Functional Programming in Haskell

Prof. Madhavan Mukund and S. P. Suresh

Chennai Mathematical Institute

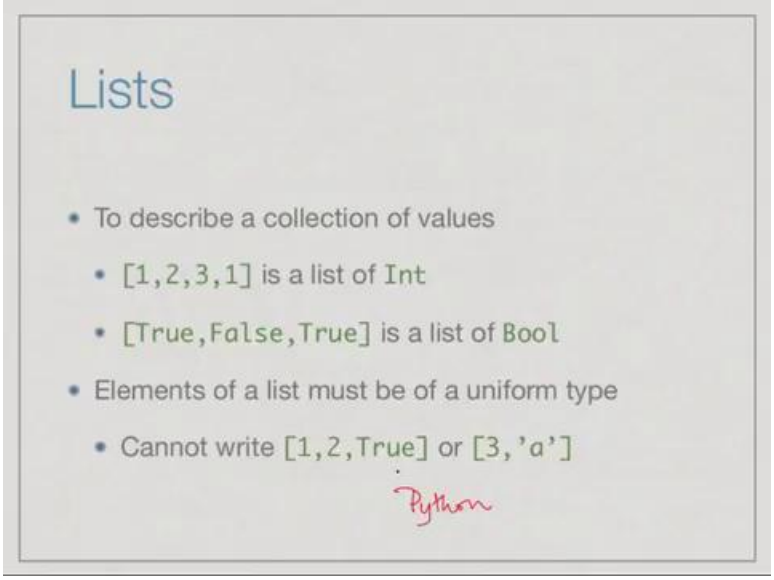
Module # 02

Lecture - 01

Lists

In any programming language it is important to be able to collect together values and refer to them by a single name. In Haskell these are done using lists.

(Refer Slide Time: 00:12)

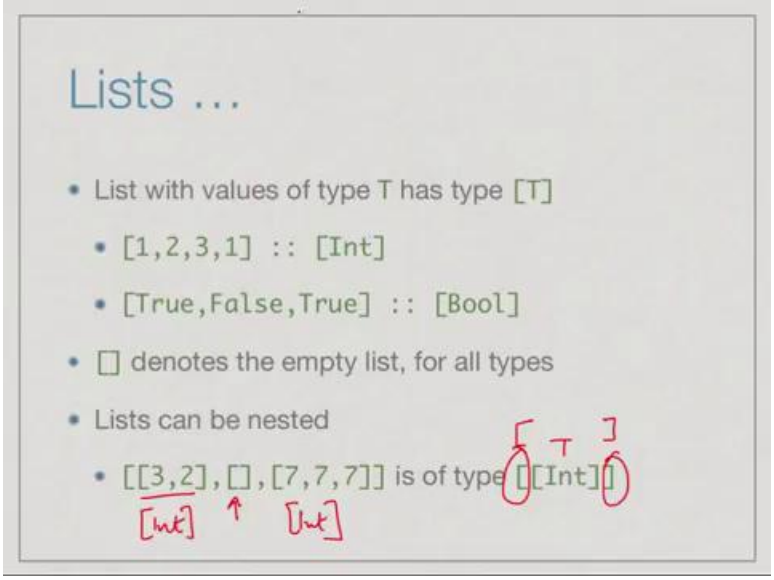


And other programming languages that you may know like C, C plus plus or Java we usel data structure for collecting a bunch of values together is in array. So, you might have come across lists before, but in Haskell it is fundamental that the basic collective type is list. So, list is the sequence of values and more importantly it is the sequence of values of a uniform type, lists are denoted using this square bracket and comma notation. So, these square brackets denote the beginning and the end of the list and the values are separated by commas.

So, the list 1, 2, 3, 1 is a list of integers, so it is a list of type Int, True, False, True could be a list of values of type Bool and so on. So, the crucial thing is that the underlined elements must have a uniform type. So, we cannot write for example, a list with mixed types like, 1, 2, True or 3 and a. So, if you are familiar with language like python, python does allow this kind of mixed thing and called it is a list. So, these are not the same as

python list, in python list the values redrawn to a uniform type and in Haskell list, it must be of uniform it.

(Refer Slide Time: 01:27)

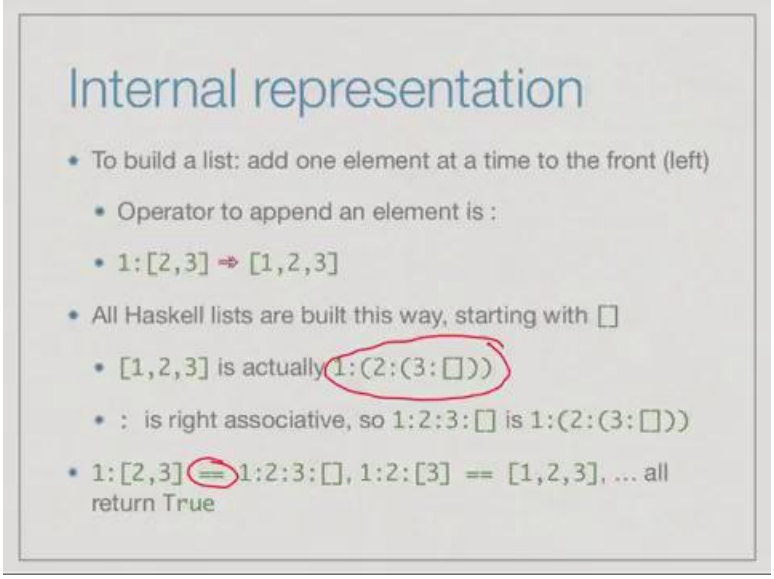


So, since the underlined elements are of uniform type, we can assign a type to the list itself. So, if the list of values has type T then the list itself has type square bracket of T. So, now, since here inside we have ints in the first list, we say that the type of the list is square bracket Int, similarly since these are Bool, the type of this list is square bracket Bool. The notation for an empty list is just an opening bracket followed by closed bracket.

Now, technically there is an empty list of type, list of Int; there is an empty list of type list of Bool and so on, but we will uniformly use the same notation without specifying what type it is. So, for every type of list the empty list is given by the same symbol, so this as a type should be read as list of int. So, if you want to read it out what is the type of this value, this value has type list of int. So, of course, list can be nested, so we can have list of lists, but again each of the internal list must have the uniform type.

So, here now we see that each of these is itself a list of int, this is the famous empty list which could be denoting any type, but we like contacts it is a list of int, again this is a list of int. So, we have three list of int themselves enclosed in the list, so this overall thing is list of list of int. So, this is my underline T and my outer bracket, so my underline value type is list of int and now I have a list of this.

(Refer Slide Time: 03:06)



So, list are build up in a very specific way in Haskell, list of build up by adding elements at the front and this operator is denoted by colon. So, if I have a list and if I want to put an element in front of it, then I can put the colon and I have the value in front. So, 1 colon 2 3 gives me the list 1, 2, 3, so what we will be doing in this notes we will try to do at least…We use this symbol to denote the fact that this value returns this value. So, it is not like an equality, but this expression is equivalent to this expression.

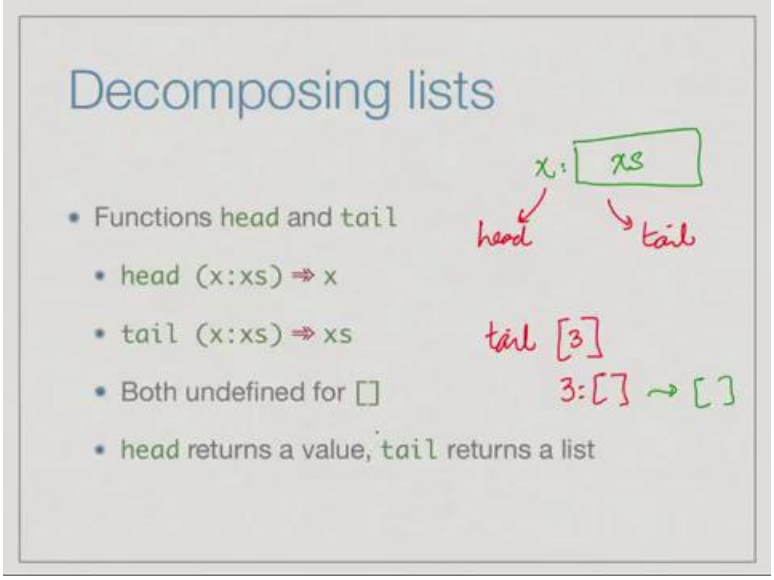
So, Haskell actually builds up list in a standard way using this operator. So, we always start with the empty list and then keep adding elements in front of it. So, when we write in this 1, 2, 3, but actually Mills means that it was built in reverse by first appending 3 to the empty list, then appending 2 to the 3 then appending 1, 2, 3. So, the list 1 comma 2 comma 3 that we write for eligibility is actually in terms of this fundamental colon operator. It is actually 1 colon and 2 colon and 3 colon, empty and so this is the initial step, then this is a next step and this is the final step.

Now, of course it is always tedious to write all these brackets and so similarly to types. So, if you remember we said that for rate int star int star int is actually bracketed from the right. So, we have implicitly that this is the bracket int, so this is what is called a right associated operator, we associated from the right. So, similarly colon is a right associated operator, so if I just write the colons without brackets it means a bracket from the right. So, I can write 1, 2, 3 and 2 as 1 colon 2 colon 3 colon empty list without writing the brackets and it will mean the correct thing.

Now, it is important to note that these are all just different ways of writing the same thing. So, if I write 1 colon and the list 2 comma 3, but if I write 1 colon 2 colon 3 comma then colon the empty list, all of these are actually the same value. So, if I try to ask the interpreter by this is equal to that. So, this remember is our equality check, then it will turn to, similarly if I say 1 colon 2 colon and the list 3 this will be equal to 1 comma 2 comma 3 and so on.

So, whenever we write these we should remember that the way we write it does not change the fact, but all of these forms are internally this form, straight for internally the only form that matters is there of the form 1 colon 2 colon 3 colon empty list. How we write it as flexible.

(Refer Slide Time: 05:50)

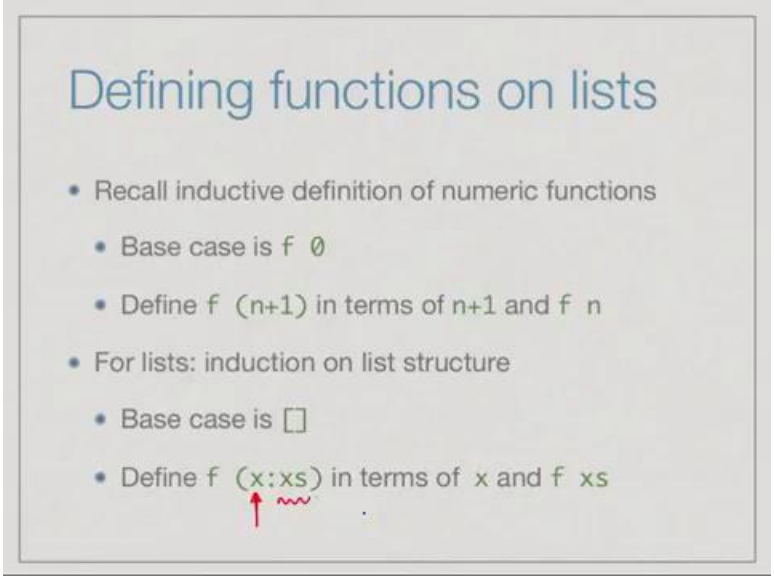


So, when we have a list now we therefore, have in some sense the first value and then we have the rest of the values attached by colon. So, it is conventional if we use the term x for the first value to use x s. So, this is an x followed by a bunch of x s that is always precedence. So, now, what we have is this operator that puts them together, so we have functions that take them apart. So, this is called the head of the list and this is called the tail of the list.

So, the head of the list is the value and the tail of the list is another list, in particular if I take tail of list with one value, this is the same as 3 colon empty. So, here the tail will be the empty list, so we can only do head and tail on list which have at least one value,

because otherwise I cannot split it I cannot split the empty list. So, both head and tail are defined on non empty list, head returns the value, tail returns the list.

(Refer Slide Time: 06:57)

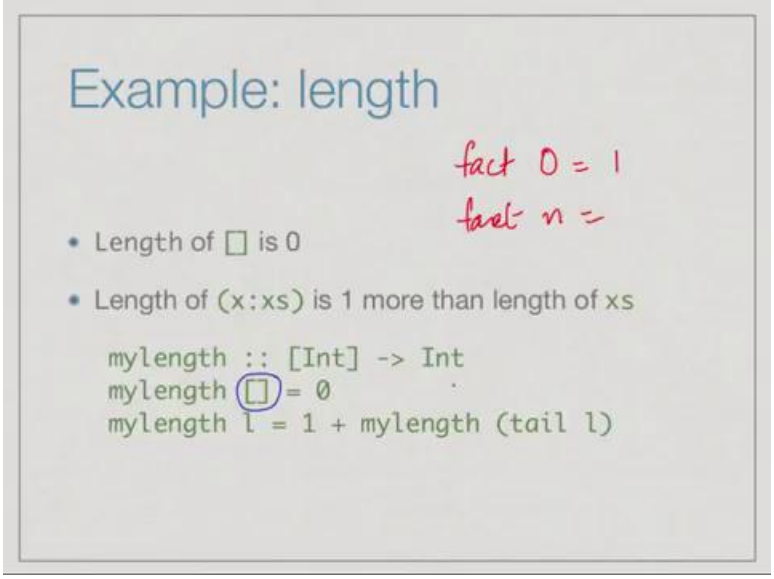


So, now we would of course, like to define functions at operators on list, so it terms out that induction is a good way to define functions on list. So, recall how we did inductions from numeric functions, especially for functions on third languages. So, we said that we would define a base case for the value 0 and then for a value n plus 1, you would define f in terms of the value n plus 1 itself and the value of the function f on a smaller value f of n.

So, for example, we said that n plus 1 factorial, this n plus 1 the value n plus 1 multiplied by n factorial. So, this is the value or the function, so this is f of n, so this is how we did inductive definitions on numbers. So, what is the natural notion of induction for the list? Well, it turns out that for the list the natural notion of induction is the structural list or you can think of it in terms of the length of the list. So, the base case is the list we start with, which is the empty list and then we add one element at a time.

So, if we have a non empty list we can strip of the outer most colon and define the function in terms of the value that we get, the head and the value that we have inductively computed on the tail.

(Refer Slide Time: 08:19)

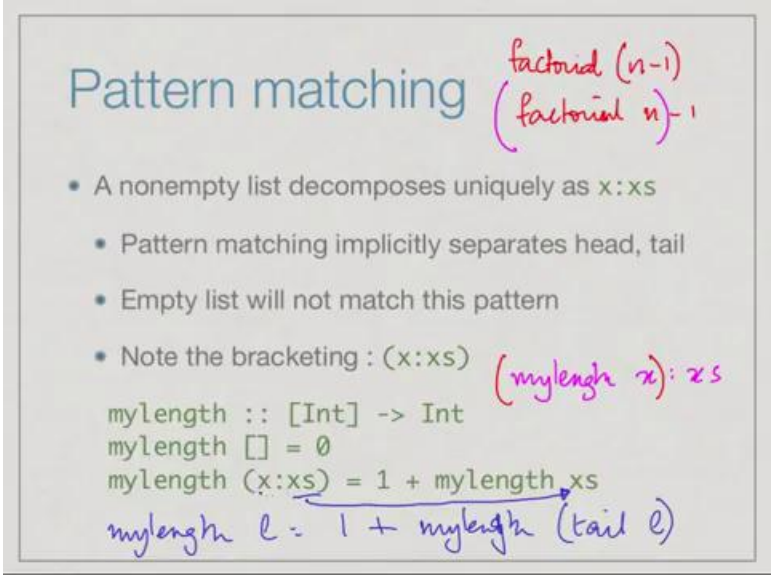


So, let us look at an example, so supposing we want to compute the length of the list. So, it is very clear with the length of the empty list is 0 and the length of any list to more than one element is one more then the list length of it is tail. So, here is the function which we called my length, because there is a build in function length and we do not want confuse Haskell interpreter if you actually write try out. So, my length takes the list of integers and in turns an integer which is the length of that list.

So, the base case as we did for, so if you remember we use to write factorial of 0 is equal to 1 and factorial of ints in something. So, again we have two cases we have my length on the empty list which is 0 and now if it is not the empty list then it will not match this pattern. So, this is like a pattern now, so if I given argument to the not the empty list, it will not match this definitions to go to the next definitions.

Now, I am guaranteed that the list is not empty, the list is not empty it has a head and it has a tail. So, I can said that the answer is 1 plus inductively the length of the tail. So, this is the typical inductive definition of a function operating on the list.

(Refer Slide Time: 09:34)

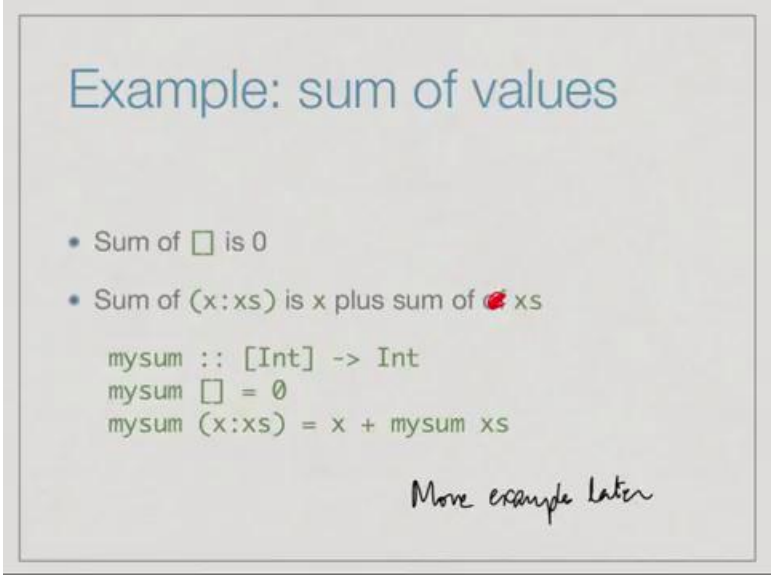


Now, it turns out that we can use pattern matching, because of the unique way in which list are build up. So, we do not have to actually explicitly called head over tail when we write inductive definitions, every non empty list can be uniquely decompose as it said and it is tail and this can be represented as an expression of the form X colon X S. So, the first variable we first to the head, the second variable the first to the tail, so we can write this same function is as follows instead of writing tail in the second definition.

So, earlier we had did in my length l is 1 plus my length tail of l this was our earlier definition. So, now, we say well let us directly break up our l which is not empty into the head and the tail and just use the variable that matches the tail as for recursive call, now notice this bracketing. So, we had the same problem the beginning when we wrote and our first week, we said that we must write this, because of you write factorial of n minus 1 without brackets then it will Haskell will try to bracket it at this way, because function application as a tighter braining it has precedence over arithmetic.

So, similarly here function binding will have precedence over this. So, it will try to say that this is my length per head my length of x colon x without this it will try to say it is my length of X colon X S in a little probably give a type mismatch. So, we have to put these brackets when we use a list pattern for a non empty list.

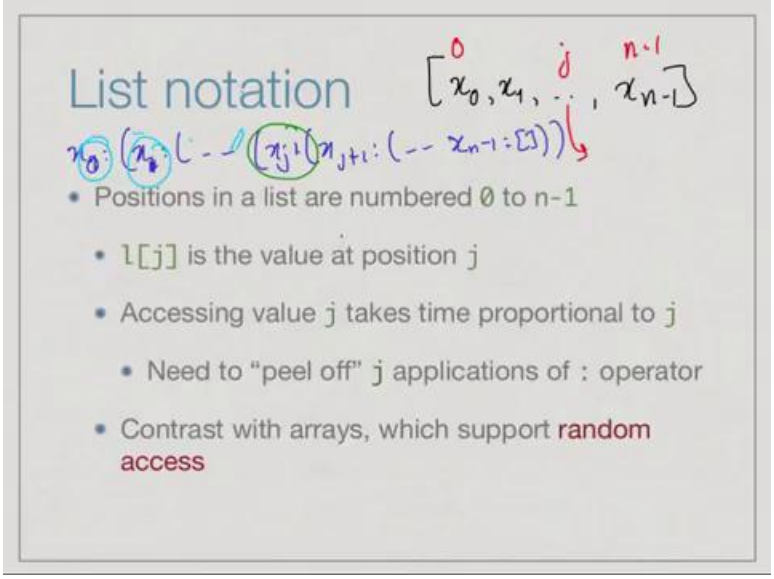
(Refer Slide Time: 11:16)



So, just to get some practice let us look at another function which is inductive define, supposing now we want to take a list of integers and add them up. So, we want to sum up the values in a list. So, again the sum up the empty list is just 0, because there are no values and now if I want to sum up a non empty list I inductively sum up the tail and I add the head which, so it is a x plus the sum of the tail. So, it is easy enough write my sum up the empty list is 0 my sum of X colon X S is x plus my sum of X S, so this is the general pattern.

So, you write inductive definitions by defining the value for a non empty list in terms of the head and the value of the function on the tail. So, we will see more examples in a subsequent lecture this week of more interesting questions and just length and sum we will see several examples to get familiar with this one set.

(Refer Slide Time: 12:21)



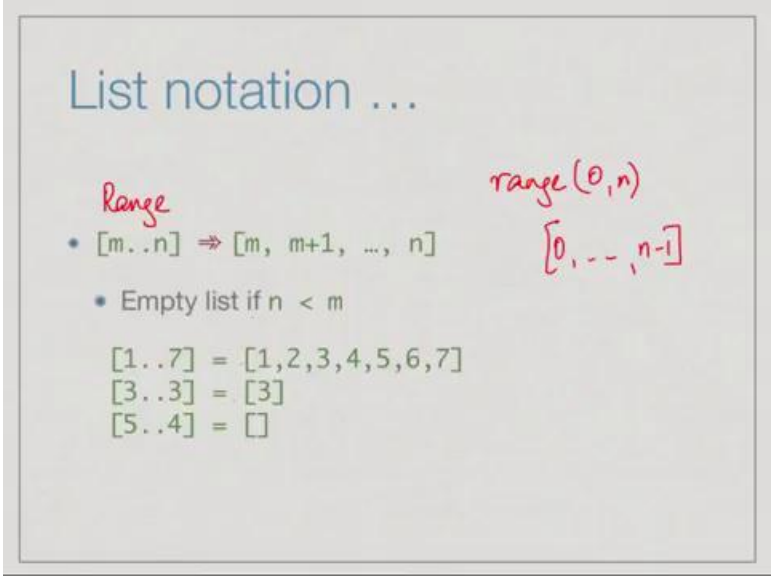
So, list is a sequence, so we have implicitly values in at which are numbered with respect to 0. So, if I have a list with n values I should think of that list as consisting of x 0, x 1 up to x n minus 1, so we are positions 0 to n minus 1 this is a quite traditional. So, arrays in languages like C, C plus plus, Java start with 0, list in python start with 0 and so on, so the positions are 0 to n minus 1. So, Haskell gives as a expression which looks like an array access to the square bracket to access the jth position, to jth position remember will be a j plus 1th value is the position start from 0.

So, if this is position j when l j will be the value at that position, now one important thing to remember is that actually this is only a short form for the following expression which is x 1 colon x 2 colon and so on and then x j colon x j plus 1 colon and so on then finally, x n minus 1 colon empty. So, this is what the list will which like, so in order to get to the j'th value have to first I have to extract the first this colon then act to extract this colon. So, I have to peel off each then until I reach this value, so takes me j steps we are actually get through, so this should be these are x 0 x 1. So, to get from x 0 x 1 to x j takes me j times.

So, therefore, accessing a value in a list is not for constant time operation unlike an array, if you have an array in memory when you can compute the position of any value in the array by looking at the offset, because these are all define to be contiguous values of uniform sides in a list we only know that it was attached. So, we have to detach the preceding elements to get to the internal elements, so it is not a random access device.

So, you take time proportional to the position in order to get to an internal value inside the list. So, this is important when we saw looking at functions at trying to determine their efficiency and complexity. So, some functions which work well on arrays which can access a, j in unit time will not work, so well on Haskell list.

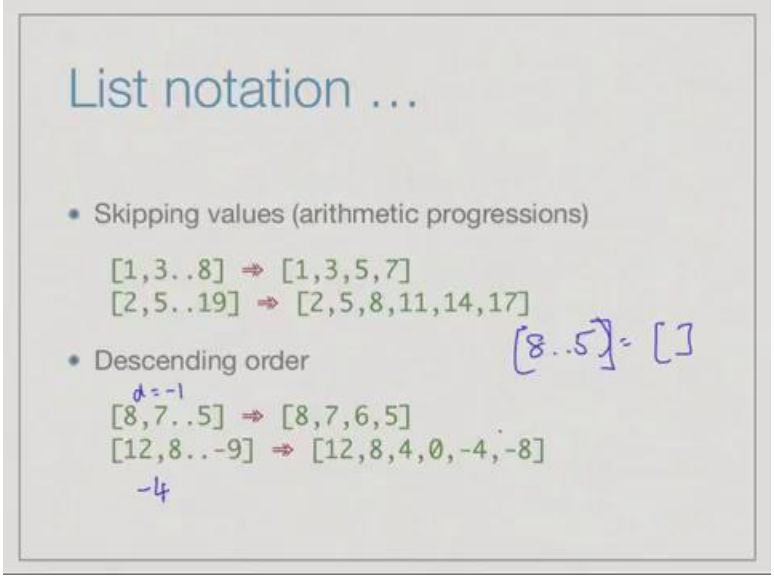
(Refer Slide Time: 14:49)



So, Haskell has some nice notation to denote sequences of values. So, if you want range of values from m to n which is write m colon colon n, this means the list m, m plus 1 m plus 2 and so on up to n of course, now if then upper limit is smaller than the lower limit then this will not give as a list. So, if I say 1 colon colon 7 I get 1, 2, 3, 4, 5, 6, 7 notice that the last value is included some of you if you note python will see that in python if I write range 0 n then I will get the list are goes from 0 to n minus 1.

So, in Haskell it is not like that and Haskell the upper limit is included in the final list. So, if I say 1 to 7 I get 1 and I get 7, so if I say n and I use the same upper limit is a lower limit then for the value of the list 3 colon colon 3 dot dot 3 is 3 and if I use on upper limit which a smaller then the lower limit then I will get an empty list, because starting from 5 I cannot go up. So, if I idea is that you start from the lower think and keep adding one, so long as you do not cross the right once a cross the right. So, once I go to 8 a stop and I threw it away. So, therefore, if I say 5 dot dot 4 I get the empty list.

(Refer Slide Time: 16:11)



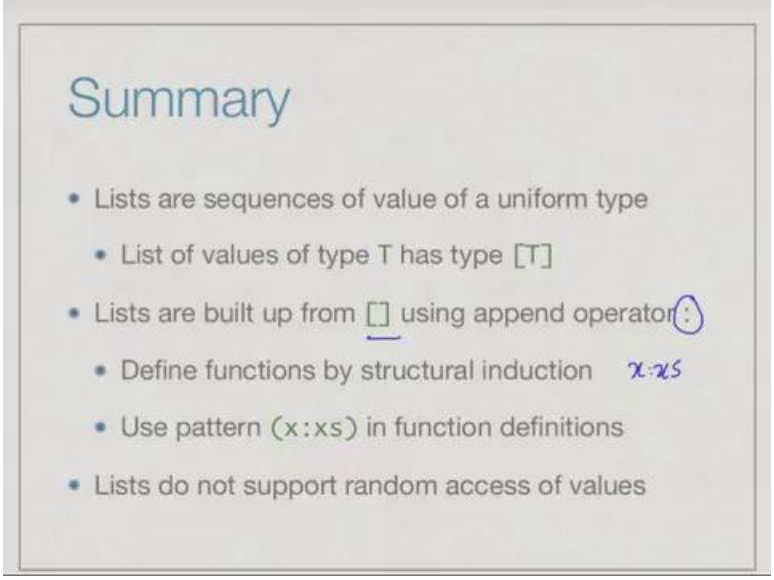
Now, sometimes we would like to get not every value, but values by skipping a fixed amount want you call it arithmetic progressions. So, we want a, a plus d, a plus 2 d and so on and now of course, when I say a, a plus d, a plus 2 d there is no guarantee that the number I choose on the upper bound is actually part with the progressions. So, here if I do the way we do it an Haskell is you give that difference by example. So, we give the first value and then we give the second value. So, implicitly this as the d is equal to 3 minus 1 is equal to 2.

So, which says that I must be skipping 2, so 1, 3, 5, 7 then the next value generate could be 9, but 9 would cross this upper limit a way and so we stop it. Similarly, if I say 2 5 then here it says the d is equal to 3 and so I have 2, 5, 8, 11, 14, 17 the next value would be 20, but 20 goes beyond 90, so we have to start. So, we said that if we do something like 5 dot dot 4 will get to the empty list, but it is often use suit to be able to count backwards and not forwards.

So, we can use this idea of an arithmetic progression to count backwards by giving the second value is smaller than this value. So, if I just say 8 colon dot dot 5 this would give me the empty list. Because, it would implicitly try to add one if I say 8 comma 7 dot dot 5 then I fix d to be equal to minus 1. So, now, it will start counting down it will go 8, 7, 6, 5 and the ideas of crossing, it is not greater than or less than if I cross the right hand side limit then I stop.

So, when I go to 4 across that limit, so I stop likewise, here the difference is minus 4. So, if I keep going down after minus 8 the next one could be minus 12. So, I would a cross minus 9 so I stop. So, it is quite intuitive and it is very clear out specified ranges in a Haskell.

(Refer Slide Time: 18:04)



So, to summarize a list is a collective set of values of an uniform times. So, it is a sequence if position 0 to n minus 1 and because there of uniform type a list of values of type t as type list of t denoted by it is square brackets around the T. So, list are internally represented in a canonical way all list have built up from the empty list using this append to the left operator, which adds one element to the left each time which is denoted by colon per name from the operators colon.

And because we have this canonical way and which lists are built up you can decompose list using same operator and define functions by structural induction, we can define functions as the base case on the empty list and then define it on list of the form X colon X S and we can use this pattern X colon X S is an function definitions to easily defines such inductive functions. In the last thing to remember is that, because the way that list are represented and implemented these are not random access this way, these are not like arrears in C, C plus plus, Java it does take time to postponed to the position to reach an internal position in a list.