

# Introduction To Haskell Programming

Prof. S. P. Suresh

Chennai Mathematical Institute

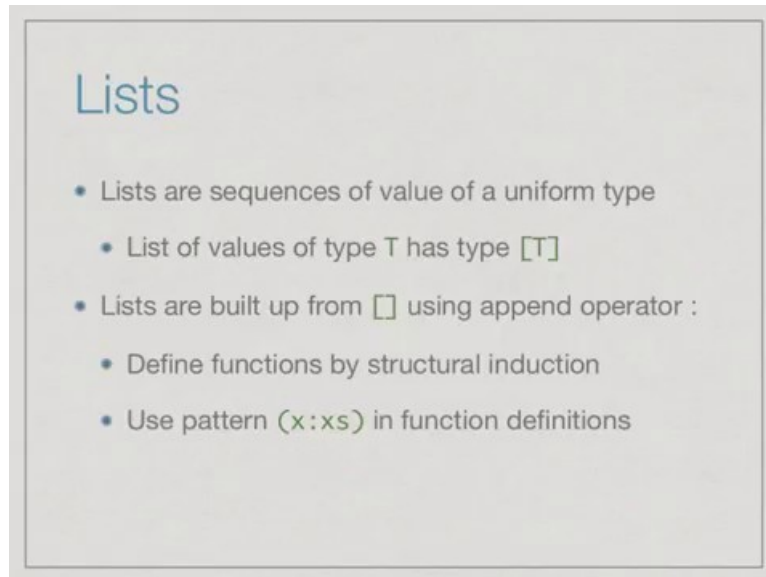
Module # 02

Lecture - 02

Functions on Lists

Last time we looked at lists and saw some simple functions on lists. So, let us get some more practice on defining functions on lists.

(Refer Slide Time: 00:10)



The slide is titled "Lists" in a large, light blue font. Below the title, there is a list of five bullet points, each preceded by a small blue dot. The text is in a light gray font on a white background. The bullet points are: 1. Lists are sequences of value of a uniform type. 2. List of values of type T has type [T]. 3. Lists are built up from [] using append operator :. 4. Define functions by structural induction. 5. Use pattern (x:xs) in function definitions.

- 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
  - Use pattern (x:xs) in function definitions

So, recall that a list is a sequence of values of a uniform type and the type of a list whose underlying values have type T is given as list of type [T]. And all lists in 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:xs) to decompose a non empty list into its head and its tail in our inductive definitions. So, now, let us look at inductive definition of some interesting functions on lists.

(Refer Slide Time: 00:51)

Example: appendright

$\sim [y_0 \dots y_{n-1}]x$

- Add a value to the end of the list
- An empty list becomes a one element list
- For a nonempty list, recursively append to the tail of the list

```
appendr :: Int -> [Int] -> [Int]
appendr x [] = [x]
appendr x (y:ys) = y:(appendr x ys)
```

Handwritten notes:  $[ ]_x$ ,  $[x]$ ,  $x:[ ]$

So, the built in append operator denoted by colon attaches a value to a list on it is left. So, what if we want instead to attach a 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 a single value, which of course, the same of  $x:[ ]$ . So, append right to the base case just gives me a singleton list and otherwise, it is inductive.

So, if I have a list  $[y_0, y_1, \dots, y_{n-1}]$  and if I try to attach an  $x$  on the right, then I can pretend 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 as the first argument an integer, second argument a list and returns a list. So, it is of type  $\text{Int} \rightarrow [\text{Int}] \rightarrow [\text{Int}]$ , 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, `appendr x [] = [x]`.

And if I want to append right to a non empty list, I can pull this out and try to insert  $x$  into the  $ys$ . So 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  $ys$ . 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)

### Example: attach

- Attach two lists to form a single list
  - `attach [3,2] [4,6,7] ⇒ [3,2,4,6,7]`
- Induction on the first argument

```
attach :: [Int] -> [Int] -> [Int]
attach [] l = l
attach (x:xs) l = x:(attach xs l)
```
- Built in operator ++
  - `[3,2] ++ [4,6,7] ⇒ [3,2,4,6,7]`

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, 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 `attach` now takes two lists, the first argument to the `attach` is a list of integer, it takes a second list of integers and finally produces 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 this second argument. The base case says that attaching the empty list to any other list `l`, just 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 `xs` to `l` using induction, then I can stick back the `x`.

So, that is precisely what it says, it says inductively attach `xs` to `l`, now `x` is smaller than `(x:xs)` by 1. So, it is a smaller list, so this function is inductively defined and now, I can use the colon operator to stick the `x` back. So, of course, this is an extremely useful function. So, it turns out the Haskell has a built in operator `++`, which does precisely this, so `[3,2] ++ [4,6,7]` gives us `[3, 2, 4, 6, 7]`.

(Refer Slide Time: 04:09)

Example: reverse  $[1,2,3] \rightsquigarrow [3,2,1]$

- Remove the head
- Recursively reverse the tail
- Attach the head at the end

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

Handwritten annotations: A red bracket groups the list  $[1,2,3]$ . A red arrow points from the head '1' to the reversed tail  $[3,2]$ . Another red arrow points from the head '3' of the reversed list to the '++' operator in the function definition. A green arrow points from the 'xs' in the function definition to the 'xs' in the recursive call.

So, what if we want to reverse a list, so we want to take a list such as  $[1,2,3]$  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 `++`. 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 `++` or we could use `appendr` if you wrote last time go to ((Refer Time: 05:05)), but since `++` is more compact we just use `++ [x]`. So, this is saying that adding the value `x` at the end is the same as attaching the list containing the single element `x`.11

(Refer Slide Time: 05:18)

Example:  $[3,4,4]$   $3 \leq 4$   $[4,4]?$   
 $4 \leq 4$   $[4]?$

- Check if a list of integers is in ascending order
- Any list with less than two elements is OK

```
ascending :: [Int] -> Bool    x:(y:ys)
ascending [] = True
ascending [x] = True
ascending (x:y:ys) = (x <= y) && ascending (y:ys)
```

- Note the two level pattern

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

So, since we are talking about ascending order, this is like checking if an array is sorted. So, anything which has zero or one element is 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 element, then ascending is definitely true, so ascending takes a list of integers and produces True or False. So, it is  $[Int] \rightarrow Bool$ . So if it has no values or only one value, then ascending is trivially True. Otherwise, and now notice that if it has reached this point, then the pattern  $(x:y:ys)$  make sense, because we have definitely at least two values in the list and remember that this is a short form for  $x:(y:ys)$  which in turn will go further and so on.

So, I can decompose any number of levels provided that number of levels exist. So, this pattern will not be matched unless there are two values but if there are two values then the first value will come to x, second value will come to y and whatever remains which may be empty will go to the ys. So, now the inductive definition of ascending says check that the first

two values are correctly ordered, that  $x \leq y$  and that, what remains after that is also sorted.

So, this will walk down for instance, if we want to check it, if I say  $[3,4,4]$  for instance, the first thing we check  $3 \leq 4$  and now is  $[4,4]$  sorted. So, this will say  $4 \leq 4$  and is this single list  $[4]$  sorted, and now the singleton list will match the base case and it will say ok. But the interesting thing about this particular definition is that we can have a two level pattern like this which allows us to access not just the first element, but the second element or even the third element when we have a 3 level pattern provided the list has that many elements we can individually name all those elements directly in the pattern.

(Refer Slide Time: 07:58)

**Example: alternating**

- Check if a list of integers is alternating
  - Values should strictly increase and decrease at alternate positions
- Alternating list can start in increasing order (~~downup~~) or decreasing order (~~updown~~)
  - tail of a **downup** list is **updown**
  - tail of an **updown** list is **downup**

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 keeps going down and up. So, we want to check, if a list of integers is alternating, so the value should strictly increase and decrease at alternate positions. We have seen, that there are two possibilities it could start strictly increasing.

So, this is position 0 and position 1 is bigger, and position 2 is smaller, 3 is bigger and so on or it could be other way round. At position 1 it goes down and at 2 it goes up and 3 goes down and so on. So, let us look at the second case, so this should be reversed, so if it starts in an increasing order I will say it is an updown list, it goes up first, then its goes down and if it starts in decreasing order I will say it is a downup list, it goes down and goes up.

Now, if I cut off this list at this point, so it started as an updown list, then at this position the remaining part must be downup. So, that is the observation that updown and downup are

connected. So, if I start an updown list and I remove the head, the tail must be of the opposite type. Similarly if I have a downup list and removed the head then the tail must be of opposite type. So, this is an interesting definition, because we will define updown and downup in terms of each other, so just let us look at the definition.

(Refer Slide Time: 09:31)

Example: alternating ...

*Mutually recursive*

```

alternating :: [Int] -> Bool
alternating l = (updown l) || (downup l)

updown :: [Int] -> Bool
updown [] = True
updown [x] = True
updown (x:y:ys) = (x < y) && (downup (y:ys))

downup :: [Int] -> Bool
downup [] = True
downup [x] = True
downup (x:y:ys) = (x > y) && (updown (y:ys))

```

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

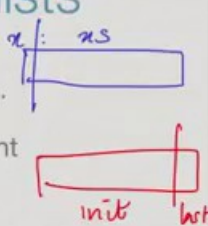
On the other hand, if I have two values then updown must start by going up. So, I want x and then y and I have wanted to go up, so I want  $x \leq y$  and now, as we observed once you go up the remaining part must go down. So, the next step must be down, so the list starting from y onwards will be going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

So, these are what are called mutually recursive functions, so alternating says that the list must either be of the form updown or downup. Updown and downup both have their base cases. And then, they are defined in terms of each other. The first position determines whether it is up or down and then you reverse. So, this is a kind of interesting mutually recursive function on list.

(Refer Slide Time: 11:03)

### Built in functions on lists

- `head`, `tail`, `length`, `sum`, `reverse`, ...
- `init l`, returns all but the last element  
`init [1,2,3] ⇒ [1,2]`  
`init [2] ⇒ []`
- `last l`, returns the last element in `l`  
`last [1,2,3] ⇒ 3`  
`last [2] ⇒ 2`



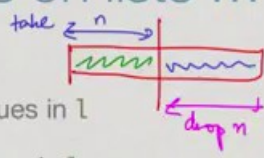
So, Haskell of course, like you would expect has many built in functions on list. So, some of the functions we have seen and some, which we have defined are actually built in functions, so `head` and `tail` we have seen, `length`, `sum` and `reverse` we wrote, but actually they are built in functions. So, you can take `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` knock off the first element append the colon and the remaining thing, now it maybe that you want to knock off the last element. So, then this is called the initial segment and this is called the last value. So there is a function `init`, which returns everything except the last element. So, if I have a 3 element list it will return the first two, if I have a one element list it will return empty because if I remove the last element nothing is left and `last` will be the value at the end. So, `last [1,2,3]` is 3, `last [2]` is 2 and once again `init` and `last` will only work if the list has at least one value. So, you cannot define `init` or `last` for an empty list.



(Refer Slide Time: 12:15)

### Built in functions on lists ...



- `take n l`, returns first `n` values in `l`
- `drop n l`, leaves first `n` values in `l`
  - Do the “obvious” thing for bad values of `n`
- `l == (take n l) ++ (drop n l)`, always

Sometimes you don't want the first or the last, but you want to break a list at some point. So, supposing you have a list you might want to take either the left hand part up to a certain position or you might want the right hand part from a certain position of values. So, these are functions called `take` and `drop`, `take` gives me the first `n` values, 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 then I will get the `n+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 try to take a negative number of values, if I take -5 values I get an empty list, if I take more values than the list has it will give me the entire list. So, it is not going to actually worry about whether `n` is within the range  $(0, \text{length of list} - 1)$  and the same with `drop`, if I say `drop -5` values it will drop nothing. If I say `drop everything`, then it will give me an empty list.

So, the useful thing to remember is that between `take` and `drop` there is no gap. So, if I take `n` elements and I drop `n` elements then every element either gets in to the first part or the second part. So, if I then combine them using `myattach` or concatenate operator then 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  $(\text{length of list} - 1)$  or not, `(take n l) ++ (drop n l)` is always equal to `l`, where `l` is the list.

(Refer Slide Time: 13:58)

### Built in functions on lists ...

- Defining take

```
mytake :: Int -> [Int] -> [Int]
mytake n [] = []
mytake n (x:xs)
  | n == 0 = []
  | n > 0 = x:(mytake (n-1) xs)
  | otherwise = []
```

Handwritten annotations:

- Diagram showing a list  $(x_0, x_1, \dots, x_{n-1})$  with indices  $k$  and  $k-1$  above the first two elements.
- Blue arrows labeled  $n$  and  $l$  pointing to the `Int` and `[Int]` parameters in the type signature.
- A red arrow labeled  $n$  pointing to the `n` parameter in the pattern `n (x:xs)`.
- A red arrow labeled  $n$  pointing to the `n` parameter in the guard `n == 0`.
- A red arrow labeled  $n$  pointing to the `n-1` in the recursive call `mytake (n-1) xs`.
- A red bracket under the `otherwise` clause with the text "negative n" below it.
- The text  $n \leq 0 = []$  written in red at the bottom right.

So, just to exercise our skills in writing functions on list, let us try to define our own version of take. So, remember that take must take a list, first a number and then the list and give me back a list, so it takes an Int, a list of Int i.e. [Int] and gives me back a list of Int, [Int]. So, if I take n, any n values from empty list I get nothing, so from empty list I cannot take anything as there is nothing to get. So, for any n mytake n of the empty list is an empty list, now if it is a nonempty list I will use induction, I will go by cases of n.

So, if I am taking no values, if I say 0 values had been taken, then I get back the empty list. If, I want to take n values, so I have  $[x_0, x_1, \dots, x_{n-1}]$  this is my thing. Supposing I want to take k values from this, then the idea is I will take out this thing and then, I will take k-1 values from what remains. So, I pull out the first value and then I take 1 less value from the rest of xs and remember that, if n is very large, then at some point x has become empty and this will just return.

And finally, we have a last case, which takes care of negative n value, so it says if n is actually less than 0, I should give you nothing. So, notice that actually these two cases are together, so I could have said  $n \leq 0 = []$ , and it would have done the same.

(Refer Slide Time: 15:33)

The slide is titled "Summary" in a blue font. It contains a list of four bullet points. The second bullet point is followed by a handwritten note in red ink. The third bullet point has the word "length" highlighted in green.

- Functions on lists are typically defined by induction on the structure
- Point to ponder *length: [Int] → Int*
- Is there a difference in how *length* works for [Int], [Float], [Bool], ...?
- Can we assign a more generic type to such functions?

So, we have seen a number of examples of functions on lists and almost any interesting function that you write on a list will be defined 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 wanted to write `length` for a list of `Float`, or a list of `Bool`, a list of `Char` or list of anything else, then it is clear that 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 values inside the list.

So, the type of the value inside the list is really irrelevant to `length`. Similarly, `reverse`. If I just want to reorder, it is just a structural property. If I have a bunch of boxes and I want to reverse the sequence of boxes I do not need to look inside the box to find out what is there, I just need to move them around. On the other hand the way we have written types for list we are forced to say this is of type `[Int]` or this is the `[Float]` or this is a `[Char]`.

So, question to think 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 values, but just need to know the structure of the list.