up, so I want x less than equal to y and now, as we observed once you go up the remaining part must go down. So, the next step must be down up, so the list starting from y onwards will be going down and then, up and then, so this part will be down up a symmetrically here if I want to start going down when I want that the first x is bigger than the second y for, then after this it is up down.

So, these are what are called mutually recursive functions, so alternative says that the list must either be of the form up down or down up, up down and down up both of their base

cases. And then, they are defined in terms of each other the first position determines is whether it is up or down then you reverse. So, this is a kind of interesting mutually recursive function on list.

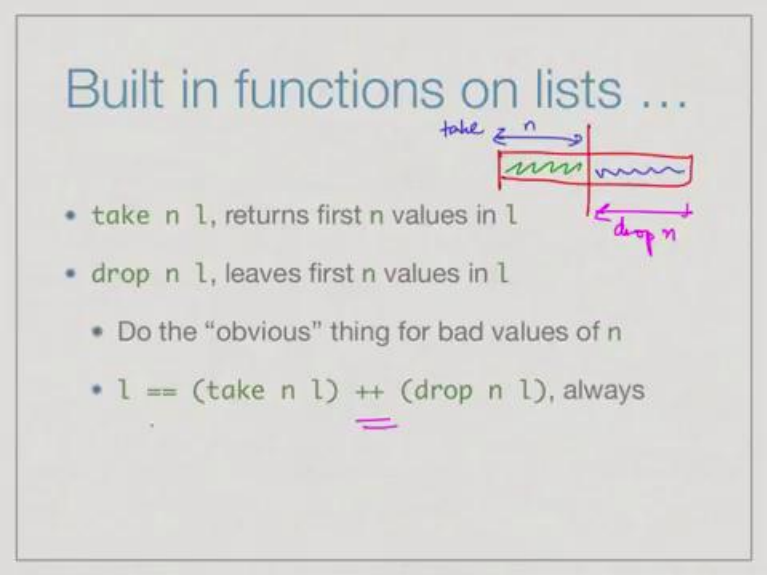
(Refer Slide Time: 11:03)



So, Haskell of course, like you would expect as many built in functions on list. So, some of the functions we have seen and some, which we are defined or actually built in functions, so head and tail we have seen length sum and reverse we wrote, but actually they are built in functions. So, you can take a length of a list sum of a list reverse a list and so on. Now, the opposite of head and tail is to decompose the other way.

So, remember that head and tail the knock of the first element from the colon and the remaining 2, now it maybe that you want to knock of the last element. So, then this is called the initial statement and this is called the last value, so there is a function in it, which will returns everything except the last element. So, if I have a 3 element list from the first two have a one element list it will turn empty because if I move the last element nothing is there and last will be the value at the l. So, last of this list is 3 last of the list containing 2 is 2 and once again in it and last will only work if the list has at least one value. So, you cannot define in it or last for an empty list.

(Refer Slide Time: 12:15)

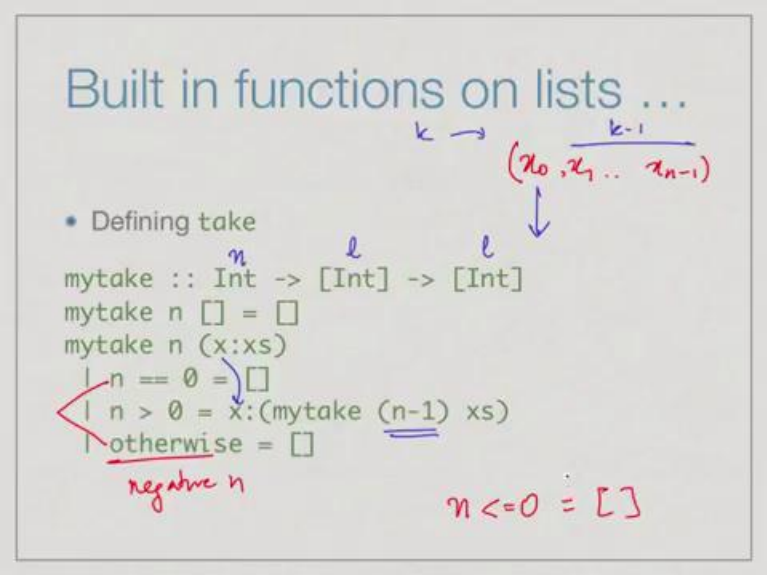


Sometimes you go and want the first of the last, but you want to break a list at some point. So, supposing you have a list might want to take either the left hand path part up to a certain position are you might want the right hand point from a certain position of values. So, these are functions called take and drop take gives me the first n value, so I will have n values here, if I take and drop will on the other hand give me these values.

If I drop the first n values and I will get the n plus 1 value onwards and take and drop will do the obvious thing. So, if I try to take 0 values I will get an empty list if I take try to taken negative number of values. If I take minus five values I get a empty list, if I take more values in the list can has it will give me the entire list. So, it is not going to actually worry about whether n is within the range 0 to l length of the list minus 1 and the same with drop if I say drop minus five values if I drop nothing if I say drop everything, then it will give me an empty list.

So, the useful thing they remember is the between take and drop there is no gap. So, if I take n elements and if, drop n elements then every element either gets in to the first part to the second part. So, if I then combine the using my attach or concatenate operator when I will get back the original, so for any n any value of n, whether n is a sensible value or not, whether it is within the range 0 to length of list minus 1 or not take n l plus, plus drop n l is always equal to l.

(Refer Slide Time: 13:58)

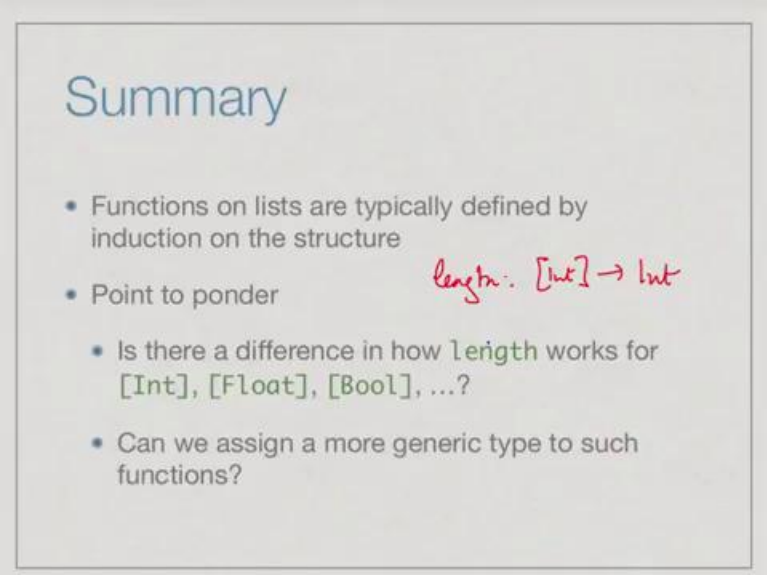


So, just to exercise are skills in writing function on list, let is try to define our own version of take. So, remember the take as take a list a first number and the list and give me back a list, so I takes a Int a list of Int, which make a list of Int. So, if I take n any n values from empty list I get nothing, so empty list I cannot take anything it was nothing get. So, for any n my take n of the empty list is n, now if it is a nonempty list I will use induct value, I will go by cases of n.

So, if it, if I am taking no values, if I says 0 values at we taken, then I get back to empty list. If, I want to take n values, so I have x 0 x 1 this is my take supposing I want to take k values from this, then the ideas I will take out this thing and then, I will take k minus 1 value somewhat remains. So, I pull out the first value and then I take 1 less value from the rest of x s and remember that, if n is very large, then a sum point x is become empty and this will just return.

And finally, we have a last case, which takes care of negative n value, so it says a n is actually less than 0 I should give you nothing. So, notice that actually these two cases are you get, so I could of set n less than equal to 0 is equal to empty and return on the same.

(Refer Slide Time: 15:33)



So, we have seen a number of examples of functions on lists and almost any interesting function to write on a list will be define using the structure of the list by induction. So, something that we should think about is when we wrote length we explicitly wrote a length for a list of integers. Now, if we want to write length for a list of flow a list of move a list of part list of anything else, then it is clear of the function will do the same thing it will just say it is 1 plus the length of the tail it does not have to look at the value is inside the list.

So, the type of the value inside the list is really irrelevant to list to length. Similarly, reverse that is want to reorder it just a structural part I want to take the, if I have a bunch of boxes and I want to reverse the sequence of boxes I do not need to look at inside the box we find out, what is there, I just need to move them around. On the other hand way we are return types for list you have force to say this is a list of Int or this is the list of float or this is a list of Bool.

So, question to thing about is, what would be a good way to assign a more generic notion of a type to functions like this, which do not actually need to look internally into the value, but I just need to note this structural things.