# Introduction To Haskell Programming

**Prof. S. P. Suresh**

**Chennai Mathematical Institute**

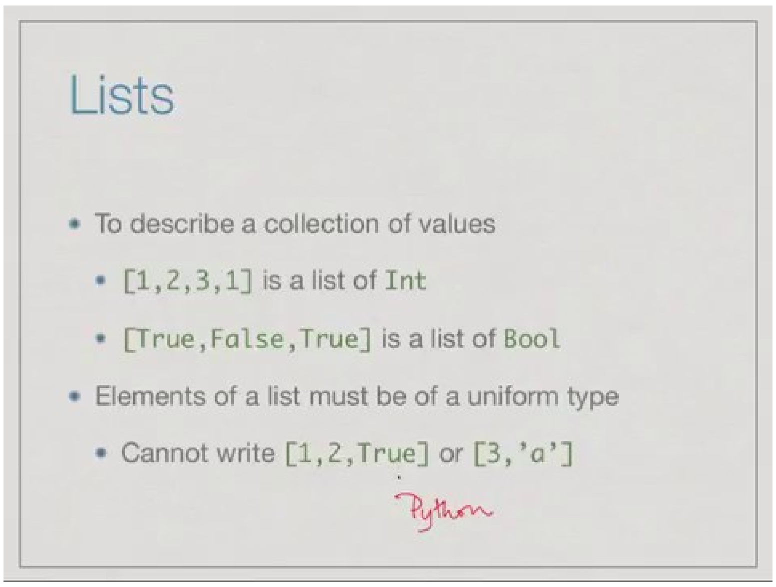
**Module # 02**

**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)



### Lists

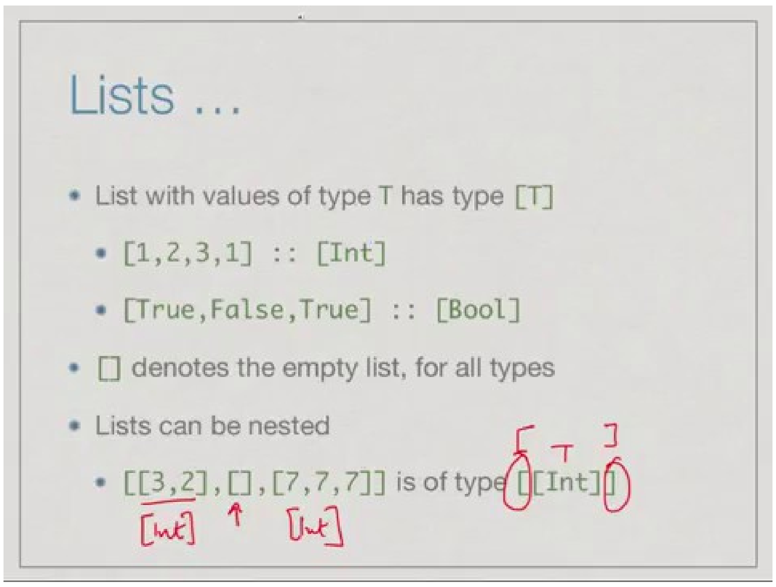
- To describe a collection of values
  - [1,2,3,1] is a list of Int
  - [True,False,True] is a list of Bool
- Elements of a list must be of a uniform type
  - Cannot write [1,2,True] or [3,'a']

Python

In other programming languages that you may know like C, C++ or Java , the usual data structure for collecting a bunch of values together is an 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 a sequence of values and more importantly it is a 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 underlying elements must have a uniform type. So, we cannot write for example, a list with mixed types like, [1, 2, True] or [3 , 'a']. So, if you are familiar with language like python, python does allow this kind of mixed thing and calls it a list. So, these are not the same as python list, in python list the values need not be of uniform type and in Haskell list, it must be of uniform type.

(Refer Slide Time: 01:27)



## Lists ...

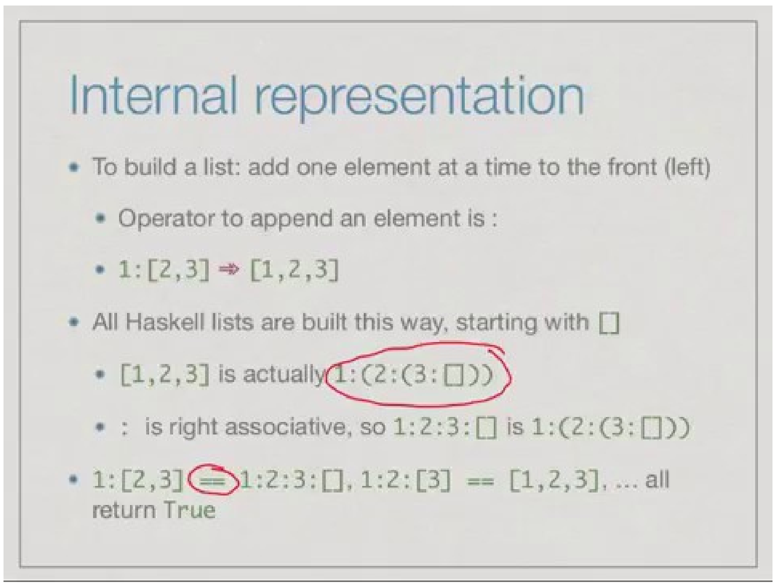
- List with values of type T has type [T]
  - `[1,2,3,1] :: [Int]`
  - `[True,False,True] :: [Bool]`
- [] denotes the empty list, for all types
- Lists can be nested
  - `[[3,2],[],[7,7,7]]` is of type `[[Int]]`

So, since the underlying 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 [T]. So, now, since here inside we have Ints in the first list, we say that the type of the list is [Int], similarly since these are Bool, the type of this list is [Bool]. The notation for an empty list is  [ ] which just an opening bracket followed by closed bracket .

Now, technically there is an empty list of type [Int],  there is an empty list of type [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 an 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 by context it is a list of Int, again this is a list of Int. So, we have three lists of Int themselves enclosed in the list, so this overall thing is list of list of int denoted as [ [Int] ]. So, this [Int] is my underlying T and my outer bracket, so my underlying value type is list of int and now I have a list of this.

(Refer Slide Time: 03:06)



## Internal representation

- To build a list: add one element at a time to the front (left)
  - Operator to append an element is :
  - `1:[2,3]` ⇒ `[1,2,3]`
- All Haskell lists are built this way, starting with `[]`
  - `[1,2,3]` is actually `1:(2:(3:[]))`
  - `:` is right associative, so `1:2:3:[]` is `1:(2:(3:[]))`
- `1:[2,3] == 1:2:3:[]`, `1:2:[3] == [1,2,3]`, ... all return `True`

So, lists are build up in a very specific way in Haskell. Lists are 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 : [ 2 3] gives me the list [1, 2, 3] . 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.

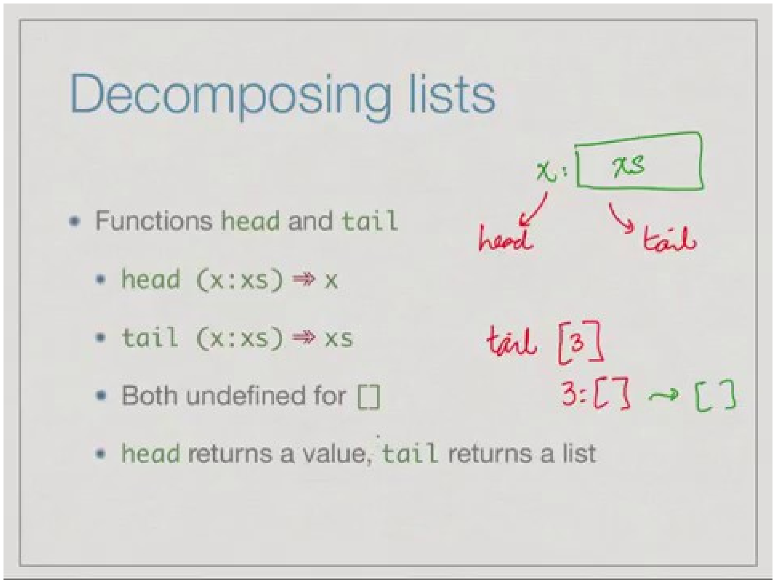
So, Haskell actually builds up list in a standard way using this operator. So, we always start with the empty list and then keep adding elements in front of it. So, when we write the list [ 1, 2, 3] but actually means that it was built in reverse by first appending 3 to the empty list [ ], then appending 2 to the [3], then appending 1 to [2, 3]. So, the list [1,2,3] that we write for legibility is actually in terms of this fundamental colon operator. It is actually 1: (2 : (3 : [ ] ) ) 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 * int * int is actually bracketed from the right. So, we have implicitly that this is the bracketing, so this is what is called a right associated operator, we associate 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] as 1 : 2 : 3 : [ ] 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: [2,3], but if I write 1:2:3: [], all of these are actually the same value. So, if I try to ask the interpreter whether this is equal to that. So, just remember this is our equality check, then it will return True, similarly if I say 1:2:[3] this will be equal to [1,2,3] 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, so internally the only form that matters is there of the form 1:2:3:[]. How we write it is flexible.

(Refer Slide Time: 05:50)



## Decomposing lists

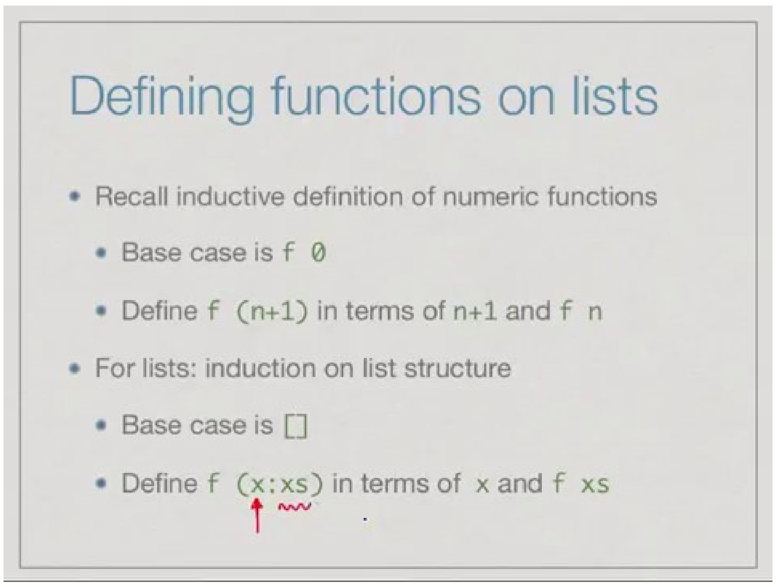
- Functions head and tail
  - head (x:xs) ⇒ x
  - tail (x:xs) ⇒ xs
  - Both undefined for []
  - head returns a value, tail returns a list

x: xs
head
tail
tail [3]
3:[] ⇝ []

So therefore, when we have a list, we 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 xs for the rest. So, this is an x followed by a bunch of xs that is how you are supposed to read it. 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 a value and the tail of the list is another list, in particular if I take tail of a list with one value, this is the same as 3:[]. 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 lists, head returns the value, tail returns the list.

(Refer Slide Time: 06:57)



## Defining functions on lists

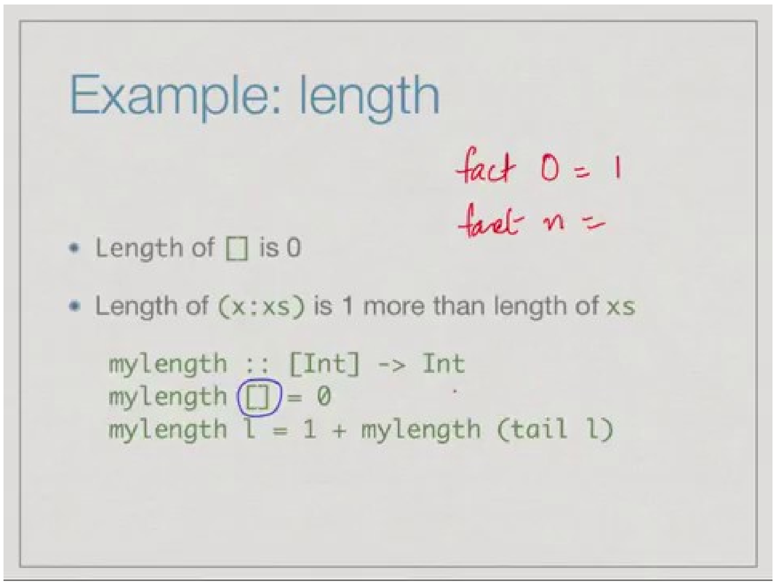
- Recall inductive definition of numeric functions
  - Base case is `f 0`
  - Define `f (n+1)` in terms of `n+1` and `f n`
- For lists: induction on list structure
  - Base case is `[]`
  - Define `f (x:xs)` in terms of `x` and `f xs`

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

So, for example, we said that (n+1)! has the value (n+1)*n!. So, this is the value of the function, so this is f 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 a 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)



## Example: length

fact 0 = 1

fact n =

- Length of [] is 0
- Length of (x:xs) is 1 more than length of xs

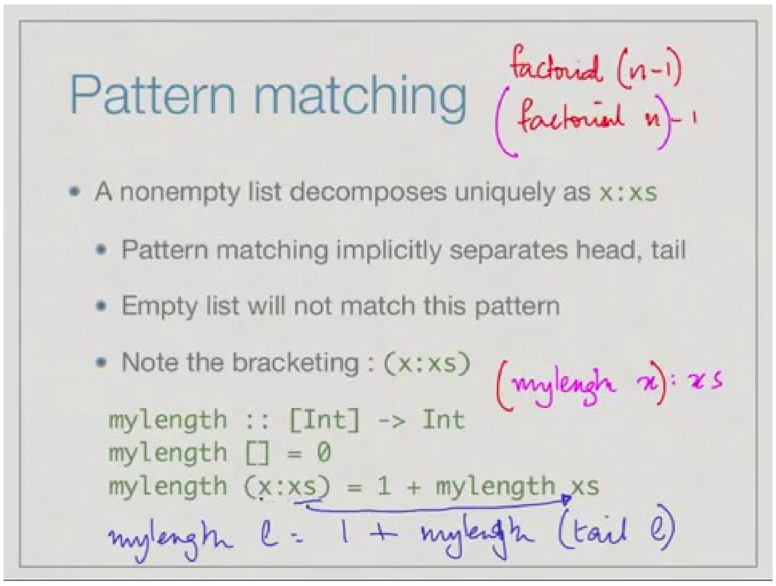
```
mylength :: [Int] -> Int
mylength [] = 0
mylength l = 1 + mylength (tail l)
```

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

So, the base case as we did for, so if you remember we used to write factorial 0 = 1 and factorial of int is something. So, again we have two cases we have mylength 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 give an argument which is not the empty list, it will not match this definition, it will go to the next definition.

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

(Refer Slide Time: 09:34)



## Pattern matching

$\text{factorial } (n-1)$

$(\text{factorial } n) - 1$

- A nonempty list decomposes uniquely as `x:xs`
    - Pattern matching implicitly separates head, tail
    - Empty list will not match this pattern
    - Note the bracketing : `(x:xs)`

(mylength x) : xs

```
mylength :: [Int] -> Int
mylength [] = 0
mylength (x:xs) = 1 + mylength xs
```

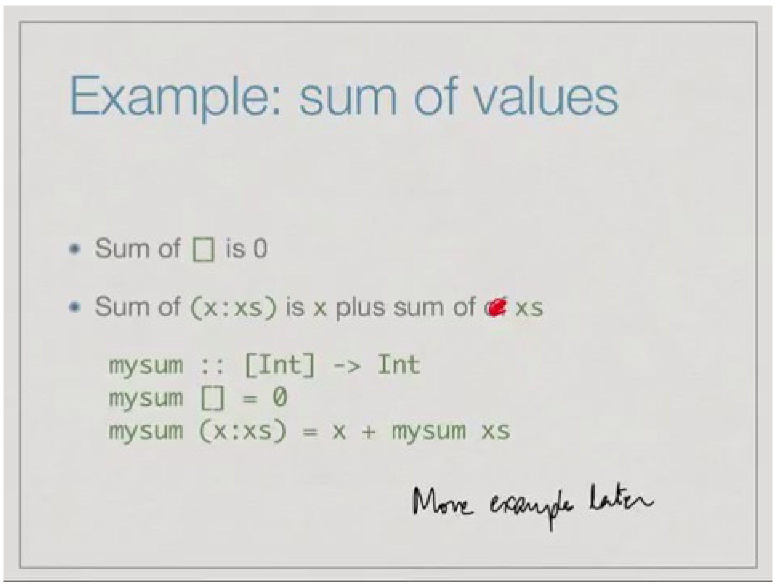
mylength l = 1 + mylength (tail l)

Now, it turns out that we can use pattern matching, because of the unique way in which lists are built up. So, we do not have to actually explicitly call head or tail when we write inductive definitions, every non empty list can be uniquely decomposed as its head and its tail and this can be represented as an expression of the form (x : xs). So, the first variable refers to the head, the second variable refers to the tail. So we can write this same function as follows instead of writing tail in the second definition.

So, earlier we had written mylength l =1+mylength(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 the recursive call, now notice this bracketing. So, we had the same problem  in the beginning when we wrote  in our first week, we said that we must write this, because if you write factorial  n - 1 without brackets then Haskell will try to bracket it this way (factorial n ) -1, because function application has a tighter binding, it has precedence over arithmetic.

So, similarly here function binding will have precedence over this. So, it will try to say that this is mylength. If I write mylength x:xs without this it will try to say it is (mylength x) : xs. Then it will 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)



## Example: sum of values

- Sum of [] is 0
- Sum of (x:xs) is x plus sum of xs

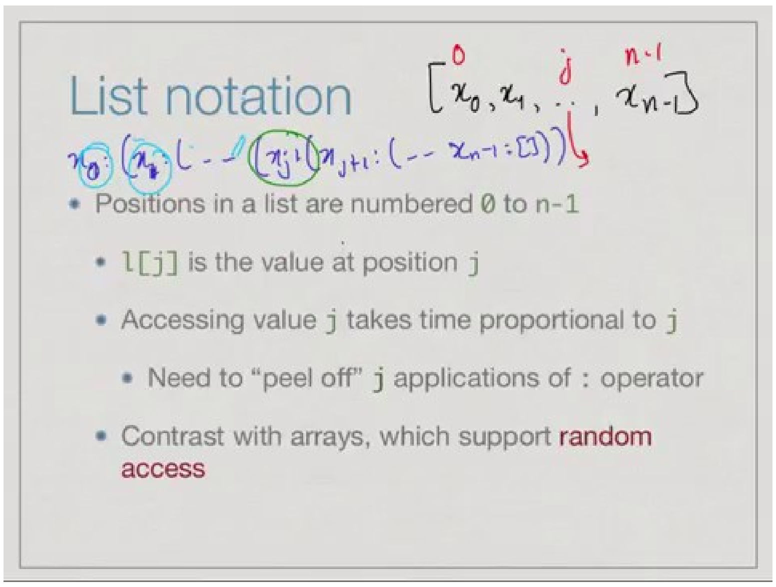
```
mysum :: [Int] -> Int
mysum [] = 0
mysum (x:xs) = x + mysum xs
```

More example later

So, just to get some practice let us look at another function which is inductively defined, 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 of the empty list is just 0, because there are no values, mysum [] = 0. And now if I want to sum up a non empty list I inductively sum up the tail and I add the head to it, so it is x plus the sum of the tail. mysum (x : xs) = x + mysum xs, 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 than just length and sum. We will see several examples to get familiar with this concept.

(Refer Slide Time: 12:21)



## List notation

$[\overset{0}{x_0}, x_1, \overset{j}{\ldots}, \overset{n-1}{x_{n-1}}]$

$x_0 : (x_1 : ( \ldots (x_j : (x_{j+1} : ( \ldots x_{n-1} : [])) \ldots$

- Positions in a list are numbered `0` to `n-1`
  - `l[j]` is the value at position `j`
  - Accessing value `j` takes time proportional to `j`
    - Need to "peel off" `j` applications of `:` operator
  - Contrast with arrays, which support random access

So, list is a sequence, so we have implicitly values in it 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 [x0 , x1, … xn-1], so we have positions 0 to n-1, this is quite traditional. So, arrays in languages like C, C++, Java start with 0, list in python start with 0 and so on, so the positions are 0 to n-1. So, Haskell gives us an expression which looks like an array access with the square bracket. To access the jth position, the jth position, remember, will be the (j+1)th value because the position starts from 0.

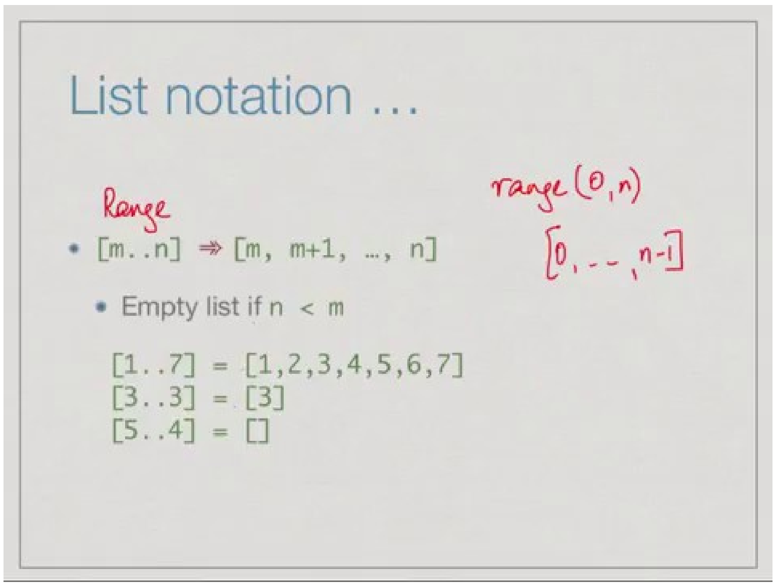
So, if this is position j when l[j] will be the value at that position, now one important thing to remember is that actually this is only a short form for the following expression which is

x0 : ( x1 :… (xj : (xj+1 : … (xn-1 : [] ) ) ) ) ). So, this is what the list will look like, so in order to get to the j'th value I have to extract first this colon then I have to extract this colon. So, I have to peel off each thing until I reach this value, so it takes me j steps to actually get through. So, to get from x0 to x1 to xj takes me j steps.

So, therefore, accessing a value in a list is not a constant time operation unlike an array, if you have an array in memory then you can compute the position of any value in the array by looking at the offset, because these are all defined to be contiguous values of uniform size. 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, it takes time proportional to the position in order to get to an internal value inside the list. So, this is important when we start looking at functions and 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 lists.

(Refer Slide Time: 14:49)



## List notation ...

Range

- [m..n] ⇒ [m, m+1, …, n]
  - Empty list if n < m

```
[1..7] = [1,2,3,4,5,6,7]
[3..3] = [3]
[5..4] = []
```

range(0,n)

[0, - - , n-1]

So, Haskell has some nice notation to denote sequences of values. So, if you want a range of values from m to n , we just write [m..n], this means the list [m, m+1,m+2, … n]. Of course, now if the upper limit is smaller than the lower limit then this will not give us a list. So, if I say [1..7] I get [1, 2, 3, 4, 5, 6, 7] . Notice that the last value is included some of you if you know python will see that in python if I write range(0,n) then I will get the list that goes from [0, ..., n-1].

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

(Refer Slide Time: 16:11)

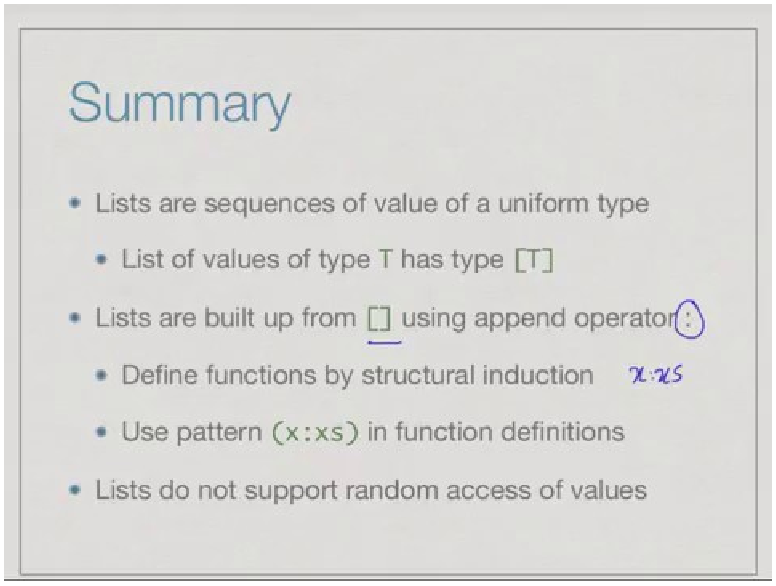
Now, sometimes we would like to get not every value, but values by skipping a fixed amount what you call it arithmetic progression. So, we want [a, a + d, a + 2 d ,.. ] and so on and now of course, when I say [a, a + d, a + 2 d, .. ] there is no guarantee that the number I choose on the upper bound is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

So, which says that I must be skipping 2, so [1, 3, 5, 7 ]then the next value generated 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 that 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 19, so we have to stop.

So, we said that if we do something like [5..4] we will get an empty list, but it is often useful 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 smaller than the first value. So, if I just say [8..5] this would give me the empty list. Because, it would implicitly try to add one but if I say [8,7..5] then I fix d = -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, I crossed that limit, so I stop. Likewise, here the difference is -4. So, if I keep going down after -8 the next one would be -12. So, I would have crossed -9 so I stop. So, it is quite intuitive and it is very clear how to specify ranges in Haskell.

(Refer Slide Time: 18:04)



## Summary

- Lists are sequences of value of a uniform type
  - List of values of type T has type [T]
- Lists are built up from [] using append operator :
  - Define functions by structural induction $x{:}xs$
  - Use pattern (x:xs) in function definitions
- Lists do not support random access of values

So, to summarize a list is a collective set of values of an uniform type. So, it is a sequence of positions 0 to n -1 and because they are of uniform type, a list of values of type T has type list of T denoted by [T]. So, lists are internally represented in a canonical way. All lists are 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, the name of the operator is colon.

And because we have this canonical way in which lists are built up you can decompose lists 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 : xs) and we can use this pattern (x:xs) in function definitions to easily define such inductive functions. And the last thing to remember is that, because of the way the lists are represented and implemented these are not random access lists, these are not like arrays in C, C++, Java. It does take time proportional to the position to reach an internal position in a list.