



# FUNCTIONAL PROGRAMMING IN HASKELL

# **Prof. Madhavan Mukund and Prof. S P Suresh**

# **Computer Science and Engineering**

## **Chennai Mathematical Institute**



# INDEX

S. No	Topic	Page No.
	<b><i>Week 1</i></b>	
1	Functions	1
2	Types	10
3	Haskell	18
4	Running Haskell programs	35
5	Currying	46
6	Examples	53
	<b><i>Week 2</i></b>	
7	Lists	66
8	Functions on lists	78
9	Characters and strings	89
10	Tuples	101
	<b><i>Week 3</i></b>	
11	Computation as rewriting	112
12	Polymorphism and higher-order functions	124
13	Map and filter	132
14	List comprehension	140
15	Folding through a list	157
	<b><i>Week 4</i></b>	
16	Measuring efficiency	170
17	Sorting	182
18	Using infinite lists	199
19	Conditional polymorphism	218
20	Defining functions in ghci	230
	<b><i>Week 5</i></b>	
21	User-defined datatypes	235
22	Abstract datatypes	245
23	Modules	258
	<b><i>Week 6</i></b>	
24	Recursive data types	276
25	Binary search trees	295
26	Balanced search trees	318

## *Week 7*

27	Arrays	340
28	Input/Output	366

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

Module # 01

Lecture – 01

Functions

So, welcome to the first lecture of the course Functional Programming in Haskell.

(Refer Slide Time: 00:06)

**Programs as functions**

- Functions transform inputs to outputs

```
graph LR; x[x] --> f[f]; f --> fx[f(x)]
```

A diagram illustrating a function transformation. An input 'x' is shown on the left, with an arrow pointing to a central box labeled 'f'. Another arrow points from the box 'f' to the output 'f(x)' on the right.

- **Program:** rules to produce output from input
- **Computation:** process of applying the rules

So, functional programming starts with the point of view that the program is a function; abstractly a function can be thought of as a black box; that takes inputs and produces outputs. In other words, a program is a function that transforms inputs to outputs. So, when we write a program in a functional programming language, what we are doing is specifying the rules that describe how to generate a given output from a given input. And when we compute in a functional programming language, we apply these rules to the input that is given to us in order to actually produce the output that is expected.

(Refer Slide Time: 00:50)

## Building up programs

How do we describe the rules?

- Start with built in functions
- Use these to build more complex functions

So, the first thing we have to ask ourselves is, how do we build up these programs. Now, it is impossible to start from nothing, so we have to assume that we have some built in functions and values. So, we assume that we begin with something that is given to us and then, we use these built in functions and values to build more complex functions.

(Refer Slide Time: 01:17)

## Building up programs ...

Suppose

- ... we have the whole numbers,  $\{0, 1, 2, \dots\}$
- ... and the successor function, `succ`

```
succ 0 = 1  
succ 1 = 2  
succ 2 = 3  
...
```

$f(x)$   
 $f x$

- Note: we that write `succ 0`, not `succ(0)`

So, let us look at a concrete example. Suppose we start with the whole numbers, that is the integers 0, 1, 2 and so on, the non-negative integers and suppose the only thing that we know how to do with these numbers is to add 1. So, we have a successor function, which you can think of as plus 1 function. So, it says that 0 plus 1 is 1. The successor of 1 is 2, so 1 plus 1 is 2 and so on.

So, this is what is given to us. We have the whole numbers and the successor function. Note by the way, that we are using a slightly non-standard notation for the functions. Normally one would write successor of 0 with brackets like this  $\text{succ}(0)$ . So, we write  $f(x)$  normally, but in functional programming, we will see that, we will normally write  $f\ x$  eliminating the brackets. So, this has two advantages, one is you have fewer brackets to worry about, but it also has some interesting technical advantages, which we will describe in a later lecture in this week.

(Refer Slide Time: 02:18)

## Building up programs ...

We can **compose** `succ` twice to build a new function

- `plusTwo n = succ (succ n)`

```
graph LR; plusTwo[plusTwo] -- "n" --> succ1[succ]; succ1 -- "n+1" --> succ2[succ]; succ2 -- "n+2" --> plusTwo;
```

So, now, that we have a successor function, we can apply twice to an input for example, and I get a new function which adds 2. So, `plusTwo n` takes  $n$ , applies successor and then applies successor to that, so it is as though we had two boxes with us called successor. So, we feed  $n$  here, then we get  $n + 1$ , we feed it to the second box and we get  $n+2$  and what we are saying is that, now we can take these two boxes and call this outer box `plusTwo`. So, this is what it means to compose two functions, you take the output of the first function and feed it to the second function. So, in this case, we have composed the same function twice, we have taken the successor, of successor.

(Refer Slide Time: 03:05)

## Building up programs ...

We can **compose** succ twice to build a new function

- `plusTwo n = succ (succ n)`

If we compose plusTwo and succ we get

- `plusThree n = succ (plusTwo n)`

```
graph LR; n((n)) -- "n" --> plusTwo[plusTwo]; plusTwo -- "n+2" --> succ[succ]; succ -- "n+3" --> n3((n+3))
```

However, we can also combine two different functions, for instance we can take the plusTwo which we have defined and feed its output to successor. So, we have plusTwo and now, we have successor. So, we already know that, if we take a number n and feed it to plusTwo, we get  $n + 2$  and then, we feed it to successor and we get  $n + 3$  and now, this gives us a new box, which we called plusThree. So, in this way we can combine functions by function composition, which is well known to us from mathematics.

(Refer Slide Time: 03:42)

## Building up programs ...

How do we define plus?

- `plus n m` means apply succ to n, m times

$$((n+1)+1)+1\dots+1 \quad \text{m times}$$

But, now suppose we want to extend our definition of plusTwo and plusThree to plusFour, plusFive, plusSix. In general, we want to define plus with two arguments n and m, where we

mean that when we say plus n m; we apply successor to n, m times. In other words, we start with n and then, we do plus 1 and then, we do plus 1 and then, we do plus 1. So, totally we add 1 to n uptill m times.

So, this is what plus means and this is how we were taught the definition of plus when we were in kindergarten, you just keep adding 1 by 1 by 1 until m times. So, how would we describe this in our setting, because we need to describe this family of plus 1's.

(Refer Slide Time: 04:29)

## Building up programs ...

How do we define plus?

- plus n m means apply succ to n, m times
  - Again note: plus n m, not plus(n,m)
- plus n 1 = succ n  
plus n 2 = succ (plus n 1) = succ (succ n)  
...  
plus n i = succ(succ(...(succ n)...))  
*i times*
- How do we capture this rule for all n, i

So, by the way again note this notation, we don't write a function of two arguments with bracket as usual, but we just write the arguments one after the other. So, plus followed by the first argument followed by the second argument, you can just think of it right now as a peculiar syntax which eliminates arguments, brackets and commas. But, as we will see this is a very interesting way of thinking about functions from a computational stand.

So, what are the rules for plus, well ,we know that plus n 1 is same as successor, the built in function add 1. Plus n 2 is, we have seen the successor of successor, but we can think of succ n as plus n 1. So, we are taking plus n 1 and then, applying successor to that. So, in the same way, plus n i, as we saw before will apply the successor i times and the question is, how do we write a rule which captures this mysterious dot, dot, dot which says do something a fixed number of times, but that fixed number of times depends on the value of the argument.

(Refer Slide Time: 05:33)

## Inductive/recursive definitions

- plus  $n \circled{0} = n$ , for every  $n$
- plus  $n \circled{1} = \text{succ } \underline{n} = \text{succ } (\text{plus } n \circled{0})$
- Assume we know how to compute plus  $n m$
- Then, plus  $n \circled{(m+1)} = \text{succ } (\text{plus } n m)$ 
$$n + \circled{(m+1)} = (n+m) + \underline{1} \text{ given}$$

So, this brings us to the realm of inductive or recursive definitions. So, an inductive definition is one where we specify a base case and then, we specify the value for larger arguments in terms of smaller arguments. So, for instance, we know that, if we add 0 then we do nothing, so plus  $n 0$  is always  $n$ . So, this is the base case, we do not have to do any computation, we just return the first argument.

Now, plus  $n 1$  consist of adding 1 to  $n$ , but we can also think of applying the previous value 0, so we get plus  $n 0$ . So, we just do not think of it as  $n$ , but we think of it as the base case and now, we apply successor to the base case. Similarly, in general, if I want to add something to  $m + 1$ , then assuming that I know how to do  $n + m$ , then I can add 1 to that. So, this is just saying that the value of  $n + m + 1$  is the same as the value of  $n + m$ , which I know how to do inductively, plus 1 which is given to me. So, this is my given function.

So, this is the basis of inductive or recursive definitions. That is, you take the base case, whose value is obvious and for larger values, you describe it in terms of operations that you know how to do plus the operation you are trying to define on smaller values, which is inductively known to be true.

(Refer Slide Time: 07:07)

0 1 2 ..

succ 0 = 1  
succ 1 = 2

plus m (succ m)  
= succ (plus n m)

- Unravel the definition
- plus 7 3
  - = plus 7 (succ 2)
  - = succ (plus 7 2)
  - = succ (plus 7 (succ 1))
  - = succ (succ (plus 7 1))
  - = succ (succ (plus 7 (succ 0)))
  - = succ (succ (succ (plus 7 0)))
  - = succ (succ (succ 7)) *(base case  $\Rightarrow 7$ )*
  - .....
  - 10      9      8

So, how does computation work, well it just unravels the definition. So, supposing I want to add 7 to 3, now in our universe, we have 0, 1, 2, etc and we know that succ 0 is 1; succ 1 is 2 and so on. So, I can think of 3 not as 3, but as succ 2 and now, I have a rule which says, plus n (succ m) is equal to succ (plus n m); So, if I have plus 7 (succ 2), this is succ (plus 7 2); when I plug in 7 for n and m for 2.

Now , once again I can expand this : 2 as succ 1 and apply that rule again , and I get succ (succ ( plus 7 1 ) ) and then, once again I can replace the number 1 by the expression succ 0 and then, I get succ ( succ (plus 7 0) ). Now, this is the base case which is 7. So, I get succ (succ 7) and then, if I continue with this computation, this will give me 8 by the built in rule, this will give me 9 by the built in rule and this will give me 10 by the built in rule.

So, this is how I will get plus 7 3 equal to 10. The important thing to note in this is that computing the value 10 does not involve understanding anything about numbers, it is just syntactically replacing expressions by expressions and this is something that we will see more formally later. So, this is how computation works in a functional programming language, you have rules which tell you how you can replace one expression by other expression and you keep replacing expressions, until you reach the value which cannot be simplified.

(Refer Slide Time: 09:06)

## Recursive definitions ...

Multiplication is repeated addition

- $\text{mult } n \ m$  means apply  $\text{plus } n$ ,  $m$  times
- $\text{mult } n \ 0 = 0$ , for every  $n$
- $\text{mult } n \ (\text{succ } m) = \text{plus } n \ (\text{mult } n \ m)$

$$n \cdot (m+1) = n + \underbrace{(n \cdot m)}_{\text{Inductive}}$$

known

So, let us look at another recursively defined function. So, just as addition is repeated application of the successor function, multiplication as you may remember from school is repeated addition. When I say multiply  $m$  by  $n$ , it means add  $n$  to itself  $m$  times. So, we have to apply plus  $n$ ,  $m$  times starting from 0. So, the base case says that if I multiply any number by 0, then I will get 0.

So, multiplying  $n$  by 0 is 0 for every  $n$  and now,  $n * (m+1)$  is just  $n + (n * m)$ . So, this is the inductive case, I know how to do this, because this is smaller and this is now a known function, because we have already defined plus. So, plus was not a given function, successor was the given function, but we already defined plus in terms of successor. So, now, we can use plus to define multiplication.

(Refer Slide Time: 10:13)

## Summary

- Functional programs are rules describing how outputs are derived from inputs
- Basic operation is function composition
- Recursive definitions allow repeated function composition, depending on the input

So, to summarize, a functional program describes rules to tell us how to compute outputs from inputs, what we have seen is that the basic operation that we use in functional programming is to combine functions. So, we use function composition where we feed the output of one function as the input of another function.

Often we have to apply this function composition more than once, but not a number of times that we know in advance. So, when we did plusTwo and plusThree, we knew exactly how many times, we have to compose the function, but when we wrote the general plus or the general multiply, we have to apply these functions depending on the value of the argument. So, for such functions, we saw that recursive definitions are a good way to capture the dependence of the number of times the function has to be composed based on the input value.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

**Module # 01**

**Lecture – 02**

**Types**

(Refer Slide Time: 00:03)

## Programs as functions

- Functions transform inputs to outputs



- **Program:** rules to produce output from input
- **Computation:** process of applying the rules

In the previous lecture, we saw that we would like to look at programs as functions that transform inputs to outputs. So, our program is basically a set of rules that describes how to produce a given output from an input. And a computation consists of applying these rules and transforming the input to the output by repeatedly rewriting expressions based on the rules given.

(Refer Slide Time: 00:29)

## Building up programs

- Start with built in functions
- Use function composition, recursive definitions to build more complex functions
- What kinds of values do functions manipulate?

So, the way we write a program is to start with some built in functions and values. And then, we use function composition and things like recursive definitions to build more complex functions from the basic built in functions and values given to us. So, another thing that we need to be aware of when we are dealing with functions is to look at the values that they manipulate, so these are what we call data types or just types.

(Refer Slide Time: 00:59)

## Types

Functions work on values of a fixed type

- `succ` takes a whole number as input and produces a whole number as output
- `plus` and `mult` take two whole numbers as input and produce a whole number as output
  - Can also define analogous functions for real numbers

So, functions are defined over fixed input and output types. For instance, we saw last time the function successor named `succ` which adds one to the input. Now, the successor function that

we defined assumes that inputs are the whole numbers 0 1 2 3 and as output it produces another whole number. We also looked at definitions of plus and multiply. So, plus and multiply each took two whole numbers as input and produced a whole number as output. Notice that if you add two whole numbers or you multiply two whole numbers you will necessarily get a whole number as an output. Now, you could also define similar functions as plus and multiply for values that are not whole numbers or even successor.

So, plus 1 could be defined for any value, but successor means something. Successor means the next number. So, it makes sense to say that the next whole number after 1 is 2, but it does not make sense to say that the next fractional number after 1.5 is 2.5. There is no next fractional number, which is why successor is usually defined only for whole numbers. But, if you think of successor not as plus 1, then we can define plus and multiply, the general addition even for fractional numbers, but these will be different functions.

(Refer Slide Time: 02:16)

The slide has a title "Types" in blue. To the right, there are two equations written in purple:  
 $\sqrt{4} = 2$   
 $\sqrt{2} = 1.414\dots$

Below the equations, the text "How about `sqrt`, the square root function?" is displayed in green.

- Even if the input is a whole number, the output need not be—may have a fractional part
- Numbers with fractional values are a different type from whole numbers
  - In Mathematics, whole numbers are often treated as a subset of fractional or real numbers

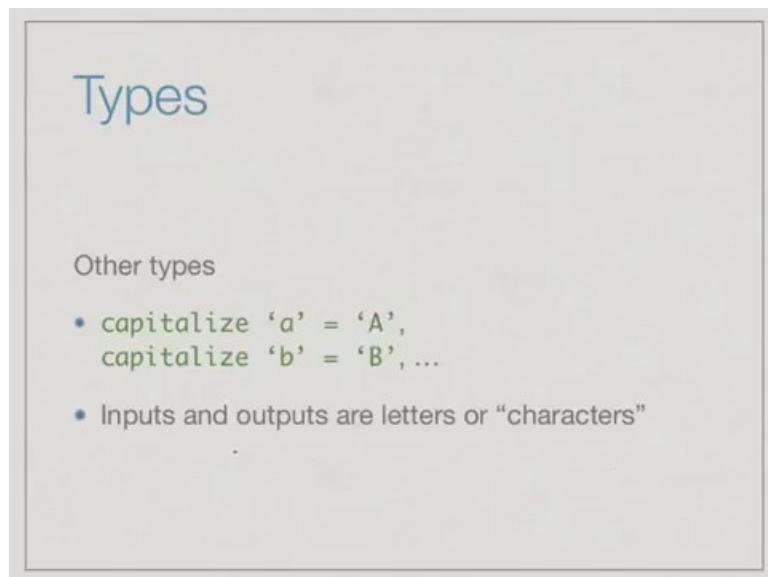
At the bottom, there is a red mathematical expression:  $\mathbb{Z} \subseteq \mathbb{R}$ .

So, what about a function like square root that takes the square root of a number. So, here, notice that even if the input is a whole number, the output need not be. The square root of 4 is 2, but the square root of 2 is in fact an irrational number, but at the very least it is the fractional number which is neither 1 nor 2. So, we have two different types of numbers that we deal with, we deal with these whole numbers or integers and we also deal with numbers which have fractional components.

The problem with fractional components is that the value after the decimal point can be arbitrarily long, it could be infinite in general. So, to precisely describe a fractional value, the description requires an infinite amount of information. In mathematics, typically, you write sets like R for the real numbers and Z for the integers and you assume implicitly that the integers are those real numbers which have a 0 fractional part.

Now, when we write programs it turns out that representing integers and representing fractional numbers are very different and therefore, these are distinct types also. So, we will always differentiate between whole numbers and fractional numbers when we write functional programs, even though mathematically one is often seen as a subset within the other.

(Refer Slide Time: 03:40)



Now, many interesting programs manipulate values, which are not numbers. For instance, when you use a text editor or you enter an url in a browser, you are typing some text and then the program that is the web browser or your text editor has to interpret this text. So, characters are a very important type in programming. So, think of very simple function that we can write, something that takes an input letter and provided it is a valid letter of the alphabet not a punctuation mark or a number, it capitalizes it. So, capitalize 'a' is capital A, capitalize 'b' is capital B and so on. So for such a function the input and the output consists of characters.

(Refer Slide Time: 04:22)

The slide has a light gray background with a thin black border. At the top center, the text 'Functions and types' is written in a blue font. Below it, a large red plus sign symbol is centered. In the lower-left quadrant of the slide, there is a bulleted list:

- We will be careful to ensure that any function we define has a well defined type
- The function `plus` that adds two whole numbers will be different from another function `plus` that adds two fractional numbers

So, whenever we write functions in a functional programming language we will have to ensure that the functions' type is well defined. In fact, it will turn out that the functional language we are going to use, which is Haskell will not allow us to write a function whose type is not defined. Now, not defined can mean many things, but one of the things it means is that the output must be uniform. So, you cannot have a function, which on some values produces an integer as an output and on some other values produces a character as an output.

So, this will be the kind of limitation that we will see. So all functions will have to have a well defined type. And therefore, the same function which say mathematically does a same thing like `plus`, which mathematically does a same thing for whole numbers and for fractional numbers will actually consist of two different functions. Now it may be that for convenience we use a symbol like '+' or the name '`plus`' to define both functions. But, in practice, actually, internally they are two different functions, because they have different types.

(Refer Slide Time: 05:23)

## Functions have types

plusTwo n = succ (succ n)  
apply twice (f) n = f (f n)

- A function that takes inputs of type  $A$  and produces output of type  $B$  has a type  $A \rightarrow B$ 
  - In Mathematics, we write  $f : S \rightarrow T$  for a function with domain  $S$  and codomain  $T$   $\text{range}(P) \subseteq T$
  - A type is just a set of permissible values, so this is equivalent to providing the type of  $f$

So, now, not only do the inputs and the outputs of functions have types, the functions themselves have types. We will see that we can write functions, which take other functions as input and manipulate them. For instance we can think of a function, ok we saw that, for instance, plusTwo n is succ ( succ n ).

Now in general, we could say why to fix the number of times we apply successor. Instead, why not we say that the successor has to be applied twice. So, we take as input a function and a number and we say that the output is the result obtained when you are applying the function twice to that number. So, here notice that the function itself is being supplied to our new function and it is being used twice in the output. So, there is nothing that prevents us from writing functions, which actually take other functions as arguments. And now, we assumed that all functions must have well defined types for inputs and outputs, so functions must also have types.

Now, the notion of a function type is not very new to us. In mathematics, if we write a function from a set  $S$  to a set  $T$  we write  $f : S \rightarrow T$ . This means inputs come from  $S$ . So,  $S$  is often called the domain of the function  $f$  and outputs are from the set  $T$ . So,  $T$  is often called the co domain. In particular, the range of the function is that subset of the co domain, which is actually reached by values from  $S$ . So, we have this notion of an input set and an output set.

So, a type of a function in a programming language is exactly that. It just takes the input type and the output type and says it transforms the values of type  $A$  to values of type  $B$ . So, a type

is just a set of values. When we say we have the set of whole numbers, then it just means that the values permitted are 0 1 2 3. We say we have set of characters when we have a set of symbols that we are allowed to type. When we say fractional numbers, we have certain values that we can define with decimal points. So, a type is just a set of values and just like in a mathematical notation we write a function from one set to another, we use the same notation to denote the type of function and this is important, because one function can be actually fitted to another function as input as we will see.

(Refer Slide Time: 07:48)

The slide has a light gray background with a title 'Collections' in blue. To the right of the title, there is handwritten text: 'Whole numbers 0, 1, 2...' and 'Sets of whole numbers {1, 3, 7}'. Below the title is a bulleted list:

- It is often convenient to deal with collections of values of a given type
  - A list of integers
  - A sequence of characters — words or strings
  - Pairs of numbers (7.3, 4.2)
- Such collections are also types of values

The other interesting motion is that of building collections of values. So, if you have whole numbers then we have individual values 0 1 2... Now you can have sets of whole numbers. For example in mathematics, a set of whole numbers is a collection of whole numbers say 1 3 7 and we write it with some notation to indicate that this is a collection, may be curly brackets and commas.

So, similarly in programming we often want to deal with collections of values. We might want to talk about a sequence or a list of integers. Or, when we type words, then a word consists of a number of characters, so that is a sequence of characters. Or, we might to talk about points in space, x y coordinates. So, an x y coordinate is just a pair of values like 7.3 and 4.2, so we want pairs of numbers. So, if we have types for the components, then we have a type for the whole, so we can say that I have a pair of fractional numbers, I have a sequence of integers or I have a sequence of characters.

So, collections are also important types. So, a type remember, now just a set of permitted values. So, if I tell you the type then you can tell me whether a given value is legal for the type or not. If I tell you that the type is whole numbers and I ask you whether seven point five is a whole number you can say that it is not a whole number. If I ask you whether the list consisting of 1 followed by 2 is a list of integers ,you say yes it is the list of integers and so on.

(Refer Slide Time: 09:26)

## Summary

- Functions manipulate values
- Each input and output value comes from a well defined set of possible values — a **type**
- We will only allow functions whose type can be defined
  - Functions themselves inherit a type  $f: A \rightarrow B$
  - Collections of values also types

So to summarize, we are talking about functions that manipulate values, but the values must come from well specified sets or types. So, each input and output value comes from a well defined set of possible values and we will only allow function definitions whose type is precisely specified. And given the types of the input and the output we saw that the function itself will have a type, which is from the input type to the output type.

And finally, we saw that it is important to have collections of values, so lists of integers or pairs of characters. So, we want to combine individual values or smaller values into larger values, which can store multiple values of the same type.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

**Module # 01**

**Lecture - 03**

**Introduction to Haskell**

We have looked at programs as functions and we have looked at data types or types describing the values that a function can take as input and produce as output. So, with this abstract background let us concretely look at the language Haskell, which we are going to use in this course.

(Refer Slide Time: 00:21)

The slide has a title 'Haskell' in blue. Below it is a bulleted list:

- A programming language for describing functions
- A function description has two parts
  - Type of inputs and outputs
  - Rule for computing outputs from inputs
- Example

Below the list is a diagram showing the Haskell code for the square root function:

```
sqr :: Int -> Int
sqr x = x * x
```

Annotations explain the components:

- A green arrow points from 'sqr' to the first colon, labeled 'Type definition'.
- A red arrow points from the second colon to the multiplication operator (\*), labeled 'Computation rule'.
- A red bracket underlines the multiplication operator (\*) and is labeled 'built in op'.

So, Haskell is basically a programming language for describing functions. In Haskell, a function definition has two parts, the first part describes the types of the inputs and the outputs, what values does this function manipulate and the second part which is of course, the important part is the set of rules for computing the outputs from the inputs.

So, here is a very simple example for Haskell function, the first line describes the type, so the notation. So, this is the name of the function `sqr`, then this notation with two colons indicates that this is the type definition. As we will see `Int` is a built in type in Haskell, so `Int` stands for integers which are the whole numbers including negative and positive numbers. And finally,

this funny notation which is a minus followed by greater than is indicative of the mathematical arrow. So, you will see in many Haskell syntactic forms that in Haskell that we try to describe in characters something that is of familiar symbols from mathematics.

So, this says that `sqr` is a function which takes integers as input and produces integers as output. So, this is the type definition and now we have a rule which tells us what to do with the given input, so it says if I am given an input  $x$  then the output should be  $x*x$ , where this  $*$  is the built in operation. So, this is what a Haskell program looks like, a type definition and a computation rule, now this is a very simple function with one computation rule. We will see later that it could have more than one computation rule depending on the value of the input.

(Refer Slide Time: 02:08)

The slide has a light gray background with a white rectangular content area. At the top left, the title "Basic types" is written in a blue font. To the right of the title, there is handwritten text: "5/3 = 1.666 ...". Below the title is a bulleted list of Haskell basic types:

- Int, Integers
  - Operations: +, -, \*, / (Note: / produces Float)
  - Functions: div, mod
    - div 5 3 ~ 1
    - mod 5 3 ~ 2
  - Float, Floating point ("real numbers")
  - Char, Characters, 'a', '%', '7', ...
  - Bool, Booleans, True and False

So, before we look at how to write more complicated rules in Haskell, let us look at the basic types that Haskell supports. The most familiar type in any programming language is the set of integers, which in Haskell is called `Int` with a capital I. And note that in Haskell, in all variable names and types the upper case, lower case matters and all types by conventions start with the capital letter. So, `Int` stands for the set of integers this is of course, the usual set of numbers 0, +1, -1, +2, -2 and so on.

And there are the usual arithmetic operations supported for integers, addition which is denoted `+`, subtraction `(-)`, multiplication `(*)` and division `(/)`. But, note that division does not produce integer division, it is the regular division, so if I say 5 divided by 3 then I will get a number which looks like 1.666. I will not get 1. Integer division is a function `div` and

remainder is function mod, but these are functions, so I should write `div 5 3` and this will give me 1, because if I divide 5 by 3 it goes one time and not two times and similarly `mod 5 3` will be 2, this says that the remainder of 5 when divided by 3 is 2.

So, I should be careful not to write `5 div 3`. There is a way to write it like this, but not the way I have written it here. So, you can take functions and treat them as operators and vice versa. We will look at this later. But, for the moment just note that when it is a function you must write it as a function and provide the arguments after the function and as we saw before, if a function has multiple arguments you separate them one after the other with spaces, no brackets, no commas.

So, the numbers other than integers are of course, the so called real numbers and in Computer Science, they are often called floating point numbers to indicate that the decimal point is not at a fixed position, but floating point. And because of reasons of precision you can only represent a finite amount of information, the floating point numbers do not actually capture all the real numbers. So, only up to a certain precision, but anyway that does not bother us right now.

But, we have two different types `Int` which are integers, `float` which are floating point numbers. Then as we saw before, a very important type for programs is text. So, we have the basic text unit as a character and a character is written with single quotes like '`a`', '`%`', '`7`'. So, any character which we can type on the keyboard can be represented in single quotes as a character name.

And finally, a very useful type for programs to decide how to apply values depending on the inputs is the Boolean type. The Boolean type has two constant values in Haskell denoted as `True` beginning with a capital T and `False` beginning with a capital F. So, Booleans are denoted `True` and `False`, and not as 0 and 1 as they are in some other programming languages. So, it is a separate type with two specific constants, `True` and `False`.

(Refer Slide Time: 05:16)

Basic types ...

- Bool, Booleans, True and False
- Boolean expressions  $\text{sqr } x = x * x$
- Operations: &&, ||, not  
and or
- Relational operators to compare Int, Float, ...
- $=, \neq, <, \leq, >, \geq$   $\neq$

So, just as we have arithmetic operators to combine arithmetic values like addition, subtraction and so on, we know that we have Boolean operations to combine Boolean values. So, we have the Boolean AND denoted as ‘&&’ which is true provided both its inputs are true, we have the Boolean OR denoted as ‘||’ which is true provided either one or both of its inputs are true and then, we have the function NOT denoted as ‘not’ which negates the value, not (True) is False, not (False) is True. And notice that we use the word ‘not’, we do not use the single form.

Another very useful thing that all programming languages support is to allow us to compare values and then return a True or a False. So, we have relational operators to compare say integers or floating point numbers. The single equality symbol ‘=’ in Haskell will be used for function definitions that we saw when we said  $\text{sqr } x = x * x$ . So, ‘=’ is really equality in the sense of the function definition.

So, equality of values as in many programming languages is denoted ‘==’ and then we have ‘<, <= , >=’. What is probably slightly unusual is that the symbol for not equal to in some programming languages is written with exclamation mark as !=. But, Haskell as we mention before tries to mimic the symbols we actually use in mathematics. So, ‘/=’ is supposed to be a representation for the normal equality symbol = with a slash across it which we use to indicate not equal to when we write mathematics by hand. So, just remember that /= is the symbol for not equal to in Haskell and not exclamation mark.

(Refer Slide Time: 06:58)

## Defining functions

$a \parallel b$  if  
is True  
a is True  
b is True  
both are True

- xor (Exclusive or)
  - Input two values of type Bool
  - Check that exactly one of them is True

Input 1      Input 2      Output  
 $xor :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$  (why?)  
 $xor b1 b2 = (b1 \&& (\text{not } b2)) \parallel ((\text{not } b1) \&& b2)$  ↗ Boolean exp

$\text{sqr } x = x * x$  ← another exp

So, let us now define a more complicated function than square, let us for instance define a function on Boolean values. So, we saw the function OR  $\parallel$ , so we have  $a \parallel b$  is True, if a is True, b is True or both. So, this is the so called inclusive OR which is True if either one or both of the values are True, the Exclusive OR xor is the function which requires precisely one of the values to be True. So, you take two inputs of type Bool and check that exactly one of them is True.

So, now, here is our first surprise that how do we write the type of a function with two input values, well just take it for granted right now, that we just write the types of the values in succession followed by the output. So, this is the input 1, this is the input 2 and the last one is the output (Refer Slide Time: 06:58), why this is the case will become clear a little later in this week's lectures, when we talk about this whole notation of functions not using brackets and so on.

But, for now let us just assume that for a function of many inputs, we write the type of each input one after the other separated by this arrow symbol followed by the output type. So, a function which takes two Booleans and produce a Boolean is  $\text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$  and what is the rule for this, well if I am given two input values  $b1$  and  $b2$  then either I want  $b1$  to be true and  $b2$  to be false, but  $b2$  is false provided  $\text{not } b2$  is true. So, I want  $b1$  and  $\text{not } b2$  to be true or I want  $\text{not } b1$  and  $b2$ , so I want  $b1$  to be false and  $b2$  to be false, so this is the exclusive OR called xor function.

So, this is very similar to our earlier function which says `sqr x = x * x`. So, there we use an arithmetic expression but here we are using a Boolean expression. So, in principle there is no difference between a function which takes numeric values and using numeric expression and a function will takes Boolean inputs and uses a Boolean expression, this form is very similar, it is just that we have to be clear about what values our functions are manipulating.

(Refer Slide Time: 09:25)

## Defining functions

- `inorder`

<ul style="list-style-type: none"> <li>• Input three values of type Int</li> <li>• Check that the numbers are in order</li> </ul>	$\begin{array}{c} \text{inorder } 3 \ 7 \ 8 \rightarrow \text{True} \\ \text{inorder } 7 \ 3 \ 8 \rightarrow \text{False} \end{array}$
---	--

```

 $\begin{array}{c} 1 \ 2 \ 3 \ \text{Output} \\ \text{inorder} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \\ \text{inorder } x \ y \ z = (x \leq y) \ \&& \ (y \leq z) \end{array}$ 

```

So, let us look at another function which mixes the two types. So, supposing we want to take three integers and check that the three integers are in order. So, we want to say something like `inorder` of say 3, 7, 8 should be True, but on the other hand if I say `inorder` of 7, 3, 8 this should be False. So, we are given three inputs and we want to check that the three inputs are in ascending order. So, this is now a function which takes three Ints and produces a Bool, now given our earlier discussion about how we write a type, this is the first input, this is the second input, this is the third input and this is the output.

So, it is `Int -> Int -> Int -> Bool`, so the first three Ints corresponds to first three inputs in that sequence and the last one is the output type. So, it takes  $x$ ,  $y$  and  $z$ , let these three be the input values, then check that  $(x \leq y) \ \&& \ (y \leq z)$ . So, we are now taking integer values and producing a Boolean output.

So, it will check three integers as input and produce a Boolean as an output and what it will do is that it will apply this relational operator ' $\leq$ ' to the first two inputs, relational operator ' $\leq$ ' to the second pair of inputs and use a Boolean operator ' $\&&$ ' to combine them. So, it

produces a Boolean expression from three integers. So, this is a function which mixes types from the input to the output, so it converts integer inputs to Boolean output.

(Refer Slide Time: 11:05)

## Pattern matching

- Multiple definitions, by cases
  - xor :: Bool -> Bool -> Bool
  - xor True False = True
  - xor False True = True
  - xor b1 b2 = False
- Use first definition that matches, top to bottom
  - xor False True matches second definition
  - xor True True matches third definition

So, now let us illustrate more complicated ways of defining functions using the function that we saw before. So, recall that xor is the function, which tries to check that exactly one of its inputs is True, now in the case of Boolean values we only have two possibilities for the inputs, they are either True or False. So, you have a function with two Boolean inputs as we know, we can write what is called a truth table.

So, we can say that we have  $b_1$ ,  $b_2$  and we have the output, so we can enumerate these things. You can say  $b_1$  is True,  $b_2$  is False,  $b_1$  is True,  $b_2$  is True,  $b_1$  is False,  $b_2$  is False,  $b_1$  is False,  $b_2$  is True. These are the four possible values and then we can specify the outputs, we will say that this is okay, this is okay and these two are not okay. So, this is the truth table for exclusive OR, now we can actually write a function which checks these patterns, we say that if this is the case then xor True False is True.

Similarly, if this is the case you say xor False True is True and finally, we say that if it is not one of these two, then XOR of any  $b_1$  and  $b_2$  which is not of the form True, False or False True must be False. So, what we are saying is that when we give an input to this function, we look at the input and see whether it matches the definition. So, if I give xor False True, it does not match the first definition, because the first definition works only if the first input is True

and the second input is False, so this does not match. So, we check the second definition, so you go top to bottom, so this matches this definition.

Similarly, if I give True, True it does not match the top definition, because this is wrong, it does not match the second definition, because this is wrong. So, finally, it has to fall through to here and it says True, True. Then the third definition applies and the answer is False. So, what we need to make a little bit more precise is what it means for a function call to match a definition that is quite straight forward.

(Refer Slide Time: 13:11)

## Pattern matching ...

$\text{xor } \underline{\text{True}} \underline{\text{True}} \times \text{xor } \underline{\text{True}} \underline{\text{False}} = \text{True}$

- When does a function call match a definition?
  - If the argument in the definition is a constant, the value supplied in the function call must be the same constant
  - If the argument in the definition is a variable, any value supplied in the function call matches, and is substituted for the variable (the “usual” case)

$f(x) = x + 3$        $f(7)$

So, if the argument in the definition is a constant, so supposing I write xor True False = True and now if I say xor True True, then this does not match, because this is a constant and this does not match. So, if I have a definition which uses the constant then the function call matches the definition only if the same constant appears in the function. On the other hand, as we saw before if we have b1 or b2 then if the definition is a variable then any value matches that call and then you substitute as well.

So, this is what we normally use when we have function, we write  $f(x) = x + 3$  and then when we write  $f(7)$ , 7 is substituted for x and then everywhere in this where I write  $x + 3$  then 7 becomes the value here as well. So, I will get 10, so this is how we normally write functions, we write variables to indicate the arguments and when we call the function with the concrete value that value substituted for that variable uniformly in the definition of the function.

So, this is the usual case, but what is the unusual case in Haskell is that I can give these constant patterns and say that if my variable is of a fixed value then that pattern will be matched and that definition could be matched. So, we will see more examples of this as we go along.

(Refer Slide Time: 14:39)

## Pattern matching ...

- Can mix constants and variables in a definition
  - or :: Bool -> Bool -> Bool
  - or True b = True
  - or b True = True ~~or~~ or True True ?
  - or b1 b2 = False
- or True False matches first definition
- or False True matches second definition
- or False False matches third definition

So, here is a definition of the built in function OR, but we are writing it ourselves. So, remember that the truth table for OR is that if either of the function values is true then the output is true. The only case which is outside this is when both values are false. So, what is here now is that I am mixing constants and variable. The first definition ‘or True b = True’ says that if the first argument is definitely True and whatever second argument may be the output is True. So, when I call the function as ‘or True True’, then I am either in the first case or I am in the second case.

Now the second definition ‘or b True = True’ says that if I have the second argument true then whatever the first argument maybe the answer is true. So, now, I am in this first case or in this second case and finally, if neither of these things hold then the first argument is not true and the second argument is not true, then I must be in the third case, so the output must be False. So, here ‘or True False’ matches the first definition, ‘or False True’ matches the second definition, ‘or False False’ matches the third definition.

And what about or True True. Well, or True True matches both the first and second definition, because the first argument is true and second argument anything, second argument

is true and first argument anything. But, remember that we will do it in order, so it will actually be using the first definition and not the second definition to compute the value and to this case in both cases the output is true. But, something like this which matches multiple definitions will match the first one from the top which succeeds.

(Refer Slide Time: 16:34)

## Pattern matching ...

- Another example

$\text{and} :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$

$\rightarrow \text{and True } b = b$   
 $\text{and False } b = \text{False}$

- In the first definition, the argument supplied is used in the output

b <sub>1</sub>	b <sub>2</sub>	out
T	T	F → F
T	F	T → T
F	T	F
F	F	F

So, here is a slightly more illustrative example, so supposing we want to define not OR, but AND. So, the truth table for AND says that if the first input is true and the second false then the output is false, if it is true and true then it is true, if it is false and true then it is false and if it is false and false then it is false. So, basically there is only one case in which AND will give true which is both inputs are true, but notice that if the first input is true then the output is exactly the second input, if the second input is false the output is false, if the second input is true the output is true.

So, we can now capture that in this rule. It says that if the first argument is true, then look at this second argument and return it directly as the output. If the second argument is False the output is False, if the second argument is True the output is True. So, that is capturing the first two rules and finally, the last two rules both have this common fact that the first argument is False. So, we say if the first argument is False no matter what the other argument is, the output is False.

So, here we are mixing first of all constants and variables as we did in the OR case, but we are also using the fact that the argument that we provide is reflected in the output which we

did not do in the OR case. In the OR case we explicitly computed a constant output for each case, here we are saying the output is a variable output depending on the input variable.

(Refer Slide Time: 18:09)

## Recursive definitions

- Base case:  $f(0)$
- Inductive step:  $f(n)$  defined in terms of smaller values,  $f(n-1), f(n-2), \dots, f(0)$
- Example: factorial
  - $0! = 1$
  - $n! = n \times (n-1)!$
$$n! = n \cdot (n-1) \cdot (n-2) \cdots 1$$

$$n-1 \cdot (n-2)!,$$

So, as we saw in the beginning when we are defining functions we typically need to write functions which apply operations or compose functions an arbitrary number of times depending on the value of the input and what we said was that inductive or recursive definitions are the way to this. So, to recursively define a function, we will specify a base case, for example, if it is for the whole numbers we would define the value for  $f(0)$  and then inductively for  $f(n)$  we would write an expression involving smaller values of the arguments, so  $f(n - 1), f(n - 2)$  down to  $f(0)$ .

So, you can define it in terms of any smaller value assuming that those have been inductively computed already. So, this is the meaning of a recursive or an inductive definition and one of the most standard examples used to illustrate recursive and inductive definitions is a factorial function. So, by definition  $0!$  is 1 and  $n!$  is  $n * (n-1)!$  and of course, if you unravel this, this in turn becomes  $(n - 1) * (n - 2) !$  and so on.

So, we will get the familiar expression that  $n!$  is  $n * (n-1) * (n-2) \dots$  up to 0,  $n!$  is the product of all the numbers from 1 to  $n$ , but this is recursively captured by these two rules :  $0!$  is 1 and  $n!$  is  $n * (n-1)!$ . So, now, we can translate this using pattern matching very directly into Haskell.

(Refer Slide Time: 19:42)

## Recursive definitions ...

- In Haskell
  - factorial :: Int -> Int
    - factorial 0 = 1
    - factorial n = n \* (factorial (n-1))
- Note the bracketing in factorial (n-1)
  - factorial n-1 would be read as (factorial n) - 1
- No guarantee of termination: what is factorial (-1)

So, we say that factorial is a function which takes an integer and produces integer ((Refer Time: 19:48)) factorial 0 = 1. So, this is the base case. If my input is already the base case I know the answer, if it is not that base case then I apply the rules saying I take n \* ( factorial (n-1)). So, first note this expression, so we have been careful to put (n-1) in brackets before providing it a factorial, because the way Haskell works if I just write factorial n - 1 without brackets, then it will put the brackets around factorial n as (factorial n) - 1. So, it will compute factorial n and then subtract 1 which is not we want. In fact, that would leave us into a kind of infinite recursion of computation, because factorial n is then defined again in terms of factorial n. So, we will have to apply the same rule and we wont get any answer. So, this is one thing to note.

Another thing to note is that Haskell does not guarantee you the fact that you have written a recursive definition means that it always works correctly on all inputs. Now, here the base case 0 make sense if n is positive, because if n is positive then I come down to a smaller numbers. So, I start with 7 and I come down to 6, 6 will come down to 5 and so on and eventually I will get 0 and it will become 1. What if I start with -1. The function call factorial (-1) will say -1 is not 0. So, the first rule does not apply, so the second rule must apply. So, it will be -1\*factorial (-1 - 1) which is -1\*factorial(-2). Then again the rule does not apply to -2, so we call -3 and so on.

So, this function as stated works correctly for positive values of n, but note that the input type is Int. Haskell's built in type Int includes negative values and there is no Haskell type built in type which consist of only the non negative integers. So, if I write factorial I have to write its type as a Int to Int and this does not prevent me from feeding negative values to factorial and this definition of factorial does not terminate for negative inputs. So, just writing something that looks inductively correct does not guarantee that the computation will terminate for all legal inputs.

(Refer Slide Time: 22:01)

## Conditional definitions

- Use conditional expressions to selectively enable a definition
- For instance, “fix” factorial for negative inputs
 

```
factorial :: Int -> Int
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 0 = n * (factorial (n-1))
```

So, we can fix this by checking if the input is negative, so we need to now extend our syntax for defining functions to use conditional expressions to say when a definition is enabled. So, for instance we have the base cases as before the factorial of 0 is 1, now if it is not 0 it could be positive or negative , the positive case is as before : if  $n > 0$  then I want to say that the output is  $n * \text{factorial}(n-1)$ , but what if  $n < 0$  well of course, factorial of a negative number is not mathematically defined, but for the sake of computation I can always make it work by reversing the sign of inputs.

So, if I ask you factorial of -6 then I will convert it to factorial of 6 and give you that answer. So, at least computationally the value will be produced in finite amount of time instead of going into an infinite condition. So, I say that if  $n < 0$  then  $\text{factorial}(n)$  is computed by taking  $\text{factorial}(-n)$ . So, -6 will become  $-(-6)$  which is +6. So, what we have here is we have a conditional expression  $n < 0$  and so we have this equality which is a function definition.

So, it says that factorial(n) if  $n < 0$  is factorial(-n) and if  $n > 0$  then it is  $n * \text{factorial}(n-1)$ . So, this vertical bar ‘|’ signifies options, so it says evaluate this condition and if this condition is true use this definition; otherwise, go to the next condition , if this condition is true go to the next conditions and so on.

(Refer Slide Time: 23:50)

## Conditional definitions ..

```

factorial :: Int -> Int
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 0 = n * (factorial (n-1))

```

- Second definition has two parts
  - Each part is **guarded** by conditional expression
  - Test guards top to bottom
  - Note the indentation

So, the second definition has two parts. Each part is guarded, so these are called guards. So, it is like a watchman or a security guy saying you can use definition by proving what your value is, only then you can go forward. We check the guards from top to bottom and we use the first guard that works. In this case only one of them : if n is not 0 either  $n < 0$  or  $n > 0$ . Now, notice that n must be an integer, because we have already described a factorial is from Int to Int.

So, if you provide an input which is not an integer Haskell will say this is not a legal value. So, if it is an integer then if it is not 0 it must be positive or negative and exactly one of these things will become true. We will see later that exactly one being true is not required, but it will test them from top to bottom and pick the first guard that works, if no guard works it will go to the next definition, the other thing to note is that we have indented list.

So, technically this whole thing these three lines together constitute one definition with guards. So, there are overall two definitions in this function, there is this line which is the base case with the pattern and this long definition with conditional guards which specifies what happens when the pattern is not matched.

(Refer Slide Time: 25:07)

## Conditional definitions ..

```
factorial :: Int -> Int
pattern factorial 0 = 1
factorial n
condition | n < 0 = factorial (-n)
           | n > 0 = n * (factorial (n-1))
```

- Multiple definitions can have different forms
  - Pattern matching for factorial 0
  - Conditional definition for factorial n

So, therefore, we could have multiple definitions with different forms. So, as we said the first one is a pattern match and this is condition.

(Refer Slide Time: 25:21)

## Conditional definitions ...

- Guards may overlap

```
factorial :: Int -> Int
factorial 0 = 1
factorial n
| n < 0 = factorial (-n)
| n > 1 = n * (factorial (n-1))
| n > 0 = n * (factorial (n-1))
```

factorial 2

match  
3rd rule factorial 1

Now, suppose we fiddle with this definition, this is not very meaningful computation, but I can just I can split this second case  $n > 1$  as first  $n > 1$  and  $n > 0$ . Now notice that if  $n$  is strictly greater than 1 it is also strictly greater than 0, but because we go top down, if I say factorial of 2 then it will say that it is not 0 then it will come here then it will say it is not less

than 0 it will come here and it will be matched. So, though it matches the third rule it will not reach the third rule.

So, the only value that will reach the third rule is factorial of 1. Because, factorial of 1 will say it is not 0, so it will come here it is not less than 0, so it will come here, it is not greater than 1 will come here. So, finally, this will only match this rule. So, guards may overlap, there is no requirement that the guards must be exclusive that exactly one of them is true. So, you need not worry about that.

(Refer Slide Time: 26:31)

## Conditional definitions ...

- Guards may not cover all cases

```
factorial :: Int -> Int
factorial 0 = 1
factorial n
| n < 0 = factorial (-n)
| n > 1 = n * (factorial (n-1))
```

- No match for factorial 1

```
Program error: pattern match failure: factorial 1
```

Now, the other thing is, just as we saw before the fact that recursive definition of Haskell need not terminate, Haskell need not promise you that the function definition actually terminates and gives you a value. For instance, when we did not take care of negative input factorial of -1 was giving us an unterminating computation.

So, here supposing I write this definition and I have made a mistake, so I got the guards as  $n < 0$  and  $n > 1$ . So, now, if I look at the set of integers then I have this factorial  $0 = 1$  for the base case. So, this is the definition and I have everything from 2 onwards covered by this definition. Now, I have basically a problem because, I covered the negative inputs and inputs greater than one, but if say factorial of 1 it does not match the base case, it does not match either of the guards and now what will happen is that if I actually run this in Haskell it will give me an error saying that no patterns match.

So, the function definition can miss cases and in some situations when you miss cases some values may give you outputs which are correct for some values. So, here in this particular definition factorial of 0 will work, because it will not look at the second part. But, any factorial which requires a number bigger than 0, to come down to 0 it will have to eventually compute factorial 1. If I say factorial of 3 it could be 3\*factorial 2, 2\*factorial 1 and when I try to evaluate factorial 1 and get this error message saying that program has a match failure.

So, not just if I provide factorial 1, but if I provide factorial of anything even though the first few times it may match eventually the recursive computation will come down to factorial 1 and when it comes to factorial 1 , I will get this error. So, it is your responsibility to make sure that the conditional definitions cover all the cases. So, that more function value is undefined.

(Refer Slide Time: 28:43)

## Summary

- A Haskell function consists of a type definition and a computation rule
- Can have multiple rules for the same function
  - Rules are matched top to bottom
  - Use patterns, conditional expressions to split cases

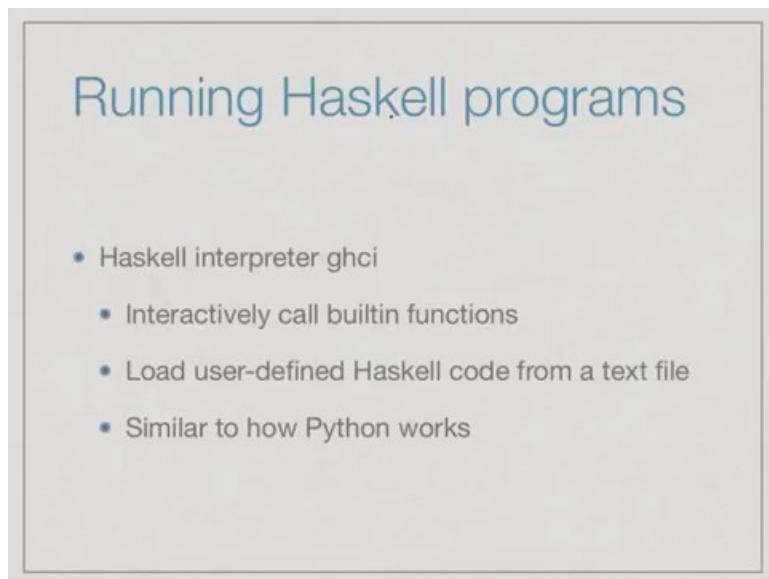
So, to summarize, a Haskell function consists of a type definition and a computational rule. Now, these computational rules can be used to define for example, recursive functions. We can have multiple rules for the same function and the rules are matched top to bottom and the way we defined rules for different input values is to use patterns or conditional expressions to split the cases that are possible for the input values.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

**Module # 01**  
**Lecture - 04**  
**Running Haskell Programs**

Having seen how to write Haskell programs, let us now look at how to run them on the computer.

(Refer Slide Time: 00:08)



Running Haskell programs

- Haskell interpreter ghci
  - Interactively call builtin functions
  - Load user-defined Haskell code from a text file
  - Similar to how Python works

The easiest way to run a Haskell program is to use the Haskell interpreter ghci. Using ghci will be very familiar to people who have used languages like python. So, ghci is an interpreter, you load ghci and then you can interact with it. You can directly call built in functions or you can load user defined Haskell code which you have previously entered into a text file.

(Refer Slide Time: 00:35)

## Setting up ghci

- Download and install the Haskell Platform
  - <https://www.haskell.org/platform/>
  - Available for Windows, Linux, MacOS

ghci is freely available on the internet; the easiest way to get ghci working on your system is to download what is called the Haskell platform from the web URL given here.

(Refer Slide Time: 00:51)



So, here is a screenshot of what the Haskell platform website looks like, as you can see you can download versions of the Haskell platform for any operating system for windows, for Mac OS and for Linux.

(Refer Slide Time: 01:10)

## Using ghci

- Create a text file (extension .hs) with your Haskell function definitions
- Run ghci at the command prompt
- Load your Haskell code
  - `:load myfile.hs`
- Call functions interactively within ghci

So, how do we actually use ghci? We first create a text file with the extension ‘.hs’ to indicate that it is the Haskell file containing your Haskell function definitions exactly as we wrote them in our previous lecture, then we run ghci from the command line. Inside ghci you can then feed it further commands and the most basic command you need to know is a command to load Haskell code, this command is written as `:load` followed by the file name, note the use of colon. So, colon is an indicator to ghci that it has to perform some internal action, anything without a colon is interpreted as a Haskell function that it must evaluate. So, in this way you can interactively call function from within ghci, let us look at some examples.

(Refer Slide Time: 02:01)

```
boolean.hs      factorial-fail.hs    factorial-good.hs
factorial-positive.hs factorial.hs      intreverse.hs
madhavan@dolphinair:...o-lectures/week1/code$ vi factorial.hs
madhavan@dolphinair:...o-lectures/week1/code$ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load factorial.hs
[1 of 1] Compiling Main           ( factorial.hs, interpreted )
Ok, modules loaded: Main.
*Main> factorial 7
5040
*Main> factorial 5
120
*Main> 4 + 5
9
*Main> True || False
True
*Main> div 7 3
2
*Main> sqr x = x * x

<interactive>:8:7: parse error on input '='
*Main> |
```

So, here is the command window in a Unix like system. So, in this we have already created some text files, but let us start from scratch, so supposing we want to create a version of factorial program. So, we open an editor like vi or Linux. So, let us say vi factorial.hs

(Refer Slide Time: 02:22)



```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
-
```

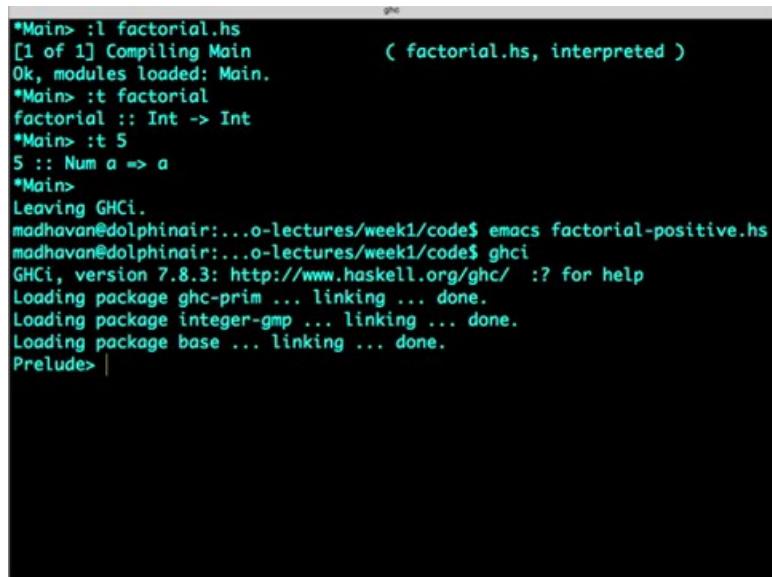
A screenshot of a terminal window titled "vim". It contains the first few lines of a Haskell program. The code defines a function named "factorial" that takes an integer and returns an integer. It has two cases: if the input is 0, it returns 1; otherwise, it returns the input multiplied by the factorial of the input minus one. The cursor is positioned at the end of the third line.

So, this is the first factorial program that we wrote, so there is an error, so let us fix that. So, it says that the type of factorial is Int -> Int, factorial 0 = 1 and factorial n = n \* factorial (n-1). So, now, we can exit from the editor ((Refer Time: 02:37)), save the file and load ghci. So, ghci will load up and give us a command prompt of its own, which in this case is with the prefix Prelude. Now, we can load the file that we just created using the :load command that we have discussed and now if all is well you will get message like this saying it is okay. Having done this, we can now ask it to evaluate versions of factorial like factorial of 7 is 5040, factorial of 5 if you want to check a familiar number is 120 and so on.

You can also evaluate built in functions, for instance you can ask 4+5. So, you can use it as a calculator, this is pretty much how you will use it in things like python, you can say true or false or you can say div 7 3. So, it is very similar to the python interpreter if you use that, in that you can invoke any built in function or any function which you have defined in the file that has been loaded. As we will see one of the things that you cannot do in this, which you can do in a python interpreter is to define functions here. So, I cannot write a new function here which says something like  $\text{sqr } x = x*x$ . So, this kind of a definition which is allowed in

python, is not allowed in the interpreter, you have to write this in a separate file and then load.

(Refer Slide Time: 03:56)



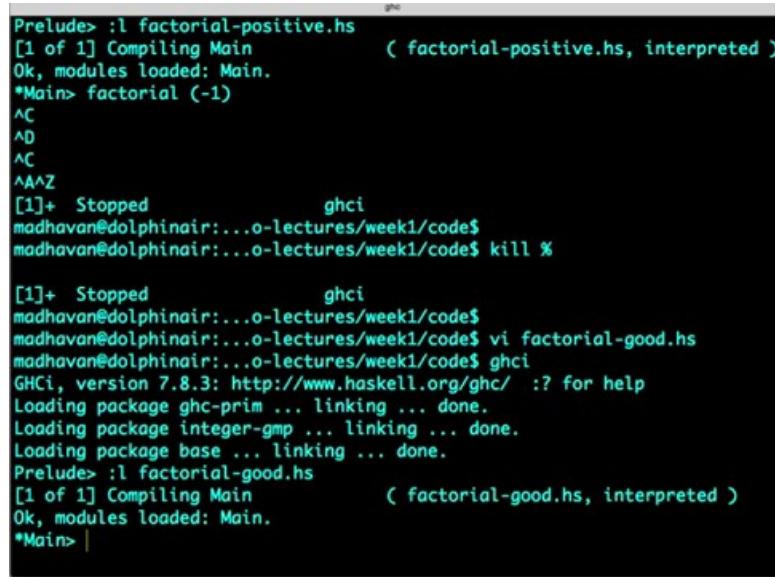
```
*Main> :l factorial.hs
[1 of 1] Compiling Main             ( factorial.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t factorial
factorial :: Int -> Int
*Main> :t 5
5 :: Num a => a
*Main>
Leaving GHCi.
madhavan@dolphinair:...o-lectures/week1/code$ emacs factorial-positive.hs
madhavan@dolphinair:...o-lectures/week1/code$ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  ?: for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> |
```

So, as a shortcut for instance you can use :l instead of load, first of all if you say :h you get a list of all the possible commands that you can do in ghci in case you want to investigate, but :l is a short form. So, I can say :l factorial.hs and now it will just replace the old factorial by a new factorial. The other thing is that you can query types, for instance I can now ask what is the type of factorial and it tells me that the type of factorial is Int->Int.

So, if you have a function whose type you are unsure of you can ask this, but beware that the types reported here can be more complicated than the types we have seen. For instance, you might ask what is the type of a number 5, the number 5 you would expect as type Int, but actually it gives you this complicated thing which says Num a => a. We will see later what these things mean.

But, for functions that you define you can verify that the types are indeed what is say they are. So, let us get out of the interpreter for a moment and now let us look at this factorial that we wrote. So, I have got the same code in a file called factorial-positive and this is to indicate that this factorial handles only the positive case. So, this is the same factorial we just used except I added now a comment. A comment in Haskell starts with two hyphens on the first two columns of line. So, the first line is a comment which says this is a factorial that does not handle negative inputs, which we already know, but lets see what it does.

(Refer Slide Time: 05:39)

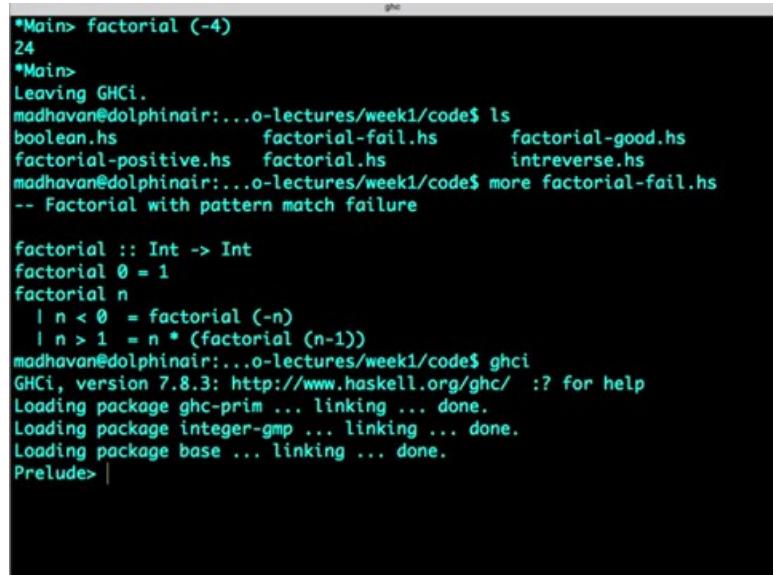


```
Prelude> :l factorial-positive.hs
[1 of 1] Compiling Main           ( factorial-positive.hs, interpreted )
Ok, modules loaded: Main.
*Main> factorial (-1)
^C
^D
^C
^C^Z
[1]+  Stopped                  ghci
madhavan@dolphinair:...o-lectures/week1/code$ vi factorial-good.hs
madhavan@dolphinair:...o-lectures/week1/code$ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l factorial-good.hs
[1 of 1] Compiling Main           ( factorial-good.hs, interpreted )
Ok, modules loaded: Main.
*Main> |
```

So, if I go back to ghci and I load this factorial-positive.hs and I ask it to do factorial say -1, then as we would expect it goes into an infinite computation of factorial -2 , -3 and so on. Because, we have not handled the negative case and the only way to get out of this is to physically break the program from running, and one way to do this is use a control C or control D or nothing works unfortunately.

So, we have to figure out a way to abort the program. So, now, instead we have written a better version which we had done already in the class, which said if  $n < 0$  then you negate the factorial. So, now this is a factorial with negative inputs.

(Refer Slide Time: 06:45)

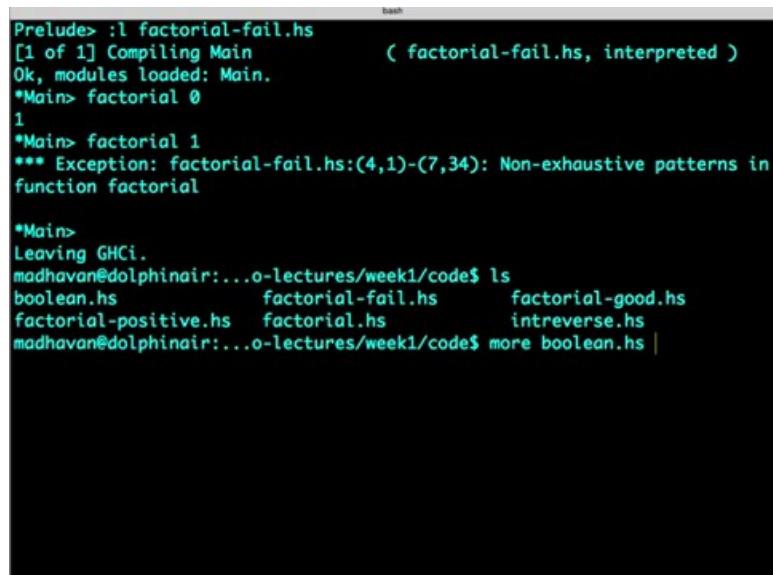


```
*Main> factorial (-4)
24
*Main>
Leaving GHCi.
madhavan@dolphinair:...o-lectures/week1/code$ ls
boolean.hs      factorial-fail.hs    factorial-good.hs
factorial-positive.hs factorial.hs    intreverse.hs
madhavan@dolphinair:...o-lectures/week1/code$ more factorial-fail.hs
-- Factorial with pattern match failure

factorial :: Int -> Int
factorial 0 = 1
factorial n
| n < 0  = factorial (-n)
| n > 1  = n * (factorial (n-1))
madhavan@dolphinair:...o-lectures/week1/code$ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> |
```

So, now, if I load this version and now they say factorial of -4 for instance then it will be converted to +4 and give me factorial of 4. Now, we also saw that we could have this kind of factorial function, where we have made a mistake. So, instead of  $n > 0$  we have written  $n > 1$  and this says that there will need not be any case when factorial 1 works. So, let us see what happens when we load this.

(Refer Slide Time: 07:10)



```
Prelude> :l factorial-fail.hs
[1 of 1] Compiling Main           ( factorial-fail.hs, interpreted )
Ok, modules loaded: Main.
*Main> factorial 0
1
*Main> factorial 1
*** Exception: factorial-fail.hs:(4,1)-(7,34): Non-exhaustive patterns in
function factorial

*Main>
Leaving GHCi.
madhavan@dolphinair:...o-lectures/week1/code$ ls
boolean.hs      factorial-fail.hs    factorial-good.hs
factorial-positive.hs factorial.hs    intreverse.hs
madhavan@dolphinair:...o-lectures/week1/code$ more boolean.hs |
```

So, now, if I say factorial of 0 it does not go into the problematic case, so it is fine. But, if I say factorial of 1 then I get some kind of pattern match failure. So, it says it is an exception

factorial fail and so on. So, the actual error message depends on the version of ghci, the error message written on the slides is slightly less expressive than this. So, you get some kind of error message and now finally, we have also written a version of the XOR and OR that we did. So, we have xor and or.

(Refer Slide Time: 07:54)

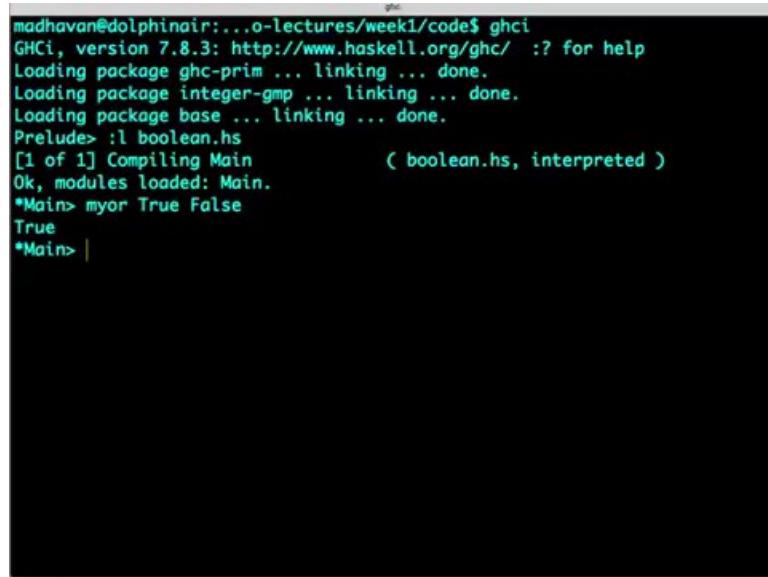
```
batch
madhavan@dolphinair:...o-lectures/week1/code$ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load boolean.hs
[1 of 1] Compiling Main           ( boolean.hs, interpreted )
Ok, modules loaded: Main.
*Main> xor True False
True
*Main> or True False

<interactive>:4:1:
  Ambiguous occurrence `or'
    It could refer to either `Main.or', defined at boolean.hs:19:1
    or `Prelude.or',
      imported from `Prelude' at boolean.hs:1:1
      (and originally defined in `GHC.List')
*Main> Main.or True False
True
*Main>
Leaving GHCi.
madhavan@dolphinair:...o-lectures/week1/code$ emacs boolean.hs
madhavan@dolphinair:...o-lectures/week1/code$ |
```

Now, here we have another problem which comes up and which you should be aware of. So, if I say load boolean.has now if I say ‘xor True False’ there is no problem. But if I say ‘or True False’ then I get an error message saying that is confused by the word ‘or’, because the built in symbol for OR is the two vertical lines ‘||’, but ‘or’ as an English word can also be used in a Haskell program.

So, ‘or’ is a built in function and it says that the built in function ‘or’ is in conflict with the ‘or’ which is defined in this file. Now of course, one could say Main.or and this Main is the version that we have just loaded or even better than this is to actually go back and edit the file and use a new name. So, it is conventional to use the prefix my, so whenever you have a function whose name looks like a familiar function it is best to rename it with something which distinguishes it from the familiar function.

(Refer Slide Time: 09:03)

A terminal window showing a session in the Haskell interpreter ghci. The session starts with the GHCi version (7.8.3) and package loading (ghc-prim, integer-gmp, base). It then loads a module named boolean.hs, which contains a function myor. The interpreter shows the result of calling myor with arguments True and False, returning True. The session ends with a prompt \*Main>.

```
madhavan@dolphinair:...o-lectures/week1/code$ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l boolean.hs
[1 of 1] Compiling Main           ( boolean.hs, interpreted )
Ok, modules loaded: Main.
*Main> myor True False
True
*Main> |
```

So that now if I load this and say ‘myor True False’ it will work fine. So, we have seen how to load files into the interpreter, some kinds of error messages that you see. You can also query types. So, it is very easy to use this interpreter and since Haskell program tend to be small, it is not very difficult to debug them as we will see. So, please get familiar . Do download ghci onto your system and start playing around it.

(Refer Slide Time: 09:37)

## Caveats

- Cannot define new functions directly in ghci
  - Unlike Python
  - Must create a separate .hs file and load it

So, as we mentioned when we were showing the use of ghci, one of the differences between ghci and python is that you cannot create function definitions on the fly. So, within the

interpreter you cannot define a new function, you must load it as a separate function from an outside file. So, you must first create a .hs file and then load it into the interpreter.

(Refer Slide Time: 10:00)

## Compiling

- ghc is a compiler that creates a standalone executable from a .hs file
  - ghc stands for Glasgow Haskell Compiler
  - ghci is the associated interpreter
  - Using ghc requires some advanced concepts
  - We will come to this later in the course

So, this is an interpreter, but Haskell also comes with a full fledged compiler. So, ghc is a compiler that creates an executable file from a .hs file. So, ghc stands for the Glasgow Haskell compiler and ghci is the associated interpreter which is the one that we have seen. Now, unfortunately we cannot use ghc directly on the Haskell function that we have been writing. We need some more advanced concepts which we will see as we go along. So, at some point we will start compiling our code, but for the moment we will just stick to ghci the interpreter.

(Refer Slide Time: 10:33)

## Summary

- ghci is a user-friendly interpreter
  - Can load and interactively execute user defined functions
- ghc is a compiler
  - But we need to know more Haskell before we can use it

So, to summarize the easiest way to run Haskell code is to use the interpreter ghci, which is quite user friendly and in which we can load and interactively execute user defined functions, there is an associated compiler called ghc but we need to know more Haskell before we can use it.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

**Module # 01**

**Lecture – 05**

**Currying**

Let us now turn to the mysterious notation we have been using for functions with multiple inputs.

(Refer Slide Time: 00:08)

So, what we have been seeing is that when we have a function which takes two inputs like plus. Instead of using the familiar notation, where we use a bracket and we put the arguments inside the brackets separated by commas, we just write the arguments one after the other separated by spaces. Now, when we write a function like plus in the usual way, we indicate that it takes two arguments and this is called its arity. So, the arity of a function is how many arguments it takes.

So, in this case we say that plus is a binary function. So you have binary functions that take two arguments, unary functions take only one argument and so on. So, this becomes part of the definition of the function and when we use the function we have to ensure that we supply the right number of arguments. So, this is the conventional point of view. Now our radical departure from this is to assume let all functions take only one input and this is the principle behind the notation that we have been using and we will see in a minute how it works and

just as a piece of terminology, this style of writing functions where every function is the function of only one argument is called currying.

So, it has nothing to do with cooking, but rather it is named after the logician Haskell Curry, who made it popular. So, Haskell Curry is not in fact, the person who invented it. The person who invented it was another logician called Schonfinkel, but Haskell Curry was a logician who made this style of notation for functions popular. And another piece of interest for us is that the language Haskell, you might have wondered where the name Haskell came from, where Haskell is actually named after the logician Haskell Curry.

(Refer Slide Time: 01:53)

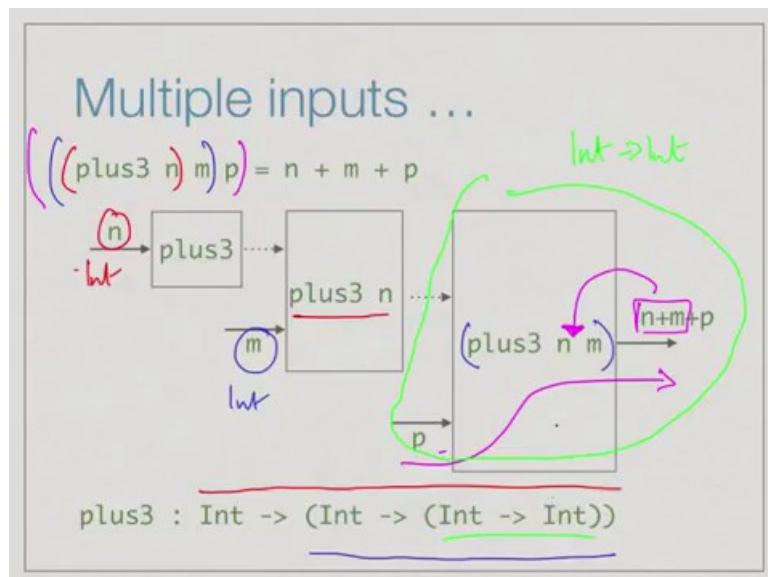
So, let us look at these two view points of functions. So, on the left we have a familiar plus which takes two arguments n and m and produces an output or the answer  $n+m$ . The right is a picture of how we are supposed to think about the curried version. So, what we are saying now is that plus as we have defined it takes only one argument. So, the one argument it takes is the first argument, so effectively we start by consuming this n. So, plus consumes n, so that is this box and it produce the new function in which n has now been observed.

So, now, we have a function which will add n to whatever it gets, so that is the principle. So, instead of consuming multiple arguments in one shot, you consume one argument at a time and then you transform yourself into a new function in which part of the functionality is internalised. So, plus has consumed the first argument and became a fixed function called plus n, which will add n to whatever argument it gets. So if the first argument to plus was 2, then now this new function we call plus 2. It will add 2 to whatever it gets.

And now this new function is going to consume the  $m$ . This is now going to take this  $m$ , add  $n$  to it, which has already been built in, and give us  $n+m$ . So, the idea in currying is that instead of multiple arguments you get a sequence of functions. So, you consume one argument at a time, each argument transforms the function in some way by internalizing the most recent argument and creating a new function which will consume one more argument and so on.

So, let us try to look at the type of the new plus that we have defined. So, if we start from the end the last thing that happens is function  $\text{plus } n$  which takes an Int as an input and produce an Int as output. So, this particular function here is  $\text{Int} \rightarrow \text{Int}$  and this whole thing is the output of  $\text{plus}$ . So, therefore, the type of  $\text{plus}$  is something that consumes an Int and produces this function. So, that is why we get that the type of  $\text{plus}$  is from an input Int to an output  $\text{Int} \rightarrow \text{Int}$ , that is,  $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ . So, this output  $\text{Int} \rightarrow \text{Int}$  is this box and the input Int is the original Int. So, working backwards we can take a curried function and work from the last box backwards to construct its type like this.

(Refer Slide Time: 04:34)



Let us look at another example one that takes maybe 3 inputs. So, supposing we have a  $\text{plus3}$  which will add 3 numbers. So, now, again in the same way when I first consume the first  $n$  I will get something which internalizes the  $n$ , so now it will add  $n$  to whatever it is going to get as a next input, now we are not done. So, we will now consume one more input which is this one and we will now have something which adds  $n + m$  to whatever it gets.

And finally, when we consume the third input here we will have this function which takes us

from a number p to the number n + m + p and the point is this n + m is kind of built into the function. It has already been hard wired in a sense, because we have consumed n and m as previous inputs. So, now, again working backwards, so this last thing is the last box here, so this is an Int  $\rightarrow$  Int function, so that is this type.

So, therefore, if we go to the previous box, so this now takes as input an Int and produces that. So, that is this function and finally, the outermost takes an Int and produces Int  $\rightarrow$  Int  $\rightarrow$  Int and therefore, the type of plus3 this is whole expression. So, if we have a curried function we consume the inputs one at a time, each input transforms a function into one, where one value is frozen in some sense. So, the function now becomes one where one argument is fixed. So, we keep consuming one argument at a time and keep transforming the function. So that, it now has some values fixed in to it and we can recover the type of the function by working backwards from the last box to the first box.

(Refer Slide Time: 06:20)

## Multiple inputs ...

- Consider a function with many arguments  

$$f \ x_1 \ x_2 \ \dots \ x_n = y$$
- Suppose each  $x_i$  is of type Int,  $y$  is of type Bool
- Type of  $f$  is  

$$f :: \text{Int} \rightarrow (\text{Int} \rightarrow (\dots (\text{Int} \rightarrow \text{Bool}) \dots))$$
- Correspondingly, we should write  

$$(\dots ((f \ x_1) \ x_2) \ \dots) \ x_n = y$$

$$\underbrace{f \ x_1}_{f_1} \ \underbrace{x_2}_{f_2}$$

So, in general we could have a function which takes say n arguments and produces an answer. So, suppose in this particular function that we are looking at each of these inputs 1 to n is Int and say the last one is of type Bool then by our earlier description we would start by working backwards. So the last box would take the last input  $x_n$  and produce  $y$ . So, this would be Int  $\rightarrow$  Bool, so this is this last thing and then the previous box would have taken  $x_{n-1}$  and produces function that would be Int  $\rightarrow$  Int  $\rightarrow$  Bool and so on. So, we will get this nested thing sequence of Int  $\rightarrow$  Int  $\rightarrow$  Int  $\rightarrow$  Int and finally with an Int  $\rightarrow$  Bool.

And logically if we look at the expression, the original function  $f$  first consumes  $x_1$  and it

becomes a new function. So, this is our second box, so this is like an  $f'$  or an  $f_1$ .  $f_1$  consumes  $x_2$  and this becomes a new function which has  $x_1$  and  $x_2$  into it, we call  $f_2$ . So, we have this implicit bracketing of the types in one sense, where the innermost bracket corresponds to the right most function, the last input in the last output and we have the corresponding bracketing of the way the function is used which is that we first consume  $x_1$  then we consume  $x_2$  and so on.

(Refer Slide Time: 07:50)

Now, fortunately Haskell knows this, so there is an implicit bracketing which Haskell uses depending on the type of expression that we used, we are familiar with the implicit bracketing for arithmetic for example, we have this BODMAS. So, we have in BODMAS it says that if I write for instance  $7 + 2 * 3$  then it is not going to be  $9 * 3$ , this is not a correct answer, so this is not  $9 * 3$  rather it is  $7 + 6$ . So, there is a precedence which says that it is bracketed like this. So, that says division and multiplication are bound tighter than addition and subtraction.

Now, in a similar way when it sees expression like these arrows, then Haskell knows that it must bracket them in a particular way and in particular it will bracket this starting from the right. So, it will first put bracket only to `Bool` then around the previous one and so on. So, it will produce from the upper expression without any brackets a lower expression and we can freely use the upper expression without worrying about all these messy nested brackets.

(Refer Slide Time: 08:50)

## Multiple inputs ...

- Likewise, function application brackets from left
- So
$$f \ x_1 \ x_2 \dots x_n$$
means
$$(\dots((f \ x_1) \ x_2) \ \dots) \ x_n$$
- Which is why we have to be careful to write
$$\text{factorial } (n-1)$$
because
$$(\text{factorial } n)-1$$
means
$$(\text{factorial } n) - 1$$

The same way when we write a function with arguments as a call to the function, it will implicitly assume that the function is bracketed from the left. So, it will put in these brackets it will start with bracket around  $x_1$  then a bracket around  $x_2$  and so on and finally, it will produce bracket around this. So, we do not have to worry about this bracket, so that is very nice for us, so it means that the built in bracketing rules in Haskell take care of the usual thing that we expect. So, unless we want to do something unusual, we do not need brackets.

Now, we have seen an example where the scan creates a problem, remember when we defined factorial we had to be careful to put a bracket around the  $n-1$  and that is precisely because of this rule. Because, Haskell when it sees  $\text{factorial } n - 1$  without bracket will first take factorial and bind it to nearest thing and this is like having two different operators, one is factorial with an argument and the other is subtraction.

So just like division and multiplication will get bound ahead of subtraction, so will the function call. So, this becomes  $(\text{factorial } n) - 1$  and this is not what we expect, so we have to be a little aware of how this bracketing is done. Because, in such situations we may need to insert brackets to ensure that Haskell is understanding the expression the way we intend it.

(Refer Slide Time: 10:07)

## Summary

- Haskell uses currying
- Every function takes only one argument
  - A multi argument function is a sequence of single argument functions
- Helpful bracketing conventions for types and function application

So, to summarize Haskell uses currying as a notation for functions and in currying, the basic simplification is that functions do not have arities, we do not have to say a function is a binary function or a k-ary function taking k arguments, every function takes only one argument. So, if a function takes multiple arguments it consumes these one after the other, each argument transforms the function by internalizing that argument and making it into a new function where one of the arguments is frozen.

So, this becomes very convenient we will also see that we can use these intermediate functions in other contexts. So, actually if we define a function of two arguments then a function in which only one argument is provided in currying is partially another function, let us say if I take plus m n and I feed 7 then plus 7 is a new function which will add 7 to whatever it gets. I can actually treat this partially instantiated function as a real function in many contexts we will see.

And associated with currying is the implicit bracketing which is right to left for types and left to right for function application, but fortunately this bracketing is built into Haskell's bracketing rules along with arithmetic and Boolean expressions. So, we do not have to use brackets unless we really want to disambiguate something.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

**Module # 01**

**Lecture – 06**

**Haskell Examples**

For the last lecture of this week, we will look at some examples of programs in Haskell to get used to the idea of programming in this language.

(Refer Slide Time: 00:10)

Pattern matching

- Recall our example

```
and :: Bool -> Bool -> Bool
and True b = b
and False b = False
```

- The second argument is used differently in the two definitions
  - First definition: the value `b` determines the answer
  - Second definition: the value `b` is ignored

Before we do that, we will revisit a couple of the notions that we have seen for writing Haskell functions and explain a few concepts that we did not do when we introduced them. So, we begin with pattern matching. Remember that in Haskell functions, we can use patterns to illustrate cases. So, here is an example of the ‘and’ function, so the ‘and’ function does the pattern match on the first argument.

If the first argument is True, then the first definition is used and in this case it returns the value of the second argument. If the first argument is False, we already know that the answer is False. So, we can ignore the second argument and just return False. So, here we have True and False as constant patterns and `b` is a usual kind of pattern, which is matched by whatever it is in box.

Now, one thing to notice in this is that, the value b that we write here in the two definitions actually has two different uses. In the first case, the value b is important, because whatever we feed as b to ‘and’, comes back to us as the output. So, it is important to connect the output b to the input b, the b in the answer is same as the b that is given as input to ‘and’.

On the other hand, in the second case, we see that this particular thing gets discarded, because we do not really care, what we use, False and anything is False. So, the answer is immaterial, the value of b is immaterial, the answer is independent of b. So, Haskell then ask why we need to provide a name, if we are going to ignore the argument.

(Refer Slide Time: 01:43)

## Pattern matching ...

```
and :: Bool -> Bool -> Bool
and True  b = b
and False _ = False
```

- Symbol `_` denotes a “don’t care” argument
  - Any value matches this pattern
  - The value is not captured, cannot be reused

So, there is a special notation in Haskell, which allows us to specify a wild card. So, this symbol underscore denotes an argument which matches everything. So, it is like using a variable, remember a variable matches everything and it will be substituted by the value we provide. So, the difference between a wild card, which is the don’t care kind of argument and a name like b is that, the value you give is going to be thrown away.

So, any value matches our underscore, but the value is not captured, cannot be reused, but two things, one is it avoids unnecessarily putting a name in there, we do not have to think of a name. Second thing is that it also makes it transparent that this function really does not care about second value. So, from the point of view of reading the code it is immediately obvious that ‘and’ with False in the first argument does not require the second argument to give its answer.

(Refer Slide Time: 02:36)

## Pattern matching ...

```
or :: Bool -> Bool -> Bool
or True _      = True
↓ or _      True = True
or F   F      = False
• Can have more than one _ in a definition
```

So, here is more extreme version of that, here is the version of ‘or’ that we have written, but now with these wild cards. So, ‘or’ says that if the first argument is True, it does not matter what the second argument is, the answer is True. Similarly, if the second argument is True, it does not matter what the first argument is, the answer is True. And now, we have skipped both of these, no matter, what I give, the answer is False, the last definition on its own would be totally useless, but because it comes after the first two definitions. So, we have already gone past these two cases, we know that, this must be False, this must be False in this case, but what it says is, if I have reached here at this point, if I reached the third line and I have not matched the first two lines, then no matter what the value of the two arguments, the answer must be False. Notice, you were using two underscores in the same definitions. So, it is really emphasizing the fact that underscore is not a name of a value.

We cannot write two arguments with the same name in a normal definition, because then there will be confusion about which value we are referring to use it on the right hand side. So, if we say plus m n; we cannot write plus x x; because we do not know in the right hand side, which x we are going to use, whereas, if you use underscore, you cannot use the value on the right hand side. So, you need only one underscore across all definitions, because it is just going to be a value that is ignored.

(Refer Slide Time: 03:53)

## Conditional definitions

- A variant of factorial

```
factorial :: Int -> Int
factorial n
| n == 0 = 1
| n > 0  = n * (factorial (n-1))
| n < 0  = factorial (-n)
```

- Have we really covered all cases?
- Can some invocation fall through with Pattern match failure?

The other new concept that is useful to use is to do with conditional definitions. So, here is a variation of our factorial function which handles negative numbers. Earlier, we had given factorial 0 as a pattern match, so we had written something like factorial 0 = 1 as the separate case. And then, we had written this conditional thing only for  $n > 0$  and  $n < 0$ .

But, we can as well use the conditional to give a completely conditional function. So, this in a more conventional language, you will say something like, if  $n == 0$ , then the answer is 1, else if  $n > 0$ , then the answer is  $n * \text{factorial}(n-1)$ . Else, if  $n < 0$ , then factorial is the value  $\text{factorial}(-n)$ . So, now, notice that, we are forcing ourselves to give a guard for each case.

And in particular, we might reach a situation, where we wonder whether there is a value of  $n$  for which there is no guard. Here we can check that  $n$  must be 0 or greater than 0 or less than 0. But, if it is for a more complicated situation, it may well be difficult to determine whether every input actually matches one of these things. So, have you really covered all the cases or is there a situation, where we have missed something in which case some inputs might give us some error about pattern match failures.

(Refer Slide Time: 05:09)

## Conditional definitions ...

- Replace the last guard by otherwise

```
factorial :: Int -> Int
factorial n
| n == 0 = 1
| n > 0 = n * (factorial (n-1))
| otherwise = factorial (-n)
n<0
```
- “Catch all” condition, always true
- Ensures that at least one definition matches

So, in the equivalent of the else in a normal programming language, normally we say if condition 1 then something ,else condition 2 then something and so on and finally, the last one is the catch all else. So, here is a catch all word called otherwise. Instead of saying explicitly that the last case is  $n < 0$ , we can use this word otherwise to say that, if it is not 0 and is not greater than 0, then do this.

So, ‘otherwise’ is the special reserved word in Haskell, which is a condition that is always True. So, if we reach the ‘otherwise’ condition in a sequence of guards, then that guard will always evaluate to True and whatever definition is provided to that guard will match. So, this ensures that every invocation will match at least one of the definitions in a condition. So, we have these two new things, we have this don’t care patterns and ‘otherwise’ and we will use these to simplify some of the code that we will develop in the examples of this type.

(Refer Slide Time: 06:08)

Example: gcd

$$\begin{aligned} \text{gcd}(18, 12) \\ = \text{gcd}(12, 6) \\ = \text{gcd}(6, 0) \Rightarrow 6 \end{aligned}$$

- Euclid's algorithm for  $\text{gcd}(a,b)$ , assume  $a \geq b$ 
  - If  $b$  is 0, define the answer to be  $a$
  - Otherwise,  $\text{gcd}(a,b) = \text{gcd}(b, \text{mod } a \ b)$

So, let us look at our first example, the first example is a very traditional example and one which illustrates many interesting concepts in programming in all languages. So, this is the gcd algorithm. So remember that the gcd of a and b is the greatest common divisor, it is the largest number that divides both a and b and since 1 definitely divides both a and b, this is always well defined, it is at least 1. So, if the gcd is 1, then they are said to be co prime.

So, assuming that a is the bigger of the two numbers, then what Euclid proposed was the following algorithm. It is that we keep replacing the gcd of a, b by the gcd of the smaller number and the remainder of the larger number divided by the smaller number. So, for instance, if we start with the gcd of say 18 and 12, then this will be replaced by gcd of this smaller number 12 and the remainder if I divide 18 by 12. I get 6 as a reminder because it is 1 with a remainder 6. Then I will get gcd of 6 and the remainder of 12 divided by 6. It is 0 and then, this would be the base case and this will say gcd is 6. If b is 0, the answer is a. So, this is Euclid's algorithm, we will not worry about its correctness, but it is useful for you to think about why this works. It keeps reducing gcd to smaller and smaller numbers and you are guaranteed that it will terminate. In fact, it is a very efficient algorithm; it turns out to be proportional to the number of digits. So, it is actually a linear algorithm in the size of its inputs.

(Refer Slide Time: 07:42)

## Example: gcd

- Euclid's algorithm for  $\text{gcd}(a,b)$ , assume  $a \geq b$ 
  - If  $b$  is 0, define the answer to be  $a$        $a < b$
  - Otherwise,  $\text{gcd}(a,b) = \text{gcd}(b, \text{mod } a \ b)$

```
gcd :: Int -> Int -> Int
gcd a 0 = a
gcd a b
| a >= b    = gcd b (mod a b)
| otherwise  = gcd b a
```

So, this is a very simple function to code in Haskell. So, gcd first of all takes two Integers and produces an Integer; that is the type `Int -> Int -> Int`. So, the base case is easy,  $\text{gcd } a \ 0 = a$  , for any  $a$  is just the value  $a$  itself. Then, if we assume as we have done here; that  $a > b$ , then we take the gcd of the smaller number and we have the mod function which gives us the remainder.

So, it is,  $\text{gcd } a \ b$ . If  $b$  is not already 0 it is given by  $\text{gcd } b$  and the remainder of  $a$  divided by  $b$  and in case it turns out that this assumption is false. So, we actually have  $a < b$ , then we just reverse it. So, if, here we are using this new word that we have discovered called ‘otherwise’. So,  $a$  happens to be smaller than  $b$ , then we just invoke gcd in the reverse order, after that notice that since  $b$  is a smaller number then the remainder will be smaller than  $b$ , all successive calls to gcd will have this property. So, only the first call which may be wrong in which case we reverse it. So, this is like our earlier case where we said factorial of a negative number can be fixed by taking the negation at the end.

(Refer Slide Time: 08:48)

## Example: Largest divisor

- Find the largest divisor of n, other than n itself
- Strategy: try n-1, n-2, ... In the worst case, we stop at 1

```
largestdiv :: Int -> Int
largestdiv n = divsearch n (n-1)

divsearch :: Int -> Int -> Int
divsearch m i
| (mod m i) == 0 = i      i divides m ✓
| otherwise        = divsearch m (i-1)
Aux function
```

So, here is another example, supposing we want to find the largest divisor of n other than n itself. So, of course, n divides n, but then other than n, what is the next smallest divisor. So, here is the simple strategy, we know that 1 divides n. So, 1 is certainly a candidate just like in GCD, we know that 1 is a GCD, so 1 will always work. So, here we can just iteratively try n-1, n-2.

So, we go back we want the largest one. So, we go back in reverse order from n, we start with n-1, because we do not want n itself and we check for each of these. So, it is just a normal programming, this would just be a loop, we are just trying for each i from n-1, n-2, down to 1, if i divides n, then we stop and report that. So, this loop can be simulated using a simple recursive call, so we say that the operative thing is actually the second function.

So, here is an example where the function we want to write is expressed in terms of another function. So, the largest divisor of n can be obtained by searching for divisors of n starting from n-1. How do we search, well, if we get an answer, if the second argument divides the first argument, so if m divided by i is 0, then we have found the answer, so we return i.

So, this means that i divides m, otherwise we will go to the next one and the reason which terminates is that, eventually i-1 is going to become 1 and 1 will always divide. So, we have an auxiliary function. So, this is something about Haskell, you can write a bunch of functions together, it does not matter in what order you write them, it does not have to be before or after and these functions can call each other.

So, we have largestdiv the function we want, which calls this divsearch. The interesting thing about divsearch is that it is a kind of recursive version what you normally call as loops, so it is looking at i and if it is not i, it is going to i-1.

(Refer Slide Time: 11:02)

## Example: integer logarithm

- $\log_k n = y$ 
  - $k^y = n$ 
    - if we divide  $n$  by  $k$   $y$  times, we reach 1
  - Integer logarithm:  $\text{intlog}_k n = j$ 
    - Dividing  $n$  by  $k$   $j$  times gives a number  $\geq 1$
    - Dividing  $n$  by  $k$   $j+1$  times takes us below 1

So, we know what the logarithm is, logarithm tells us what power we need to get back to the number. So, the  $\log_k n = y$ , provided  $k^y = n$ . Another way of thinking about this, if we start with  $n$  and then we divide by  $k$  and then, we divide by  $k$  and so on, how many times can we do this until we reach 1. So, usually the logarithm is some fractional number.

So, let us do a simpler version which we call the integer logarithm. So, the Integer logarithm is a number such that, if we divide by that number, we stay above 1, if we divide by one more number we go below 1. So, this is the Integer part of the actual number.

(Refer Slide Time: 11:49)

Integer logarithm ...

Hand-drawn diagram:

$60 \xrightarrow{2} 30 \xrightarrow{2} 15 \xrightarrow{2}$   
↓  
 $1 \leftarrow 3 \leftarrow 7$   
 $5 \quad 4 \quad 3$

- $\text{intlog}_2 60 = 5$  because  $60/2^5 > 1$ ,  $60/2^6 < 1$
- Divide **n** by **k** until we reach 1, or go below

Base      Number

```
intlog :: Int -> Int -> Int
intlog k 1 = 0
intlog k n
| n >= k    = 1 + intlog k (div n k)
| otherwise = 0
```

So, as an example, if you take the base 2 and we take the number 60, then the  $\text{intlog}_2 60 = 5$ , because if we divide 60 by 2 for 5 times, then we stay above 1. 25 by the way equal to 32. So, 60 divided by 32 is more than 1, but 26 is 64. So, 60 divided by 26 will be less than 1. So, therefore by our definition, 5 is the largest number of times we can divide by 2 and stay above 1.

So, this gives us a strategy to compute the Integer logarithm,  $\text{intlog}$ . We start with  $n$ , and we keep dividing by the base  $k$  until we reach 1 or go below. If you reach 1 exactly, then we get the exact logarithm. So, for example, if we had done  $\text{intlog}_2 64$ , then we would have got 6 and this is exactly, because 64 divided by 26 is equal to 1. But, because we start with something which was off that we got something which went below 1, so either we reach 1 or we go below.

So, here is a function. Now we should be clear that the first argument is the base and the second argument is the number. So, the first argument to  $\text{intlog}_2 1$ , because of the way we write it as  $\text{intlog}_2$  base number, so,  $k$  is a base, 1 is a number. So, anything raised to 0 is 1. So, if we get the number 1 as input, then with respect to base  $k$ , the integer logarithm is going to be 0 by definition, otherwise provided we can divide it.

So, provided  $n$  is bigger than  $k$ , we go ahead and divide and then, we compute its logarithm. So, it is a recursive algorithm, we keep dividing it and notice that, we do not need to actually divide exactly. it's enough to do an integer division. So, we go from 60, in our case, we go to

30 over here, then we go to 15 and now, if we divide once more we will go to 7.5. But, it is enough go to 7 and then, go to 3 and then, go to 1 and then so on, so this is how it works. So, we went on 1, 2, 3, 4, 5 times and at 6th time, you will go below 1 and therefore, we will solve. So, this is the integer logarithm function, it just does this divide by n, dividing n by k, until we reach 1 or go below.

(Refer Slide Time: 14:13)

## Example: Reverse digits

- `intreverse 13276` should yield `67231`
- Strategy
  - Split `13276` as `1327` and `6` using `div` and `mod`
  - Recursively reverse `1327` to get `7231`
  - Multiply `6` by suitable power of 10 and add:  
 $60000 + 7231 = 67231$
  - Use `intlog` to determine the power of 10

So, for a change, let us look at a function which has nothing to do with numbers per se, it is more like something which manipulates sequences of characters. So, we want to reverse the digits of a number, of course we will do it numerically, but later on we will see how to treat these as list and do it in a different way. So, we want to write a function `intreverse`, which will take a multi digit number and return the digits in reverse.

So, if we give it `13276`, it should give us `67231`. So, how would we do this, so here is the strategy. So, if we take `13276` and we divide by 10 using integer division, then we get `1327`. And we take its remainder with respect to 10, we get `6`. So, using `div` and `mod`, we can split this number into the last digit and the rest. Now, using this style that we are used to in Haskell, we will do something inductive.

So, we will reverse the first part, this is the smaller number. So, here the induction is on the length of the number, earlier on induction you stay on the argument itself, we do `n` and then `n-1`, here the induction is on the length of the number. So, we will see that for things like list and sequences, the length is something we can write inductive definitions based on.

So, if we inductively assume we know how to reverse this number, then we will get the last part. So, this 7231 is now been reversed. we need to stick 6 in front of it. So, how do we get 6 in front of it, we have to shift 6 to the left which means we have to pad it with 0's. So, we have to multiply 6 by suitable power of 10 and add it. So, we have to take 6 and make it 60,000 and add it. And the question is how do we get the suitable power of 10. So it turns out that the suitable power of 10 is just the number of digits that we have here.

So, in this case, if we take the integer logarithm of this, it gives us 1 less than the number of digits, if we divide this by 10, 4 times, we get 6.7 something and if you divided 5 times, we get 0.67 something. So, if we take the integer logarithm, it is precisely 1 less than the number of digits in the decimal number and that is precisely the number of times we need to multiply 6. So, we can use a function we have defined in the last example in order to get this last step.

(Refer Slide Time: 16:43)

```

Reverse digits ...

intreverse :: Int -> Int
intreverse n
| n < 10 = n
| otherwise = (intreverse (div n 10)) +
              (mod n 10)*
              (power 10 (intlog 10 n))

power :: Int -> Int -> Int
power m 0 = 1
power m n = m * (power m (n-1))

```

plus repeated succ  
 mult " plus  
 power " mult

$$m^n = m \cdot m^{n-1}$$

So, here is the function. So, it says that, I want to reverse an integer, if the number is single digit number, if it is less than 10, I do nothing. Otherwise, I recursively reverse the first k-1 digits, then I take the last digit and multiply it by suitable power. Suitable power is the intlog to the base 10 of the original number and how do I define power, well we saw in the first few examples that plus is repeated successor, mult is repeated plus.

So, power or exponentiation is repeated multiplication, When we say  $2^7$  it means  $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$  to 7 times. So, just as exactly we wrote plus and mult, we can write a power function which says anything to the power 0 is 1 and if something to the power n, say  $m^n$ . This is equal to  $m \cdot m^{n-1}$ .

1. So, using this function and the intlog function that we wrote last time, we can compute the reverse of any set of digits.

(Refer Slide Time: 18:00)

## Summary

- Use `_` as a don't care wildcard pattern
- Use `otherwise` as a catch all conditional guard
- Several examples

So, what we have seen is a bunch of examples, but along the way we have also looked at some new syntax involving function definitions. One is this underscore as a don't care pattern, which allows us to specify arguments, which are not used in evaluating the function. So, it also gives us two options, one is not to use a variable where we do not need it, and second , it makes it more transparent that a function is independent of the particular argument.

The other thing is that in a conditional definition, we can use ‘otherwise’ as a catch all phrase at the end to make sure that any argument which does not match any of the previous conditions will definitely match this one. So, this ensures that, we do not have any pattern match failures.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

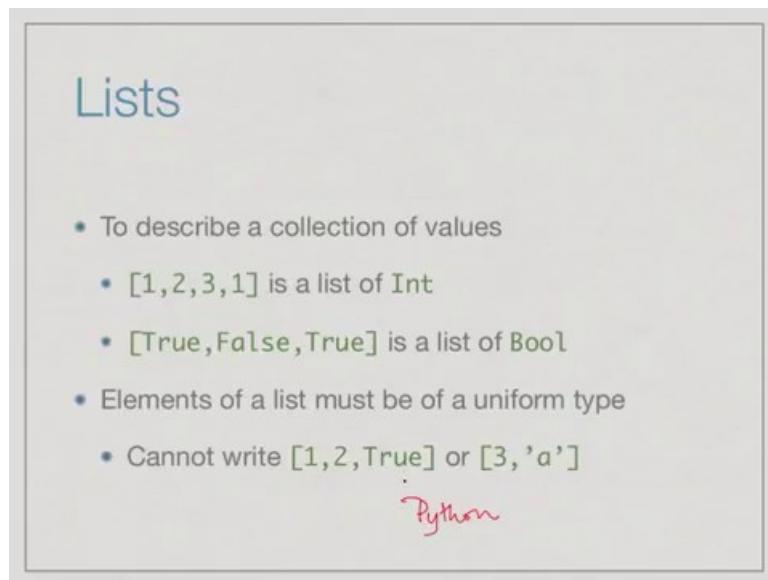
**Module # 02**

**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)



**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]]  
[Int] ↑ 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 :
  - $1:[2,3] \Rightarrow [1,2,3]$
  - 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  $\Rightarrow$  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 ( $x:xs$ )  $\Rightarrow x$
  - tail ( $x:xs$ )  $\Rightarrow xs$
  - Both undefined for []
  - head returns a value, tail returns a list

Diagram illustrating list decomposition. A list node is shown with 'x' above the colon and 'xs' below the colon. Red arrows point from 'head' to 'x' and from 'tail' to 'xs'. Below this, an example shows 'tail [3]' and '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  $\square$
  - 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 off 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

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

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

$\text{mylength} :: [\text{Int}] \rightarrow \text{Int}$   
 $\text{mylength } [] = 0$   
 $\text{mylength } (x:xs) = 1 + \text{mylength } xs$

$\text{mylength } l = 1 + \text{mylength } (\text{tail } l)$

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  $\text{mylength } l = 1 + \text{mylength}(\text{tail of } l)$ . 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  $\text{factorial } n - 1$  without brackets then Haskell will try to bracket it this way  $(\text{factorial } n) - 1$ , 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  $\text{mylength}$ . If I write  $\text{mylength } x:xs$  without this it will try to say it is  $(\text{mylength } x) : xs$ . 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  $\square$  is 0
- Sum of  $(x:xs)$  is  $x$  plus sum of ~~the~~  $xs$

```
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,  $\text{mysum } [] = 0$ . 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.  $\text{mysum } (x : xs) = x + \text{mysum } xs$ , 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**  $[x_0, x_1, \dots, x_{n-1}]$

- 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  $[x_0, x_1, \dots, x_{n-1}]$ , 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

$x_0 : (x_1 : \dots (x_j : (x_{j+1} : \dots (x_{n-1} : [])))$ ). 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  $x_0$  to  $x_1$  to  $x_j$  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*

- $[m..n] \Rightarrow [m, m+1, \dots, n]$
- Empty list if  $n < m$

*range(0,n)*

$$[0, \dots, n-1]$$

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

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  $[m, m+1, m+2, \dots, n]$ . 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  $\text{range}(0,n)$  then I will get the list that goes from  $[0, \dots, n-1]$ .

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  $[a, a + d, a + 2d, \dots]$  and so on and now of course, when I say  $[a, a + d, a + 2d, \dots]$  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 values 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.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

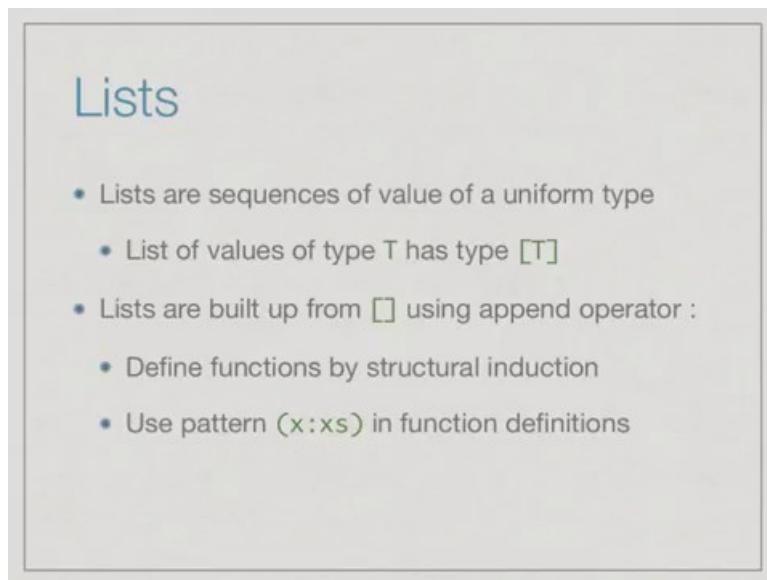
**Module # 02**

**Lecture - 02**

**Functions on Lists**

Last time we looked at 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 has a light gray background with a thin black border. The title 'Lists' is centered at the top in a blue font. Below the title is a bulleted list of six items, each starting with a blue bullet point:

- Lists are sequences of values 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)`

So, the built in append operator denoted by colon attaches a value to a list on its left. So, what if we want instead to append 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,  $\text{appendr } x [] = [x]$ .

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
  - $\text{attach } [3,2] \ [4,6,7] \Rightarrow [3,2,4,6,7]$
- Induction on the first argument

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

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  $[1, 2, 3] \rightarrow [3, 2, 1]$

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

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

The diagram illustrates the recursive steps for reversing the list [1, 2, 3]. It shows the list [1, 2, 3] being transformed into [3, 2, 1]. A green arrow points from the head '1' to the start of the tail '2'. A red arrow points from the end of the tail '3' back to the start of the tail '2'. A red curved arrow shows the movement of the tail '2' and '3' to become the new head '3, 2'.

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: is sorted  $[3, 4, 4]$   $3 \leq 4$   $[4, 4]?$

- 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  $[Int] \rightarrow \text{Bool}$ . 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.

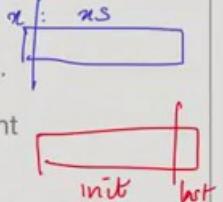
On the other hand, if I have two values then updown must start by going up. So, I want x and then y and I have wanted to go up, so I want  $x \leq y$  and now, as we observed once you go up the remaining part must go down. So, the next step must be down, so the list starting from y onwards will be going 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]  $\Rightarrow$  [1,2]
  - init [2]  $\Rightarrow$  []
- last l, returns the last element in l
  - last [1,2,3]  $\Rightarrow$  3
  - last [2]  $\Rightarrow$  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  $n$
- $l == (\text{take } n \text{ } l) ++ (\text{drop } n \text{ } l)$ , 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 take 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 myappend 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,  $(\text{take } n \text{ } l) ++ (\text{drop } n \text{ } l)$  is always equal to  $l$ , where  $l$  is the list.

(Refer Slide Time: 13:58)

## Built in functions on lists ...

$k \rightarrow (x_0, x_1, \dots, x_{n-1})$

- Defining take

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

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)

## Summary

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

*length : [Int] → Int*

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.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

**Module # 02**

**Lecture - 03**

**Characters and Strings**

So, let us turn our attention to a data type we have not really looked at very much so far, namely the data type of characters and associated with that the data type of strings.

(Refer Slide Time: 00:12)

The datatype Char

- Values are written with single quotes
  - 'a', '3', '%', '#', ...
- Character symbols stored in a table (e.g. ASCII) Unicode
  - Functions ord and chr connect characters and table
    - char -> num
    - num -> char
  - Inverses:  $c == \text{chr}(\text{ord } c)$ ,  $j == \text{ord}(\text{chr } j)$
  - Note: import Data.Char to use ord and char.

So, characters are the symbols that we can type from our keyboard for example. So, these are the type of items which are manipulated in software like word processors, so it is an important part of computational software that people write. So, the way that Haskell represents characters in a program is to put the symbol between single quotes, so you have 'a' or '%' and so on.

Now as all of you know a character is typically stored in some binary format in the memory. So, there will be a table, which tells us how to interpret a particular bit, so if you want to call this a character, then this represents a character Z for example. So, there is a table look up, which tells us how to treat a bit string, which is essentially an integer and convert it into a character.

So, there are various standards for this, so the old standard which is very common is called ASCII, a modern standard which allows more characters sets, for example, in non English and especially symbols in languages from say places like India or Asia, which is called a Unicode. So, Unicode is a two byte character representation, ASCII is a one byte thing, basically there is this table, so now, we need a way to go backwards and forwards.

So, you need to find out the code of a character and you need to find out, what the given code corresponds to. So, these are the two functions that Haskell provides for that, it is called `ord` and `chr`, so `ord` takes a character and gives us a number which is the code for the character and `chr` gives us the character for a given number, so these are inverses.

So, if I take a character take its number and then, convert to character back I will get the same character. Similarly, if I take a number, convert it into a character and then, extract its number I will get back the same number. So, if you want to use these functions in our Haskell code, they are not included by default, so you have to import a character module. So, you have to use this statement at the top of your .hs file, `import Data.Char`, if you want to use `ord` and `char`.

(Refer Slide Time: 02:45)

## Example: capitalize

- Function to convert lower case to upper case
- Brute force, enumerate all cases

```
capitalize :: Char -> Char
capitalize 'a' = 'A'
capitalize 'b' = 'B'
...
capitalize 'z' = 'Z'
capitalize ch = ch
```

26  
Inference

So, let us look at a function that I might want to write, so this is the function we mentioned as an example earlier on. So, supposing we want to write a function that converts all lower case letters to upper case letters and does not do anything for non letters. So, it should keep numbers as numbers plus punctuation marks as punctuation marks, but ‘a’ to ‘z’ will become ‘A’ to ‘Z’.

So, one way to do this without using any information about how characters are represented is just to exploit the fact that there are only finitely many characters. Say for instance, we are dealing with the English or what is more properly the Roman alphabet it has only 26 letters a to z, so then we can just write out 26 patterns. So, there are now 26 patterns capturing all the given inputs that need to be mapped and finally, we have something which will just copy the input to the output, if it does not match any of these patterns.

So, if I feed it a character which is not a to z, then it will just return. So, '%' will come back as '%' , '?' as '?', 9 as 9, but small a will become capital A. So, this of course, is a very brute-force thing and it relies on the fact that we can enumerate. Of course, this is a very tedious way to write such a program.

(Refer Slide Time: 04:03)

### Example: capitalize ...

- Can assume that 'a', ..., 'z' and 'A', ..., 'Z' and '0', ..., '9' each have consecutive `ord` values
- A more intelligent solution

```

capitalise :: Char -> Char
capitalise ch
| ('a' <= ch && ch <= 'z') =
  = (chr) (ord ch + (ord 'A' - ord 'a'))
| otherwise = ch
    offset
  
```

So, what we can do is to use the fact that in any of the encodings that Haskell systems will use, the letters are actually in blocks. So, these small letters 'a' to 'z' will all occur consequently in the table, so will 'A' to 'Z', so will 0 to 9. So, for instance if I have the overall set of encodings, then maybe I have one block, which is 'a' to 'z', maybe I have another block which is 'A' to 'Z'.

And now, the crucial thing is that the distance from 'a' to 'A' is the same as the distance from 'z' to 'Z'. So, if I look at the encodings they have the same distance apart, so that is an offset. So, what we can do is now we can write this function which says that because the encodings of characters can be compared '`a` < '`b`' < '`c`' and so on. So, if the given character

lies between ‘a’ and ‘z’, if  $\text{ch} \geq \text{a}$  and  $\text{ch} \leq \text{z}$ , then we add to its code the offset.

So, we add to the code of the given character, the offset which will shift it from this block to this block by an uniform amount, wherever it is, it will shift by the same amount and then, we could recompute the character back and we get the ‘A’. So, ‘a’ will go to ‘A’ and then, we recompute using the `chr` function with this. So, this is a way of capitalizing using `chr`, `ord` and the additional property that the `ord` numbering for characters are actually consequent.

(Refer Slide Time: 05:44)

## Strings

- A string is a sequence of characters
- In Haskell, `String` is a synonym for `[Char]`
  - $['h', 'e', 'l', 'l', 'o'] == \text{Hello}$
  - $"" == \text{True}$
- Usual list functions can be used on `String`
  - `length`, `reverse`, ...

So, usually programs that manipulate characters do not manipulate individual characters, but manipulate strings, so `string` is just a sequence of characters. So, when we have a text processor or something like that, you need a line of input, which will be a sequence of characters or even many lines of input and it will do something. So, in Haskell the word `String` with the capital S is a data type, but it is only a synonym, it is exactly the same in every context as `list of char` or `[Char]`.

So, in other words I can write strings using the familiar double quote notation, so I can write a sequence of symbols with double quotes. So, this is the string as you would see it in any other programming language. But, internally as far as Haskell is concerned, this notation is identical to having a list with five elements each of which is an individual character, so this is basically how Haskell works.

So, therefore, the empty string, which is just two consecutive double quotes “” is the same as

the empty list, if I say double quote, “” == [], the answer will be True. So, this is just an alternative notation what sometimes programming language people called syntactic sugar. So, this is just the different way of writing the same thing, which is more convenient for the program.

Now, the useful thing is that, because these are all lists, everything that one can do with list directly applies to strings. We do not need separate functions for strings, because we can take length of the string, because length of the string is just the length of the underlying list. We can reverse the string, because we are just reversing the underlying list and so on. So, all the things that we saw for list like head, tail, take, drop all these functions work as well on strings as they do want regular lists.

(Refer Slide Time: 07:37)

## Example: occurs

- Search for a character in a string
- `occurs c s` returns True if c is found in s

```
occurs :: Char -> String -> Bool
occurs c "" = False
occurs c (x:xs)
| c == x = True ✓
| otherwise = occurs c xs
```

So, similarly if one wants to write a function on a string, one would think of it as a list and use induction on the string and then, talk about the function on the empty string as the base case and what to do if you have a string extended with a one letter to the left. So, here is a simple function, we want to check, if a given character occurs in a given string. So, occurs is the function, which takes as input a character, a string to search for a character and returns whether or not the character is there.

So, the base case says that if I have a character and I search in the empty string, then it is not there, so the answer must be false. And now, if I search in the non empty string , then I compare it with the first character, so if it is the first character, then I found it, if it is not the

first character I must search in the rest. So, I continue my search by looking for occurs c xs. So, we are doing the familiar induction that we did for list on the string, if we are doing the base case for the empty string and the inductive case for the non empty string. So, its exactly like a programming a list.

(Refer Slide Time: 08:47)

## Example: touppercase

- Convert an entire string to uppercase  $[c_0, c_1, \dots, c_{n-1}]$
- Apply `capitalize` to each character  $[c'_0, c'_1, \dots, c'_{n-1}]$

```

touppercase :: String -> String
touppercase "" = ""
touppercase (c:cs) = (capitalize c):
                     (touppercase cs)
  
```

- Apply  $f$  to each element in a list—will come back to this later

So, here is another function, supposing we want to take a string and convert every lower case letter in that string to upper case. Now, we know how to convert one letter, because we have already written this function `capitalize`. So, again we can apply induction we say that, if you want to convert the empty string to upper case, then nothing happens, we just get back the empty string. And now, if we have a non empty string, what we do is we change the first letter to upper case and then, inductively do this. We capitalized the first letter and then, we applied `touppercase` to the rest.

So, this is actually a situation, where I have, say a string, which consists of n characters. And each of these I am going to now apply `capitalize` to get a new set of characters, which if they are not capitalizable, so the punctuation marks, numbers will be the same, otherwise they will be in capital. So, what we are doing abstractly is, if we are taking a function f, this case the function f is `capitalize` and we are applying it uniformly to a list, so that we get a new list of the same length where I have instead of  $c_0 f(c_0)$  which is  $c'_0$  and instead of  $c_1$  we have  $f(c_1)$  and so on.

Now, this is a very important operation, which we need to understand for list when we will

definitely comeback to this later. But, this is a specific example of it where we have just inductively defined how to capitalize a string.

(Refer Slide Time: 10:15)

$0 \dots n-1 \underline{n}$   
 $\underline{\text{length} = n}$

### Example: position

- `position c s`: first position in `s` where `c` occurs
- Return `length s` if no occurrence of `c` in `s`
  - `position 'a' "battle axe"`  $\Rightarrow 1$   
0 1 2      battle axe
  - `position 'd' "battle axe"`  $\Rightarrow 10$   
0      battle axe  
10

`position :: Char -> String -> Int`  
`position c "" = 0`  
`position c (d:ds)`

```

| c == d    = 0
| otherwise = 1 + (position c ds)

```

a battle axe  
+ a battle axe  
1 + 0 = 1

So, moving on to another example, supposing, we want to find the first position of a character in a string, remember that in any list the positions of a list of length  $n$  are from 0 to  $n-1$ . So, if I take the length as  $n$ , then the valid positions are 0 to  $n-1$ , so any position, which is in outside this range is an invalid position. So, we could for example, treat  $n$  as the default answer, if the string is not present.

So, either it should give us the first position in `s`, where `c` occurs or if there is no occurrence of `c` in `s`, it gives us the default value, which is the length, which is outside, so this is beyond the last position. So, for example, if I take this string ‘battle axe’, then there are two occurrences of ‘a’, here and here but because of our numbering scheme this is 0 1 2 so on. So, the number we should return is 1, because position 1 is the first occurrence.

On the other hand, the letter ‘d’ does not occur in ‘battle axe’. The length of this string is 10 the positions are numbered 0 to 9, so I return a value of 10 saying it is not found. So, how do we write positions is quite a straight forward inductive function again, so we take a character, take a string and return an integer. So, if it is not found, if it is an empty string, it is definitely not found. Remember that the length of the empty string is 0, so I should return 0 by our specification and this is correct.

Because, if it were a valid position 0, then we have actually one element in the string. So it has no elements in the string, so there were no valid positions, so 0 is the first invalid position. And on the other hand, if I take a non empty string as before I check, whether it is the first position. If it is first position, then with respect to this string it is 0, but if not founded, then I would have accumulated the positions I have skipped so far. So, if I do not find it, I add 1 to the position of the character in the remaining part.

So, if I find, for example in ‘battle axe’ I would first check ‘a’ with first character of ‘battle axe’ and I would say that this does not match. So, it will be 1 plus the position of ‘a’ in ‘attle axe’ and now it comes back with a 0, so I get 1 plus 0 is equal to 1, which is the correct answer. So the reason we are going through all these functions is we just get some practice in working with these inductive definitions on list strings, which are all basically different versions of the same.

(Refer Slide Time: 12:49)

## Example: Counting words

- wordc : count the number of words in a string
- Words separated by white space: ' ', '\t', '\n'

```

whitespace :: Char -> Bool
whitespace ' ' = True
whitespace '\t' = True
whitespace '\n' = True
whitespace _ = False

```

wild card

So, as a final example using strings, let us look at something a little more complex. So, supposing, we want to count the number of words in the string. So we have to define what a word is. So, word by convention is a sequence of characters which does not have a blank space, so blank space could be either a real blank. So, this represents a blank character, this is a tab. So, you can press the tab button on your keyboard and you get will some number of spaces depending on how the tabs are set and this ‘\n’ is a new line.

So, this is the familiar encoding that many programming languages use. ‘\t’ stands for tab, ‘\

'n' stands for new line. So, these are whitespace characters, so we can first define a function, which classifies a single character as whitespace or not. So, all it does is for the three cases which we have defined whitespace we can, we might want to later on add cases, but right now we are only saying ' ', '\t', '\n' as whitespace, everything else is not whitespace and notice that we do not need this value here.

So, we have this wildcard pattern we mentioned last. If you do not require the input in the output, we can just ignore the name and just use this '\_' as a wildcard pattern. So, this says everything which is other than these three characters are not punctuations.

(Refer Slide Time: 14:12)

### Example: Counting words

- Count white space in a string?

```

wscount :: String -> Int
wscount "" = 0
wscount (C cs)
  | whitespace c = 1 + wscount cs
  | otherwise      = wscount cs

```

- Not enough!
- Consider "abc      d"

So, the first thing that we could do is assume that, if I have word 1 word 2 and so on. Then, they will each be separated by white space, so I would have in this case, two white spaces, so if I can count the number of white spaces, if I add 1 I should get the number of words. So, the first attempt would be to just count the white space, so if I count the white space in the empty string it is 0, if I count the white space in a non empty string, then depending on whether the first character is a white space or not I add 1 or I just look at the rest.

So, if it is the first character, if c is a white space, then I add 1 and I continue inductively to count the remaining white spaces, otherwise I do not count this, because this is not a white space and I just recount ((Refer Time: 15:01)). So, this is a very straight forward way to count the white spaces, but unfortunately this would give us the wrong answer. Because, when a function does a thing like this, where I have may be four blanks between two words as in "abc d", we could say that there are 4 white spaces and therefore, this constitutes 4 or

5 words when actually there are only two words. So, actually, what is important is not the number of white spaces, but how many times one goes from a word outside and back.

(Refer Slide Time: 15:27)

### Example: Counting words



- Keep track of whether we are **inside** a word or **outside** a word
  - **Outside a word:** ignore whitespace, but non-whitespace starts a new word
  - **Inside a word:** ignore non-whitespace, but whitespace ends current word
  - Count the number of times a new word starts

So, one could write a program to check whether we are inside the word or outside a word. If we are outside a word, so we have a word and then, we have another word, so when we are outside a word there maybe many white spaces, but we do not care. Now, when we hit a non white space we transfer to inside a word, now inside a word we will ignore non white space, but when we hit a white space we will leave the word.

So, we can use this idea to count words, so actually it is enough to either check, how many times we enter a word or how many times we leave a word, So, in this case we will count the number of times a new word starts.

(Refer Slide Time: 16:12)

The slide is titled "Example: Counting words". It contains handwritten notes and code:

- Annotations:
  - "`list == []` char `c` / = [c] variable" with arrows pointing to the first character of a string and its representation as a list.
  - "`list == []` char `c` / = [c] variable" with arrows pointing to the first character of a string and its representation as a list.
  - "`list == []` char `c` / = [c] variable" with arrows pointing to the first character of a string and its representation as a list.
- Code:

```
wordcaux :: String -> Int
wordcaux "" = 0
wordcaux (c:ds)
| whitespace c) && not (whitespace d) =
  0 + wordcaux (d:ds)
| otherwise = wordcaux (d:ds)

wordc :: String -> Int
wordc s = wordcaux (':s')
```

So, new words start whenever you go from a white space, so from a space to say the character c or from a tab to the character x. So, whenever we transfer from one position which have a white space to a position that does not have a white space then by definition a new word starts. So, of course, now if I have a sentence like ‘the bat’, then I must make sure that I count this as the beginning of a word, so I must say that I go from a non white space to white space, but I start directly with the letter t.

So, what I will do is I will always take whatever string I have and insert a space before, so the very first word gets counted correctly. So, the word count, so this should be wordcount, so the function wordcount, which I am going to write on a given string will call this auxiliary function by first adding a blank space before this string. So, now, this makes sure that the very first word is counted correctly, now we are fine, what this says is that, if I have just a single character, then I cannot make a transition.

So, I say there are no words, but if have more than one character see notice that I will always call word count aux ‘wordcaux’ with a non empty string, because even if this is empty this whole thing is the string containing one blank. So, I will never call this the wordcaux with an empty string, so I wont need a pattern match case for the empty string. So, if I have one character I will declare it has no words, because if I have only one character, then it cannot be a word, it cannot be the starting point of a word because I would have flagged it as a character before.

If I have two characters I check whether the current character is a white space and the next

character is not white space, this is exactly this transition, if so I add one to the word count, otherwise I continue counting words as before. One important thing here to notice is that when the two empty strings, we said that the “” == []. This is an equality. However, “c” is not in general equal to c, because when I write “c” this is the character c, where as in [c], c is a variable.

So, when I write a function definition like this I need that this c stands for any character, if instead by mistake I had written double quote, it would say that this pattern matches precisely when I have a one element character c, a single ‘c’. So, therefore, one should be careful in some situations you have to use the list notation even though you are dealing with strings.

(Refer Slide Time: 19:07)

## Summary

- `String` is a synonym for `[Char]`
- All list functions apply to `String`
- Use structural induction for `String` functions

So, to summarize we have looked at the character data type and, what we have observed is that a string, which is a usual unit in which one consumes and manipulate characters is just a sequence of characters. And hence, Haskell provides the type `String` as a synonym for a list of `Char` and because it is a list all the usual list functions can be applied to `String` like `length` or `reverse` or `take` or `drop` and so on.

And also, because these are just lists we can write functions that manipulate lists or manipulate strings using the structural induction, exactly the same way that we use structural induction for a list by looking at the base case which is the empty string and the inductive case which is a non empty string where we separate out first character and the rest.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

**Module # 02**

**Lecture - 04**

**Tuples**

(Refer Slide Time: 00:02)

**Tuples**

- Keep multiple types of data together
  - Student info: Name, ID, Date of birth  
("Om Puri", 1007, "18 Oct 1950")  
Annotations: 'name' under "Om Puri", 'id' under 1007, 'date' under "18 Oct 1950".
  - List of marks in a course  
[("Amitabh", 89), ("Naseeruddin", 92), ("Aamir", 45)]  
Annotations: 'name' under "Amitabh", 'marks' under 89, 'name' under "Naseeruddin", 'marks' under 92, 'name' under "Aamir", 'marks' under 45.

So, we have seen lists, which are sequences of uniform type and in particular, we also looked at Strings, which are the special case of lists, where the uniform type is character. But, very often we need to keep multiple types of data together as a unit. So, this is what in a language like C or C++ would be a Struct or it could be basically a unit of information.

For example, if you want to keep information about the student, you might want to record the name, maybe the roll number and date of birth together in one place. So, if you do that, then you would have a String and then, you have say an Int and maybe another String. Now, you cannot make this a list, because this is not of a uniform type. Now, you might want to take a group of such things and make it into a bigger item.

For instance, you might want to keep a list of marks for a number of students, where you keep the name and the marks together for a given student and then, we have a list of such items. So, Tuples are Haskell's way of doing this, so notice that, we have used here this round bracket, not the square bracket of this. So, we have the square bracket, which applies to list and then a round bracket which allows us to collect together values of different types.

(Refer Slide Time: 01:26)

## Tuples ..

- Tuple type  $(T_1, T_2, \dots, T_n)$  groups together multiple types
  - $(3, -21) :: (\text{Int}, \text{Int})$
  - $(13, \text{True}, 97) :: (\text{Int}, \text{Bool}, \text{Int})$
  - $([1, 2], 73) :: ([\underline{\text{Int}}], \underline{\text{Int}})$

So, Tuple basically takes some n types and groups it together as a single unit, where we use this round bracket and comma to separate the given parts of the unit and the type of the overall Tuple is inherited from the underlying types. So, if I have a pair of integers  $(3, -21)$  written in this way, then its type is the tuple or the pair in this case,  $(\text{Int}, \text{Int})$

Well, if I have three values like this  $(13, \text{True}, 97)$  with round brackets, then the first is an Int, second is a Bool, third is an Int, so its type is tuple of  $(\text{Int}, \text{Bool}, \text{Int})$ . Now, because these types need not be uniform, we could have different structures, for instance, we can have a list as the first component and integer as the second component. So, here we have a tuple  $([1, 2], 73)$  which consists of list of Int and Int.

(Refer Slide Time: 02:18)

## Pattern matching

- Use tuple structure for pattern matching
- Sum pairs of integers
  - $\text{sumpairs} :: (\text{Int}, \text{Int}) \rightarrow \text{Int}$   
 $\text{sumpairs } (x, y) = x + y$
- Sum pairs of integers in a list of pairs
  - $\text{sumpairlist} :: [(\text{Int}, \text{Int})] \rightarrow \text{Int}$   
 $\text{sumpairlist } [] = 0$   
 $\text{sumpairlist } (x, y) : \text{zs} = x + y + \underline{\text{sumpairlist } \text{zs}}$

So, because the syntax is very structured, there is only one way to decompose the tuple, which is to use the comma to separate it out, so we can directly use pattern matching to extract each component. So, if I want to take pairs of integers and add them up, then I can just directly say that the first element of the pair will come out as x and the second, so this is the pattern which matches the tuples.

So, when I give it as an input say (7,12), then 7 will get mapped to x, 12 will get mapped to y, because of the pattern matching and then, the answer will be 7+12, which is 19. Now, this pattern matching can be combined with other pattern matching like for lists. So, if you want to take the pair, a list of pairs and add them up across the entire list, so I want  $x_1+y_1$ ,  $x_2+y_2$  and so on.

Then, we use list induction to say that, if I have an empty list of course, the sum is 0, if I have a non empty list, then I have a first value and a second value. So, this is the usual list pattern, it says use colon to separate the z from the zs and the z itself is the pair. So, use x , y to split this as two values x , y and now, I just take x + y as the sum of the first pair and add whatever I get by doing the rest. So, tuples are particularly easy to manipulate in terms of programs, because it is very easy to split them using patterns to get the integer components.

(Refer Slide Time: 03:48)

### Example: Marks list

- List of pairs (Name,Marks) –  $[(\text{String}, \text{Int})]$
- Given a name, find the marks
 

```
lookup :: String -> [(String, Int)] -> Int
lookup p [] = -1
lookup p ((name, marks): ms)
| (p == name) = marks
| otherwise     = lookup p ms
```

So, for example, supposing as before we had this list of marks of students, where each pair consists of a name and the marks for that name, so we have a pair which consists of a String and an Int and we have a list of such pairs. So,  $\text{String} \rightarrow [(\text{String}, \text{Int})]$  is the type of our input and now you want to look up the name, the marks for a given student. So, we have given a

name which is the String and we have given this list of marks for the entire class [(String, Int)], and we want to return those marks that this particular student got.

So, we use our usual list induction, so if we have no names or we have not found this name in this list, then we have to return something. So, let us assume that everybody gets a positive mark, so as a default value, we can get a -1 saying that, it was not found. But, as if we still have marks to process, then we break up the marks list into the rest and the first element.

First element again because of the structure will make it up into the name and marks, we match the given argument p with name, if it matches we return the marks, if it does not match we skip this value and look up this. So, this is the familiar list induction and this is this double level pattern matching, one is to do the list pattern with the colon and then, the tuple pattern with the comma.

(Refer Slide Time: 05:12)

## Type aliases

- Tedious to keep writing [(String, Int)]
- Introduce a new name for this type

```
type Marklist = [(String, Int)]
```

- Then

```
lookup :: String -> Marklist -> Int
      ||  
      [Char]
```

So, Haskell gives us a little bit of flexibility in how to refer to these kinds of types. Very often, because we are grouping together these types as a unit, we want to maybe give it a name, which makes sense for that unit. So, we might want to say instead of writing this stuff as [(String, Int)] we might want to indicate in our program that this particular unit has some meaningful association for us.

So, maybe, we want to give it a name like a marklist. So, Haskell has this declaration called type, which says that the name marklist is a synonym for the name [(String, Int)]. So, this means, we can now change our earlier definitions. So, instead of writing here [(String, Int)] as we have done earlier, once we had this type definition in our program, you can use that,

instead of this type.

Now, this is just a synonym. In the same way, that String is the same as [Char], there is no difference, there is no difference between writing marklist and [(String, Int)]. But, this is just very useful for us to make our program more legible and also of course, to cut down the amount of steps we have to write and now, a nice thing is that if we change this slightly, then everywhere the type remains the same, provided of course, the functions are so updated to use the new type.

(Refer Slide Time: 06:29)

## Type aliases ...

- A type definition only creates an alias for a type
  - Both `Marklist` and `[(String, Int)]` are the same type
  - `String` is a type alias for `[Char]`

So, both marklist and [(String, Int)] are exactly the same type and the String is type alias of [Char], so this is not like it is creating a new type. So, it is only creating a new name for a complex type.

(Refer Slide Time: 06:44)

## Example: Point

```
* type Point2D = (Float,Float)
  distance :: Point2D -> Point2D -> Float
  distance (x1,y1) (x2,y2) =
    sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1))

* type Point3D = (Float,Float,Float)
  distance :: Point3D -> Point3D -> Float
  distance (x1,y1,z1) (x2,y2,z2) =
    sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) +
      (z2-z1)*(z2-z1))
```

So, here is another example, so supposing we want to represent points in two dimensional space by x, y coordinates, then a typical x, y coordinate will be of type (Float,Float), would be a pair of real numbers. We might want to encapsulate this (Float,Float) as a single name, and call it say 2D point, let us use Point2D to refer to this. Then, we can compute the usual Euclidean distance between (x<sub>1</sub>, y<sub>1</sub>) and (x<sub>2</sub>, y<sub>2</sub>) using Pythagoras formula, you take (x<sub>2</sub>-x<sub>1</sub>)<sup>2</sup> + (y<sub>2</sub> - y<sub>1</sub>)<sup>2</sup> and take its square root.

So, the distance between (x<sub>1</sub>, y<sub>1</sub>) and (x<sub>2</sub>, y<sub>2</sub>), so now, we know that the underlying type is Point2D. Point2D is itself a tuple. So, we can use this pattern matching to extract the coordinates without having to ask, what is the first coordinate, what is the second coordinate and then, use these values directly in the function. So, this is an example, now you could correspondingly expand this to a 3D point and have the three dimensional version of the distance formula, which includes the z coordinate and so on.

So, this is an example of how we could use Tuples to make your type of your program more meaningful. Now, you know that you are taking the distance between points, so it makes your program more legible.

(Refer Slide Time: 08:03)

## Type aliases are same type

- Suppose
  - $f :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Point2D}$
  - $g :: (\text{Float}, \text{Float}) \rightarrow (\text{Float}, \text{Float}) \rightarrow \text{Float}$
- Then  $\text{Point2D} \quad P_{\text{2D}}$ 
  - $g (f \underline{x1 \quad x2}) (f \underline{y1 \quad y2})$  is well typed
  - $f$  produces  $\text{Point2D}$  that is same as  $(\text{Float}, \text{Float})$

Now, to illustrate the fact that these are only type aliases, now suppose that we have a function  $f$ , which takes two Floats and produces a point. Now, remember that this is just by our earlier definition the same as  $(\text{Float}, \text{Float})$ , so we had this thing we said type  $\text{Point2D}$  equal to  $(\text{Float}, \text{Float})$ . So, this definition gives us an alias, now I have another function  $g$ , which takes  $(\text{Float}, \text{Float})$ ,  $(\text{Float}, \text{Float})$  and produce a Float.

So, the whole point is that this should be the same as  $\text{Point2D}$  and there is no difference we claim between  $\text{Point2D}$  and  $(\text{Float}, \text{Float})$  and indeed, if I take the value of  $f$  produced for some, so if I take  $f$  of  $x_1, x_2$  then this produces a  $\text{Point2D}$ . Again, if  $y_1$  and  $y_2$  produces a  $\text{Point2D}$ , now I can feed these to  $g$ , with  $g$  is expecting only  $(\text{Float}, \text{Float})$  and  $(\text{Float}, \text{Float})$ , it is not expecting  $\text{Point2D}$ . But, because  $\text{Point2D}$  and  $(\text{Float}, \text{Float})$  are exactly the same thing, Haskell will not complain about the type, it will say, this is well typed.

So, in every context, if I have a value of  $\text{Point2D}$ , it is exactly compatible with any expectation of a value of tuple  $(\text{Float}, \text{Float})$ . So, these are just different names for the same thing, they are not different things about.

(Refer Slide Time: 09:29)

## Local definitions

- Let us return to distance
- ```
type Point2D = (Float,Float)

distance :: Point2D -> Point2D -> Float
distance (x1,y1) (x2,y2) =
    sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1))
```
- Introduce `sqr` to simplify expressions

So, let us get back to this distance function. So, this particular expression is rather tedious. So, let us say that, we want to write a function which actually takes an argument and squares it. So, we can introduce the function `sqr`.

(Refer Slide Time: 09:51)

## Local definitions

- ```
sqr :: Float -> Float
sqr. x = x*x
```
- ```
type Point2D = (Float,Float)

distance :: Point2D -> Point2D -> Float
distance (x1,y1) (x2,y2) =
    sqrt(sqr (x2-x1) + sqr (y2-y1))
    
$$\sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}$$

```
- But now, auxiliary function `sqr` is globally available

So, we can say square of  $x$  takes a `Float` and produces a `Float` and it just returns  $x * x$ , then this is a much more legible way of writing the same function. So, it says take the square root of  $\sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}$ . So, this is much clearer that  $\sqrt{(x_2-x_1)^2}$  is  $(x_2-x_1)^2$ , than the long thing of the multiplication as in  $(x_2-x_1)*(x_2-x_1)$ .

But, now the only problem will this formulation is, that we have produced a function `sqr`,

which is used in distance, but now becomes available everywhere in our program, because I have defined it separately. So, what we would like is to localize this auxiliary function and push it, so that, this is available only within distance and does not affect the rest of the program, I can or cannot use, may or may not want to use sqr elsewhere. So, if I want to I can write it again, but it does not conflict with this definition of sqr.

(Refer Slide Time: 10:47)

## Local definitions

```
type Point2D = (Float,Float)
sqr?
distance :: Point2D -> Point2D -> Float
distance (x1,y1) (x2,y2) =
    sqrt(sqr (x2-x1) + sqr (y2-y1))
where
    sqr :: Float -> Float
    sqr x = x*x
• Definition of sqr is now local to distance
```

So, Haskell has this other key word, which we have not seen before called ‘where’. So, Haskell says, you can write  $\sqrt{\text{sqr}(x_2 - x_1) + \text{sqr}(y_2 - y_1)}$  and then, say ‘where’ sqr is a function given then. So, this is the localized definition, now sqr is available within distance, but it is not available outside. So, distance can make use of sqr and the function sqr is defined inside.

So, it is local, its scope is only within distance. Outside, if I said sqr, there is no sqr. So, this is an obvious advantage of having local definitions, which is I mean using this where command which allows you to localize the definition.

(Refer Slide Time: 11:35)

## Local definitions

- Another motivation

```
type Point2D = (Float,Float)

distance :: Point2D -> Point2D -> Float
distance (x1,y1) (x2,y2) =
    sqrt(xdiff*xdiff + ydiff*ydiff)
    where
        xdiff :: Float
        xdiff = x2 - x1
        ydiff :: Float
        ydiff = y2 - y1
```

A slightly more subtle point is that, we might want to avoid duplicating a complication. So, here is another use of ‘where’. So, we have not put square, we have gone back to the multiplied version, but instead of  $x_2 - x_1$ , we have written xdiff, instead of  $y_2 - y_1$  we have written ydiff and then, we have said  $xdiff = x_2 - x_1$ ,  $ydiff = y_2 - y_1$ . So, notice one other feature here which is that the values that came into the function are available inside the where.

So, I have  $x_2$  and  $x_1$  coming from the function call and they are available inside the ‘where’. So, I can refer to the arguments of the function inside the ‘where’, I can refer to the values defined in the ‘where’ in the function.

(Refer Slide Time: 12:24)

## Local definition

- $\underline{xdiff} \cdot \underline{xdiff}$  vs  $\underline{(x_2 - x_1)} \cdot \underline{(x_2 - x_1)}$
- With  $xdiff \cdot xdiff$ ,  $(x_2 - x_1)$  is only computed once
- In general, ensure that common subexpressions are evaluated only once

So, what does this give us, what it says is that, if I write `xdiff*xdiff` as opposed to `x2-x1` Here, Haskell will actually not recognize, there is no way Haskell will recognize this `x2-x1` is the same as this `x2-x1`, it does not try to optimize any calculation, so it would actually do it twice. Now, in a subtraction it might not matter but if this were a complicated function that had to be called, then if you use the ‘where’ and give it a new name and use the same expression with the name twice. It knows that it has to compute `xdiff`, but having you computed `xdiff` is the same `xdiff` here.

So, this is one way in which you can control the efficiency of the program by ensuring that the common expressions, which occur in multiple parts of your program are evaluated only once for a given set of arguments.

(Refer Slide Time: 13:12)

## Summary

- Tuples allow different types to come together in a single unit  $(v_1, v_2, v_3)$
- Pattern matching to extract individual components
- `type` statement creates type aliases
- `where` allows local definitions

So, to summarize what we have seen is that lists have sequences of uniform type and if you want to combine values which are of different types into single units then we use tuples. So, tuples are written using this notation of round brackets and commas and then, we can use pattern matching to get the individual components when we write function definitions to process tuple data.

Along the way, we have also seen two new Haskell constructs, `type` and `where`. ‘`type`’ allows us to create new names for whole types. So, this is a convenience for writing programs and for making programs more legible, ‘`where`’ allows local definitions and it also helps us to avoid wasteful recomputation expressions.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

**Module # 03**  
**Lecture - 01**  
**Computation as Rewriting**

In order to explain some of the unique features that are available in Haskell, we need to understand how Haskell computations, how Haskell programs are executed; in other words, how computation progresses in Haskell.

(Refer Slide Time: 00:13)

The slide has a light gray background with a white rectangular content area. At the top, the title 'Computation as rewriting' is written in a blue font. Below the title is a bulleted list of four items:

- Use definitions to simplify expressions till no further simplification is possible *Reduction*
- An “answer” is an expression that cannot be further simplified
- Built-in simplifications

Below the list are two examples of simplification:

3+5  $\Rightarrow$  8

True || False  $\Rightarrow$  True

In general, in functional programming languages like Haskell, computation is a mechanical process of rewriting definitions. So, we have definitions which allow us to replace expressions on the left hand side by expressions on the right hand side and the process of computation is just to keep on doing this, until no further reduction is possible. So, this process of replacing left hand side by right hand side is often also called reduction.

So, when no further reduction is possible we have potentially an answer. Now, of course, if this expression is of the type that we want, if it is an Int or Bool or something, then it is a valid answer, if it is of a form which is not in the type that you like then we will get an error saying that some definition is missing.

To start with Haskell has built in definitions for the built in types. So, when we have an

arithmetic expression like  $3 + 5$ , then Haskell knows that it can be simplified to 8. It is not important for us how these definitions are built into Haskell, but these are obvious things that we all know and this is what it means to say that these are available to us to start with. So, similarly Boolean expressions like  $\text{True} \parallel \text{False}$  would be rewritten as  $\text{True}$ .

(Refer Slide Time: 01:25)

## Computation as rewriting

- Simplifications based on user defined functions

```

power :: Int -> Int -> Int
power x 0 = 1      base  $x^0 = 1$ 
power x n = x * (power x (n-1))

$$x^n = x \cdot x^{n-1}$$


```

Now, in addition; obviously, we provide new definitions through our Haskell function definitions or our Haskell programs. So, here is a definition of the exponentiation function, so it says that anything raised to the power 0 is 1. So, this is the base case, this says  $x^0$  is always 1, then it says,  $x^n$ , if it is not zero, it is going to be  $x \cdot x^{(n-1)}$ . So, this is just a basic inductive definition of the exponentiation function.

(Refer Slide Time: 01:58)

## Computation as rewriting

$$3^2 = 9$$

- power 3 2
  - $\rightarrow 3 * (\text{power } 3 (2-1))$  user definition
  - $\rightarrow 3 * (\text{power } 3 1)$  built in simplification
  - $\rightarrow 3 * (3 * (\text{power } 3 (1-1)))$  user definition
  - $\rightarrow 3 * (3 * (\text{power } 3 0))$  built in simplification
  - $\rightarrow 3 * (3 * 1)$  user definition
  - $\rightarrow 3 * 3$  built in simplification
  - $\rightarrow 9$  built in simplification

Now, suppose Haskell is confronted with an expression of the form power 3 2 that is, we are supposed to compute  $3^2$ . So, at this stage the only thing that can be simplified is the definition of power itself. So, using the function that we have defined we get the definition that power  $x n$  is  $x * x^{(n-1)}$ . So, now, this  $n-1$ ,  $2-1$  is now an arithmetic expression that can be internally simplified. So, using the built in simplification we replace  $2-1$  by 1, so we get  $3 * (\text{power } 3 \ 1)$ .

Now, once again we cannot evaluate this multiplication, because the right hand side is not yet a value that can be multiplied by 3. So, we have to again apply the definition of power and expand power 3 1 as  $3 * (\text{power } 3 \ (1-1))$ . This is just a blind substitution, this is what you should remember , it just says that power  $x n$  should be replaced by  $x * (\text{power } x \ (n \text{ minus } 1))$ . So, this is just a plain substitution, it does not require any intelligence on the part of the system.

It just says if I see a pattern which looks like this I can replace it with a pattern that looks like this, where the corresponding values will be transported to the current one, corresponding places. So, now, again  $1-1$  is a built in expression, so we can make it 0, now fortunately power 3 0 has a simple form which is 1. So, using the user definition for power 3 0, I get to 1 and now I can start a plain arithmetic internally. So, I get  $3 * 1$  is 3, so I get  $3 * 3$  and then  $3 * 3$  is 9.

And so by this process of simplifying using a combination of the definition of power which we provide and the arithmetic expressions for subtraction and multiplication which are built in to Haskell, we reach the conclusion that  $3^2$  is 9. This is not because Haskell knows arithmetic or it knows anything about exponentiation, it is just that the way we have defined the computation rules, it ensures that the answer is meaningful.

(Refer Slide Time: 04:16)

## Order of evaluation

- $(8+3)*(5-3) \Rightarrow 11*(5-3) \Rightarrow 11*2 \Rightarrow 22$   
 $(8+3)*(5-3) \Rightarrow (8+3)*2 \Rightarrow 11*2 \Rightarrow 22$
- power  $(5+2)(4-4) \Rightarrow$  power 7  $(4-4)$   
 $\Rightarrow$  power 7 0 = 1
- power  $(5+2)(4-4) \Rightarrow$  power  $(5+2) 0 \Rightarrow 1$
- What would power  $(\text{div } 3 0)$  0 return?  
division by zero

Now, there may be situations where more than one thing is possible, in our previous example at every stage more or less we could do only one thing, maybe at this point we could do two things, where we could multiply 3, with the bracketing we can do one thing. We do  $3*1$  with the 3 but, we did not have brackets we could possibly do this in different orders. But, there will be situations where it is ambiguous, so here for instance we have an arithmetic expression which has two sub expressions which need to be evaluated.

So, I have  $(8 + 3) * (5 - 3)$ , so if I choose to go left to right, then I first replace  $(8 + 3)$  by 11 and then replace  $(5 - 3)$  by 2 and finally, we have something where we can multiply and get 22. But, we can of course, just as well do it the other way, we could start with  $(5 - 3)$  and make it 2 and then move to  $(8 + 3)$  and make it 11 and still we get the same answer. So, we are familiar with the fact in arithmetic that this will not matter, it does not matter with me, whether we simplify  $(8 + 3)$  first or  $(5 - 3)$  first.

So, either order will reach the same result, now the same could happen when we invoke a function like power. For instance, suppose we invoke power with the arguments  $(5 + 2)$  and  $(4 - 4)$ , then if we choose to start with  $(5 + 2)$  then this becomes power of 7  $(4 - 4)$ . Then, I replace  $(4 - 4)$  like 0 and then the power definition of 0 gives me 1, but if we do it the other way something interesting happens.

So, now, we start with  $(4 - 4)$  and replace it by 0 and at this stage now there are actually two definitions that we can use. Because, the definition of power for 0's, since power  $x 0$  is 1, in other words we could just as well have written power  $_ 0$  is 1 because the  $x$  plays no role

here, so the x is not used. So, we said in the beginning that if we have a pattern in which the variable that we are matching is not used in the answer, we can as well make it an underscore. So, even though we have said x, the value of x is not used.

So, in particular here Haskell can look at the definition of power and that the second argument is 0 and just ignore this argument and give 1. So, we have evaluated power in two ways, but the second way we have done something which is somewhat unexpected, which is that we have reached the final answer while not evaluating something in between. So, this is perhaps a curiosity, but what would be it, something like this, so we know now that power anything 0 evaluates to 1 without looking at anything.

So, what do we replace that anything by a division by 0, so logically if I write div 3 0 then I should get an error, but if I say power div 3 0 0 would Haskell expose that error or would it blindly apply the definition for power anything to 0 and give me a 1. So, we will come back in this very shortly in the same lecture.

(Refer Slide Time: 07:33)

## Lazy Evaluation

- \* Any Haskell expression is of the form  $f \ e$  where
  - \*  $f$  is the **outermost** function
  - \*  $e$  is the expression to which it is applied.
- \* In `head (2:reverse [1..5])`
  - \*  $f$  is `head`
  - \*  $e$  is `(2:reverse [1..5])`
- \* When  $f$  is a simple function name and not an expression, Haskell reduces  $f \ e$  using the definition of  $f$

So, Haskell therefore, has to make a choice about what order to evaluate expressions. So, we have seen that Haskell expressions are of the form  $f$  applied to  $e$  written as  $f \ e$ . So, we call  $f$  the outermost function and  $e$  the expression to which it is applied,  $e$  in turn could of course, contain more functions and so on, but they are inside. So, here is an example, so we have an expression which says take the head of the list, 2, append it to reverse of  $[1..5]$ . So, there is an inner function `reverse` and then, there is an inner operator colon, but the outer function in this is `head`.

So, in our terminology if we say f of e, this is head of something, so that something is e and f is head. Now, what Haskell does is it prefers to use a definition which applies to f rather than definitions to apply inside e. So, in particular here if I have a choice between expanding reverse [1..5] and taking head of the list, then I would first try to use the expansion for head. So, if Haskell goes outermost, it tries to use the outermost definition that it can use in the current expression to reduce.

(Refer Slide Time: 08:45)

## Lazy evaluation ...

- The argument is not evaluated if the function definition does not force it to be evaluated.  
 $\text{head } (x : xs) \rightsquigarrow x$   
 •  $\text{head } (2 : \text{reverse } [1..5]) \rightsquigarrow 2$
- Argument is evaluated if needed  
 •  $\text{last } (2 : \text{reverse } [1..5]) \rightsquigarrow$   
 $\text{last } (2 : \underline{[5,4,3,2,1]}) \rightsquigarrow 1$

So, in this case it turns out for example that you can take head of this list, now head(x:xs) gives me x we know this. So, this is a kind of built in rule in Haskell, so in this expression the head value is known, it is 2. So, without evaluating reverse you can extract the head, so this is what Haskell would do, it will try to get the answer with as little work as possible, if you like to think of it that way and hence this is sometimes called lazy evaluation.

So, the argument to a function in Haskell is not evaluated, if the definition of the function does not force it to be evaluated. So, I have no interest in knowing what the result to reverse[1..5] is, because it is not going to be part of my answer. So, head(2 : reverse[1..5]) just gives me 2. On the other hand if I want the last value of the list, then I do need to know what happens to the end. So, if I say last (2 : reverse[1..5]), then I must reverse this list and get 5, 4, 3, 2, 1.

So that, I can then conclude that last of that definition of list is 1. So, Haskell's lazy reductions starts from the outermost thing and if it finds, it has enough information to process the answer, it does not need to look at an argument, it will just ignore that argument.

(Refer Slide Time: 10:01)

## Lazy evaluation ...

- What would power (div 3 0) 0 return?

```
power :: Int -> Int -> Int
power x 0 = 1
power x n = x * (power x (n-1))
```
- First definition ignores value of x
- power (div 3 0) 0 returns 1

So, if we come back to the question about what happens to power when we give it a nonsensical argument in the first position, but 0 in the second position. So, Haskell's strategy would be to first look for a matching definition for power. So, it finds a matching definition of power, this matching definition does not require me to evaluate x, so it ignores x, so this would actually give me 1.

So, lets see how this works, so just to convince you that this is real, so here is a Haskell file which contains that exact same definition of power, it says power of x 0 is 1, power of x n is x times power of x n minus 1. So, now, if I run this, I say power say for example 3 2 we have seen before power 7 5 works correctly. Now, if I say power ( div 3 0 ) 0, I get 1 and this is not because div 3 0 is a sensible expression, because div 3 0 on its own gives me as I would expect a divide by 0 error. So, lazy rewriting allows me to compute some values even though the arguments are not fully defined.

(Refer Slide Time: 11:29)

Lazy evaluation ...

power (dw 30) 0

- If all simplifications are possible, order of evaluation does not matter, same answer
- One order may terminate, another may not
- Lazy evaluation expands arguments by "need"
  - Can terminate with an undefined sub-expression if that expression is not used

Fortunately in general this order of evaluation given in a functional expression does not matter, just as it does for arithmetic, it does not matter. If all the simplifications are possible which ever order we choose, then the order does not matter we get the same answer. But, as we have seen if we choose one order, for example in this power example if I choose the order in which I try to evaluate the argument before the function, then I will reach a situation where the computation does not terminate or it terminates to an error, whereas if I use the outermost things, it does terminate with sensible answer.

Now, this is not to say, I mean we are not claiming that arithmetically something like infinity to the power 0 should be 1. So, its just saying that as per our rules it is possible to conclude from this expression that the answer is 1, it does not mean that it is arithmetically a sensible answer. So, the point is that using outermost reduction in this case, you get some expression which terminates, whereas if you use innermost reduction that is you start from the arguments and move outwards, then you will get an situation where your computation either does not terminate or gives some error.

So, lazy evaluation is also called call by need, it uses the arguments only if they are needed and therefore, as we have seen it can terminate even though there are undefined sub expressions which are not used in the computation.

(Refer Slide Time: 12:56)

The slide has a light gray background with a dark gray border. At the top, the title "Infinite lists" is written in a blue font. Below the title is a list of Haskell code:

```
* infinite_list :: [Int]
infinite_list = inflistaux 0
  where
    inflistaux :: Int -> [Int]
    inflistaux n = n:(inflistaux (n+1))
```

Below the code, there are two handwritten annotations in red ink:

- 0:(inflistaux 1)
- 0 : 1 : (inflistaux 2)

So, why is this useful other than curiosities like power? Well, let us look at this definition, so here is a definition which creates an infinite list. So, infinite list, it says is of type [Int] and what it, how you get it is by calling an auxiliary function with an argument 0 and what does this function do, if it takes an n it sticks in front and then calls itself with n+1. So, if I call inflistaux with 0, this becomes 0 : (inflistaux 1) which in turn becomes 0 : 1 : ( inflistaux 2 ) and so on.

So, in other words this is going, the 0 will produce a 1, the 1 will produce a 2 and so on. So, this is going to produce a infinite list starting with 0. So, we can verify that, so here we have again the definition that we just wrote. So, we have inflist.hs, so if I put this into ghci and I ask for what inflist returns , indeed it does return infinite list, that is it just keeps going. So, Haskell is able to somehow generate this infinite list and show it to us because it generates it one at a time.

(Refer Slide Time: 14:33)

The slide has a title "Infinite lists". Below it is a list of definitions and their evaluations:

- \* infinite\_list :: [Int]  
infinite\_list = inflistaux 0  
where  
inflistaux :: Int -> [Int]  
inflistaux n = n:(inflistaux (n+1))
- \* infinite\_list => [0,1,2,3,4,5,6,7,8,9,10,12,...]
- \* head (infinite\_list) => head(0:inflistaux 1) => 0
- \* take 2 (infinite\_list) =>  
take 2 (0:inflistaux 1) =>  
0:(take 1 (inflistaux 0)) =>  
0:(take 1 (1:inflistaux 2)) => [0,1]

So, as we have seen infinite list the definition I have given above produces this infinite list and if we just run it in the interpreter and ask for the value of infinite list, we just get an unending stream of numbers. On the other hand, if we ask for some function on this list which does not use the entire stream, then we can get a sensible answer. for instance, if we ask for the head of this list then as we saw before what Haskell will do, it will try to apply a definition for head, but head will ask that I need some values in the list in order to compute the head.

So, then I will expand infinite list just enough, so infinite list will now be expanded as inflistaux 0 which will in turn become 0 : inflistaux 1 and at this point we have a value. So, head of this will come out as 0, we could also have a function which requires more of the infinite list, for instance suppose we wanted first two elements we say take two of this infinite list, then in order to get anywhere we first have to get the first element and then we take it out and then we have to recursively take one of the remaining.

So, again we have to get this one value out and then once we have got two values out then we can stop and say the take 2 of this list is 0,1. So, in other words by using lazy evaluation, we can take infinite list and productively use them in our code.

(Refer Slide Time: 15:58)

Infinite lists

[m..n]

- Range notation extends to infinite lists
  - $[m..] \Rightarrow [m, m+1, m+2, \dots]$
  - $[m, m+d..] \Rightarrow [m, m+d, m+2d, m+3d, \dots]$
- Sometimes infinite lists simplify function definition

So, in fact, a Haskell allows us to define infinite list directly using the range notation, if you remember we had this notation from  $m$  to  $n$ ,  $[m..n]$ . So, now, if we leave out the upper end then we get a list which does not have a terminating point. So,  $[m..]$  is just  $[m, m+1, m+2, \dots]$ . we can also write an infinite list with some arithmetic progression like thing. So,  $[m, m+d, m+2d, \dots]$  and so on, so you might ask why this is useful well it will turn out we will see later examples, where allowing infinite list in the definition of a function makes it simpler to argue about it.

Because, if we dont know in advance what is the upper bound of the numbers we are looking for then we can just provide an infinite list and let Haskell figure out how many of these numbers it needs.

(Refer Slide Time: 16:47)

## Summary

- In functional programming, computation is rewriting
- Haskell uses *lazy* evaluation — simplifies outermost expression first
- Lazy evaluation allows us to work with infinite lists

So to summarize, the process of computation in a functional programming language is rewriting or reduction and this applies rules from left to right to reduce expressions using the definition which are either built into the language or provided by the user. Now, this allows multiple orders of reduction, different sub expressions in the same expression may be available for reduction. So, Haskell uses what is called lazy evaluation, it simplifies the outermost expression first and one of the consequences of lazy evaluation is that we can now work with infinite list in Haskell.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

**Module # 03**  
**Lecture – 02**  
**Polymorphism and Higher Order Functions**

Let us take a closer look at types in Haskell.

(Refer Slide Time: 00:06)

Functions and types

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

myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++ [x]

myinit [x] = []
myinit (x:xs) = x:(myinit xs)

• None of these functions look into the elements of the list
    • Will work over lists of any type!
```

Early on we wrote some functions on list, which we defined as functions over list of integers. For instance, we wrote this function length, which inductively assigns 0 to the length of the empty list and adds 1 for each element in the list. We also wrote this similar inductive function to reverse a list. And finally, we wrote the reverse of the head function, which takes the last element; I mean the tail function, which takes the first segment of the list, leaving off the last element.

Now, all of these functions have one thing in common, which is that they do not actually bother about the values in the list, so it is really a structural property of the list. The length of the list really just depends on how many elements there are, when you reverse the list, we just take the elements in the list and reverse the order. Similarly, when you take the initial segment of a list, you just drop the last element.

So, in all these things, you can actually assume that similar things will work regardless of the type of element in the list. So, if you think of a list as a list of boxes in some sense and then if you need a value in the list, you need to open the box, so none of these functions actually need to open the boxes, they can just move the boxes around or select a few of the boxes or one of the boxes and so on.

(Refer Slide Time: 01:26)

## Polymorphism

- Functions that work across multiple types
- Use type variables to denote flexibility
  - $a, b, c$  are place holders for types
  - $[a]$  is a list of type  $a$

So, it would be nice if we could assign within the type system that Haskell uses for each function. The type which will allow a function to work across multiple input types like this. So, we would like reverse for instance to work on list of Int or list of Float or even in list of list. So, as long as the underlying value is of a uniform type, so in Haskell, we use type variables, usually  $a, b, c$ ; letters like these as place holders for type. So, if we say that a function takes as input type  $a$ , it means  $a$  can stand for any input. And if  $a$  stands for any input, then if you put square brackets around  $a$ ,  $[a]$ , then we denote a list of that type.

(Refer Slide Time: 02:12)

## Polymorphism ...

- Types for our list functions
  - `mylength :: [a] -> Int`
  - `myreverse :: [a] -> [a]`
  - `myinit :: [a] -> [a]`
  - `head :: [a] -> a`
- All `a`'s in a type definition must be instantiated in the same way

So, if you go back now to our list functions, what does my length do, mylength takes any list and produces the length, the length is a number, so the output type is Int, the input type is unconstrained, it can be a list of any type of a. So, we write the type as `[a] -> Int`, reverse on the other hand, takes a list and produces another list. So, once again it takes an arbitrary list, but it produces a list of the same type.

So, it is important that the a on the left and the a on the right are actually the same a, it means that, if I give it an input of `[Int]`, it will produce an output which is a `[Int]`. If I give it a list of float, it will produce a list of float as output, you cannot change the type, but the type itself is not fixed in advance. And finally, if you look at myinit, so what it does is it actually produces, so this is a small mistake here.

So, it takes again a list of a, `[a]` and produces the list of a, `[a]` and if you wrote a function like head for instance, then it would take a list of a and produce a single value of type a. So, the important thing to note is that, if I have a variable, then wherever it appears in the type it must be instantiated by the same type.

(Refer Slide Time: 03:38)

## Functions and operators

- \*  $+, -, /, \dots$  are operators — infix notation
  - \*  $3+5, 11-7, 8/9$
- \*  $\text{div}, \text{mod} \dots$  are functions — prefix notation
  - \*  $\text{div } 7\ 5, \text{mod } 11\ 3$
- \* Use operators as functions:  $(+), (-) \dots$ 
  - \*  $(+) 3\ 5, (-) 11\ 7, (/) 8\ 9$
- \* Use (binary) functions as operators:  $\text{'div'}, \text{'mod'}$ 
  - \*  $7 \text{'div'} 5, 11 \text{'mod'} 3$

So, let us digress a little bit about notation of operators and functions in Haskell. So, we have seen arithmetic operators like  $+$ ,  $-$ ,  $/$ , we also have you know Boolean operators like OR and AND, so we could have you know  $x$  and  $y$  and so on. So, these are operations which operate on two values, but we write them in this infix notation; that is we put the operation in between the arguments. So, we have two arguments and the operator comes in between, so the operator is said to be infix.

On the other hand, we have seen functions, which are predefined like  $\text{div}$  and  $\text{mod}$ , which are written as functions. In other words, you write the name of the function and then, you write the arguments. So, this is what is called prefix notation, because the function comes before the arguments. Now, sometimes you may want to switch between these two notations or two views.

So, in particular, we can take any binary operator which we have defined and make it into a function just by putting round brackets around it. So, the operator ' $+$ ' can be written as a function by writing  $(+)$ . So, this expression  $(+) 3, 5$  is exactly the same as  $3 + 5$  and so on. So, in this way, we can use this notation to convert operators to functions and we can also do the reverse, provided of course, the function takes two arguments, because otherwise it is not clear, what it means to be infix, infix means it is between two arguments.

So if we have binary functions like  $\text{div}$  or  $\text{mod}$ , then we can use this reverse quote ` , this is the back quote key, this is not the normal apostrophe but the back quote, which we will find

on your keyboard. So, if we enclose the name of the function within these reverse quotes, then we can use them as operators. So, if you want to say  $11 \bmod 5$  and not  $\bmod 11, 5$ ; then we write  $11 `mod 5$ , so here we have written  $7 `div 5, 11 `mod 3$  and so on.

(Refer Slide Time: 05:40)

## Functions and operators ...

- `plus :: Int -> Int -> Int`  
`plus m n = m + n`
  - `(plus m) :: Int -> Int` adds `m` to its argument
  - Likewise, `m + n` is the same as `(+) m n`
  - Hence `(+ m)`, like `(plus m)` adds `m` to its argument
    - `(+ 17) 7 = 24`

So, this is useful in the sense that we have seen, suppose we have written a function plus, which takes two integers and adds them. What we saw is that because of currying plus actually consumes the arguments one at a time. So, in an intermediate stage, when it has consumed the first argument, we have a new function which we can write call `plus m n`, which actually takes, whatever it takes as a second argument, namely the `n` and it will add `m` to it.

So, `m` is now fixed, so in the same way, we can think of this using the normal plus operator. So, `m + n` is the same as this `(+) m n`. So, this `(+)` is the function which exactly like the `plus`, we wrote above, which takes two arguments and if you take one of them, then it becomes the function `plus m`. So, `(+) m` adds `m` to its arguments.

So, you can try this out in the Haskell interpreter in `ghci`. So, if you take something like `(+ 17)` and you applied it to `7`, written as `(+ 17) 7`, then `(+17)` will add `7` to `17` and give you `24`. So, using the fact that we can convert operators to functions by using the round bracket, we can also get partially instantiated functions, where one of the arguments has been fixed and use these, when we manipulate functions in Haskell.

(Refer Slide Time: 07:07)

## Higher order functions

- Can pass functions as arguments
- $\text{apply } f x = f x$ 
  - Applies first argument to second argument
- What is the type of apply?
  - A generic function  $f$  has type  $f :: a \rightarrow b$
  - Argument  $x$  and output must be compatible with  $f$
- $\text{apply} :: (a \rightarrow b) \rightarrow a \rightarrow b$

Now, manipulating functions in Haskell takes us to this notion of a higher order function. So, in Haskell, there is no particular problem or specific thing that we need to do to pass a function as an argument to another function. So, here is the most obvious function, we could write like this  $\text{apply } f x = f x$ . So, we can take a function called  $\text{apply}$ , which takes two arguments, and applies the first one to second one.

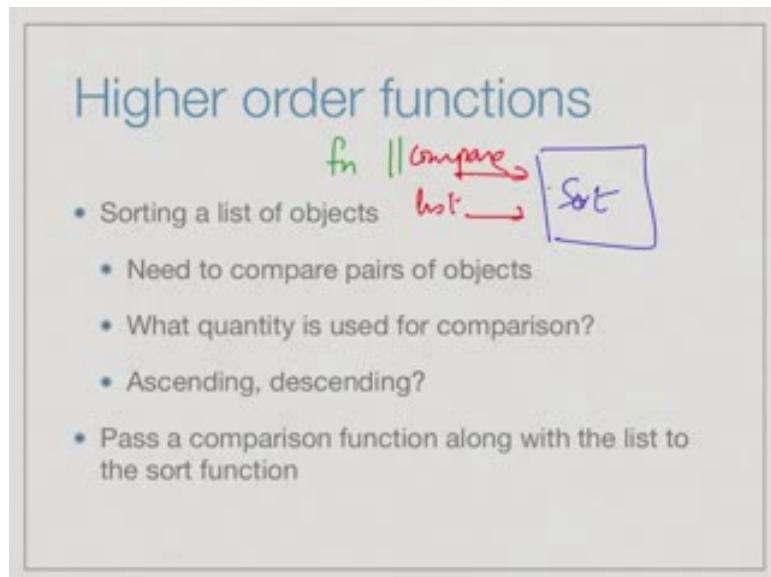
So, it really reads the function as its first argument, it reads the value as the second argument and applies the function it gets to the value it gets. So, it just says given  $f$  and given  $x$  return the value  $f$  of  $x$ . So, here for instance, if we say something like  $\text{apply plus } 17 \ 7$ ; then this should reduce by the application of  $\text{plus } 17$  to  $7$  to the value  $24$ , this is the intention.

So, what is the type of this function, well the first argument is the function and we wanted to be possible to send all possible functions to  $\text{apply}$ . So, if we make no assumption about the function, then it takes an input type and it takes an output type, which may be different. So, generic function  $f$  has type from  $a \rightarrow b$ , where  $a$  and  $b$  are possibly different types.

So, there is no constraint  $a$  and  $b$  may be the same type, may be different types, we do not constrain it in any way. Now, given that  $f$  takes an input of type  $a$  and produces an output of type  $b$ , this constrains the type of  $x$ , because  $x$  is now going to be passed to  $f$ . So,  $f$  takes  $x$ , so  $x$  must be an  $a$  and the result must be a  $b$ . So, the argument  $x$  and the output must be compatible with this choice.

So, this must be of type a and this must be of type b. So, this gives us the following type, so at this function, so this is not f but apply. So, apply takes as its first argument, some unknown function f and it takes as a second argument, a value to be passed to this function. And it produces an output, which is the application of x, f to x, which because f has type a to b, this is of type b. So, apply has type from a function a to b and an input of type a, produce an output of type b.

(Refer Slide Time: 09:33)



So, higher order functions will turn out to be very useful. So, here is a very familiar thing that you would have seen before. So, suppose we want to sort a list of objects, and the heart every sorting algorithm is a comparison between objects. So, we need to take pairs of objects and order them in the right way. So, in some situations, it might be important to know how to compare these objects, if we have pairs, then we use only one component, we use dictionary ordering and so on.

And also we might want to know in which direction to order, we want to order them in an ascending order or descending order and so on. So, to be fully flexible, what we would like is that, we have the sort function and it should take actually two inputs, it should take the compare function and it should take the list and then, use this compare function to sort this list, according to the sorting algorithm.

So, by changing the compare function for instance, we can make it go from ascending to descending and so on. So, this is a natural situation in which we pass a function to another function and we will see other examples of this.

(Refer Slide Time: 10:43)

## Summary

- Haskell functions can be polymorphic
  - Operate on values of more than one type
  - Notation to use operators as functions and vice versa
  - Higher order functions
  - Arguments can themselves be functions

So, to summarize Haskell functions typically those we have seen, which are structural functions on list, things like take the length or reverse it or take an initial segment or a final segment, can be polymorphic. In other words, they can operate on list of more than one type. We have also seen that we can use this round bracket and the backquote to convert operators to functions and vice versa.

And finally, we have seen that in Haskell, it is no big deal to pass a function to another function. So, we have higher order functions. So, to speak for free and this is very useful and we saw an example in the context of sorting and we will see more examples in the lectures to come.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund, S. P Suresh**  
**Chennai Mathematical Institute**

**Module # 03**  
**Lecture - 03**  
**Lists: Map and Filter**

So, we have seen that Haskell supports higher order functions, functions that take other functions as arguments and apply them to yet another argument. In the context of lists, map and filter are the two most important higher order functions that one can use.

(Refer Slide Time: 00:19)

So, to introduce map, let us look at two functions we have seen before. The first function is one that converts the entire string into upper case. So, this function is defined inductively using the base function capitalize, which converts a single letter to upper case. So, touppercase of the empty string is the empty string and if we have a non empty string, then we capitalize the first letter and then recursively upper case the list.

Another function which we have seen is one which squares all the elements of a list of integers. So, the square of an empty list is an empty list and if we have a non empty list, we square the first number and then recursively square the rest. So, in

both of these functions we are applying the given function to each element of the list. In the first case we are capitalizing every character in the string, in the second case we are squaring every number in the given list.

So, this is what the built in function map does. So, map in general takes a function f and applies it to every element of the list, so we have a list  $x_0$  to  $x_k$ , then  $x_0$  will be replaced by f of  $x_0$  and so on. So, we have this kind of function which takes each element and just applies the function one element at a time. So, this produces a new list of exactly the same length as the old is, in which each  $x_i$  is replaced by f of  $x_i$ , so it maps the function to the list.

(Refer Slide Time: 01:49)

**Examples** [[1,2,3], [2,1,0]]

↓      |      |

\*  $\text{map } (+ 3) [2, 6, 8] = [5, 9, 11]$  sum [2, 1, 0] = 3

\*  $\text{map } (* 2) [2, 6, 8] = [4, 12, 16]$  2+4+0=3

\* Given a list of lists, sum the lengths of inner lists

```
sumLength :: [[a]] -> Int  
sumLength [] = 0  
sumLength (x:xs) = length x + (sumLength xs)
```

\* Can be written using map as:

```
sumLength l = sum (map length l)
```

So, here is an example, remember we saw last time that if you take an operator like  $+$ , then you can make it a function by using the round bracket  $(+)$ . And therefore, if I have plus applied to n and m, then the function with one argument consumed plus m, actually adds m to every element that is applied. So, now, if you take the function  $(+ 3)$  it adds 3 to whatever it is applied to. So, if I map this onto this list, I get  $(+ 3) + 2$ , which is 5, 6 + 3 is 9 and 8 + 3 is 11.

So, this is one easy example of map, here is the similar one using multiplication. So, now, you have  $2 * 2$  is 4,  $6 * 2$  is 12,  $8 * 2$  is 16. So, here is a slightly more involved example, supposing we have a list of lists, so say we have a list consisting of say 1, 3 and then 5 and then we get an empty list. So, what we want to do is we want to compute the sum of the lengths of these lists, so the sum of this, the length of this

list is 2, the length of this list is 1, the length of this list is 0.

So, for this we want a function that will return 3, this  $2 + 1 + 0$ , so it is not the length of the whole list, but the length of the inner lists. So, we have a list of any type contain, the list of lists of any type and we want to return an integer and here is the usual inductive definition. We said that the sum of the lengths of the empty list of the lists is 0, because there are no lists inside and if I have a non empty collection of lists, then I compute the length of the first one and then inductively compute the rest.

So, this is the traditional inductive or recursive definition of this function. Now, on the other hand, what we can do is, we can use maps, we say take the list and apply length to each one of them, does this map. So, this takes for example in the list that we had seen earlier, so the effect of map is to replace this by 2, this by 1 and this by 0. So, now, we have this new list, which is after we map length of 1.

And then the built in function sum adds up all those things, so if I apply sum of these I will get  $2 + 1 + 0$ , which is 3. So, sumLength can now be successively described in terms of map and sum, rather than writing out an inductive definition like this.

(Refer Slide Time: 04:20)

## The function `map`

- The function `map`
$$\text{map } f \ [ ] = []$$
$$\text{map } f \ (x:xs) = (f\ x):(map\ f\ xs)$$
- What is the type of `map`?
$$\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

So, the function map itself can be defined inductively pretty much the way we have been doing the specific cases. So, if you map f to the empty list, then we get the empty list, if you map f to a non empty list, then we apply f to the first element of the list and recursively apply f to all the other elements. So, what is the type of map?

So, map takes of course as input some list, so this is a list of some type a.

And now this is the function that operates on elements of this list, so this function must start with an input a, but it can produce an output of any type. So, it will be in general of type a  $\rightarrow$  b and the outcome of taking a list of a, and applying a function that goes from a  $\rightarrow$  b is, of course produce a list of b. So, if we ask a type of map it will say, give me a function from some a to some b, give me a list which is compatible with the input type of this function. Because, I want to apply to each element of the list and it will in turn produce a list, which is of the output type of a function.

(Refer Slide Time: 05:37)

## Selecting elements in a list

- Select all even numbers from a list

```
even_only :: [Int] -> [Int]
even_only [] = []
even_only (x:xs)
| is_even x = x:(even_only xs)
| otherwise = even_only xs
where
  is_even :: Int -> Bool
  is_even x = (mod x 2) = 0
```

So, map takes the function from a  $\rightarrow$  b, it takes a list of type a and it produces a list of type b, So map is one important higher order function. Now, let us look at another higher order function, which involves selecting elements in the list. So, here is the specific example, supposing we have a list of integers and we want to select from this list of integers those that are even. So, inductive definition would say, that if I have the empty list of integers, then I get the empty list of integers.

If I have a non empty list of integers, then I have to decide whether to include the first element or not. So, I have a function here which checks whether a number is even, it just checks that the remainder of x divided by 2 is 0, so that is definition of even. So, if x is even, then include x and continue extracting the even numbers in the rest otherwise x is also excluded and just go to the rest and this is again a traditional

recursive definition of this.

But, we would like to do this in general, so here it says you want to select all the even numbers. In general we may have some other property, which is True and False of some elements and pick out all those elements for which the property is True.

(Refer Slide Time: 06:48)

## Filtering a list

- filter selects all items from list l that satisfy property p

```
filter p [] = []
filter p (x:xs)
| (p x)   = x:(filter p xs)
| otherwise = filter p xs

filter :: (a -> Bool) -> [a] -> [a]
even_only l = filter is_even l
              :: Int -> Bool
```

So, this is called filtering, so in filtering we take a property p, so property p is a function that takes a type and maps it to a Boolean value. So, it takes say integers and decides whether they are even or not, take letters, characters and decide whether they are vowels or not for example. So, filter takes a property and a list and it extracts exactly those items in the list that satisfy property p. So, filter takes p and if I have to filter an empty list [] with the property p, I get nothing, because no elements are there to satisfy the property.

If I have a non empty list, then I check the property on the first element, if it is True I include the first element and continue, otherwise I exclude. So, in our previous case the property was is\_even, but this is a generalization. So, as we said a property is just something, which takes the type of the underlying list and decides whether or not each element satisfies the property.

So, it takes a function which maps the type a to Bool, it takes an input list and produces an output list, which is the subset or sub list of the input list that of the same type. It is not like map, which produces a new list, which is the result of transforming the elements by a function. So, in a map each element is transformed

by a function from  $a \rightarrow b$ , so you get a list of these, in filter you are not transforming anything, you are really selecting a sub list, so the output list and the input list have the same type.

Of course, it is possible that none of the elements in the list satisfy  $p$ , in which case the output list maybe empty, where it will be of the same type. So, if we go back to our function even\_only, then the property in this case is the function is\_even, if you wrote before. So, is\_even remember it takes integers and produces Booleans, just checks whether the remainder is 0 when divided by 2. So, if I filter a list given this function, then I will extract exactly the even numbers from the list, so this is a much more succinct way of writing the same function.

(Refer Slide Time: 08:59)

## Combining map and filter

- Extract all the vowels in the input and capitalize them
- filter extracts the vowels, map capitalizes them

```
cap_vow :: [Char] -> [Char]
cap_vow l = map touppercase (filter is_vowel l)

[is_vowel :: Char -> Char
is_vowel c = (c=='a') || (c=='e') ||
(c=='i') || (c=='o') ||
(c=='u'))
```

So, very often map and filter occur in conjunctions, so very often what we want to do is to take a given list, extract some elements from that list, which satisfy given property and then transform those elements to some new elements. So, here is an example supposing we want to extract all the vowels. So, this is the filter and then we want to capitalize these letters, so that is a map. So, filter extracts the vowels and map capitalizes them, so we have first a filter function called is\_vowel similar to is\_even, which just checks whether the character is an a e i o or u.

So, we first apply filter to the list with is\_vowel as the property, this will result in a list, which has exactly those characters, which are vowels in the original list. And then, we use our earlier function touppercase which capitalizes every element in a

string to apply to this list, which contains the vowels. So, filter followed by map is a very common form that we will find in many functions that we use.

(Refer Slide Time: 10:06)

## Combining `map` and `filter`

$[1, 2, 6, 9, 11, 14]$

$\begin{matrix} \cancel{1} & - & \cancel{6} & \cancel{9} & \cancel{11} & \cancel{14} \\ \downarrow & & & & & \\ 4 & 36 & 196 \end{matrix}$

- Squares of even numbers in a list

```
sqr_even :: [Int] -> [Int]
sqr_even l = map sqr (filter isEven l)
```

So here is another simple example of filter and maps. Supposing we want to square all the even numbers. So, we first filter the list by getting only the even numbers out of them, then we map the square function to this list. So, this will square each individual element of that list and give us the squares of all the even numbers, so if we have list [1, 2, 6, 9, 11, 14], then first filter will give us 2, 6 and 14, others are excluded. So, I get 2, 6 and 14 and then map will give me 4, 36 and 196.

(Refer Slide Time: 10:49)

## Summary

- `map` and `filter` are higher order functions on lists
- `map` applies a function to each element
- `filter` extracts elements that match a property
- `map` and `filter` are often combined to transform lists

So to summarize, map and filter are extremely useful higher order functions of list. So, maps take a function and a list and applies the function to each element of the list. Whereas, filter takes a property that evaluates to True or False, and extracts elements from the list that match the property, that is those elements for which the property is True and we often use these in combination, so to extract the sub list and then apply a function to that list.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S P Suresh**  
**Chennai Mathematical Institute**

**Module # 03**  
**Lecture – 04**  
**List comprehension**

We have seen two important higher order functions on lists, map and filter and we also saw that filter and map are often used together in order to produce interesting transformations on a list. So, a filter takes a list, applies a property and extracts those elements which satisfy the property and map takes a list and applies a function to each element of a list. So, by combining map and filter, you can select some items from a list and then, transform only those items. So, today, we will look at a nice notation for doing this, which is much more readable than just writing map and filter.

(Refer Slide Time: 00:37)

New lists from old

- "Set comprehension"
- Generates a new set **M** from a given set **L**
- Haskell allows this almost verbatim
- List comprehension, combines map and filter

$M = \{x^2 \mid x \in L, \text{even}(x)\}$

$[x*x \mid x < - l, \text{is\_even}(x)]$

Annotations: / = ! = → ← ∈

So, we start with basic set theoretic notation. So, we have often seen this kind of notation to describe a set. So, here it says, given a set L, take all elements from the L which satisfy some condition, in this case, they are even and square these elements. So, this is set of all  $x^2$ , such that  $x$  belongs to L and  $x$  is even, written as  $\{x^2 \mid x \in L, \text{even}(x)\}$ . So, effectively this takes a given set L and produces a new set M. So, it transforms a given set to a new set and in set theory, this notation is called set comprehension.

So, this is a technical term, defining sets in this way is defined to be using a technique called set comprehension and so this is just terminology from set theory. So, analogous to this, Haskell allows us to define lists using list comprehension. So, this notation is very similar to that, instead of a curly bracket {} we have a list square bracket [] and then, we have of course, the same vertical bar '|', which is the symbol you can type from the keyboard and now, for the element of, we use thing <- which resembles elements of.

So, remember we said that Haskell tries to use notation, which looks like what we use in real life. So, we use \= slash equal to for not equal to and now, we have already seen that, we use this -> to simulate the arrow for function. And now, we have <-, which is supposed to represent the element of operator. So, [x\*x | x <- L, is\_even(x)] says, take the elements in L, check if they are even. So, x <- L, is\_even(x) is a filter and then, if they are even apply x2, so this is a map.

(Refer Slide Time: 02:30)

## Examples

- Divisors of n

divisors 6  
 $[1, 2, 3, 4, 5, 6]$   
 $\checkmark \checkmark \checkmark \times \times \checkmark$

divisors n =  $[x \mid x \in [1..n],$   
 $\underline{\underline{(mod\ n\ x) == 0}}]$

So, here is an example. So, supposing we want to find the divisors of n, the divisors of n are those numbers that divide n without leaving any remainder. So, first of all, the first thing to note is the divisors of n must be in the range 1 to n. So, we take all elements in the range 1 to n and if the remainder, when n is divided by that number x is 0, so x divides n exactly, then we list it. So, this lists all the divisors of n.

So, if I take for example, divisors of say 6, then I will first generate 1, 2, 3, 4, 5, 6 are the possible candidates and then, applying this condition, it will say that 6 divided by 1 is okay, 6 divided by 2 is okay, 6 divided by 3 is okay, because  $6 \bmod 3 = 0$ , 6 divided by 4 is not okay,

6 divided by 5 is not okay, 6 divided by 6 will be okay. So, exactly the numbers 1, 2, 3 and 6 will survive.

(Refer Slide Time: 03:30)

## Examples

$\text{divisors } 7 = [1, 7]$

- Divisors of n  
 $\text{divisors } n = [x \mid x <- [1..n], (mod\ n\ x) == 0]$
- Primes below n  
 $\text{primes } n = [x \mid x <- [1..n], \underline{\text{divisors}\ x == [1,x]}]$   
 $\text{divisors } 1 = [1]$   
 $\text{divisors } 2 = [1, 2]$

Now, using this function divisors, we can classify whether a number is prime or not. So, primes below a given number n. So, a prime is a number, which is divisible by only itself and 1. So, if the number is prime, like a divisors of 7, you will get a list consisting of 1 and 7 and nothing else, because that is a definition of prime; that only two integers divide the number, the number itself and 1, it has no other non-trivial factors.

So, in order to check whether something is prime or not, we just check whether divisors of x is exactly the list [1,x] and then, we take all the numbers from 1 to n such that, this is true and we list them out, we get the primes below the number in the range 1 to n, 2 to n. Because, notice that if I say 1, so 1 is not a prime, divisors of 1 for the way we have defined it is going to be the list consisting of 1.

So, it will fail this test, because it is not [1,1], which is what this test requires, it requires divisors of x to be list [1,x]. So, in order for 1 to be a prime it would have to produce a list of divisors of the form [1,1] but our function divisors will not do that, it will only check 1 number and produce that number. So, divisors of 1, we just get the list containing the single element 1. So, we are okay that 1 does not come out as prime, 2 on the other hand will give as divisors [1,2] and so on.

(Refer Slide Time: 04:58)

## Examples ...

- Can use multiple generators
- Pairs of integers below 10
- Like nested loops, later generators move faster

*for each x in [1..10]  
for each y in [1..10]  
(x,y)*

$[(x,y) \mid x <- [1..10], y <- [1..10]]$ .

$[(1,1), (1,2), \dots, (1,10), (2,1), \dots, (2,10), \dots, (10,10)]$

So far we have seen examples where you use only one generator but we can use more than one generator. So, what  $[(x,y) \mid x <- [1..10], y <- [1..10]]$  says is, let x run through the list 1 to 10, let y run through the list 1 to 10 and construct the list of all pairs (x,y) that are generated by combining these values of x and y. So, this is saying something like for each x in the range 1 to 10, for each y in 1 to 10 produce (x,y).

So, therefore, if I fix a value x=1, then it will generate every possible y, then for x=2 we generate every possible y and so on. So, the later generators move fast. So, I have (1,1), (1,2), (1,3), (1,4) .. (1,10). So, x is fixed as 1, y will run through 1 to 10, then x will move to 2, again y will move from 1 to 10 and finally, we will get of course, (10,10). So, when we have multiple generators, the generators to the right are executed or they run through for each element of the generators to the left.

(Refer Slide Time: 06:10)

### Examples ...

$$x^2 + y^2 = z^2$$

- The set of Pythagorean triples below 100

```
[ $(x,y,z) \mid x <- [1..100],$   
 $y <- [1..100],$   
 $z <- [1..100],$   
 $x*x + y*y == z*z]$ 
```

$\boxed{[3, 4, 5]}$   
 $\boxed{[4, 3, 5]}$

So, here for example, this is a way to generate all pairs. So, remember Pythagoras's formula. So, if you want integers which satisfy the relation  $x^2 + y^2 = z^2$  up to a certain upper limit. Then, we can run x from say 1 to 100, y from 1 to 100, z from 1 to 100 and check that  $x*x + y*y == z*z$  and extract all of these.

So, this is using our list comprehension and multiple generators, we can generate all the Pythagorean integers. But If you see this, notice that  $[3, 4, 5]$  is a Pythagorean triple. So, it will be generated by this when we run, but then later on when x becomes 4, then we will also generate  $[4, 3, 5]$ . So, this actually generates duplicates.

(Refer Slide Time: 07:06)

### Examples ...

- The set of Pythagorean triples below 100

```
[ $(x,y,z) \mid x <- [1..100],$   
 $y <- [1..100],$   
 $z <- [1..100],$   
 $x*x + y*y == z*z]$ 
```

- Oops, that produces duplicates.

$\boxed{[3, 4, 5]}$   
 $\boxed{[4, 3, 5]}$

So, we get [3, 4, 5] and we also get [4, 3, 5]. So, supposing we do not want to list these separately, because these are essentially the same triple just in different order, then we can be a little more careful in our generator. So, generators can actually just as we said that for every x, y is generated, for every y, z is generated, the values that we generate for y can depend on the current value for x.

(Refer Slide Time: 07:33)

## Examples ...

- The set of Pythagorean triples below 100
 

```
[(x,y,z) | x <- [1..100],  
          y <- [1..100],  
          z <- [1..100],  
          x*x + y*y == z*z]
```
- Oops, that produces duplicates.
 

```
[(x,y,z) | x <- [1..100],  
          y <- [(x+1)..100],  
          z <- [(y+1)..100],  
          x*x + y*y == z*z]
```

$x \leq y < z$

$[3, 4, 5]$   
 $[4]$

What we can say is that, we always want to generate these in the order x strictly less than y strictly less than z ,  $x < y < z$  . So, x runs from 1 to 100, but for each value of x, I only check y is from  $x+1$  onwards and I only check Z's from  $y+1$  onwards. So, this gives us a set of triples without any duplicates, because now I will get [3, 4, 5], but once I set x to 4, I cannot generate 3, because y will start from 5.

(Refer Slide Time: 08:09)

## Examples ...

concat [ $\begin{bmatrix} 3, 7 \end{bmatrix}$ ,  $\begin{bmatrix} \end{bmatrix}$ ,  $\begin{bmatrix} 4, 6 \end{bmatrix}$ ]

• The built-in function concat

```
concat l = [x | y <- l, x <- y]
```

So, here is another function rewritten using list comprehension, remember the function concat, it dissolves brackets, if I said concat of say  $[3, 7]$ ,  $[]$  and  $[4, 6]$ , then what this does is it produces from this, a list  $[3, 7, 4, 6]$ . So, we are taking a list of lists ,and producing a single list by just collapsing all of them into a single list, and if is empty, it is just left out. So, this says okay, concat of l, for you take each element in l, so I take this, then I take this, then I take this, so y will be first this and y will be this, y will be this.

And now, I take each x and y, so x now is this, then this, then this, then this, these are the values that x takes and output all of these as 1's. So, it takes every y in the inner list, takes every element of that list and extracts it as in algorithm. So, therefore, this dissolves one level of brackets and behaves exactly like concat.

(Refer Slide Time: 09:13)

Examples ...

- Given a list of lists, extract all even length non-empty lists

So, let us just look at a couple of more, slightly more exotic examples. So, supposing we want to take a list of lists and extract all the even length non-empty lists. So, if I take something like this, so this is the list of lists. Now, [] has length 0, [3] has length 1, [6,8] has length 2, [5,7,9] has length 3, but I do not want this, I only want to extract 6, 8. So, I want to extract all the even length non-empty lists in a given list.

(Refer Slide Time: 09:54)

Examples ...

- Given a list of lists, extract all even length non-empty lists

```
even_list l = [ (x:xs) | (x:xs) <- l, (mod (length (x:xs)) 2) == 0 ]
```

So, the new thing here is that we want the filter to take the list of lists, check each element in the list and extract it provided its length is even. Remember, that length and anything be even is just checking, the remainder with respect to 2, we just want to take the length of the given list divided by 2 and check whether the answer is 0. But, one important feature here is we

want non-empty list, the way we can do non-empty list in one shot in this list comprehension is by providing a pattern here, which only matches non-empty list.

So, this tells do not take every element of l, only take those elements of l, which are of the form x:xs. So, in other words in our earlier example if we saw empty list as a given element in the list of lists, then this pattern wont match. So, we just skip it. So, for every non-empty list in l, check, if length is even and if so, then put out that list..

(Refer Slide Time: 10:58)

## Examples ...

- Given a list of lists, extract all even length non-empty lists

```
even_list l =  
  [ (x:xs) | (x:xs) <- l,  
            (mod (length (x:xs)) 2) == 0 ]
```

- Given a list of lists, extract the head of all the even length lists

```
head_of_even l =  
  [(x | (x:xs) <- l,  
     (mod (length (x:xs)) 2) == 0 )]
```

Now, given that we have this pattern, we also have this structure of the list. So, we can extract not just entire list, but any part of it. So supposing, we want to modify this slightly to say, we do not want all the entire non-empty list, we want the head of the even length list. So, this is again non-empty. So, this is simple enough, we just take exactly the same right hand side, we check first of all that is non-empty by putting the pattern x:xs in l, we use mod to check that length is even.

But, now we dont take entire list, we use this pattern to extract only the head of it. So, the message from this example is that, when we write a generator it is not just a simple variable, we can actually use a pattern and use that pattern to generate elements of l.

(Refer Slide Time: 11:55)

## Translating list comprehensions

- List comprehension can be rewritten using `map`, `filter` and `concat`
- A list comprehension has the form
$$[e \mid q_1, q_2, \dots, q_N]$$
where each  $q_j$  is either
  - a boolean condition  $b$  or
  - a generator  $p \leftarrow l$ , where  $p$  is a pattern and  $l$  is a list valued expression

So, list comprehensions are essentially just syntactic conveniences for us, it is not a fundamental concept, it is just way of writing map and filter in a more readable format, which is very similar to the set theory notation and easy to decode. Rather than nested map and filters and in fact, you can formally translate this comprehension using map, concat and a version of filtering.

So, list comprehension typically has this form, have an output expression, which is generated by a bunch of input conditions. So, each of these is either a Boolean condition  $b$  or it is a generator and in the generator we have patterns. So, we can either say  $p \leftarrow l$  or  $b$  and we apply condition  $b$ , it applies to the things which have been generate before.

(Refer Slide Time: 12:47)

## Translating ...

- A boolean condition acts as a filter.
$$[e \mid b, Q] = \text{if } b \text{ then } [e \mid Q] \text{ else } []$$
- Depends only on generators/qualifiers to its left

So, a boolean condition acts as a filter. So, if I have an expression form e, such that b followed by Q, then this is an expression which is allowed in Haskell, if you have not seen before. So, we can write.. this is an expression in Haskell, written using the conventional, if then else. So, ‘if b then [e | Q] else [ ]’ says if b is True then this is the output, otherwise this is the output. So, in other words for whatever list I am generating, if b holds then I continue to apply the remaining things, otherwise I just keep it.

So, for each element implicitly this applies to things which have been generated in the left. So, if the element from the left satisfies the condition, then I continue to process it using what remains, otherwise I drop it.

(Refer Slide Time: 13:33)

## Translating ...

- Generator  $p \leftarrow l$  produces a list of candidates
- Naive translation

$$[e \mid p \leftarrow l, Q] = \text{map } f l$$

where

$$f p = [e \mid Q]$$

$$\times f _- = \square$$

What about generators? So, generators produce lists of candidates. So, if I have a generator then I need to take each element of p and such that Q apply this e to it. So, I have mapped each element of p with this function. So, this is map f of l, where f of p is e, such that Q and if p is not matched, so this is because it is a pattern, then I do not do anything.

So, this is taking care of the fact this is the pattern. So, the pattern itself does not match, if this is not a pattern I would not have this case, if it just said elements I will just have f of x is, such that Q, but since I have a pattern, it says if the pattern matches then do something, if the pattern does not match, skip it. Now, this is a naïve translation and we will see why. So far we have not used concat, we have used map and we have used a kind of filtering efforts.

(Refer Slide Time: 14:30)

## Translating ...

- \*  $[n^2 \mid n <- [1..7], \text{mod } n \cdot 2 == 0]$   
⇒ map f [1..7]  
where  
 $f n = [n^2 \mid \text{mod } n \cdot 2 == 0]$

So, let us look at an example to see, why this goes off, so let us look at this simple example. So, supposing we want to square all the even numbers between 1 and 7, so we generate all the numbers from 1 to 7, check if they are even and then square it. Now the first thing that we have is a generator. So, we have to apply the map thing. So, it says map f to [1..7], where f of n is what remains; that is this and this. So, this is  $[e \mid Q]$  and now, we have to inductively recover this. So, this is a property. So, this will be replaced by if then else.

(Refer Slide Time: 15:10)

## Translating ...

$2^2, 4^2, 6^2$   
 $[4, 16, 36]$

- \*  $[n^2 \mid n <- [1..7], \text{mod } n \cdot 2 == 0]$   
⇒ map f [1..7]  
where  
 $f n = [n^2 \mid \text{mod } n \cdot 2 == 0]$
- ⇒ map f [1..7]  
where  
 $f n = \text{if } (\text{mod } n \cdot 2 == 0) \text{ then } [n^2] \text{ else } []$
- ⇒  $[], [4], [], [16], [], [36], []$

So, it says map f to the list 1 to 7, where f of n is, if n is even, then  $n^2$ , the list containing  $n^2$  else the empty list and now if I apply this to each element, then for 1 we get the empty list, for 2 we get the list containing 4 and so on. So, you will notice that, what you would expect

from this list is, we will expect 22 , 42 , 62.

So, in other words we will expect the list 4, 16, 36 to be the output of this list comprehension, but we actually get this complicated expression with the lot of spurious brackets. And this is precisely why we need the concat, we need to eliminate these brackets.

(Refer Slide Time: 15:52)

## Translating ...

- Need an extra concat when translating  $p \leftarrow l$
- Correct translation

```
[e | p <- l, Q] = concat map f l
where
f p = [e | Q]
f _ = []
```

So, the correct translation of the generator is to insert a concat. So, we do not just map f to 1, we take the resulting output and we dissolve one level of brackets. So, the result of removing a generator from this expression is to concat the result of mapping f to the list.

(Refer Slide Time: 16:14)

## Translating ...

- $[n^2 | n <- [1..7], \text{mod } n 2 == 0]$ 
  - ⇒ concat map f [1..7]
    - where
    - $f n = [n^2 | \text{mod } n 2 == 0]$
  - ⇒ concat map f [1..7]
    - where
    - $f n = \text{if } (\text{mod } n 2 == 0) \text{ then } [n^2] \text{ else } []$
  - ⇒ concat [1, 4, [], 16, 36, []]
  - ⇒ [4, 16, 36]

And now, if we do this everything works out fine for that example and you can check that it

works in general. So, we take the same list  $n$  squared, such that,  $n$  is from 1 to 7 and  $n$  is even. So, after one expansion we have this map, but now we have this concat in front of it. So, after second expression, we have this if, we still have this concat in front of it. So, now, we get concat of this earlier expression that we had and now the concat dissolves this brackets and removes the empty list. So, I just get [4, 16, 36], which is the expected output.

(Refer Slide Time: 16:49)

## The Sieve of Eratosthenes

- Start with the (infinite) list [2,3,4,...]
- Enumerate the left most element as next prime
- Remove all its multiples from the list

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...

So, let us now look at the example that we had in the introduction to, the introductory video to the course. So, this is the sieve of Eratosthenes. So, what does the sieve of Eratosthenes do, it generates all the possible primes. So, the strategy is the following. you start with the list of all numbers beginning with 2, because a first prime is 2. So, the left most number in the list at any point is the next prime, once we enumerate a prime, we remove all its multiples from the list. because they are no longer candidates to be primes.

So, let us just see how it works. So, supposing we start with this infinite list of which we have written a finite prefix. So, we have up to 20. So, now, the first number in this list is the first prime namely 2. So, we mark 2 as a prime and now, we must remove all its multiples from the list. So, the first multiple of 2 is 4, the next is 6, 8 and so on. So, we go through this infinite list knocking off all those multiples. So, this gives us a resulting list in which the first number that is left is 3. So, 3 is now a prime.

So, we knock off its multiples. So, 6 is already knocked off, you knock off 9, 12 is already knocked off, you knock off 15, 18 is already knocked off, but we knock off 21 and so on. After knocking off multiples of 3, we have also got rid of 9 and 15 and of course, a lot of

numbers to the right which we do not see. So, now, the next prime is a 5 and then, we will knock off 10, 15, 25, 20, 25 and so on. So, this is the process by which we generate the primes.

So, of course, in this we have several infinite processes. First of all we have to start with an infinite list and every time we pick out the first element, we have to remove all its multiples. So, that is again an infinite process because we have go through this infinite list and knock off all the multiples.

(Refer Slide Time: 18:48)

## The Sieve of Eratosthenes

- In Haskell,

```
primes = sieve [2..]
where
sieve (x:xs) = x (sieve [y | y <- xs, mod y x > 0])
```

[2,3,4.. ]

Nevertheless, as we claimed in the introductory video, we can write this using this intuitive notation, it says apply the sieve with the list infinite list 2 onwards. Remember, by lazy notation this is the list 2, 3, 4 and so on, because the lazy evaluation, this makes sense and result of applying sieve to a list is to extract the first element as a prime and then, remove all multiples of that element from the list.

So, if I take all this, a tail of the list for every y, I keep it only if does not get divided evenly by x and then, I recursively apply sieve to that. So, this is succinctly describing the sieve algorithm and says take the list 2 onwards and apply sieve to it where sieve extracts the first element, removes all this multiples from the tail and apply sieve recursively to that tail. So, if we look at the way that this evaluates it will actually make sense.

(Refer Slide Time: 19:47)

The Sieve of Eratosthenes

```
primes => sieve [2..]
=> 2:(sieve [y | y <- [3..], mod y 2 > 0])
=> 2:(sieve (3:[y | y <- [4..], mod y 2 > 0]))
=> 2:(3:(sieve [z |
  z <- (sieve [y | y <- [4..], mod y 2 > 0]) |
  mod z 3 > 0]))
=> 2:(3:(5:(sieve [w |
  w <- (sieve [z |
    z <- (sieve [y | y <- [4..], mod y 2 > 0]) |
    mod z 3 > 0]) |
    mod w 5 > 0])))
=> ...
```

So, we say that set of primes is a result of applying the sieve to 2 onwards. So, sieve [2..] says take out the first element and apply sieve to [3..], such that elements don't divide by 2. Now, this in turn will say extract the first element of this, so first element of this happens to be 3. So, it will say apply sieve to 3 and the rest, so, this is just saying that, if I expand this inner list of this comprehension, then this produces 3 followed by [4..] in the same property.

Having done this, I have got the first element, so sieve will extract it out and then, it will say apply sieve to the result of this original list, which we already, this is the list that is currently running. So, we take every element in that list and apply another condition. So, once again if we do this to nested list comprehension the first element that comes out will be 5. And so now, after we extract the 5, then it will say ok take the inner list that we are already working with, the two list comprehensions and apply a third one.

So, this is the way that the sieve function gets rewritten and as it is rewritten, we get more and more primes. So, do write this out for yourself in ghc and verify that, it does generate primes. So, this is not necessarily the most efficient way to generate primes. In fact, it is not the most efficient way to generate primes. But, it is certainly an interesting exercise, it says that a very direct implementation is possible, because of the combination of this list comprehension notation and lazy evaluation.

Of course, lazy evaluation is crucial otherwise we cannot go with infinite list at all and using lazy evaluation and using list comprehension, we can write a very succinct 2 or 3 line implementation of a very basic algorithm in such a way that, it is immediately obvious what

is going on and it does the expected.

(Refer Slide Time: 21:57)

## Summary

- List comprehension is a succinct, readable notation for combining `map` and `filter`
- Can translate list comprehension in terms of `concat`, `map`, `filter`

So, what we have seen is that, we often use map and filter together and list comprehension is a succinct and readable way of combining these functions. So, that we can directly understand, what is going on, but list comprehension is not in itself a new piece of notation in Haskell. It is merely what is called syntactic sugar. It is just an easy to read form of something that can be described directly using concat, map and filter.

**Functional programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

**Module # 03**  
**Lecture – 05**  
**Folding through a List**

Map and filter are two 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)

### Combining elements

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

multlist :: [Int] -> Int
multlist [] = 1
multlist (x:xs) = x * (multlist xs)

• What is the common pattern?
```

So, here 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 function sum. So, let us call it sumlist. So, that we do not confuse it with the built in function sum. So, if we have an empty list, this sum is 0, if we have a non-empty list, we add the first element to 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 in the list.

(Refer Slide Time: 01:03)

## Combining elements ...

$$\begin{aligned} \text{combine } f v \square &= v \\ \text{combine } f v (x:xs) &= f x (\underline{\text{combine } f v xs}) \end{aligned}$$

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 in this case the functions were hardwired as sum or multiplier, 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 apply the function to the current value and then recursively compute the value that we got from the rest of the list.

(Refer Slide Time: 01:35)

## Combining elements ...

$$\begin{aligned} \text{combine } f v \square &= v \\ \text{combine } f v (x:xs) &= f x (\text{combine } f v xs) \end{aligned}$$

- We can then write
  - $\text{sumlist } l = \text{combine } (+) \underline{\underline{0}} 1$
  - $\text{multlist } l = \text{combine } (*) \underline{\underline{1}} 1$

So, this is then the way, we can write our earlier functions.

$\text{sumlist } l = \text{combine } (+) 0 1$

$\text{multlist } l = \text{combine } (*) 1 1$

So, when we are doing sumlist, the function is plus (+) and when we are doing multiplying list, the function is times (\*). So, the initial value for sumlist is 0, the initial value for multiply list is 1. So, when we combine 0 as an initial value with the function plus, we get the sum of the numbers. When we combine multiply with the initial value as 1, we get the product of the list.

(Refer Slide Time: 02:02).

## foldr

- The built-in version of combine is called foldr

$$\text{foldr } f \ v \ [] = v$$

$$\text{foldr } f \ v \ (x:xs) = f \ x \ (\text{foldr } f \ v \ xs)$$

$x_1 \ x_2 \ \dots \ x_{n-1} \ x_n \ \boxed{v}$

f  $x_n : []$        $f \ v \ []$   
 $\downarrow$                    $\downarrow$

So, the function in Haskell which does this is called foldr; and 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 an initial value. For the empty list, we return the initial value and otherwise, we inductively apply the function to the current value, the head of the list and the value, we have computed in the tail of list.

So, we can visualize this as 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 operate on the last value. So, if we have  $x_n : []$ , then this will apply  $f$  to  $v$  and the empty list, it gives us  $v$  and then, apply  $f$  to this value and this value.

(Refer Slide Time: 02:55)

## foldr

- The built-in version of `combine` is called `foldr`

$$\text{foldr } f v [] = v$$
$$\text{foldr } f v (x:xs) = f x (\text{foldr } f v xs)$$

So, the first thing we get is that, we combine  $x_n$  with the initial value  $v$  and we get a new value  $y_n$ . Then, in the previous step, we apply  $x_{n-1}$  to the result of this value and we get a value  $y_{n-1}$ . So, in this way working backwards from the right, so this  $r$  in `foldr` 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 in 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, to fold this function  $f$  from the right through this list with an initial value  $v$ .

(Refer Slide Time: 03:40)

## Examples

- `sumlist l = foldr (+) 0 l`
- `multlist l = foldr (*) 1 l`
- `mylength :: [Int] -> Int`  
`mylength l = foldr f 0 l`  
where  
 $f x y = y+1$
- Note: can simply write mylength = foldr f 0
  - Outermost reduction: mylength l  $\Rightarrow$  foldr f 0 l

So, therefore, `sumlist`, the function that we wrote, this is same as folding the function plus from the right, using the initial value 0. `Multlist` is the result of folding the function times

from the right with an initial value 1. We can even define the length of a list by an appropriate function, so if we just take the function which starts off with 0 and adds 1 for every value it sees, then we can take that function and fold it.

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, whatever value, in this case,  $y$  will be the result that has come from the remaining list and we will get  $x+1$ . So, `mylength`, the length of the list can also be expressed using `fold`.

So at this point, let us introduce the convention which is often used by Haskell programs. Notice that, we can write `mylength` directly to be this expression `fold f 0`; that is because, when we apply `mylength` to any list, we apply this expression to the list. Then, if I have an expression of this form, remember that Haskell works by outermost reduction, to rewrite the outermost definition. So, it will not try to expand  $l$ , it will try to expand `mylength`.

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

(Refer Slide Time: 05:42)

*foldr appendright [] [1,2,3]*

Examples ...

- Recall  
 $\text{appendright } x \times l = l ++ [x]$
- $\text{foldr appendright } [] = ??$

The diagram illustrates the reduction of the expression `foldr appendright [] [1,2,3]`. It shows the list `[1,2,3]` and the empty list `[]`. A red arrow labeled "appendright" points from the empty list to the list `[1,2,3]`. Below this, another red arrow points from the empty list to the result `[1,2,3]`, with the label `[ ] + [3] ↗ [1,2,3]` indicating the addition step.

So, remember that one of the early functions we wrote in Haskell was to add a value to the end of the list, instead of instead of appending at the beginning as is done when we do `x:l` we want it to take `l` and stick `x` at the end. So, we can write this for instance as `l ++ [x]`, where `[x]` is the singleton list `x` or we can write an inductive definition, but this is what we need to appendright. So, what would happen if we used appendright on the list with initial value empty.

So, lets try to execute it and supposing I apply this `foldr appendright [] [1,2,3]`. So, we have the empty list `[]` and we have `[1, 2, 3]`. So, the first thing is that, I will take this element 3 and combine it to this `[]` using appendright. So, this will give me the element on the right `[3]`. Now, we will take this element 2, stick it to the right of this `[3,2]` and you will take this element 1, stick it to the right `[3,2,1]`. So, we can see is that append r , appendright is shifting each element to the end of the list already created to it is right.

(Refer Slide Time: 07:03)

## Examples ...

- Recall  
`appendright x l = l ++ [x]`
- `foldr appendright [] = ??`
- `foldr appendright [] = reverse`

So, in other words `foldr` of `appendright` to the empty list is the same as `reverse`.

It just reverses the list.

(Refer Slide Time: 07:12)

### Examples ...

- What is `foldr (++)` □ ?
- Dissolves one level of brackets
- Flattens a list of lists into a single list

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

(Refer Slide Time: 07:39)

### Examples ...

- What is `foldr (++)` □ ?
- Dissolves one level of brackets
  - Flattens a list of lists into a single list
- The built-in function `concat`

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

(Refer Slide Time: 07:49)

**foldr**

$f: a \rightarrow b \rightarrow b$   
2 args:  $\underline{\underline{v}} \rightarrow \underline{\underline{\text{result}}}$

$\text{foldr } f \ v \ [] = v$   
 $\text{foldr } f \ v \ (x:xs) = f \ x \ (\text{foldr } f \ v \ xs)$

- What is the type of foldr?

$$\begin{array}{c} [a] \\ \xrightarrow{x_{n-1}, x_n} c \\ f \quad y_n \\ \downarrow \\ y_1 \end{array}$$

So, foldr as we saw before is defined 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 does  $f$  do?, it takes two arguments and it produces a result. So, there are three possible things. It could be in general of type  $a \rightarrow b \rightarrow c$ . On the other hand, what we have seen is that the result that it produces is the right hand side of the next. So, we have, say the value  $v$ , then we have  $x_n, x_{n-1}$ , and so on.

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 compatible with the second argument of  $f$ . So, this must actually be another  $b$  and then at the end I am eventually going to produce  $y_1$  which is the output of the function. So, that will be a  $b$ . So, it takes a function of the form  $f$  which takes two arguments of type  $a$  and  $b$  and producing output of  $b$ . So, I can keep repeating this process, it takes an input list, this whole input list is of type  $a$  and it produces an output of type  $b$ .

(Refer Slide Time: 09:06)

## foldr

```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

- What is the type of foldr?

$\text{foldr} :: \underbrace{a \rightarrow b \rightarrow b}_{f} \rightarrow b \rightarrow [a] \rightarrow b$

So, the type of foldr is the following. So, this is my function, it takes two arguments and produces an output of the same type as the second argument. It takes an initial value which is of the same type as final output. It takes a list of inputs which are compatible with this argument of the f and produces an output which is the output of the last application. So, foldr takes a function  $a \rightarrow b \rightarrow b$  and initial value b and input list of type a and produces an output value of type b.

(Refer Slide Time: 09:37)

## foldr1

- Sometimes there is no natural value to assign to the empty list
- Finding the maximum value in the list
  - Maximum is undefined for empty list

```
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

$\text{maxlist} = \text{foldr1 max}$        $\text{max} + 3 = 7$

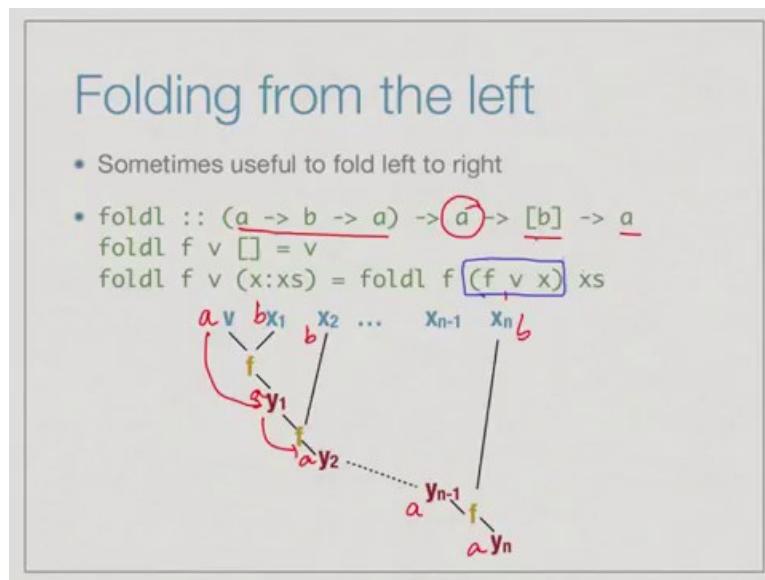
So now in some situations, there is no natural value to assign to the base case to the empty list. So, foldr basically takes two arguments the initial value and then, the function. So, the function is sometimes may not be sensible for an empty list, for example, supposing we want

to find the maximum value in a list, then there is no sensible maximum that you can assign to the empty list.

So, it makes sense to have a function which only uses a non-empty list, this in Haskell is called foldr1. So, foldr1 on a non-empty list, for a singleton will just return the value itself. So, the base case for this is a non-empty list of length 1, because 0 is not allowed. It cannot have the empty list and otherwise, it behaves exactly like foldr. So, it takes the first argument of a non-empty list and applies f along with the result to bring it to the rest.

So, in this case for instance the list, maxlist the maximum value in a list can be done by folding this which 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 it gives us the bigger of the 2. So, we start with max list of a singleton list being the value itself and if we fold it through the list we get maxlist.

(Refer Slide Time: 11:01)



So, there is a symmetric thing which you can do is to start from the left and fold from the left to the right. So, we start with the b at the beginning then we apply f to v and x1 to get y1 then we take y1 and apply f to y1 and x2 to get y2 and so on. And then, eventually we get applying the last element to get yn. So, now, we can see that, this will have a slightly different type, because now I take a function whose inputs are of type a and b, now, the outputs must be the same as the first.

So, this must again be of type a, because again I am going to map it to b, this will again be of type a and again I am going to map it to b and finally, this is of type a type b type a. So, the

function is from  $a \rightarrow b \rightarrow a$  and then, we have an initial value of type  $a$ , we have a list of  $b$ 's and it produces an output of type  $a$  and when recursively applied, notice that, the first argument is the inductive thing of the left segment and we applied this value to what remains. So, it is slightly different from `foldr`, because we are going left to right.

(Refer Slide Time: 12:16)

## Example

- Translate a string of digits to an integer  
`strtonum "234" = 234`
- Convert a character into the corresponding digit:  
`chartonum :: Char -> Int`  
`chartonum c`  
   $| ('0' \leq c) \&& (c \leq '9')$   
     $= (\text{ord } c) - (\text{ord } '0')$

So, here is an example of using this `foldl`, if we want to take a string of digits written using a character and represented as a number. Then it is natural to move from left to right, because as we see each digit, the number increases in length. So, our base function can be one which takes a single character and converts it to a digit. So, we have seen the function character to number `chartonum`, which checks whether  $c$  lies in the range 0 to 9 in characters.

Remember, we assumed that the characters are represented in tables and the digits are all contiguous in this table. So, if it lies in this range, then we just compute the number by taking its offset with respect to the table value of 0. So, 0 will be mapped to 0 , 1 will be mapped to 1 and so on.

(Refer Slide Time: 13:12)

### Example ...

"234"

$23 \times 10 + 4$

- Process the digits left to right
- Multiply current sum by 10 and add next digit

```
nextdigit :: Int -> Char -> Int
nextdigit i c = 10*i + (chartonum c)
strtonum = foldl nextdigit 0
```

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

Now, if you start with the base value 0 and go from left to right then the first step is to convert the character 2 to number 2, so I will get 2. Then, the next time I will take  $2 * 10$ , 20 and add 3, so we are get 23. So, I will get 23 then I will take 23 times 10 and add 4 and I will get 234. So, this is an example where you fold the function from the left to the right.

(Refer Slide Time: 14:12)

### Summary

- Many cumulative operations on lists can be expressed in terms of folding a function through the list
- Built in functions `foldr`, `foldr1`, `foldl`

← →

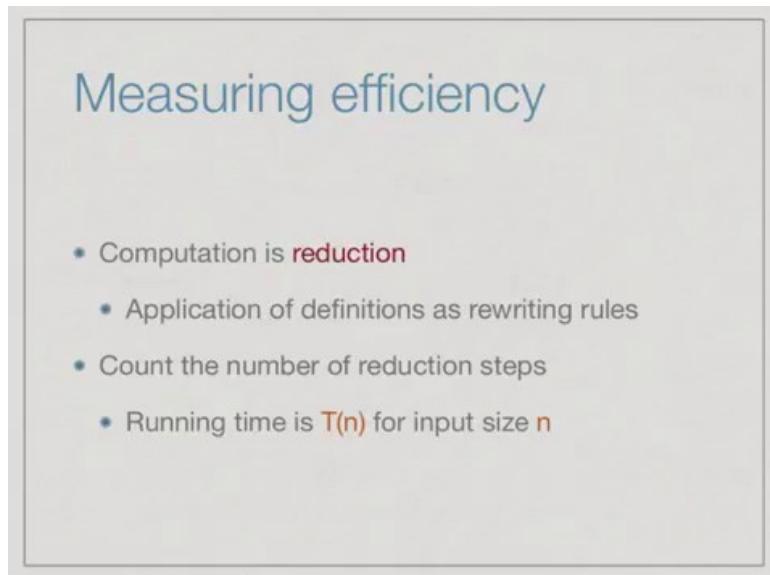
So, Map and filter are functions which transform elements to elements individually, but if you want to combine the elements in the list into a single value as an output. Cumulative operation typically involves folding a function with the list, applying it from one end to the other end. So, the basic functions in Haskell which do this are foldr and foldl. So, foldr goes from right to left, foldl goes from left to right and there are versions of these functions, which do not have a sensible value for the initial empty list called foldr1.

**Functional programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

**Module # 04**  
**Lecture – 01**  
**Measuring efficiency**

Whenever we write a program to solve a given task, we need to know how much resources it requires, how much space and how much time. Here we will focus on how to compute the amount of time required by a Haskell program.

(Refer Slide Time: 00:17)



Measuring efficiency

- Computation is **reduction**
  - Application of definitions as rewriting rules
  - Count the number of reduction steps
  - Running time is  **$T(n)$**  for input size  **$n$**

Remember that the notion of computation in Haskell is rewriting or reduction, in other words we take function definitions and when we use them to simplify expressions by replacing the left hand side of a define by its right hand side. In this way we keep applying these, rewriting rules until no further simplification is possible for the given expression. Hence it makes sense to count the number of reduction steps and use this as a measure of the running time for Haskell program.

Now normally, the running time of a program depends on the size of the input, it obviously takes more time to sort a large list than a small list. So, we typically express the running time

as a function of the input size, if the input size is n, let us write T of n is the function which describes the dependence of the time on the input size.

(Refer Slide Time: 00:12)

## Example: Complexity of ++

$\square \quad ++\ y = y$   
 $(x:xs) \ ++\ y = x:(xs++y)$

- $[1,2,3] \ ++\ [4,5,6] \Rightarrow$   
 $1:([2,3] \ ++\ [4,5,6]) \Rightarrow$   
 $1:(2:([3] \ ++\ [4,5,6])) \Rightarrow$   
 $1:(2:(3:([] \ ++\ [4,5,6]))) \Rightarrow$   
 $1:(2:(3:([4,5,6])))$
- $l1 \ ++\ l2$ : use the second rule length  $l1$  times, first rule once, always

So, let us start with an example, here is the definition of the built in function `++` that combines two lists into a single list, so that function is defined by induction on the first argument. So, if we combine the empty list with the list  $y$ , then we just get  $y$  itself, if we combine the non empty list with  $y$ , then we take the first element of the first list  $x$  and move it to be the first element of the inductively combined list  $xs++y$ .

So, let us execute this on an input such as  $[1,2,3] \ ++\ [4,5,6]$ . Since, the first list is non empty, the second definition applies and so we now have to append 1 to the result of  $[2,3] \ ++\ [4,5,6]$ . Once again we apply the second definition, so inside the bracket now we have 2 appended to the list  $[3] \ ++\ [4,5,6]$ . Once again we apply the second definition. So, we get the list 3 appended to the  $[] \ ++\ [4,5,6]$

Now the base case applies, so  $[] \ ++\ [4,5,6]$  is just  $[4, 5, 6]$ . So, we get the final answer which is 1 appended to 2 appended to 3 appended to the list, 4, 5, 6 written as  $1: (2: (3: ([4,5,6] )) )$ . So, from this it is clear that when we execute `++` for each element in the first list we have to apply the second rule once. So, the second rule is used length of  $l1$  times and finally, when the length of  $l1$  becomes zero, we will apply the first rule once and this behavior is independent of the actual values of  $l1$  and  $l2$ . We will always use the second rule length of  $l1$  times followed by one application of the first rule.

(Refer Slide Time: 03:02)

## Example: elem

```
elem :: Int -> [Int] -> Bool
elem i [] = False
elem i (x:xs)
  | (i==x) = True
  | otherwise = elem i xs

• elem 3 [4,7,8,9] => elem 3 [7,8,9] =>
  elem 3 [8,9] => elem 3 [9] => elem 3 [] =>
  False

• elem 3 [3,7,8,9] => True

• Complexity depends on input size and value
```

On the other hand, let us look at this function elem, which stands for element of. It checks if a given integer belongs to a list of integers. So, the base case says that the element i never belongs to the empty list and otherwise, we check whether it is the first element of the non empty list, if so we return True, if it is not so, we continue to search for the element in the rest of the xs. Now, if we apply the element function to a list which does not contain the value, then the second clause gets executed as many times as the length of the input until we reach the empty list and gets false.

So, we start with elem 3 of [4, 7, 8, 9] then in turn we strip off the 4, the 7, the 8, the 9 until we get to elem 3 of the empty list and then say False. On the other hand if we are lucky, we might find the element right away. For instance, if the first element of this list was not a 4, but a 3, then in one step we would find that the first element matches the pattern we are looking for and we would return True. So, in general the actual execution times of function on an input depends both on the input size and the actual value of the input.

When we executed ++, we saw that the value of the input did not play any role, we would always execute the command, the second definition as many times as the length of the first list and then execute the first definition once. But, in most functions depending on what we passed to the function the execution may take time less or more time.

(Refer Slide Time: 04:40)

## Variation across inputs

- Worst case complexity
  - Maximum running time over all inputs of size  $n$
  - Pessimistic: may be rare
- Average case
  - More realistic, but difficult/impossible to compute

So, to account for variation across the values of the inputs, the standard idea is to look at the worst possible input, this is called the worst case complexity. Now, this basically takes the maximum running time over all inputs of size  $n$  and defines this to be the worst case complexity of the function, this is perhaps a bit pessimistic, because the worst case might occur very rarely. But, on the other hand this is the only concrete case that we can typically analyze.

It would often be nice if we could actually make some kind of statistical average and compute the average case. But, in many cases it is difficult to define a standard distribution of probability across all inputs and compute a meaningful average. So, though the average case complexity is a more realistic measure of how long the function takes, it is either difficult or impossible to compute in general. So, we must unfortunately settle for the worst case complexity.

(Refer Slide Time: 05:42)

## Asymptotic complexity

- Interested in  $T(n)$  in terms of orders of magnitude
- $f(n) = O(g(n))$  if there is a constant  $k$  such that  $f(n) \leq k g(n)$  for all  $n > 0$
- $an^2 + bn + c = O(n^2)$  for all  $a,b,c$   
(take  $k = a+b+c$  if  $a,b,c > 0$ )  
$$3n^2 + 5n + 2 \\ \leq 10n^2, n > 0$$

The other feature that is usually used when analyzing algorithms is to use what is called asymptotic complexity. So, we are interested in how  $T(n)$  grows as a function of  $n$ , but we are interested only in orders of magnitude, we are not really interested in exact details of the constants involved. So, the standard way to express this is to use this so called Big O notation. So, big O notation says that  $f(n)$  is no bigger than  $g(n)$ , in other words  $f(n)$  is dominated by some constant times  $g(n)$  for every  $n > 0$ .

As an example, suppose we have the concrete function  $f(n)$  as the quadratic  $an^2 + bn + c$ . We claim that this is actually Big O of  $n^2$ . For instance, supposing we take a concrete value such as  $3n^2 + 5n + 2$  then we could take  $3+5+2$  and say that this is always  $\leq 10n^2$  for all  $n > 0$ . So, if  $a$  and  $b$  and  $c$  are all positive then we can just add up the coefficients to come up with this value  $k$ .

You can check that if any of the coefficients is negative you can just drop it. So, you can just add up this sum of the positive coefficients and that should work. So, usually it turns out to be a simple problem which is you just take the highest power.

(Refer Slide Time: 07:16)

## Asymptotic complexity

- Interested in  $T(n)$  in terms of orders of magnitude
- $f(n) = O(g(n))$  if there is a constant  $k$  such that  $f(n) \leq k g(n)$  for all  $n > 0$ 
  - $an^2 + bn + c = O(n^2)$  for all  $a,b,c$   
(take  $k = a+b+c$  if  $a,b,c > 0$ )
- Ignore constant factors, lower order terms
  - $O(n), O(n \log n), O(n^k), O(2^n), \dots$   
*polynomial exponential*

So, usually we ignore the constant factor, so we ignore constants like  $a, b$  and  $c$  and we take then among the terms that contributes to the complexity the highest power and we say that this function is  $O(n^2)$ . So, given this we typically express the complexity of the function terms of functions like  $n \log n$  or  $nk$  for some  $k$ . So, these are the so called polynomial functions or we have exponential and so on. So, this is the typical notation that we will use to describe the complexity of our functions.

(Refer Slide Time: 07:53)

## Asymptotic complexity ...

- Complexity of `++` is  $O(n)$ , where  $n$  is the length of the first list
- Complexity of `elem` is  $O(n)$ 
  - Worst case!

So, in this notation we saw that the complexity of `++` is  $O(n)$ , when  $n$  is the length of the first list, the length of the second list is immaterial. Therefore, it really is irrelevant as far as the input goes. On the other hand, for the function `elem` we again got a linear dependence  $O(n)$ , but this is not true for all inputs, we saw that it could actually terminate in one step if the first element matches the length element we are looking for. So, this is really a case where we are applying this worst case definition in order to determine the complexity of the function.

(Refer Slide Time: 08:32)

## Complexity of reverse

```

myreverse :: [a] -> [a]
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++ [x]

```

- Analyze directly (like `++`), or write a recurrence for  $T(n)$ 
  - $T(0) = 1$  ✓
  - $T(n) = T(n-1) + n$

So, let us try and analyze the complexity of function that we wrote earlier. So, this is our inductive definition of reverse, we said that we can reverse the empty list by just returning the empty list. On the other hand, if we have a non empty list then we pick the tail of the list, reverse it inductively and then append the first element afterwards. Now, unfortunately this append we know depends on the length of the tail.

So, we could try to analyze this directly or we could use the fact that we know something about `++` to write what is called a recurrence. Recurrence expresses  $T(n)$  in terms of smaller values of  $T$ . So, in this case if we have an empty list, if the list length is 0, so here this input size is the length of the list to be reversed. So, if the length is 0 then clearly we can reverse it in one step. Now, if the length is not zero then we have to first reverse the tail.

So, this means that in order to reverse the length of list of length  $n$  we have to first reverse the list of length  $n-1$  and then what we saw in our first analysis was that this function `++` will take time proportional to  $n-1$  iterations of the second definition plus 1 iteration of the first

definition and therefore, it will take  $n$  steps. So, this gives us the behavior of the time complexity of reverse in a recursive form.

(Refer Slide Time: 10:08)

## Complexity of reverse

```
myreverse :: [a] -> [a]
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++ [x]
```

- Analyze directly (like  $\text{++}$ ), or write a recurrence for  $T(n)$ 
  - $T(0) = 1$   
 $T(n) = T(n-1) + n$
  - Solve by expanding the recurrence

So, how do we solve this kind of thing to get an expression for  $T(n)$ , well the easiest way is just to expand the recurrence.

(Refer Slide Time: 10:17)

## Complexity of reverse ...

- $$\begin{aligned} T(n) &= T(n-1) + n & T(0) &= 1 \\ &= (T(n-2) + n-1) + n & T(n) &= T(n-1) + n \\ &= (T(n-3) + n-2) + n-1 + n & \\ &\dots & \\ &= T(0) + 1 + 2 + \dots + n & \\ &= 1 + 1 + 2 + \dots + n & \\ &= 1 + n(n+1)/2 & \\ &= O(n^2) \end{aligned}$$

So, here is our recurrence now. It says  $T(0)$  is 1 and  $T(n)$  is  $T(n-1) + n$ . So, now, we start with  $T(n)$  and using the second item of the recurrence we expand it as  $T(n-1) + n$ . Now, in turn we

can apply the same definition to  $T(n-1)$  and get it as  $T(n-2) + n-1$ . So, this is just an expansion of this recurrence, where  $n$  is substituted uniformly by  $n-1$ . In this way we keep expanding.

And so we are building up this term over here  $n-2 + n-1 + n$  and we have what remains, eventually we come down to the point where  $n-n$  which is 0 comes to us and we have on the right if you check this will be  $n-(n-1)$  so 1, 2, 3 and all that. So, this is the summation of  $i=1$  to  $n$  and this is well known is  $n*(n+1)/2$  and hence going by our earlier way of calculating the Big O the highest terms of this is  $n^2/2 + n/2$ . So, the highest term is  $n^2$  and if we ignore all constants this turns out to be order  $n^2$ . In other words we are actually spending  $n^2$  time in order to reverse the list of  $n$  elements which seems rather inefficient.

(Refer Slide Time: 11:44)

## Speeding up reverse

- Can we do better?
- Imagine we are reversing a stack of heavy stack of books

A small diagram showing three books stacked vertically. A green arrow points from the top book to the bottom book, indicating the direction of movement or reversal. The books are represented by simple rectangles with colored edges.

So, how do we improve on this? So, the idea is that we do not reverse the list in place as we are trying to do, but build up a second list. So, imagine that we have a stack of books, so maybe we have a red book and blue book then a green book ((Refer Time: 12:06)). So, now, what we do is we move this book to a new stack. So, we now have a green book here and we have moved green book here, then we move the red book onto this and now we have a red book here and no red book. Finally, we move the blue book to new stack, now we have a blue book on top and now notice that this second stack is the reverse of the first stack.

(Refer Slide Time: 12:33)

## Speeding up reverse

- Can we do better?
- Imagine we are reversing a stack of heavy stack of books
- Transfer to a new stack, top to bottom
- New stack is in reverse order!

So, in other words we transfer to the new stack from top to bottom, and the new stack is the old stack in reverse order. So, we can use this idea to write a more efficient version of reverse.

(Refer Slide Time: 12:46)

## Speeding up reverse ...

```
transfer :: [a] -> [a] -> [a]
transfer [] l = l
transfer (x:xs) l = transfer xs (x:l)
```

- Input size for transfer  $l_1 \ l_2$  is length  $l_1$
- Recurrence

$$\begin{aligned}T(0) &= 1 \\T(n) &= T(n-1) + 1\end{aligned}$$

- Expanding:  $T(n) = \underbrace{1 + 1 + \dots + 1}_{n+1} = O(n)$

So, here is the idea of moving a list from one side to another side and reverse. So, what we do is we transfer the contents of our first list to the second list. So, the first list is empty then there is nothing to transfer and just leave the second list as it is, if the first list is non empty then this  $x$  is the book on top of the stack, so we move it to the top of the second stack. So, if

we want to transfer  $x:xs$  to  $l$  then we keep the  $xs$  in the first stack and move this  $x$  from the first stack to the second stack.

So, now, it is clear that this function does not depend on the second list that is passed. So, this is the bit like  $++$ . So, the input size is actually the length of  $l$  and it is clear that we have a recurrence of this form which says that if I have an empty list, then I do it in one step as a first argument, if I have a non empty list then it takes me one step to produce an instance of transfer of size  $n-1$ .

So,  $T(n)$  is  $T(n-1) + 1$  and now using our expansion if we expand this  $n$  times we come down to  $T(0)$ . So, we get  $1+1+1\dots n+1$  times and  $n+1$  is just order of  $n$ , in other words as we clearly know from the way we describe the process manually transferring a list in reverse from one stack to another stack takes times proportional to the length of the list.

(Refer Slide Time: 14:24)

## Speeding up reverse ...

```
fastreverse :: [a] -> [a]
fastreverse l = transfer l □
```

- Complexity is  $O(n)$  •
- Need to understand the computational model to achieve efficiency

And now we are done, because we can just start with an empty second stack as we said before and transfer everything from the first stack to the second stack. So, we have a `fastreverse` which is written in terms of this linear function `transfer` and `fastreverse` of  $l$  is just transfer the contents of  $l$  to the empty stack. The complexity of this function is linear, so notice that we had to take a look at how lists are treated in Haskell and how computation works in order to come up with the slightly non obvious definition of `reverse`, which matches the intuitive complexity that we have for the function. So, this shows that you cannot blindly

apply the techniques from one programming language to another without understanding the computation model, if you want to achieve the efficiency that you like.

(Refer Slide Time: 15:16)

## Summary

- Measure complexity in Haskell in terms of reduction steps
- Account for input size *and value*
  - Usually worst-case complexity
- Asymptotic complexity
  - Ignore constants, lower order terms
  - $T(n) = O(f(n))$        $n, n \log n, n^k, 2^n$

To summarize, we measure the complexity of a Haskell function in terms of the number of reduction steps we take to arrive at the answer. So, reduction consists of applying a definition in a function and rewriting the left hand side by the right hand side. Now, when we compute the function we have to account for the input size, but also for the input value, we saw the function elem could return quickly or take a long time depending on whether not the value we are looking for belongs to the list.

So, to account for the input size and the value, we usually use worst case complexity, because the desirable goal of computing average case complexity is very hard. And finally, we said that we will use traditional algorithmic ideas and express the efficiency in terms of asymptotic complexity. So, we will ignore the constants, ignore lower order terms and write  $T(n)$  in terms of Big O of  $f(n)$ , where  $f(n)$  will typically be a function like  $n$  or  $n \log n$  or  $n k$  or  $2n$ .

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

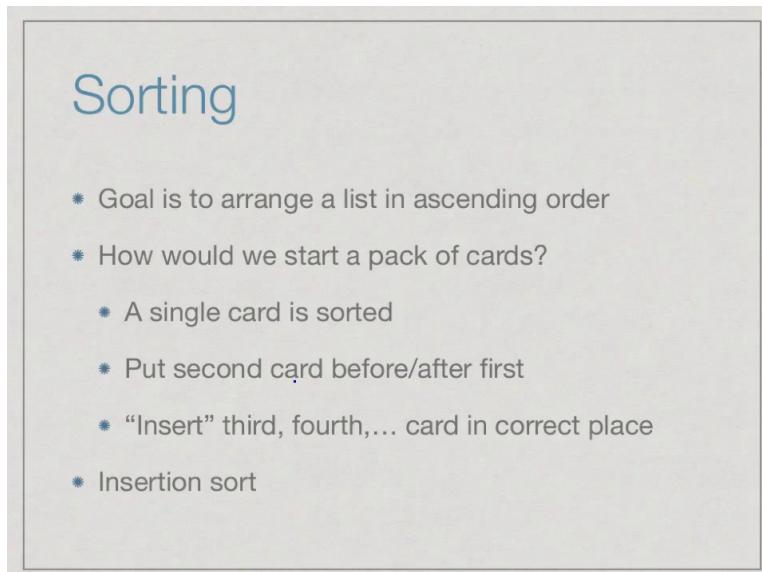
**Module # 04**

**Lecture - 02**

**Sorting**

Sorting a list is often an important prerequisite to doing other useful things on it. For example, one way to search for whether a list has duplicates is to sort it and then, check if any adjacent values in the sorted list are equal to each other.

(Refer Slide Time: 00:19)



The slide has a light gray background with a thin black border. The title "Sorting" is centered at the top in a blue font. Below the title is a bulleted list of steps:

- \* Goal is to arrange a list in ascending order
- \* How would we start a pack of cards?
  - \* A single card is sorted
  - \* Put second card before/after first
  - \* “Insert” third, fourth,... card in correct place
- \* Insertion sort

So, our goal is to arrange a list in ascending or in descending order of values. So, let us just focus on ascending order because; obviously, in descending order everything will be symmetric using greater than instead of less than. So, to understand a basic algorithm for sorting, let us try to sort a pack of cards. So, here is the simple way to sort a pack of cards that many of us are used to in practice, we start with the top card and we start forming a new pack of sorted cards.

So, since the top card is a single card by definition the new pack is currently sorted. Now, we take the second card and depending on its value with respect to the first card we picked up, we either put it above or below. So, continuing in this way we take the third card and put it in the appropriate position with respect to the first two cards and then insert the fourth card in

the appropriate position of the first three cards and so on, until the entire stack is sorted in place. So, since we keep inserting each new card into an already sorted list of the previous cards we have built up, this algorithm is quite naturally called insertion sort.

(Refer Slide Time: 01:37)

## Insertion sort : insert

- \* Insert an element in a sorted list

```
insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y:ys)
| (x <= y) = x:y:ys
| otherwise y:(insert x ys)
```

- \* Clearly  $T(n) = O(n)$

So, to describe insertion sort in Haskell, the first function we need to write is a function `insert` which puts an element into a sorted list. So, concretely let us assume we are sorting integers. So, `insert` takes an integer and a sorted list implicitly of integers and produces a new sorted list with the element we just put in the correct place. So, the base case is to insert a value into an empty list which just produces a one element list consisting of that type.

On the other hand, if you want to insert `x` into a non empty list of `ys`, we look at the first value and check whether or not `x` should come before it, if `x` is smaller than the smallest `y`, then we just take it up front, if `x` is not smaller than the smallest `y` then overall between the `x` and the `ys` the first `y` is the smallest value. So, we pull that out in front and recursively insert the `x` into the remaining `ys`. So, this in general would require us to push `x` all the way to end of the list. So, if we take the input size of `insert` to be the size of the list into which we are inserting, it is clear that the worst case complexity  $T(n)$  is  $O(n)$ .

(Refer Slide Time: 03:07)

## Insertion sort : isort

```
isort :: [Int] -> [Int]
isort [] = []
isort (x:xs) = insert x (isort xs)

* Alternatively
isort = foldr insert []
```

Now, we can express insertion sort in terms of this auxiliary function insert that have been just used. So, we want to sort an empty list then we have to do nothing. So, we just get the empty list back, if we have to sort a non empty list then we first sort the tail and having sorted the tail we insert x into it in the appropriate position. So, the insert function does all the work. An alternative way to write the same thing is to say that we fold the insert function from right to left.

So, if we start with the list  $[x_0, x_1 \dots x_{n-1}]$  then we start with the empty list and then we insert  $x_n$  into this and we get  $x_n$ . And now we will take  $x_{n-1}$  and insert it into this, then take  $x_{n-2}$  and insert it into this and so on. So, we are just folding this insert function from right to left. So, a concise definition of this recursive function is, just use our function foldr and say that isort is foldr of insert.

(Refer Slide Time: 04:12)

## Insertion sort : isort

```
isort :: [Int] -> [Int]
isort [] = []
isort (x:xs) = insert x (isort xs)

* Alternatively
  isort = foldr insert []

* Recurrence
  T(0) = 1
  T(n) = T(n-1) + O(n)

* Complexity: T(n) = O(n2)
```

So, what is the complexity of insertion sort? Well, for the empty list  $T(0)$  is 1 because in one step we get back the empty list and for the non empty case, we have to first sort  $n-1$  elements and then having sorted  $n-m$  elements, we have to insert a value into this list which takes  $O(n)$  time. Because, remember the complexity of insert is  $O(n)$  and therefore, the recurrence for insertion sort is say  $T(0)$  is 1 and  $T(n)$  is  $T(n-1) + O(n)$ . Now, we have seen this recurrence before and we know that if we expand it out, you will get  $T(n)$  is  $O(n^2)$ , because we get something like  $1+2+3+\dots+n$

(Refer Slide Time: 05:00)

## A better strategy?

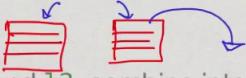
- \* Divide list in two equal parts
- \* Separately sort left and right half
- \* Combine the two sorted halves to get the full list sorted

So, can we do better than  $O(n^2)$  for sorting? So, here is a better strategy which is called divide and conquer. So, what we do is we take the given list and we divide it into two halves

and then we separately sort this half, the left half and the right half and then we combine the two sorted lists into a single over all sorted list.

(Refer Slide Time: 05:29)

## Combining sorted lists

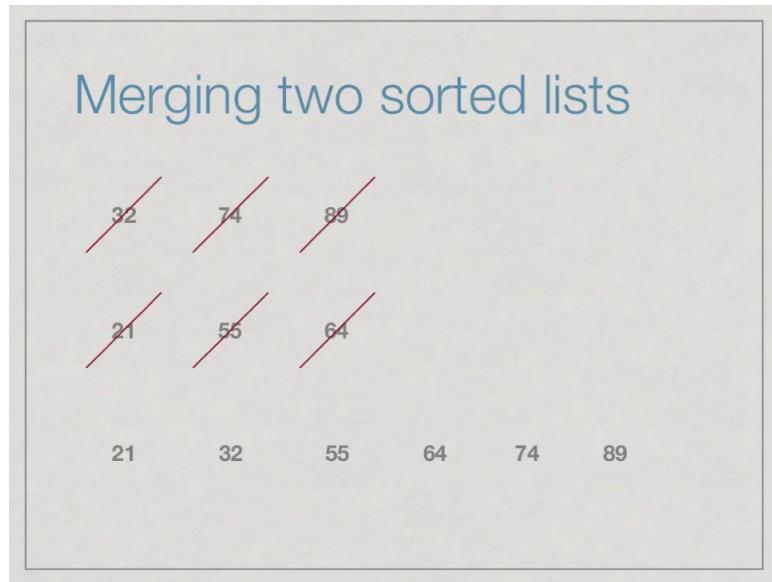


- \* Given two sorted lists  $l_1$  and  $l_2$ , combine into a sorted list  $l_3$ 
  - \* Compare first element of  $l_1$  and  $l_2$
  - \* Move it into  $l_3$
  - \* Repeat until all elements in  $l_1$  and  $l_2$  are over
- \* Merging  $l_1$  and  $l_2$

So, the final step requires us to combine two sorted lists  $l_1$  and  $l_2$  into a sorted list  $l_3$ . Now, this is easy to do, because once again imagine that you just had say two stacks of cards or papers whatever is sorted top to bottom, now you look at the top card in each thing and move the smaller of the two to the newest and each time keep looking at the top card of the two and move it to this smaller of the two.

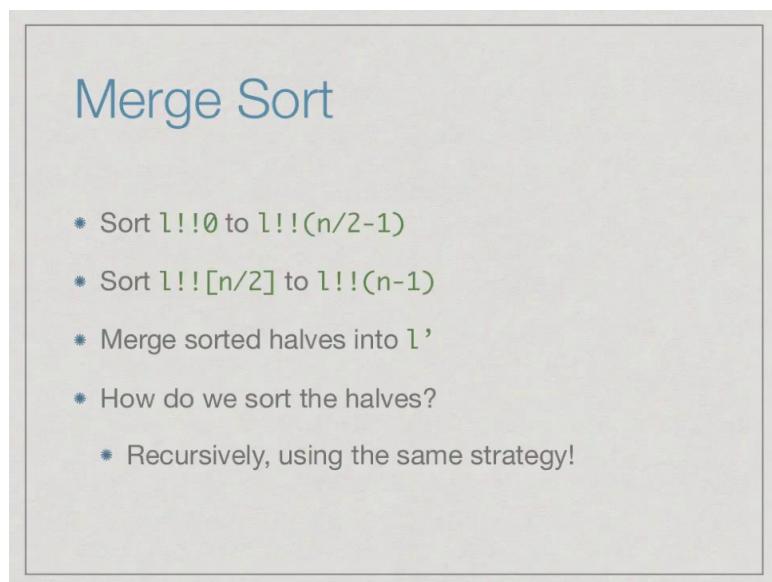
So, in other words if you are looking at lists, we look at the first element of both lists and move the smaller of the two to the new list and keep continuing, until we have exhausted both the lists, so this is called merging. So, we are merging two sorted lists into a single sorted list.

(Refer Slide Time: 06:14)



So, here is an example of how it would work and note that this is l1, so this is sorted  $32 < 74 < 89$  and this is l2, so this is also sorted. So, we start from the left we are looking at the left most element. So, since 21 is the smaller of the two, we remove it from the second list and move it to the new list. Now, since 32 now we are comparing 32 and 55 then we look at the smaller of the two which is 32 and move it to the list continuing with this we now move 55, because we are comparing these two elements and we get 55 and then 64. And now of course, we have nothing left in the second list. So, we can actually blindly copy the first list. So, we copy 74 and 89 and this is the merging procedure.

(Refer Slide Time: 06:59)



And now our earlier divide and conquer strategy was to sort the first half, sort the second half

and merge, just remember that this notation means extract the  $i$ th element. So, it says sort from 0 to the midpoint, the midpoint to the end, so this should be a round bracket. So, we sort from 0 to  $n/2 - 1$ , from  $n/2$  to  $n-1$ , and merge sort combines sorted halves into a new list  $l'$ . And how do we sort the halves? Well, use the same strategy again we divide those into two and then we merge them and so on.

(Refer Slide Time: 07:41)



So, let us run through how merge sort would work on a arbitrary list, so the first step you will divide at the midpoint. So, we will divide it into two lists and try to sort them recursively. So, we will get the left list and the right list, now in turn we will divide each of these into two, because we are applying the same strategy. So, we get two lists from the left and two lists from the right, now we pretend that we still dont know how to sort. So, we will move it one more step. So, we will split each of this list of length 2 into two, so we will get individual lists of length 1.

And now remember that a list of length 1 is by definition sorted, because it is only one value and it must be in order and now we can start merging. So, you want to merge these two, merge these two, merge these two, merge these two to get merged list which are sorted of length 2. So, if we merge 43 and 32, we get 32 followed by 43, the second pair gives us the same lists as before, the third pair again gets inverted, and the fourth pair gets inverted.

So, now, we have sorted lists of length 2 after merging lists of length 1, now you merge these sorted lists of length 2 to get two sorted lists of length 4. And finally, we merge the two sorted lists of length 4 to get sorted list of length 8. So, this is how merge sort works we

divide, divide, divide until we get singletons and then we work backwards merging, merging, merging until we get back the final sorted list.

(Refer Slide Time: 09:15)

## Merge sort : merge

```
merge :: [Int] -> [Int] -> [Int]
merge [] ys = ys
merge xs [] = xs

merge (x:xs) (y:ys)
| x <= y    = x:(merge xs (y:ys))
| otherwise = y:(merge (x:xs) ys)

* Each comparison adds one element to output
* T(n) = O(n), where n is sum of lengths of input lists
```

So, let us now write merge and merge sort in Haskell, so first we write the merge function, the merge function takes two lists which are implicitly assumed to be sorted and produces a new output which is the combination of the two sorted lists. So, we have base cases . If the first list or the second list is empty, we can just copy the other list without doing anything. Because, it is already sorted we just tag it along. On the other hand if we have something to do and both lists are non empty, then we look at the first element and if the first element on x is smaller than y then we merge the tail which is that xs with the remaining ys and then we stick x as the new first element of the overall list.

So, x is the smallest over all, if x is not the smaller one then y is the smallest over all, so this is a very direct translation of our merging strategy that we saw before. So, to analyze merge, one way to think about it is that every time we apply this second rule we are in principle adding one value to the final output and reducing the number of elements we merge by one. So, overall it will take as much time as the number of elements and the two lists put together. So, we can say that the complexity of merge is  $O(n)$ , where n is the sum of the lengths of the two input lists.

(Refer Slide Time: 10:42)

```
Merge sort
mergesort :: [Int] -> [Int]
mergesort [] = []
✓ mergesort [x] = [x]
mergesort l = merge (mergesort (front l))
                           (mergesort (back l))
where
  front l = take ((length l) `div` 2) l
  back l = drop ((length l) `div` 2) l
```

Once we have written merge then merge sort is immediate, so merge sort of the empty list is the empty list, merge sort of the one element list is the one element list and merge sort of any element list with two or more elements consists of first recursively sorting the first half and the second half, which are defined in terms of taking half of the elements. And remember that take and drop supplied with the same argument exhaustively enumerate all the elements. So, take ++ drop is always the list itself.

So, we can be sure that we are not losing any elements or missing out or duplicating elements. we are taking the length of l integer divided by 2. And then we dropping the same number of elements and using that as the back half of the list and having now constructed the sorted versions of the front and the back we are merging it. It is instructive to note that we need this case, we cannot just do with the base case of empty.

Because, otherwise if we take merge sort of x and we do not have this case, suppose we do not have this case then we will try to split into two. So, this would become merge sort of the empty list and merge sort x again now this would give us empty by the base case, but now this would again go back to the same case. So, we will end up with the loop, so we need a base case for the empty list, but we do need a base case for the singleton list as well; otherwise, we will end up with the an infinite loop whenwe come to the singleton and try to split into back and front.

(Refer Slide Time: 12:15)

## Analysis of Merge Sort

- \*  $T(n)$ : time taken by Merge Sort on input of size  $n$ 
  - \* Assume, for simplicity, that  $n = 2^k$
- \*  $T(n) = 2T(n/2) + n$ 
  - \* Two subproblems of size  $n/2$
  - \* Merging solutions requires time  $O(n/2+n/2) = O(n)$
  - \* Solve the recurrence by unwinding

So, the analysis of merge sort is slightly more involved than what we have seen for the functions so far. So, since we keep dividing by 2 it is convenient to assume that the original list of power of 2. So, let us assume for simplicity that the original list we want to sort is of length  $2k$  for some integer  $k$ , then the recurrence for merge sort says that in order to merge the lists of length  $n$  we have to mergesort the front and the back.

So, we have to solve two sub problems of half the size and then merging takes linear time the sum of the two input lists. So, once again we can use unwinding to solve this recurrence. It is just the expressions are little more complicated than we had seen for the earlier ones.

(Refer Slide Time: 13:02)

## Analysis of Merge Sort ...

- \*  $T(1) = 1$
- \*  $T(n) = 2T(n/2) + n$ 
$$\begin{aligned} &= 2 [ 2T(n/4) + n/2 ] + n = 2^2 T(n/2^2) + 2n \\ &= 2^2 [ 2T(n/2^3) + n/2^2 ] + 2n = 2^3 T(n/2^3) + 3n \\ &\dots \\ &= \underline{\underline{2^j T(n/2^j)}} + \underline{\underline{jn}} \quad j = \log n \quad 2^{\log n} \cancel{T(n)} \end{aligned}$$
- \* When  $j = \log n$ ,  $n/2^j = 1$ , so  $T(n/2^j) = 1$   $+ \cancel{\log n} n$
- \*  $T(n) = 2^j T(n/2^j) + jn = \underline{\underline{n + n \log n}} = O(n \log n)$

So,  $T(1)$  is 1 and  $T(n)$  is  $2T(n/2) + n$ , so now we recursively expand  $2T(n/2)$  and we get  $[2T(n/4) + n/2]$  and we will combine these twos and rewrite this fours. So, we will write this two times 2 as 22, we will write this 4 also as 22 for a reason that will become clear in a minute. So, now, we expand this  $n$  by 22 and we will get another division by 2. So, we will get  $2T(n/23) + n/22$ .

Now, notice that this and this cancels so we get another  $n$ . we already had 2  $n$ , this  $n$  plus 2  $n$  times  $n$  by 2. So, now, we will have 3  $n$  and then 22 into 2 this product will give us 23. Start with three steps we have 23  $T(n/23) + 3n$ . So, now, we see a pattern emerging that this is three steps and we have a 3 here and 3 here and 3 here. So, you can check that if you do this  $j$  steps you get  $2^j T(n/2^j) + jn$ .

Now, this keeps going until this becomes 1. when does this become 1? So,  $(n/2^j)$  is equal to 1; that means,  $2^j$  is equal to  $n$  and other words  $j$  is the  $\log_2 n$ . So, when  $j$  is  $\log n$ , then  $n/2^j$  is 1 so we get  $T(1)$ , so this point we have  $2\log n$ , because  $j$  is  $\log n + 1 * \log n + n * \log n$  rather, because we have got  $j$  times  $n$  so now  $j$  is  $\log n$ .

So, we have  $2\log n$  from this term we have  $T(1) + \log n * n$  and  $T(1)$  is 1. So, this goes away, so we have  $2\log n + (\log n) * n$ .  $2\log n$  by definition is  $n$ . So,  $n+n\log n$ , but then this is the smaller terms ,we throw this away and we get  $O(n\log n)$ . So, merge sort takes time  $O(n\log n)$ . We should remember that  $\log n$  is much smaller function than  $n$ , so  $O(n\log n)$  is actually a function which is much closer to  $O(n)$  than to  $O(n^2)$ . So, merge sort is a significantly more efficient sorting algorithm than insertion sort or any other  $O(n^2)$  sort.

(Refer Slide Time: 15:42)

## Avoid merging

- \* Some elements in left half move right and vice versa
- \* Can we ensure that everything to the left is smaller than everything to the right?
- \* Suppose the median value in list is  $m$ 
  - \* Move all values  $\leq m$  to left half of list
  - \* Right half has values  $> m$
- \* Recursively sort left and right halves
- \* List is now sorted! No need to merge

So, we can avoid this merging if we do not have to move elements between the left half and the right half after dividing the list into two. So, can we ensure that everything on the left is already smaller than everything on the right, then if we sort the left and we sort the right we just have to stick them together. So, suppose a median value, the median value remember is the value such that half the values are smaller and half the values are bigger.

Suppose, the median value is  $n$ , now if we take all the values  $\leq m$  and move it to the left and then we have half the values  $> m$  on the right and sort them then because  $m$  is the median value. The right hand side lies strictly to the larger side than the left hand side. So, we can just use  $++$  to combine these two from left and right, so I do not have to do any merge.

(Refer Slide Time: 16:49)

## Avoid merging ...

- \* How do we find the median?
  - \* Sort and pick up middle element
  - \* But our aim is to sort!
- \* Instead, pick up some value in list — pivot
  - \* Split list with respect to this pivot element

Now, the problem with this strategy is that it requires us to know the median and the standard way to find the median would be to sort the list and to pick up the middle element, but our aim is actually to sort the list. So, it is kind of circular to say that we are going to sort the list by finding the median. So, instead we will do this kind of splitting of the list with respect to some arbitrary value, we won't use the median exactly we will just pick up some value in the list and divide it into values just smaller than this pivot and larger than this pivot. So, we pick up some value in the list call it a pivot value and split the list with respect to this pivot element.

(Refer Slide Time: 17:28)

## Quicksort

- \* Choose a pivot element
  - \* Typically the first value in the list
- \* Partition list into lower and upper parts with respect to pivot
- \* Move pivot between lower and upper partition
- \* Recursively sort the two partitions

So, this algorithm is called quick sort and it is due to Tony Hoare. So, you choose a pivot element, typically the first value in the list, we partition the list into lower and upper parts with respect to the pivot. So, the lower part is everything smaller than or equal to the pivot, the upper part is everything greater than the pivot. And now you sort the two parts and move the pivot in between and once you sort the two parts with the pivot in between everything is automatically in order.

(Refer Slide Time: 18:02)

## Quicksort

- \* High level view

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 13 | 22 | 32 | 43 | 57 | 63 | 78 | 91 |
|----|----|----|----|----|----|----|----|

So, typically this is how quick sort would work, so you start with some arbitrary list and then you pick the first element say as the pivot. Now, you analyze the rest of the list and decide which ones are to the left and which ones are to the right. So, here for instance the yellow

values 32, 22 and 13 are smaller than the pivot and the green values are larger than the pivot. So, you re arrange the list, so that all the smaller values are to the left and all the larger values are to the right and now assuming that you can recursively sort those two, so you can sort the yellow and the green thing to get an overall sorted list.

(Refer Slide Time: 18:41)

```

Quicksort
lower ++ [spl] ++ upper

quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = (quicksort lower) ++
  [splitter] ++
  (quicksort upper)
  where
    splitter = x
    lower    = [ y | y <- xs, y <= x ]
    upper   = [ y | y <- xs, y > x ]

```

Quick sort in Haskell is extremely easy to write, so quick sort of the empty list is of course, the empty list, if I have a non empty list I pick this as the pivot. So, here its called the splitter, but it could also be called the pivot and then I take list comprehension to take all the values which are smaller than or equal to the pivot or the splitter and all those that are greater than pivot. Now, one important thing to note is that this comprehension is operating on the tail of this list.

So, the actual splitter though it is less than or equal to itself is not be included in lower, this is crucial; otherwise, we will end up as you might imagine into a situation where the list has a duplicate value, because we are going to put back this splitter here. So, this splitter is both in lower and is there on it is own then we have a problem. So, what we do is we take strictly those values excluding the splitter and we divide them into the lower and the upper, we recursively sort them using quick sort and then we put them into play.

So, we have the lower list and then we have the splitter, then we have the upper list and because the lower list is smaller than the splitter and the upper list bigger than the splitter no further merging is required.

(Refer Slide Time: 19:56)

## Analysis of Quicksort

Worst case

- \* Pivot is maximum or minimum
  - \* One partition is empty
  - \* Other is size  $n-1$
  - \* 
$$\begin{aligned} T(n) &= T(n-1) + n \\ &= \dots = 1 + 2 + \dots + n = O(n^2) \end{aligned}$$
- \* Already sorted array is worst case input!

And the problem with this strategy is that we have no control over what value is at the beginning or wherever we choose the pivot. So, if the pivot happens to be the largest or the smallest value in the list then either the lower or the upper will become empty. So, then the recursive call to lower in sorting lower and upper is not  $n/2$ , but it could be  $n-1$ . So, we may end up with a similar recurrence to insertion sort which says that in order to sort a list of  $n$  elements, we have to first split it.

So, what splitting requires us is to walk down the list and move things to lower or upper, shows that splitting phase requires  $O(n)$  time and then we may have to recursively sort something, which is as large as  $n-1$ . And of course, we know that we expand this out we get this  $1+2+\dots+n$  which is  $O(n^2)$ . So, paradoxically an array which is already sorted for example, where the first element is the smallest value is a worst case input, because it is the smallest value and it will split the list as size 0 and size  $n-1$ . So, it appears that quick sort has not achieved anything, because we have got a worst case complexity which is as bad as insertion sort, the first naive sorting algorithm that we discussed.

(Refer Slide Time: 21:11)

## Analysis of Quicksort

But ...

- \* Average case is  $O(n \log n)$ 
  - \* Sorting is a rare example where average case can be computed
  - \* What does average case mean?

Now, it turns out that quick sort or sorting in general is one of the rare situations, where we can actually compute the average case and the average case for quick sort turns out to be  $O(n \log n)$ . So, let us just quickly look at what it means to compute the average case.

(Refer Slide Time: 21:30)

## Quicksort: Average case

- \* Assume input is a permutation of  $\{1, 2, \dots, n\}$ 
  - \* Actual values not important
  - \* Only relative order matters
  - \* Each input is equally likely (uniform probability)
- \* Calculate running time across all inputs
- \* Expected running time can be shown  $O(n \log n)$

So, for sorting notice that the actual values that have been sorted are not so important as their relative order. So, we can always assume that if we are sorting a list of  $n$  elements, that the elements are actually 1 to  $n$  and the actual list given to us is some permutation of 1 to  $n$ . So, therefore, the space of all  $n$  element inputs effectively becomes the set of all permutations of 1 to  $n$  and now we can assume that each of these is equally likely. So, we can assign a sensible probability to each input saying it is  $1/n!$ .

And now we can calculate the running time across all these permutations and using standard probability theory, although it involves a bit of calculation, you can actually show that the expected running time, which is the average that we are looking for is  $O(n \log n)$ . So, this is why it is difficult to do in general, because for sorting we can kind of exhaustively characterize all the inputs of size  $n$ , but for more complicated functions it may not be so easy to do this and it may not be also so easy to assume a uniform distribution over all possible inputs and so on.

(Refer Slide Time: 22:41)

## Summary

- \* Sorting is an important starting point for many functions on lists
- \* Insertion sort is a natural inductive sort whose complexity is  $O(n^2)$
- \* Merge sort has complexity  $O(n \log n)$
- \* Quicksort has worst-case complexity  $O(n^2)$  but average-case complexity  $O(n \log n)$

So to summarize, sorting is an important starting point for many functions on list. So, it is good to be able to sort a list efficiently, insertion sort is a natural inductive sort, but its complexity in the worst case is  $O(n^2)$ . Merge sort on the other hand uses divide and conquer and has a complexity of  $O(n \log n)$ . Quick sort is a bit simpler than merge sort, because we do not have to have a merge step, we divide those things according to a pivot element and then we just paste the resulting lists together, this has a worst case complexity of  $O(n^2)$ . But, you can actually show that quick sort has an average case complexity of  $O(n \log n)$ .

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

**Module # 04**  
**Lecture – 03**  
**Using Infinite Lists**

(Refer Slide Time: 00:01)

## Lazy evaluation



- \* Recall that Haskell uses lazy evaluation
  - \* Outermost reduction
  - \* Simplify function definition first
  - \* Compute argument value only if needed

We have seen that Haskell uses lazy evaluation. Technically, this is called outermost reduction. So, given an expression of the form  $f e$ , Haskell will try to first apply a definition which simplifies the outer function  $f$ . So, it will compute this argument  $e$ , only if  $e$  is required in the expansion of  $f$ .

(Refer Slide Time: 00:28)

## Infinite lists

- \* Lazy evaluation allows meaningful use of infinite lists

```
infinite_list :: [Int]
infinite_list = inflistaux 0
    where
        inflistaux :: Int -> [Int]
        inflistaux n = n:(inflistaux (n+1))
```
- \* head (infinite\_list) => 0 ✓
- \* take 2 (infinite\_list) => [0,1] *sieve*
- \* [m..] => [m, m+1, m+2, ...]

We saw that this allows us to sensibly use infinite list. So, we wrote this function infinite\_list which generates the list of the form [0,1,2,..] without stopping. At every point n, it generates the value n followed by the infinite list, starting at n+1. Now, though this is an infinite list and it takes an infinite time to compute and therefore, the computation of this infinite list function does not terminate.

We can apply sensible functions to it and get answers in a finite moderate time, for instance if we take the head of an infinite list of this form, it only needs to generate the first element. So, once it has expanded this function once, it finds it has 0 followed by the infinite list starting from 1 and since it has the first element, it can terminate and tell us that the head of this infinite list is 0.

Similarly, if I ask to take the first two elements, it will apply the definition twice and once it has generated two elements, it will return the list [0,1] without trying to evaluate the entire argument. So, this list range notation that we said from [m..n] can be used to generate infinite list by leaving out the upper bound. So, if you write [m..] it means the infinite list [m, m+1, m+2, ..].

So far we saw only one curiosity involving this which is an implementation of the sieve algorithm by Eratosthenes for computing all the primes. So, today let us look at a couple of examples which illustrate, why it is sometimes convenient to think in terms of functions which generate unbounded list, rather than worry about, how to make them bounded.

(Refer Slide Time: 02:15)

## Graphs

- \* Graphs
  - \* A, B, ... are nodes or vertices
  - \* (A,B), (A,D), ... are (directed edges)

```
graph TD; A --> B; A --> D; B --> C; C --> B; C --> D; D --> E; E --> D; E --> F; F --> E;
```

So, our first example comes from the area of graph theory. So a graph is a picture like the one you see here, where you have nodes or vertices, here the node 6 nodes called A, B, C, D, E, F and these nodes are connected by edges. In this particular example, it is a directed graph, so the edges have a direction, you can go from A to B, but you cannot go from B to A. One can also use undirected graphs in which you say A and B are connected without specifying the direction. So, from A you are connected to B, from B are also connected to A.

(Refer Slide Time: 03:00)

## Graphs ...

```
edge :: Char -> Char -> Bool
edge 'A'->'B' = True
edge 'A'->'D' = True
edge 'B'-'C' = True
edge 'C'-'A' = True
edge 'C'-'E' = True
edge 'D'-'E' = True
edge 'F'-'D' = True
edge 'F'-'E' = True
edge _-_ = False
```

```
graph TD; A --> B; A --> D; B --> C; C --> B; C --> D; D --> E; E --> D; E --> F; F --> E;
```

So, continuing with this directed graph, we can represent this directed graph in Haskell by a function edge, which describes all the edges. So, it says that, there is an edge between A and B for example saying that, this is oriented now, the first argument is starting point of the

edge, so this says this is an edge from A to B. Similarly, there is an edge from A to D and so on, so this is probably as known, so these two edges in this should be reverse. So, it should say that, there is an edge from C to A and there is an edge from D to E.

So, this function actually represents this graph note dimension earlier. So, it says just an edge from C to A, but not from A to C, there is an edge from D to E, but not from B to B and so on. So, we exhaustively enumerate all the edges which actually exist in our graph and then in one short, we can say that any other pair of vertices, either shown here or not even shown here. Supposing, I have other vertices called z or w, then any edge involving those vertices also does not exist. So, edge therefore, is a Boolean valued function, which tells us whether a given pair of vertices is connected by an edge or not.

(Refer Slide Time: 04:12)

## Connectivity

- \* Want to check connectivity
- `connected :: Char -> Char -> Bool`
  - \* `connected x y` is `True` if and only if there is a path from `x` to `y` using the given set of edges
- \* Inductive definition
  - If `connected x y` and `edge y z` then `connected x z`
- \* Difficult to translate this directly into Haskell

The typical property that you want to evaluate and such a graph is connectivity given two vertices is their path between them. So, using the edges which are there in the graph can be started x and go to y. So, we want to write this kind of the function connected, which takes two input vertices and tells us the answer is true, if it is possible to construct a path from x to y, given the edges specified in the edge function.

So, there is a very nice inductive way of defining connectedness in terms of edges. If we have already a path from x to y and then, we have a single edge from y to z, then inductively, we have a path from x to z. So, we can define `connected x y` in terms of `connected x z` in terms of the existence of the y, such that x is connected to y and there is an x and y to z.

But, looking for this y, searching for this y is not easy to do in a language like Haskell, this is

a natural definition in other forms of programming, such as logic programming. But, in functional programming it is little hard to compute all the y's possible that are needed to check this fact. So, how do we do this?

(Refer Slide Time: 05:30)

## Building paths

$[v_0, v_1, \dots, v_k]$

- \* Inductively build up paths  $\underset{\text{Edges}}{v_0 \rightarrow v_1, v_1 \rightarrow v_2, \dots}$
- \* Only one path of length 0
- \* Extend path of length k to length k+1 by adding an edge

So, we will follow a different strategy, we will try to inductively build up all the paths that can be constructed in the graph. So, there is only one empty path of length 0, so path is nothing but, a sequence of vertices, a path is can be thought of is the list V 0, V 1 up to V k. Such that V 0 to V 1 is an edge V 1 to V 2 is an edge and so on, so these are all edges.

So, path is just a list of vertices in which every vertex has an edge from its previous element and 2 it is next element. So, we can now take a path of length k. So, notice in the path of length k has k plus 1 vertices, because start somewhere and followed k edges. So, we have a starting vertex and k new vertices. So, you can extend a path of length k to k plus 1, adding an edge.

(Refer Slide Time: 06:28)

## Building paths

- \* Inductively build up paths
  - \* Only one path of length 0
  - \* Extend path of length k to length k+1 by adding an edge

```
type Path = [Char]
extendpath :: Path -> [Path]
* extendpath [] = [ [c] | c <- ['A'..'F'] ]
extendpath p =
    [p++c | c <- ['A'..'F'], edge (last p) c]
```

So, here is a simple version of this. So, if you have an empty path, then you get paths where you just start with starting vertex. So, this is the kind of a trivial base case and we just put it there convenience, it is not really important. The really important once is the second one, we says that if I have already a path, which is a sequences of vertices, then I can add to it, the new edge provided the last element in p and the new vertex are connected by an edge. So, I can add a new vertex to this path and make it a path of length k plus 1 by adding anything, which is connected to the Haskell. So, this is my extend path function.

(Refer Slide Time: 07:11)

## Building paths ...

Handwritten notes:  
Paths of length k  
Paths of length k+1  
[P<sub>1</sub>, ..., P<sub>m</sub>] → [P<sub>1</sub>, ..., P<sub>m</sub>, P<sub>m+1</sub>]  
map extendpath over the list of paths of length k to get the list of paths of length k+1.

```
extendall :: [Path] -> [Path]
extendall [] = [[c] | c <- ['A'..'F']]
extendall l = concat [extend p | p <- l]
               = [ll | p <- l, ll <- extend p]
```

And now, I can build up paths of longer and longer length by just repeatedly extending this. So, if I first take all paths of length x or length k, so I have P 1 to P m, so these are all paths

of length k, then if I map extend on this, then P 1 will generate a list of new paths P 1 1, P 1 2, P 1 n, P 1 l say. So, these are all the extension the P 1, of the similarly P 2 will generate all exemption of P 2. So, these are all new path of length k plus 1. Now, I have this extra level of bracket, so I want to remove them, so I can apply concat.

So, extend all, so l is a list of all paths of length k, for every p and l, I extended it, this gives me a list of extensions, then I dissolve concept of brackets by using concat or I can directly use this comprehension. I can say take every path of length k, take every list that extends that path and now, collect all those list into a new list. So, this gives me a function which takes paths of length k, which is a list and generates paths of length k plus 1.

(Refer Slide Time: 08:42)

## Building paths ...

- \* Built-in function iterate
- \* `iterate :: (a -> a) -> a -> [a]`
- \* `iterate f x =>`  

$$[x; f x, f(f x), f(f(f x)), \dots]$$

$$\begin{array}{cccc} | & | & | & | \\ f^0 x & f^1 x & f^2 x & f^3 x \end{array}$$

So, now if we want to do is, we want to done this from 0 generate paths of length 1 generate paths of length 2 and so on. And at each point, we have generating using these functions extend all, which takes every path of the previous length and generates paths of one length more. So, we can use a very nice built in function in Haskell call iterate. So, iterate takes a function which has input type equal to the output type.

So, it make sense to call the function on his result, it takes a starting value and produces the list of all values of that function applied 0 or no tags. So, if I say iterate f x, so this is f to the power of 0 of x, this is the f to the 1 of x, this is f squared of x, this is f cubed of x and so on. So, iterate f x gives me a list, f of 0 of x comma f 1 of x comma x square comma f to the power 3 of x and so on.

(Refer Slide Time: 09:38)

## Building paths ...

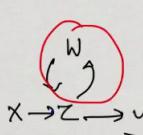
- \* Built-in function iterate
- \* `iterate :: (a -> a) -> a -> [a]`
- \* `iterate f x =>`  
`[x, f x, f (f x), f (f (f x))) ...]`
- \* To generate all paths

```
allpaths = iterate extendall [ ]
```

So, if you want to generate all the paths that we could possibly get, then what we do is, we start with a list consisting of the unique path of length 0 and we iterate this extend all function. So, extend all will take this unique path length 0 and generate all paths of length 1. So, length 1, actually consist of just a starting vertex, then it will do something useful, it generate from that all paths which have two vertices and therefore, one edge and then all paths three vertices and therefore, two edges and so on.

(Refer Slide Time: 10:12)

## Connectivity



- \* To check if `x` and `y` are connected, need to check for paths without loops from `x` to `y`

So, a little bit of thought tells us that to check, if `x` and `y` are connected, we need to check for a path from `x` to `y`, but there is no need to for this path to loop. So, if we have a path which goes through `z`, then go through `w`, comes back to `z` and then, goes to `y`, then this loop is

irrelevant, we could directly go from x through z to y.

(Refer Slide Time: 10:41)

## Connectivity

- \* To check if  $x$  and  $y$  are connected, need to check for paths without loops from  $x$  to  $y$
- \* Given  $n$  nodes overall, a loop free path can have at most  $n-1$  edges
- \* Suffices to examine first  $n$  entries in `allpaths`  
`take n allpaths`

Now, by a simple counting argument, if we are  $n$  nodes, a loop free path can have at most  $n$  minus 1 edges, because remember that for  $n$  minus 1 edges, I have a numerated  $n$  nodes. So, therefore, all the nodes have been used, if I add another edge, I will repeat a node, if I repeat a node, I automatically have a loop. And therefore, that path is not optimal.

So, I only need to look at paths which have at most  $n$  minus 1 edges, so this such that certain order to check connectivity, though my function all paths, which we defined earlier. Generates paths of arbitrary length itself, infinite list of paths of lengths 1, 2, 3, 4; for any length, if we know that, we are only looking at that graph with  $n$  vertices, it is surprises to take  $n$  entries in this paths, in this list.

(Refer Slide Time: 11:34)

## Connectivity ...

- \* Extract endpoints of paths of length at most  $n-1$

```
connectedpairs =  
[(head p, last p) | l <- firstn, p <- l]  
where  
firstn = take n allpaths  
allpaths = iterate extendall []
```

- \* Finally

```
connected x y = elem (x,y) connectedpairs
```

And now, having taken these  $n$  entries, so these  $n$  entries are generated by we treating are extend all and then, take in the first elements. We just need to take for each path in the final list that we get, the first value and the last value, because this path connects the head to the last value. And finally, to check whether two values are connected, we just have to check whether the pair starting point and ending point is one of the pairs numerated in this list of connected pairs.

(Refer Slide Time: 12:07)

## Connectivity ...

- \* `extendall` generates loops, but we don't care
  - \* For instance, path `['A', 'B', 'C', 'A', 'B', 'C']` belongs to the sixth iteration of `extendall []`

So, the point to note is that although we worried about, the fact that all not worried about we analyze the fact that loops are not relevant to us. We do not only care of path has loops are not, from the fact that loops are not relevant to us, what we reduce this that we never need to

look at path longer than length 1. So, in other words, even though we could have paths of length 6 for example, which have a loop, so this has a loop, which take me back from B to B. So, this is actually the path A, B, C, but it goes A to B to C, a back to A, B and then to C again. So, though we have such paths, it does not really matter to us.

(Refer Slide Time: 12:51)

## Connectivity ...

- \* `extendall` generates loops, but we don't care
  - \* For instance, path `['A', 'B', 'C', 'A', 'B', 'C']` belongs to the sixth iteration of `extendall []`
  - \* We just want to ensure that every pair `(x,y)` in `connected` is enumerated by the step `n`
  - \* `connected` is the reflexive transitive closure of the `edge` relation

Always saying is that, every path that we really need to look at this finally, present by step n, always looking for is a piece of evidence that I give an x and y are connectivity, it does not matter they connected with multiple ways. On the other hand, we do know that, if there are not a numerate by step n, there is no way to numerate them, because no loop free path in exist.

On more technical way of saying this is that the connected relation is the reflexive and transitive closure of the edge relation. So, in general, if we have any relation which is given to us by Haskell function like we wrote `edge` which tells us which pairs are in the relation, which pairs or not. If we find want to compute a relation which computes sequences of such pairs, which are connected one of to the other, this is the transitive reflexive transitive closure.

So, every edge is connected and everything which is connected follow by an edge is also connected. Then, we can use this trick that we have of generating all possible length paths and then, using some external criteria to decide on a maximum length path that one needs in order to capture whether or not something is connected.

(Refer Slide Time: 14:00)

## Search problems

- \* No closed form for the solution
- \* Go through a cycle of
  - \* expanding out possible solutions
  - \* undoing partial solutions when we reach a dead end
- \* Backtracking

So, in general a class of problems for which this infinite list technology is useful or they so call search problems. So, we have a space of solutions that we can generate in some form and we are looking for a particular solution with the property that we require among this space of solutions. So, what in general we would do is you keep expanding solutions and then, when we reach us, we are looking for a certain pattern, either we find the pattern or we may find the solution. We are currently expanding has no expansions possible, so we go back and try another one and this is a general strategy call backtracking.

(Refer Slide Time: 14:44)

## N queens

- \* Place  $N$  queens on  $N \times N$  chessboard so that no two attack each other
  - \* Two queens attack each other if they are on the same row, column or diagonal
- \*  $N$  queens,  $N$  rows,  $N$  columns
  - \* Exactly one queen in each row, each column

So, one of the most famous problem is which is use to illustrate backtracking is that of placing  $N$  queens on  $N$  by  $N$  chessboard. So, normally we work with 8 by 8 chessboard, just

this paid on 8 by 8 chessboard and of course, you are running one queen. The rule for queen is that a queen can attack any other piece which is on the same diagonal, same row or same column.

So, now we imagine that we are eight queens place on 8 by 8 chessboards, such that no two queens attack each other. So, in particular because any two queens on the same row and the same column will attack each other, each queen must be a different row and different column, but we have exactly as many queens and columns. So, we have N queens and we have N rows and N columns, a simple argument tells us that we must have exactly one queen at each row and an each column.

Because, we have two queens in many rows, then we have a problem. If we have no queens in some row or column, then we must have two queens in some other row or column and we have a problem.

(Refer Slide Time: 15:46)

## Heuristic

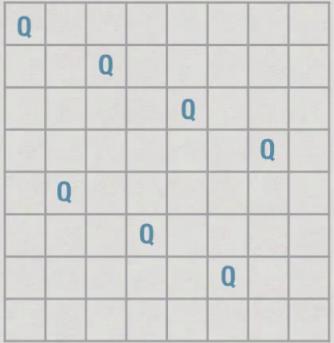
- \* Place the first queen somewhere in the first row
- \* In each succeeding row, place a queen at the leftmost square that is not attacked by any of the earlier queens

So, here is a heuristic to generate all possible solutions, you place the first queen somewhere in the first row and in each succeeding row, you place a queen at the left most square; that is not attack by any of the earlier queens.

(Refer Slide Time: 16:02)

## Backtracking ...

- \* Suppose we have an 8 x 8 board and we start at the top left corner
- \* After 7 queens, we get stuck
- \* Try other positions for queen 7
- \* Go back and try other positions for queen 6 ...



So, suppose if we have a 8 by 8 chessboard and we start with the queen of the top left corner. Now, we are to place second queen and this row, now this is in the same column as the previous queen, this is in the same diagonal. So, the first place that we can put a queen is here, so we place a queen there. Now, we will find that these squares are all ruled out, so this is attack by the first queen, this attack by the second queen.

So, now, we come to the third row in the only place we can put a queen starting from the left is here the first place within input. So, if put a queen there and continue this we come to put a queen on the 4th row, a queen on the 5th row and queen on the 6th row, queen on the 7th row. And now, unfortunately we cannot put a queen here which is the normal place, because it is a same diagonal as the very first queen we put. So, we are stuck. So, we cannot generate a potential solution.

So, what we have to do at this point is backtrack, we have to take this and say that, at this point the last thing, we did was put a 7th queen, so can we put it somewhere else. So, we backtrack and then, if we cannot do anything as we backtrack one more we change this value and so on. So, this is what backtracking means you generate resolution as far as you can go, if you are lucky generate a good one, if you are unlucky you get stuck. If so, you go back and undo the last thing you did retry that value, if you cannot get anything for all possible choice, you go back to the second last value and so on.

(Refer Slide Time: 17:42)

## Searching for a solution

mws 0 - 7  
ols 0 - 7

- \* Arrangement of queens is a list
- \* Position  $i$  describes queen in row  $i$
- \* Value is column number
- \* Previous arrangement is  $[0, 2, 4, 6, 1, 3, 5]$

So, what we want to do is use infinite list in a way that we do not have to explicitly worry about backtracking. So, the first thing we need is where of somewhere representing the state of a board with some queens on it. since, we are going from the first row to the last row, we can always describe the current queens on the board in terms of a list.

So, let us because of the Haskell terminology, number this case the eight queens, the rows as 0 to 7 and the columns as also 0 to 7. So, there are 8 rows and 8 columns which are number 0 to 7. So, the list position 0 for first 2 row 0, where is the queen in row 0, so, it will be a number between 0 and 7 again, because it is the column of that. So, this says that the queen and row 0 is in column 0.

This says that queen and row 1, because is a position, so these are the positions are 0, 1, 2, 3, 4, 5, 6. So, row 1 is a column 2, row 2 is a column 4 and so on. So, this is saying exactly what we said before which is that we have on the top left corner we have a queen and then, we skip 2 and then, we have a queen that we skip 2 more and we have a queen and so on. So, this is a representation of the seven queen that we manage to place before we got stuck.

(Refer Slide Time: 19:09)

## Searching for a solution ...

- \* extend — function to add queen  $k+1$  to current list of  $k$  queens
  - \* Should not be in same column as any earlier queen
  - \* Should not be on same diagonals — values on a diagonal agree on  $r+c$  or  $r-c$

And now, when we want to place the  $k$  plus 1 queen, we have to take a list of the first  $k$  queens and add 1 new queen, a new position for the new element, we have to extend the list by one element. The new number is the column position of the new queen, now the new queen should not be, it is by simplicity, because that the new position, it is a new row, it is not in the same row as any of the earlier columns are queens. But, we should not put it in the same column, because if it is same column, it will be attacked by a previous queen.

And we also need to check, it is not on the same diagonal. Now, it turns out the diagonals, you can do a little bit of arithmetic. So, if we have a diagonal of this form, so for example, we have the diagonal 0 0, 1 1, 2 2 and so on, all of these are the common difference of 0. If I look other diagonal starting at say 0 1, 1 2 and so on, then all of these are common difference of 1.

So, if we have a diagonal of this form going from top right left to bottom right, then all elements on the diagonal, we have a same  $r$  minus  $c$  value. Symmetrically, if we do all diagonals are look like this, then you will have the same sum. So, for instance, we have 0 7, then 1 6, 2 5 and so on, so all these are on one diagonal and they are added to 7. So, it is quite easy to check, whether it queen using the same column as an earlier queen, because of the check the number is not present in their list you have so far.

It checks on the same diagonal just have to add the position to column number and make sure or subtract to position from the column number and make sure that back difference are that sum is not seeing before. So, this extend function we will not write it explicitly that it is easy

to right such an extend function which takes a given list of k queens and returns if possible a k plus 1 position for the new queen.

(Refer Slide Time: 20:56)

### Searching for a solution ...

O f(n) f<sup>2</sup>(n) -

```
queens n = (iterate extend [[]])!!(n+1)
```

- \* All arrangements of n queens on n x n board

So, now as we did with graph, we can start with that queen a position which has no queens on it and iteratively extend it and we know that, we have to place n queens. So, if we extend this n times and we end up with the first one is the empty board, the second is all possible configurations with one queen where it is all possible configuration is two queens and so on. And so the n plus 1th entry will be the list of all configuration and which we have placed all n queens. So, n plus 1 because I start from 0, so this is f, f of n, f 2 of n and so on. So, at the end, I will have extended n times are place n queens.

(Refer Slide Time: 21:45)

### Searching for a solution ...

```
queens n = (iterate extend [[]])!!(n+1)
```

- \* All arrangements of n queens on n x n board
- \* Extract the first such arrangement

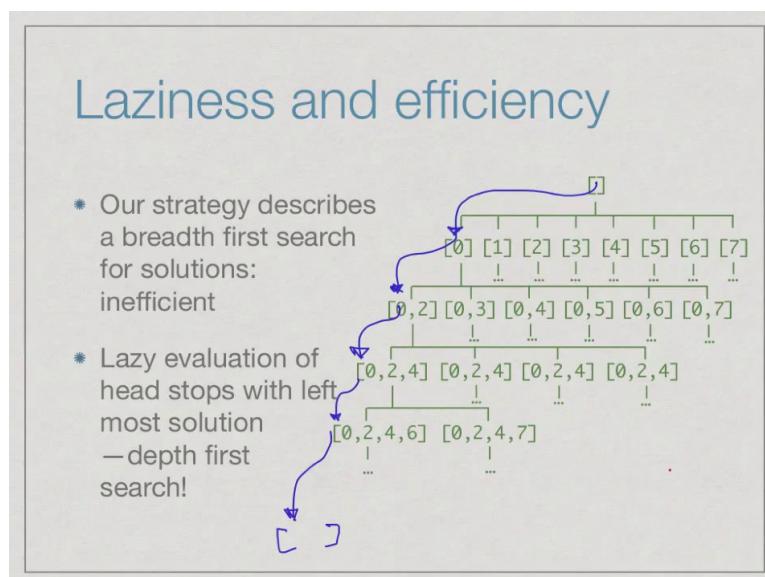
```
queensone n =
    head ((iterate extend [])!!(n+1))
```

- \* Arrangements of k queens that have no extensions die out automatically

And now, we are not said anything about which solution we want. So, it is enough to compute say the first element of that. So, you want at least one solution of the n queens. So, you iteratively extend the empty board, until you get all n queens are it and just take the head of that list. So, that will be a list of all possible arrangements of n queens, which do not attack each other and we just take the first one.

So, notice that in the backtracking problem is avoided implicitly, if we come up with the arrangement of k queens, where k is smaller than n and there is no way to extended. Then, the extend function will just reduce that one empty list, so that can we get the solution is just disappear. So, what we doing is rather than going back and generating a new solution, we are in one short generating all possible candidate solutions, those which dialogue die out, those which do not die out or die and we check which all solutions that survive in n plus 1 steps.

(Refer Slide Time: 22:42)



So, in a sense all solution consists of exploring the graph of all possible arrangements in a breadth first way. So, we generate all possible solutions, where we are place one queen in the first row, then for each of these we expanded. So, we get a second level of the this tree, with generate all possible solutions of two queens in the first 2 rows, three queens in the first 3 rows and so on.

So, this looks like a fairly inefficient strategy, because we have going to first compute all the solutions are 1 row, then all the solutions 2 rows and all the solutions 3 rows in order to reach finally, the first solution of the 8h row. But, this is where now lazy evaluation will actually do the job first, because what lazy evaluation will do is, I will say, if I want the first solution, let

me see the first solution comes from the first solution in the first row. Then, let me see that, solutions comes on the first expansion of that, then let me see the first come from that.

So, actually though we are programming a breadth first search over this three of all possible solutions, lazy evaluation actually does it first search and finds the first solution without evaluating off. Of course, if I all the solutions I have no choice that evaluate everything, but I am only interested in one solution any one solution according to a numeration I have. Then, lazy evaluation does depth first search and effectively finds as the first solution without trying to compute all the solutions we get.

(Refer Slide Time: 24:12)

## Summary

- \* Infinite lists are a useful way to think about search problems
- \* Instead of backtracking, iteratively generate all valid solutions upto a desired depth
- \* Lazy evaluation converts inefficient breadth-first search to depth-first evaluation of leftmost solution

So, we have seen that infinite list are more than just curiosity of lazy evaluation, there actually useful way to think about search problems. So, instead of warning about backtracking and deciding, when something has a particular property, we can heuristically generate all solutions up to a decide depth and then, ask how many search solutions are there or pick 1.

And in particular, we saw that lazy evaluation converts what seemingly is inefficient breadth first search over the tree of all solutions into a depth first evaluation of the left most solution. So, actually lazy evaluation buys us an efficient to way to execute or rather inefficient strategy.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

**Module # 04**  
**Lecture – 04**  
**Conditional Polymorphism**

Let us investigate polymorphism in Haskell a little more closely.

(Refer Slide Time: 00:07)

The slide has a light gray background with a white rectangular box in the center. The title 'Polymorphism' is at the top in a large blue font. Below it is a bulleted list of text in black. At the bottom of the list are four Haskell function signatures in green, each with some parts underlined in blue. There is a small blue dot character at the end of the list.

- \* Functions that only look at the structure of a list work on lists of any type

```
head :: [a] -> a
length :: [a] -> Int
reverse :: [a] -> [a]
take :: Int -> [a] -> [a]
    .
    .
```

What we have seen is that for functions that only look at the structure of a list, we can make them work on list of any type. So, for instance, the head function takes a list which is non-empty and removes the first element, it does not really care, what the value of the first element is, it just returns it. So, remember we should think of this in terms of boxes.

So, supposing we think of a list as a collection of boxes arranged in a row and somebody, ask you for the first box, you can give it to them without opening it. So, head we say takes a list of any type a and produces a value of type a. Similarly, length just counts the boxes, so it walks down the list, a list of any type a and counts, how many boxes there are and returns an integer. Reverse just rearranges the boxes, so that the first is last and the last is first and finally, take for example, gives a number.

So, take generalizes in a sense the function head, you say not the first box, I want the first k boxes. So, you take a k, then you take a collection of boxes and remove the first k of them, but once again, you do not need to look inside the box.

(Refer Slide Time: 01:15)

## Sorting?

- \* Can we write `[a] -> [a]` as the type for `isort`, `mergesort`, `quicksort`, ...?
- \* Can we sort any type of list?
- \* What about a list of functions  
`[factorial, (+3), (*5)] :: [Int -> Int]`
- \* How do we compare `f < g` for functions?

So, what about the sorting functions which we saw, we saw insertion sort, merge sort, quick sort, can we legitimately say that these functions are of type `[a] -> [a]`. Because, after all that take a list and produce a list and the way that you do, say insertion sort for integers is the same as for float and it is the same for many other types that you can think of.

So, why can we not say this, so the question is, can we sort any type of list, which is what this type would suggest. It would say that quicksort or insertion sort or merge sort will take any list of a and return back a list of the same type. Now, the problem is that, list can contain any uniform type, just as we have seen that, we can pass functions to other functions, so there is no specific limitation on what types of objects we can move around between functions. There is also no limitation on what types of objects, we can put into a list.

So, we can as well construct a list of functions. So, here is a list of functions, all of them have the type `Int -> Int`, they take an integer as an argument and produce some `Int` as a result. So, we have `factorial` which is of this type, we have `(+3)` remember `(+3)` is the function which adds 3 to whatever I give it. We have `(*5)`, which will multiply by 5 whatever I give it. So, this is the legitimate list. Now, how would I say, whether or not the function `factorial` is less than the function `(+3)`, whether `(+3)` is greater than or equal to the function `(*5)`.

(Refer Slide Time: 02:51)

## Type classes

### ord

- \* Want to assign a type as follows  
`quicksort :: [a] -> [a]` provided we can compare values of type `a`
- \* A **type class** is a collection of types with a required property
- \* The type class `Ord` contains all types whose values can be compared

So, what we want to say is that, we want to restrict the type of function like quicksort to all those lists from `[a] -> [a]`, provided we can compare values of type `a`. So, you want to rule out things like these functions, which do not have a sensible way of compare. So, in Haskell, this is captured using a notion called a type class. So, a type class is a collection of types with a required property.

In this case, the type class which is relevant to us is the type class called `Ord`, notice the capital letter `O`. So, type class is usually written with a capital letter and `Ord` contains all types, whose values can be compared, this includes things like integers and other number types like float. It also includes things like Boolean, where `False` is defined to be less than `True`.

We have characters which can be ordered depending on their internal representations. So, whatever value `ord`, the small `ord`, the function on characters return, so this allows us to order the characters and so on. So, capital `Ord` is a set of all types, whose values can be compared.

(Refer Slide Time: 04:03)

## Type classes ...

- \* `Ord t` is a predicate that evaluates to true if type `t` belongs to type class `Ord`
  - \* If `Ord t`, then `<`, `<=`, `>`, `>=`, `==`, `/=` are defined for `t`
- \* We now write  
*conditionally*  
`quicksort :: (Ord a) => [a] -> [a]`
- \* If `a` is in `Ord`, `quicksort` is of type `[a] -> [a]`

So, we can think of capital `Ord` as a predicate which evaluates to true on a type `t`, if `t` belongs to `Ord`, it is not actually a predicate, but let us just think about it this way. So, what internally it means is that, if `t` belongs to the class `Ord`, then the comparison function is `<`, `<=`, `==`, `/=`, etc defined for `t`, which in turn allows us to sort these elements. Because, these are the functions we use to compare values of this type.

So, now, we can write the type of `quicksort` in this way, it is a kind of conditional polymorphism. So, this part says that, it is from `[a] -> [a]`, but is not from any list of `a` to list of `a`, it must be a list, whose elements can be compared. So, there is a new symbol here which you should read as a kind of logical implication. So, this says if the underlying type `a` belongs to `Ord`, then `quicksort` is of type `[a] -> [a]`. So, this is not unconditionally `[a] -> [a]`, this is conditionally `[a] -> [a]` provided `a` is an `Ord` type.

(Refer Slide Time: 05:18)

## What about elem?

```
elem x [] = False
elem x (y:ys)
  | x == y = True
  | otherwise = elem x ys
```

- \* Consider the list  
`funclist = [factorial, (+3), (*5)] :: [Int -> Int]`
- \* How to evaluate `elem f funclist?`

So, this makes sense for sorting, what about the more basic function we have seen the function elem, it checks whether a value belongs to a list or not. So, inductively we said that elem of x with an empty list is always false and if we have a non-empty list, check the first value, if it is equal, then say true, otherwise, check the rest of the list (Refer Time: 05:44)).

So, here what we are really doing is, checking whether a given value is equal to another value. Now, this seems innocuous enough ((Refer Time: 05:53)) it could make sense, seems to check whether two values are the same or not. Again our problem comes, because our values include the so called higher order types, namely functions. So, let us look at our old list that we had before when we had problem sorting. So, now, the question is, what does it mean to ask, whether some other function f belongs to this list or not.

(Refer Slide Time: 06:19)

## Equality

- \* Can we check  $f == g$  for functions?
- \*  $f x == g x$  for all  $x$ ?
  - \* Recall that  $f x$  may not terminate

```
factorial 0 = 1
factorial n = n * factorial (n-1)
factorial (-1)?
```
  - \*  $f == g$  implies for all  $x$ ,  $f x$  terminates iff  $g x$  does

We need to be able to check, whether  $f$  one function is equal to  $g$  another function, is this a reasonable thing to do. So, one way of course to answer this question is to check, whether the value of the function  $f$  is the same as the value of the function  $g$  for every  $x$ . Therefore, in principle, they must be of the same type and they must agree on all the outputs. So, this is sometimes called the extensionality principle.

So, it would say for instance that any correct sorting algorithm is equal to any other correct sorting algorithm, because both of them when given an input list of the same arrangement, will produce an output list with the same arrangement. So, we are ignoring things like efficiency and other characteristics, we are just saying, the input output behaviour of the ((Refer Time: 07:06)) function are same, so can we not do this.

So, other than the minor problem that we have to check this for an infinite set of values, there is also the more serious problem that computations do not always terminate. Remember, our initial light declaration of factorial which ignored the problem of negative inputs. So, if we had a function factorial definition like this, which did not take care of negative inputs carefully, then the value of factorial on something like -1, would degenerate into an infinite computation where it called factorial -2 and then factorial -3 and so on.

And therefore, this recursive step would never terminate. So, we would never get an answer. So, even if you could in some systematic way check,  $f(x)$  equal to  $g(x)$  for all  $x$  at the very least, we need to be able to check on the way that  $f(x)$  is not going to give us a sensible value.

So, maybe we will want to say it, whenever say  $f(-1)$  is not terminating,  $g(-1)$  is not terminating.

So, in particular, we need to be able to check at least this much ,forget about checking the values, we need to at least check that they terminate on the same sets of inputs. So, not just that we are not weakening our condition, we are not checking that  $f x == g x$ , we are just asking does  $f x$  terminate exactly when  $g x$  terminates.

(Refer Slide Time: 08:30)

## Equality ...

- \* Can we write a function
  - `halting :: (a -> b) -> a -> Bool`
  - such that `halting f x` is True iff  $f x$  terminates?
- \* Alan Turing proved such a function cannot be effectively computed
- \* Hence, equality over functions is not computable

So, in other words, we need to write a function such as halting which will take the function, take an input and tell us, yes,  $f$  terminates when evaluated on  $x$ . So,  $\text{halting } f x$  is true, provided  $f x$  terminates false otherwise. So, this is the minimum that we need, in order to even address the issue of how to compare two functions and what Alan Turing proved is that, this function cannot be computed.

So, this is his famous result about the halting problem, there is no algorithm which can determine whether another program and an input to that program halts or not. So, given that we can't even compute, when two functions agree on their terminating inputs, it is clear that we cannot compute any sensible notion of equality over functions. In other words, equal to equal to is not a basic fact for every type in our system.

(Refer Slide Time: 09:28)

The type class Eq

- \* Eq a holds if ==, /= are defined on a
- elem :: (Eq a) => a -> [a] -> Bool
- \* If Eq a, Eq b then Eq [a], Eq (a,b)
- \* Cannot extend Eq a, Eq b to a -> b

So, since equality is not defined on functions, we need to have a type class called Eq, which specifies that a given type has values that can be compared for equality and inequality. So, Eq a holds provided == and /= are defined on values of that type. So, incomes of this type class, we can now give a conditionally polymorphic type for the elem function, which says that provided Eq holds for the input type, then, we can take a value of that type and the list of that type and tell whether or not the value occurs in that list.

So, fortunately all the built in types in Haskell starting with integers, floats, Char, Bool, etc, all support equality and if you have a list of values, each of which can compare to a equality, we can compare the list for equality element to element. We can compare tuples element to element and so on. So, we can extend Eq from the underlying type to structured types.

On the other hand as we have seen, even though we might be able to apply Eq to the input and the output types separately for function, it does not mean that we have Eq defined on the function type, because there is no sensible way to compute equality of functions in general.

(Refer Slide Time: 10:55)

## The type class Ord

- \* `Ord a` holds if `<`, `<=`, `>`, `>=`, `==`, `/=` are defined on `a`
- \* If `Ord a` then `Ord [a]` — lexicographic (dictionary) order
- \* If `Ord a`, `Ord b` then `Ord (a,b)`
- \* Cannot extend `Eq a`, `Eq b` to `a -> b`

In the same way remember that `Ord a` holds, if the comparison functions are defined on values of that type. Now, if we have a type `a` on which we can compare values, then we can compare list of those two values by using lexicographic or dictionary order. So, we just list out the values and look for the left most value which differs and based on the order of this left most differing value which is exactly how you look up words in the dictionary, we decide which list is smaller than the other. In the same way, we can also compare tuples ((Refer Time: 11:34)) but in the same way, we cannot compare functions.

(Refer Slide Time: 11:40)

## The type class Num

- \* Recall the function `sum`

```
sum [] = 0
sum (x:xs) = x + (sum xs)
```
- \* `sum` requires `+` to be defined on list elements
- \* `Num a` says `a` is a number, supports basic arithmetic operations

```
sum :: (Num a) -> [a] -> a
```

Another type, which is very common is the type class Num, which tells us whether arithmetic operations are allowed. So, remember the function sum, which compute sum of the list. So, sum of the list just takes 0 as the sum of negative list and adds up all the values inductively. So, sum requires the function + to be defined on the values of the list. So, Num a says that a is a number that supports basic arithmetic operations.

So, the correct polymorphic type for sum is, if Num of a, then given a [a], it produces the value of the same type. Now, why of the same type, where obviously, if you are adding up integers, you will get an integer, if you are adding up floats, you will get float and so on. So, provided the underlying values can be added, you will get a value of the same type.

(Refer Slide Time: 12:34)

The slide has a title 'Some other type classes' and a bulleted list:

- \* Integral, Frac : subclasses of Num
- \* Show : values that can be displayed
  - \* Requires a function show to display a value
  - \* For example, function types `a -> b` do not belong to Show

So, Num actually has specialized sub classes such as Integral, Frac, etc. So, you can look up some Haskell documentation to see all the various type classes that are built in to Haskell. There is a very important class called Show which tells us, when a value can be actually displayed to the user. So, it requires a function also called show with small s to be defined.

So, again for all the basic types, we have a show function which gives us a representation that we see when we work with ghci. On the other hand, if you ask, Haskell to show what the value of a function is, then obviously, there is no sensible way to describe a function, because remember a function is a black box, it is an input output box.

So, we can't possibly describe a function, in terms of all its input output behaviors and display it to the user, at the same time, there may be many different ways of describing a specific computational implementation of a function. So, there is no sensible way to assign a function type to the type class Show.

(Refer Slide Time: 13:38)

## Signatures

- \* In general, a type class is defined by a **signature**—functions that the types in the class must support
  - \* **Ord** is defined by the signature `<`, `<=`, `>`, `>=`, `==`, `/=`
  - \* **Eq** is defined by the signature `==`, `/=`
  - \* ...
- \* Later we will see how to define our own type classes and add types to a type class

So, what we have seen is that, the way that Haskell deals with type classes is to specify them in terms of signatures. It says that, if you want a type to belong to Ord, you must give sensible definitions for `<`, `<=`, `>`.. By sensible it merely means that you must give a value of the correct type, it does not introduce any meaning to it, you can give a completely unnatural definition of `<` or `<=` and still claim that something belongs to Ord.

It is just that things like sorting will not behave in the expected way. Similarly, any function any type can be made a member of Eq by providing the suitable definition for `=` and `/=` and so on. So, we will see how to do this later on, we will see how to add our own type classes or to add types to a given type class and so on.

(Refer Slide Time: 14:29)

## Summary

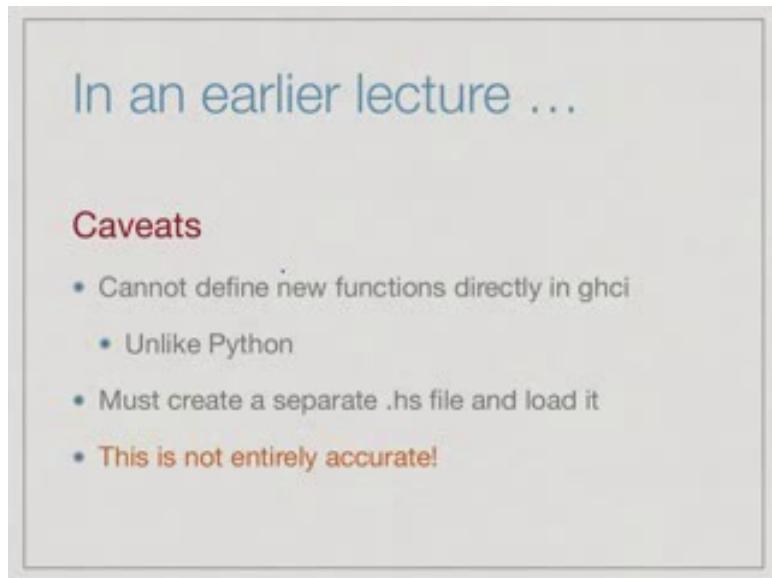
- \* A type class is a collection of types satisfying additional properties over their values
  - \* Check for equality, compare magnitude ...
- \* Additional properties are defined in terms of signatures — additional functions that must be defined on all values of the type
- \* Can use type classes to specify conditionally polymorphic types for functions

So to summarize, a type class is a collection of types, that satisfies additional properties over their values. For instance, the ability to check for equality, to compare values by magnitude and so on and these additional properties are defined in terms of signatures. So, these are additional functions that must be defined on all values of that type in order to use them as part of the type class. And then, once we have type classes, we can give conditionally polymorphic types for functions, such as sorting, which require the underlying values to have a certain property or for elem and so on.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund & S.P Suresh**  
**Chennai Mathematical Institute**

**Module # 04**  
**Lecture – 05**  
**Defining functions within ghci**

(Refer Slide Time: 00:03)



In an earlier lecture ...

**Caveats**

- Cannot define new functions directly in ghci
  - Unlike Python
- Must create a separate .hs file and load it
- **This is not entirely accurate!**

So, in an earlier lecture in the first week when we introduced the interpreter ghci, we said that you cannot define new functions directly in ghci, unlike say in python and you must create a separate Haskell file with extension .hs and load it into ghci. It turns out that this is not entirely accurate.

(Refer Slide Time: 00:24)

## Using let

- In normal Haskell code, `let` is similar to `where`
- $\text{dist } (\text{x1}, \text{y1}) (\text{x2}, \text{y2}) = \sqrt{(\text{diffx} * \text{diffx}) + (\text{diffy} * \text{diffy})}$   
`where`  
`diffx = x2 - x1`  
`diffy = y2 - y1`
- $\text{dist } (\text{x1}, \text{y1}) (\text{x2}, \text{y2}) =$   
`let` `diffx = x2 - x1`  
`diffy = y2 - y1`  
`in` `sqrt (diffx * diffx + diffy * diffy)`

So, in order to use functions, definitions in ghci we need to learn a little bit more about Haskell. So, we have seen the use of the word ‘where’ in order to specify local definitions and functions. So, we can define a function in terms of some local definition and use these definitions in the function. An equivalent way of doing this is to use an expression called ‘let’. So, we can instead of using where, we can say `let diffx = x2 - x1, diffy = y2 - y1` in this definition. So, these are seemingly two equivalent ways to write it, we have looked at where so far, we have not seen let.

(Refer Slide Time: 01:10)

## let vs where

- `let ... in ...` is a Haskell expression, like  
`if ... then ... else ...`
  - Can be used wherever an expression is allowed
- At an introductory level, the distinction between `let` and `where` is minor
  - But they are not equivalent!
- See [https://wiki.haskell.org/Let\\_vs.\\_Where](https://wiki.haskell.org/Let_vs._Where)

So, let \_ in \_ is a Haskell expression, is like if \_ then \_ else \_ which is also Haskell expression. It can be used wherever an expression is allowed. At a level at which we are using Haskell, there is no significant difference between let and where, which is why we have used where so far. The distinction is minor for us, but actually because let is a full fledged Haskell expression and where is not, they are not equivalent in more complicated context as we may see when we go along. If you are curious, you can look up this [https://wiki.haskell.org/Let\\_vs.\\_Where](https://wiki.haskell.org/Let_vs._Where) to find out some ways in which let differs from where.

(Refer Slide Time: 01:49)

## let in ghci

- Within ghci, use let for on-the-fly definitions
- Prelude> let sqr x = x\*x
- Use :{ and :} for a multiline definition
- Prelude> :{
   
Prelude(1) let fact :: Int -> Int
   
Prelude(1) fact n
   
Prelude(1) | n < 1 = 1
   
Prelude(1) | otherwise = n \* fact (n-1)
   
Prelude(1) :}

At this moment what is relevant for us is, that we can use let inside ghci to define functions on the fly. At a basic level we can use let to define single line values, so we can write something like let sqr x = x \* x and now sqr will be available within ghci. So, therefore, it is definitely possible to actually define functions indirectly in ghci and not exactly the way you would do it outside in a Haskell file, but by using let.

Of course, this is a single line definition. What if we want a multiple line definition, like say factorial which is defined separately for 0 and n. So, we can use this notation :{ and :} to enclose a multiple line definition. So, if we say :{, notice that you will find a slight change in the prompt that ghci gives you. Instead of the greater than sign >, it will give you something else perhaps a pipe symbol like this | .

Until you put a close brace and then the next line will return back to the usual form and in between, you can write a multiple line let for example, we can say something like,

let fact:: Int -> Int be the function where fact n, if n < 1 is 1, otherwise, it is n\* fact (n-1).

(Refer Slide Time: 03:08)

```
Last login: Mon Aug  3 13:29:40 on ttys000
mohan@Dolphinair:~$ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package Integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> let sqr x = x*x
Prelude> sqr 8
64
Prelude> sqr 16
256
Prelude> :{
Prelude| let fact :: Int -> Int
Prelude|   fact n
Prelude|     | n < 1 = 1
Prelude|     | otherwise = n * fact (n-1)
Prelude| }
Prelude> fact 7
5040
Prelude> fact (-2)
1
Prelude> []
```

So, here let us actually do this to verify that this works. So, we say ghci, then we can say let sqr x = x\*x . Now if we say sqr 8 for instance, we get 64. We say sqr 16 we get 256 and so on. Now, we want a multiline definition, we can prompt changes. Now, we can say let fact:: Int -> Int be the function given by fact n, such that n < 1 is equal to 1, otherwise, it is equal to n\* fact (n-1) and then, we close this multi line definition, we get back to the old prompt.

Now, we say, what is fact 7, we get 5040, we say what is fact (-2), we get 1 and so on. So, we can use let with the open brace close brace if necessary to define functions in ghci indirectly, not exactly the same way we do it in the Haskell file if we source through the load function. But, effectively we can write functions on the fly.

(Refer Slide Time: 04:21)

## Summary

- Cannot directly define functions within ghci, unlike Python
- However, can use (multiline) let instead

**Acknowledgment**

- Thanks to Oleg Tsybulskyi from Odessa for pointing this out

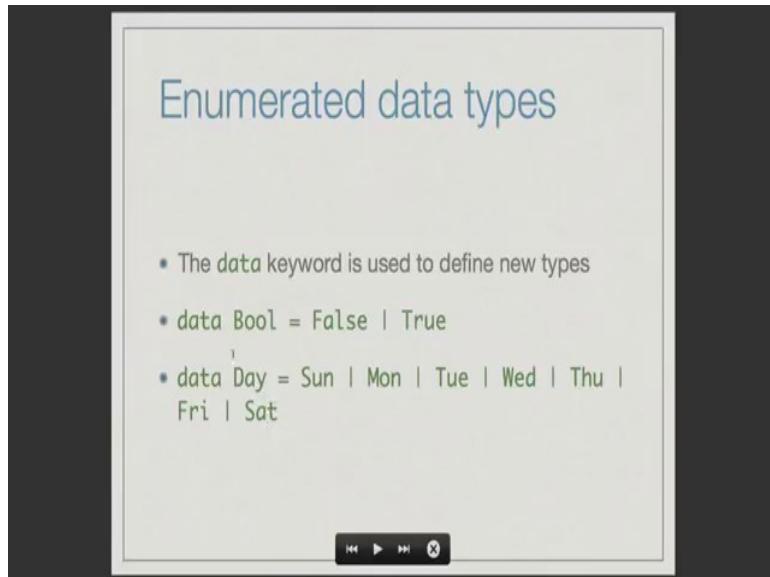
So to summarize, we cannot directly define functions within ghci unlike python. In python the same def command which is used to define functions in a file, is exactly what you use in the interpreter. In Haskell, you have to use let perhaps with this multi line :{ and :} and I would like to thank Oleg Tsybulskiy from Odessa for pointing this out on the discussion forum.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund & S.P Suresh**  
**Chennai Mathematical Institute**

**Module # 05**  
**Lecture - 01**  
**User Defined Data Types**

Welcome to week 5 of the NPTEL course on Functional Programming in Haskell.

(Refer Slide Time: 00:19)



I am S.P. Suresh and I shall be taking over from Professor Madhavan Mukund for the next few weeks. In today's lecture, we shall be looking at user defined data types in Haskell. The simplest way to define new data types in Haskell is the so called enumerated data types, here is an example `data Bool = False | True`. You define the new data type using the `data` keyword and give the name of the type here `Bool`.

Notice the capitalization and then, you enumerate all the values that will be part of the type, in this case it is either `False` or `True`, notice the capitalization again. Another example is `Day`, you declare it by saying `data Day = Sunday | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday`. These are all the possible values of the type `Day`.

(Refer Slide Time: 01:15)

The slide has a light gray background with a dark gray header bar at the top. The title 'Data types with parameters' is centered in a large, dark font. Below the title, there is a list of code snippets:

- `data Shape = Circle Float`  
  | `Square Float`  
  | `Rectangle Float Float`
- `Circle 5.0, Square 4.0, Rectangle 3.0`  
  `4.0`

At the bottom right of the slide area, there is a small navigation bar with four icons: a double-left arrow, a left arrow, a right arrow, and a double-right arrow.

Here is another example, data Shape equals either a Circle or a Square or a Rectangle, but a Circle has a parameter namely the radius and a Square comes with the parameter, namely the length of the side. Similarly, a Rectangle also has a parameter namely the length and breadth. This enables us to declare new data types, which can take infinitely many values, you have Circle 1, Circle 2.0 or Circle 3.0 and so on, one Circle object for every float value. Here are some examples, Circle 5.0, Square 4.0, Rectangle 3.0, 4.0; this is a Rectangle with breadth 3 and length 4.

(Refer Slide Time: 02:15)

The slide has a light gray background with a dark gray header bar at the top. The title 'Functions on data types' is centered in a large, dark font. Below the title, there is a list of code snippets:

- Functions can be defined using pattern matching
- `weekend :: Day -> Bool`  
`weekend Sat = True`  
`weekend Sun = True`  
`weekend _ = False`
- `area :: Shape -> Float`  
`area (Circle r) = pi*r*r`  
`area (Square x) = x*x`  
`area (Rectangle l w) = l*w`  
where  
`pi = 3.1415927`

You can define functions on user defined data types in the usual manner, here are some example functions defined using pattern matching. For instance, you can define what a weekend is. `weekend` is a function from `Day -> Bool` and here is the definition, `weekend` of

Saturday is True, weekend of Sunday is True and weekend of anything else is False. Similarly, you can define an area function, which is a function from Shape to Float, area of a Circle with radius r is  $\pi \cdot r^2$ , area of a Square with length of side x is  $x \cdot x$  and area of a Rectangle with length l and width w is  $l \cdot w$ . Are there other ways to define functions on user defined data types, yes.

(Refer Slide Time: 03:11)

Functions on data types

- What about  
weekend2 :: Day -> Bool
- weekend2 d
  - | (d == Sat || d == Sun) = True
  - | otherwise = False
- Error - No instance for (Eq Day) arising from a use of '=='

For instance, here is another way to define the weekend function called weekend2, the definition is here. Weekend2 of d is True if d is equal to Saturday or d is equal to Sunday and it is equal to False otherwise. But, if you enter this program as it is in ghci, you will get an error, the error message will be something like this. No instance for (Eq Day) arising from a use of the equality operator; let us see how to fix this later.

(Refer Slide Time: 03:49)

Functions on data types

- How about this function?
- nextday :: Day -> Day
  - nextday Sun = Mon
  - nextday Mon = Tue
  - ...
  - nextday Sat = Sun
- Invoke nextday Fri in ghci will lead to error
- Error - No instance for (Show Day) arising from a use of 'print'

Here is another function you can write, which is a function from Day -> Day, it gives the nextday. For instance, nextday Sun = Mon, nextday Mon = Tue and so on, nextday Sat = Sun. Now, if you load this function in ghci and invoke nextday Fri, it will again lead to an error and it is the following error. No instance for (Show Day) arising from a use of print, we will see what this means and how to fix this in a later slide.

(Refer Slide Time: 04:26)

Add data types to typeclasses

- To check equality of two values of a data type a, a must belong the type class Eq
- We add Day to the type class Eq as follows
- `data Day = Sun | Mon | ... | Sat deriving Eq`
- Default behaviour: `Sun == Sun, Tue /= Fri, ...`
- Now `weekday2` compiles without error

To check equality of two values of a data type a, a must be declared to belong to the type class Eq. We have seen the notion of a type class in earlier lectures and we have also seen the type class Eq in detail. So, to get weekday2 to work we need to add Day to the type class Eq and we add Day to the type class Eq as follows, `data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat deriving Eq`, the keyword is 'deriving'.

When we declare Day to derive from equality, the equality operator has the default behavior. Sunday ==Sunday, Tuesday /= Friday, Monday == Monday, etcetera. Once you declare Day to be deriving from Eq, then weekday2 compiles without error.

(Refer Slide Time: 05:32)

The type class Show

- To make nextday work, we must make Day an instance of Show
- `data Day = Sun | Mon | ... | Sat`  
deriving (Eq, Show)
- The type class Show consists of all data types that implement the function show

To make the nextday function work, we must make Day an instance of the type class called Show with a capital S. We have to declare it as follows, `data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat` deriving (Eq, Show). The type class Show consists of all data types that implement the function show with a small s.

(Refer Slide Time: 05:59)

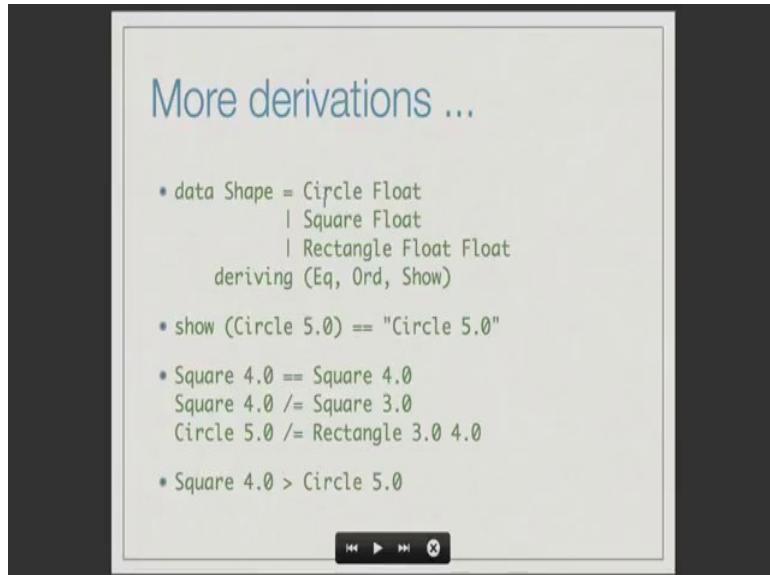
More derivations

- show converts its input to a string which can be printed on the screen
- Default text representation
- `show Wed == "Wed"`
- `data Day = Sun | Mon | ... | Sat`  
deriving (Eq, Show, Ord)
- `Sun < Mon < ... < Sat`

The function show converts its input to a string which can be printed on the screen, we need not define it explicitly for the data type Day. If we do not give an explicit definition, there is a default definition that Haskell provides, which is just to give a default text representation for the data value. For instance, `show Wed` is just the string “Wed”, we can also derive Day as an instance of the type class Ord, Ord which is an ordinal type.

When we do this, an order is defined on the data type Day as follows, Sun < Mon < ... < Sat and this order is determined by the order in which the data values are enumerated in the data type declaration.

(Refer Slide Time: 07:01)



We can also derive Shape to belong to various type classes, we say `data Shape = Circle Float | Square Float | Rectangle Float Float deriving (Eq, Ord, Show)`. Now, you can use all these functions on data on values of type Shape. For instance, `Show` `Circle 5.0` will just be the string that say `Circle 5.0`, `Square 4.0 == Square 4.0`, the equality check is derived from the equality on floating point numbers as well as the names, `Square` or `Circle` or `Rectangle`.

For instance, `Square 4.0 == Square 4.0`, because both the parameters and the name here are equal, `Square 4.0 /= Square 3.0`. Because, even though the names are equal the parameters are different, `Circle 5.0 /= Rectangle 3.0, 4.0`, because there are two different types of shape, one is a `Circle` and other is a `Rectangle` and we have also derived it to belong to the type class `Ord`. So, there is an order defined on shapes, `Square 4.0` is for instance greater than `Circle 5.0`, because `Square` comes later in the declaration than `Circle`.

(Refer Slide Time: 08:37)

## Constructors

- Square, Circle, Sun, Mon, ... are constructors
- They are functions
  - Sun :: Day
  - Rectangle :: Float -> Float -> Shape
  - Circle :: Float -> Shape

The names Square, Circle, Sun, Mon, etcetera that we have used are called constructors. They are nothing but, functions; Sunday for instance is a function that is of type Day, Sun:: Day. It is a function that accepts no input, but produces an output. Rectangle is a function with two parameters, so it takes two floats as inputs and it produces a shape as output. So, Rectangle is a function whose type is Float -> Float ->Shape, similarly Circle is a function whose type is Float -> Shape.

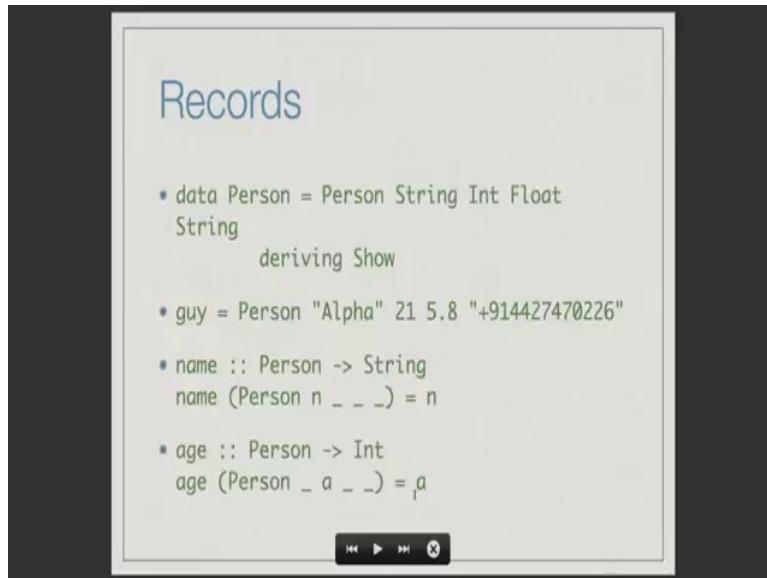
(Refer Slide Time: 09:21)

## Constructors ...

- Constructors can be used just like other functions
- Circle 5.0 :: Shape
- map Circle :: [Float] -> [Shape]
- map Circle [3.0, 2.0] = [Circle 3.0, Circle 2.0]

These constructors can be used just like any other function, for instance Circle can be invoked on the input 5.0 to give a Shape. You can map the Circle function over a list of Floats to get a list of Shapes. For instance, map Circle on the list [3.0,2.0] which will give you the list consisting of [Circle 3.0, Circle 2.0].

(Refer Slide Time: 09:49)



The screenshot shows a code editor window titled "Records". The code defines a data type Person and a constructor guy, along with two functions name and age.

```
* data Person = Person String Int Float String
  deriving Show

* guy = Person "Alpha" 21 5.8 "+914427470226"

* name :: Person -> String
  name (Person n _ _ _) = n

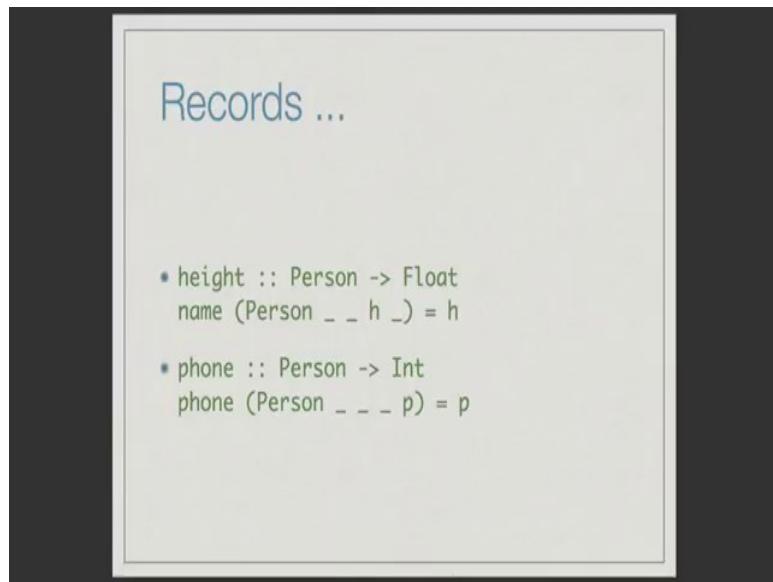
* age :: Person -> Int
  age (Person _ a _ _) = a
```

Here is another way of defining types, `data Person = Person String Int Float String`. Now, you will see two occurrences of the word `Person` here, one on the left one on the right. On the left, it denotes the name of the type; on the right it is the name of the constructor. Now, unlike in a type like `Shape` where we had three constructors `Circle`, `Square`, `Rectangle`, here we have only one constructor, though we have many parameters.

The convention in Haskell is that, if a data type has only one constructor then you use the same name for both the type and the constructor. So, the `Person` that appears on the right is a constructor and the `Person` that appears on the left is the name of the data type. So, here we say that `data Person = Person String Int Float String`, the intention is that the first string is the name of the person, this int here is the age of the person, the float here is let us say the height of the person and the last string here is the phone number.

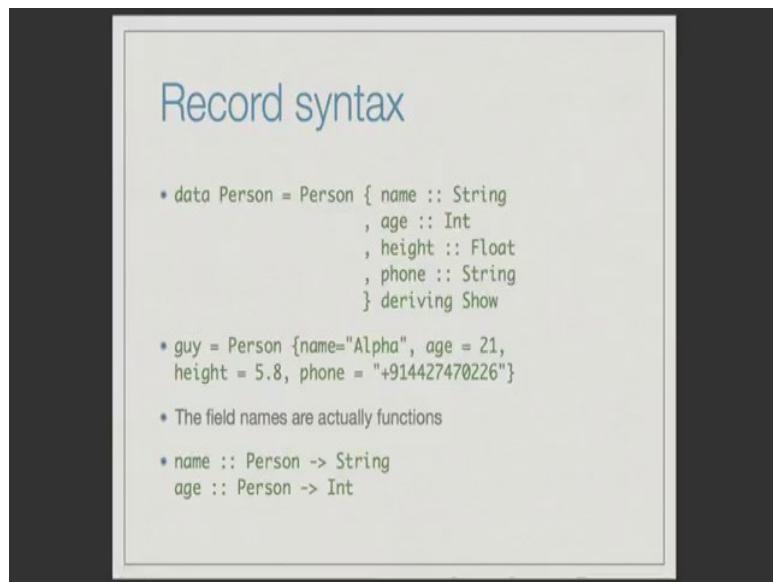
So, for instance I might say `guy = Person "Alpha" 21, 5.8` and some phone number. How do you extract the name of a `Person` object? You write a function `name :: Person -> String`, whose input is `Person` and the output is `String` and the definition is this. `Name, person n, any age, any height and any phone number equals n` written as `name (Person n _ _ _) = n`, here is a function that extracts the age of a person, `age :: Person -> Int`, it is a function from `Person` to `Int` and the definition is `age (Person _ a _ _) = a`.

(Refer Slide Time: 11:45)



You can write a height function which says height (Person \_ \_ h \_) = h and here is a function that accepts the phone number of a person; phone (Person \_ \_ \_ p) = p. There is a pattern matching and there is a don't care pattern in all these definitions, but this kind of definition is quite cumbersome. So, Haskell offers an alternative easier syntax which is also familiar from other languages.

(Refer Slide Time: 12:21)



You can define the Person type as for this : data Person = Person {name::String, age :: Int, height :: Float, phone :: String} deriving Show. data Person equals Person and within braces you say name which is of type String, age which is of type Int, height which is of type Float and phone which is of type String. So, you are naming all the fields directly in the data definition itself and these names here, name, age, height, phone etc are really nothing but,

the functions that we defined earlier. Name is a function from Person -> String, age is a function from Person -> Int.

You declare new objects of type person in this syntax as follows, guy = Person {name = "Alpha", age= 21, height =5.8, phone ="+914427470226"} guy equals Person, within braces you say name equals alpha, age equals 21, height equals 5.8 and phone equals phone number. This is an easier syntax that Haskell offers.

(Refer Slide Time: 13:24)

The slide has a light gray background with a dark gray border. At the top, the word 'Summary' is written in a blue font. Below it is a bulleted list of six items, each preceded by a small blue square bullet point. The list describes various Haskell data types and their characteristics:

- The keyword `data` is used to declare new data types
- The keyword `deriving` to derive as an instance of a type class<sup>1</sup>
- Data types with parameters - `Shape`, `Person`
- Sum type or union - `Day`, `Shape`
- Product type or struct - `Person`

At the bottom right of the slide area, there are four small black icons: a double left arrow, a single left arrow, a single right arrow, and a double right arrow.

To summarize, you have seen simple ways of defining new data types, the keyword `data` is used to declare new data types. The key word `deriving` is used to derive the data type as an instance of a type class and typically, the standard functions that are supposed to be defined on types of the type class are defined by default. For instance `Eq`, `Ord`, `Show` etc, then you can also define data types with parameters as in the example of `Shape`, `Person`, etc.

You have two different types of user defined data types, one you might call the Sum type or union type, where there are multiple constructors on the right. The example here is `Day` or `Shape` or the other is the Product type or the struct type that you might be familiar from other languages, where you have only one constructor on the right, the example of this is a `Person`.

**Functional Programming in Haskell**  
**Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

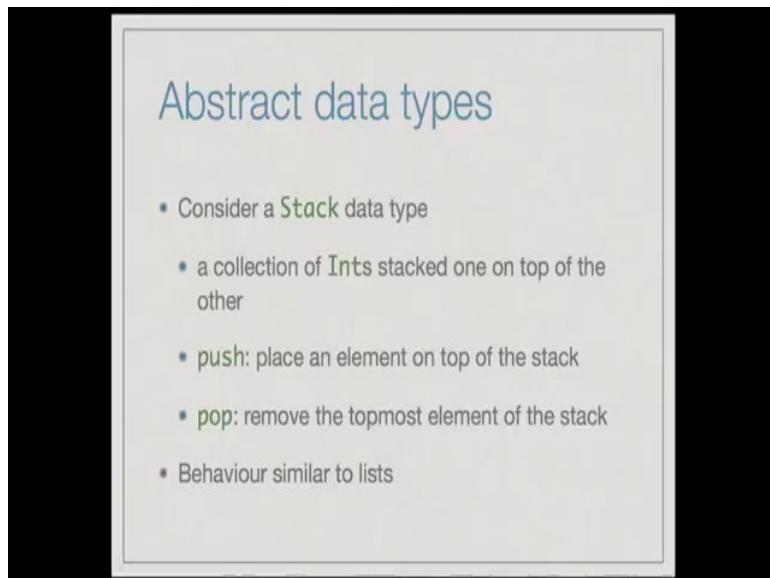
**Module # 05**

**Lecture - 02**

**Abstract Data Types**

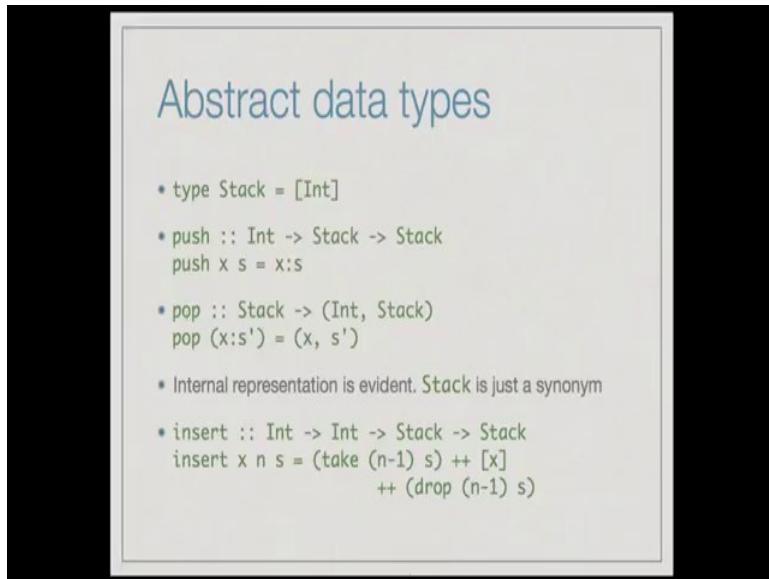
In this lecture, we shall be introducing Abstract Data Types in Haskell.

(Refer Slide Time: 00:07)



Consider the example of a stack data type, a stack is a collection of integers, stacked one on top of the other. It supports two operations push and pop, push places an element on top of the stack and pop removes the top most element of the stack. We can see that this behavior is similar to lists, the push corresponds to adding a new element at the head of a list by using the colon operator, pop corresponds to retrieving both the head and tail of the list.

(Refer Slide Time: 00:47)



With this in mind, here is a definition of stack, we define stack as a type synonym for list of integers, type Stack = [Int]. Now, the push function can be defined as follows, push is a function which takes two arguments an integer and a stack as inputs and outputs a stack. push x s = x:s. The head of the list is the top of the stack, pop is a function that takes a stack as input and produces an Int and a stack as output. The Int is the top element of the stack and the stack that is output is the stack after the top element has been removed.

The definition of pop is as follows, pop (x:s') = (x, s'). So, with this definition the internal representation of stack is very evident, stack is just a synonym for list of Ints. The drawback is that this allows one to write functions that uses the internal representation. For instance, here is a function insert that inserts an element at the nth position from the top of the stack, insert x n s, the x is an element to be inserted, n is the position and s is the stack and the output is also a stack, insert x n s = ( take (n-1) s ) ++ [x] ++ ( drop (n-1) s ).

You see that here the internal representation of the stack, namely that it is just a list of integers is used. One does not always want to allow such a use, for a stack data type you would want to allow only the push and pop operations and perhaps to check if the stack is empty.

(Refer Slide Time: 02:51)

## Abstract data types

- `data Stack = Stack [Int]`
- The value constructor `Stack` is a function that converts a list of `Int` to a `Stack` object
- Internal representation hidden

This motivates the definition of stack as an abstract data type hiding the internal representation. We can define `Stack` as a user defined data type, `data Stack = Stack [Int]`. Recall that the `Stack` on the left is the name of the type, the new type that we are defining and the `Stack` on the right is the so called value constructor. The `Stack` on the right is just a function from the list of `Int` to `Stack`, the value constructor `Stack` is a function that converts a list of `Int` to a `Stack` object. The internal representation is hidden, you can access a `Stack` only through its constructor.

(Refer Slide Time: 03:44)

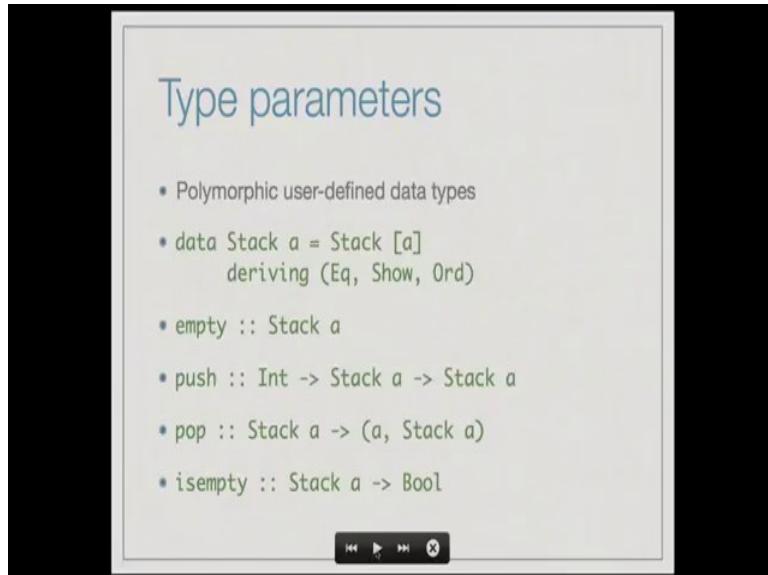
## Abstract data types

- `empty :: Stack`  
`empty = Stack []`
- `push :: Int -> Stack -> Stack`  
`push x (Stack xs) = Stack (x:xs)`
- `pop :: Stack -> (Int, Stack)`  
`pop (Stack (x:xs)) = (x, Stack xs)`
- `isempty :: Stack -> Bool`  
`isempty (Stack []) = True`  
`isempty (Stack _) = False`

Now, you can define the following functions on this new data type, `empty` which returns an empty stack and the definition is `empty = Stack []`, `push` which takes an int and a stack and returns the stack, `push x (Stack xs) = Stack (x:xs)`. Here we use pattern matching on the user

defined data type, which is something familiar to us. Pop is a function that takes a stack and returns an int and a stack, pop (Stack (x:xs)) = (x, Stack xs), isempty is the function that checks if the Stack is empty, it is a function from Stack -> Bool, isempty (Stack []) = True, isempty (Stack \_) = False.

(Refer Slide Time: 04:40)



The Stack data type that we defined earlier was very specific in the sense that it stored only integer values. We might want to implement a Stack that stores data of any type whatsoever. This is achieved by using polymorphic user defined data types, which use type parameters. The definition is as follows, `data Stack a = Stack [a]`, recall that the Stack on the left is a type name and the Stack on the right is a constructor.

When we use type parameters, the functions on the right are called value constructors and the name of the type itself is called a type constructor. Because, for every instantiation of the type `a`, you get a new type `Stack a`. For instance, if `a` is `Int` you will get a Stack of integers, if `a` is `Float` you will get a Stack of Floats etc. So, the Stack on the right is called a value constructor and the Stack on the left is called a type constructor.

In a polymorphic type, the value constructor is nothing but, a polymorphic function which takes a list of `a` as input and produces an object of type `Stack a` as output. In this particular case we are also deriving `Eq`, `Show`, `Ord`, etc for this data type. We can define the following functions again, `empty` which is a function that just returns a stack of type `a`, `push` which is a function that takes an `Int` and a `Stack a` as input and produces a `Stack a` as output.

`Pop` which is a function that takes a `Stack a` as input and produces the pair `(a , Stack a)`, one

value of type a and a stack which is of type Stack a as output, isempty is a function that takes a stack as input and produces a Bool as output. The function definitions are exactly the same as given in the previous slide, except now that the types are more general.

(Refer Slide Time: 07:00)

Type parameters...

- Suppose we want to sum all elements in a stack
- `sumStack (Stack xs) = sum xs`
- What is the type of `sumStack`?
- Applicable only if the stack has numeric elements
- `sumStack :: (Num a) => Stack a -> a`

Sometimes functions on these polymorphic data types will not be completely polymorphic, but only conditionally polymorphic. Here is an example, suppose we want to sum all elements in a stack you would achieve it as follows, `sumStack (Stack xs) = sum xs`, the function `sum` applied to `xs`, the function `sum` adds all the elements in the input list. What is the type of `sumStack`? Notice that, the type of `sum`, `sum` is a polymorphic function which is conditionally polymorphic, its type is `(Num a) => Stack a -> a`. Therefore, the `sumStack` function is applicable only if the stack has numeric elements, in other words the `sumStack` function has type `(Num a) implies Stack a -> a`.

(Refer Slide Time: 08:01)

## A custom show

- `show (Stack [1,2,3]) == "Stack [1,2,3]"`
- deriving `Show` defines a default implementation for `show`
- Suppose we want something mildly fancy
- `show (Stack [1,2,3]) == "1->2->3"`

Earlier we said we can derive `show`, we can derive `Stack` as a type which belongs to the class `show`. But, this defines the default implementation for `show`, `show (Stack [1,2,3])` is just this string which says “`Stack [1,2,3]`”. Suppose you want something fancier, let say we want to say `show (Stack [1,2,3]) == “1 -> 2 -> 3”`. The string “`1 -> 2 -> 3`” means that you would have to define our own custom `show` function.

(Refer Slide Time: 08:51)

## A custom show

- One can change the default behaviour
- ```
printElems :: (Show a) => [a] -> String
printElems [] = ""
printElems [x] = show x
printElems (x:xs) = show x ++ "->" ++
printElems xs
```
- `instance (Show a) => Show (Stack a) where`  
 `show (Stack l) = printElems l`

One can change the default behavior of `show` as follows, `instance (Show a) => Show (Stack a)` where `show (Stack l) = printElems l`. We say `instance (Show a)` implies `Show (Stack a)`, this declaration means that `Stack a` is an instance of the type class `Show` provided `a` is an instance of the type class `show`, where the new definition of `show` the small `s` `show` that we are defining is `show (Stack l) = printElems l`. The `printElems` function is given here,

`printElems` is the function from list `a` to `String` provided `a` belongs to the type class `Show`, `printElems` of the empty list is just the empty string, `printElems` of the singleton list containing `x` is just `show` of `x`, `printElems` of `x:xs`, is `show x ++ "->" ++ printElems xs`.

Notice the recursive call here, in this manner we can define custom implementations of the functions that are provided by type classes. If we just say deriving capital `Show` you would get a default implementation of the `Show` function. But, we are always allowed to redefine the `Show` function in which case we allow to declare our data type to be an instance of the type class `Show`.

(Refer Slide Time: 10:30)

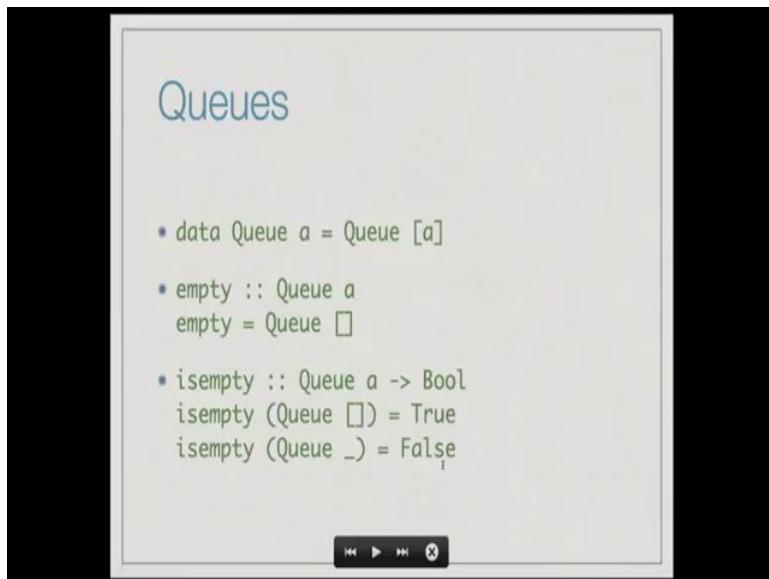
The slide has a light gray background with a dark blue header bar at the top. The title 'Queues' is centered in the header. Below the title is a bulleted list of properties:

- Consider a Queue data type
  - a collection of `Ints` arranged in a sequence
  - `enqueue`: add an element at the end of the queue
  - `pqp`: remove the element at the start of the queue

At the bottom of the slide is a dark blue footer bar with white icons for navigation: back, forward, and search.

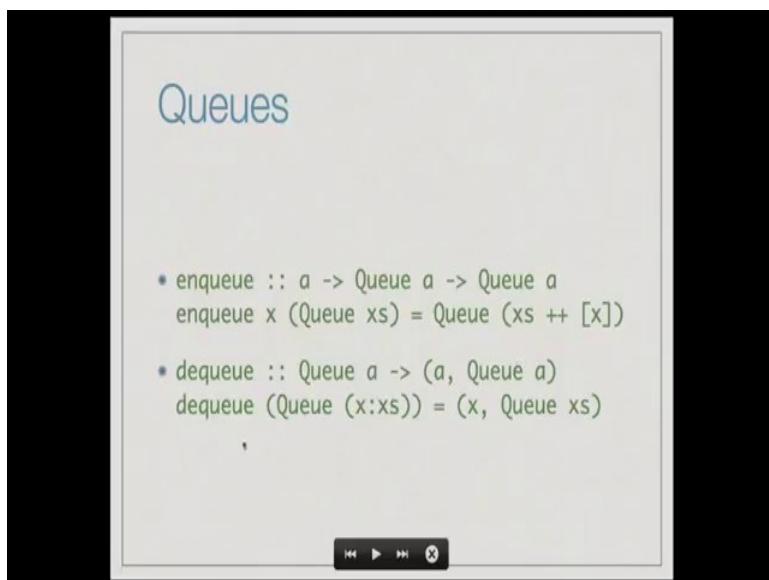
Let us consider another data type queue, a queue is a collection of integers arranged in a sequence, enqueue adds an element at the end of the queue, dequeue removes the element at the start of the queue.

(Refer Slide Time: 10:48)



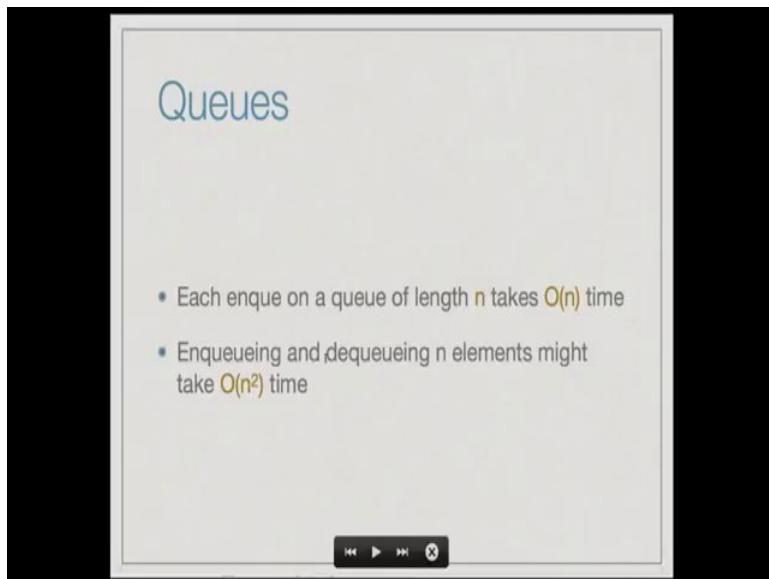
Here is the definition, `data Queue a = Queue [a]`, recall again that `Queue` is the type constructor and the `Queue` on the right is a value constructor, which is a function from `[a]` to the type `Queue a`. The empty queue is just given by `Queue []`, `isempty` is a function from `Queue a -> Bool`, `isempty (Queue []) = True`, `isempty (Queue _) = False`.

(Refer Slide Time: 11:24)



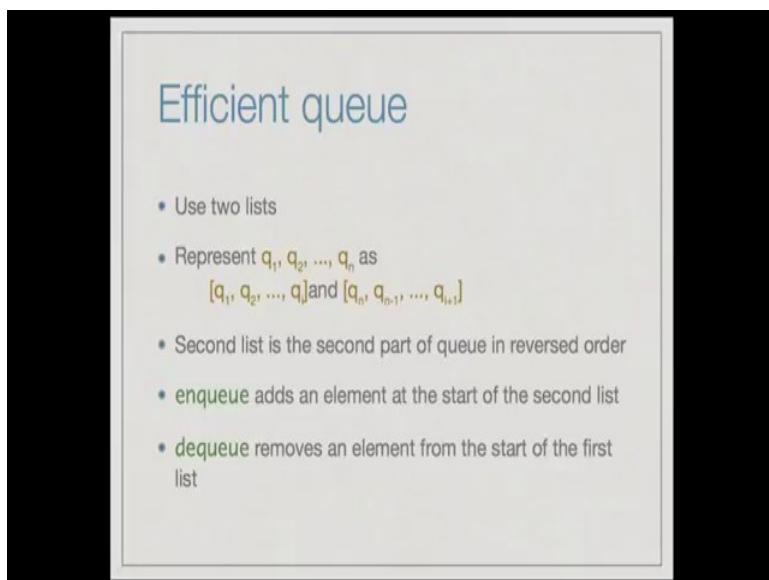
`enqueue` adds an element to the end of the list, so `enqueue x (Queue xs) = Queue (xs ++ [x])`, `dequeue` is a function that takes a `Queue` as input and produces an element of type `a` and a `Queue a` object as output. `dequeue (Queue (x:xs)) = (x, Queue xs)`.

(Refer Slide Time: 12:01)



In this implementation each enqueue on a queue of length  $n$  takes  $O(n)$  time, because we are adding the element at the end of the list. So, enqueueing and dequeuing  $n$  elements might take  $O(n^2)$  time depending on the operations that we use, this is the worst case time that could be consumed by a sequence of  $n$  enqueue and dequeue operations.

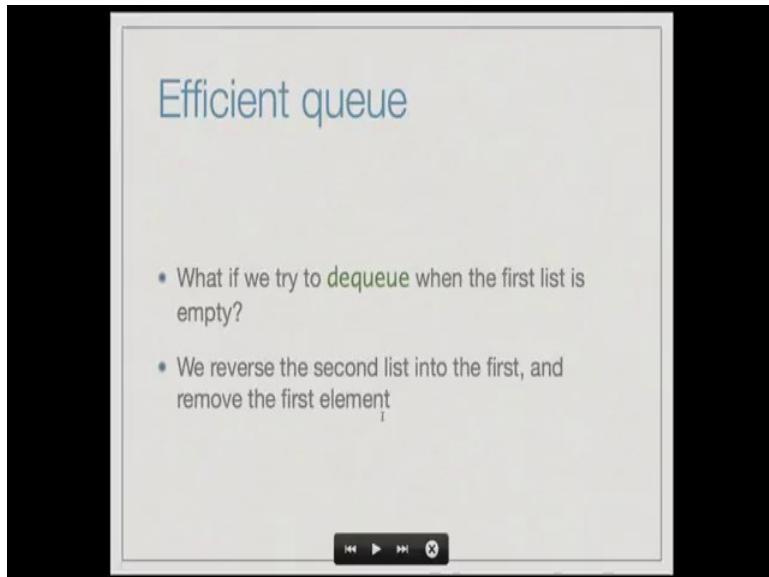
(Refer Slide Time: 12:31)



Here is a more efficient implementation of a queue. we use two lists and represent the queue consisting of elements  $[q_1, q_2, \dots, q_n]$  in this order as two lists, the first list consisting of some elements in this order  $[q_1, q_2, \dots, q_i]$ . And the second list consisting of the remaining elements in the reverse order  $[q_n, q_{n-1}, \dots, q_{i+1}]$ . The second list is the second part of the queue after  $q_1$  to  $q_i$  in reversed order, Now enqueue can be made to add an element at the start of the second list and dequeue removes an element from the start of the first list. Recall

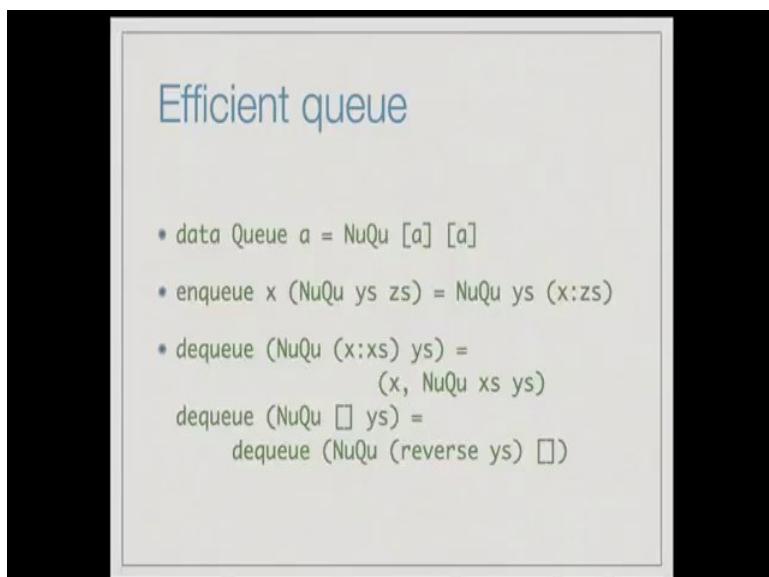
that adding an element at the start of a list is a much simpler operation than adding an element at the end of the list.

(Refer Slide Time: 13:28)



Now, there is a problem if we try to dequeue from a queue, where the first list is empty, if the queue itself is empty then we cannot dequeue from it, but if the queue is non empty, but the first list is empty in our representation then we need to reverse the second list into the first and remove the first element.

(Refer Slide Time: 13:55)



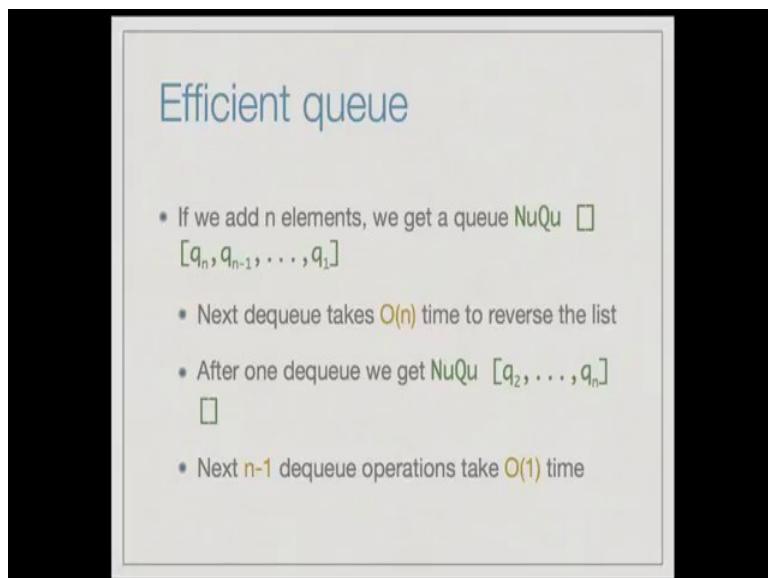
This leads us to the following implementation, `data Queue a = NuQu [a] [a]`. `NuQu` is the new constructor, typically the convention is that the value constructor has the same name as the

type constructor or type name. But, here just to distinguish this from the earlier implementation we call it NuQu, and it is a constructor which takes two lists as arguments. So, data Queue a = NuQu [a] [a], now the enqueue function as we described earlier adds an element to the start of the second list. enqueue x (NuQu ys zs) = NuQu ys (x:zs), the first list is retained as it is and x is added to the front of the second list x:zs.

dequeue (NuQu (x:xs) ys) this is the case, where the first list is non empty is nothing but, (x, NuQu xs ys) . We have removed the first element from the first list and retained the second list as it is. dequeue (NuQu [] ys ) this is the case where the first list is empty, what we do is reverse the second list in to the first position and take the second list to be the empty list and dequeue from here. So, dequeue (NuQu [] ys ) is nothing but, dequeue (NuQu (reverse ys) [])

This is seemingly a better implementation, because every time we enqueue we add to the beginning of the second list, but there are times when you have to reverse the second list in to the first list, namely when the first list is empty, how much does this cost.

(Refer Slide Time: 15:56)



If we add n elements to the queue we get NuQu [] [qn, q n - 1, ..., q1]. The next dequeue takes  $O(n)$  time to reverse the list qn to q1 into the first list and if we dequeue once now we get NuQu [q2, .., qn] [], the beauty is that the next n-1 dequeue operations take only  $O(1)$  time, we paid  $O(n)$  time when we did this dequeue operation. But, we were repaid by having to spend only constant time on the next n-1 dequeue operations, because all we have to do is just remove the element from the front of the first list therefore, on average we will still be

consuming  $O(n)$  time.

(Refer Slide Time: 16:55)

The slide has a light gray background with a dark border. At the top, the title 'Amortized analysis' is written in a blue, sans-serif font. Below the title is a bulleted list of six items, each preceded by a small blue asterisk. The list discusses various touch counts for elements in a queue. At the bottom of the slide is a dark footer bar containing four small white icons: a double-left arrow, a right arrow, a double-right arrow, and a close (X) button.

- How many times is an element touched?
  - Once when it is added to the second list
  - Twice when it is moved from the second to first
  - Once when it is removed from the first list
  - Each element is touched at most four times
  - Any sequence of  $n$  instructions involves at most  $n$  elements
  - So any sequence of  $n$  instructions takes only  $O(n)$  steps

This is made precise by something called amortized analysis, here we precisely count how much time it takes to complete a sequence of operations, rather than a single operation. Even though a single dequeue may take as much as  $O(n)$  time, when we consider a sequence of  $n$  instructions this story is different, it is not that we take  $O(n^2)$  time for completing a sequence of  $n$  instructions. In fact, we only take order  $n$  steps to complete a sequence of  $n$  instructions.

And the way to show it is as follows, look at how many times an element has touched once it enters the queue, it is touched once when it is added to the second list as part of the enqueue and it is touched twice when it is moved from the second queue to the first queue. This is when we do a dequeue operation when the first list is empty and this element is touched once when it is removed from the first list, this is when we dequeue from the first list when this element was at the head. So, therefore, each element is touched at most four times, any sequence of  $n$  instructions involves at most  $n$  elements and therefore, any sequence of  $n$  instructions takes only  $O(n)$  steps.

(Refer Slide Time: 18:24)

The slide has a light gray background with a dark gray header bar at the top. The title 'Summary' is centered in the header in a blue font. Below the title is a bulleted list of six items, each preceded by a small blue square. At the bottom of the slide is a dark gray footer bar with four small white icons: a double left arrow, a right arrow, a double right arrow, and a close button.

- Abstract data types
- We can define polymorphic user-defined data types by supplying type parameters
- Conditional polymorphism for functions defined on such data
- The `instance` keyword to define non-default implementation of functions
- Efficient queues and amortized analysis

To summarize, we defined abstract data types in this lecture through the example of the stack and a queue. We have also defined polymorphic user defined data types by showing how you can generalize this stack and queue data types to store not just integers, but any data type `a`, whatsoever by the use of supplying type parameters. The functions on these data types are polymorphic, but sometimes they are conditionally polymorphic, not freely polymorphic.

For instance, for the function `sumStack` etcetera, because the function makes sense only if the type parameter satisfies certain properties, like being a numeric data type for instance. We have shown that we can use the `instance` key word to define non default implementations of functions like `show`. Finally, we looked at an example of how we can implement queues more efficiently and analyzed the efficiency of this new implementation by using the technique of amortized analysis.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

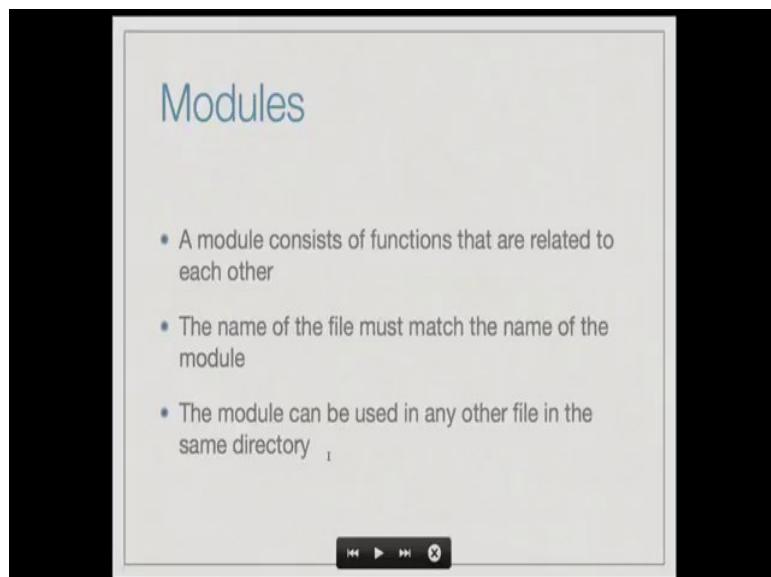
**Module # 05**

**Lecture – 03**

**Modules**

In this lecture, we shall introduce another abstraction device namely Modules.

(Refer Slide Time: 00:08)



A module consists of functions that are related to each other and defined in a single file. The name of the file must match the name of the module, the module can be used in any other file in the same directory by using the command import as we shall see later.

(Refer Slide Time: 00:29)

The slide has a light gray background with a dark gray header bar. The title 'Queue module' is in a blue sans-serif font at the top left. Below the title is a bulleted list of text and code snippets:

- \* The Queue module, defined in Queue.hs
- \* 

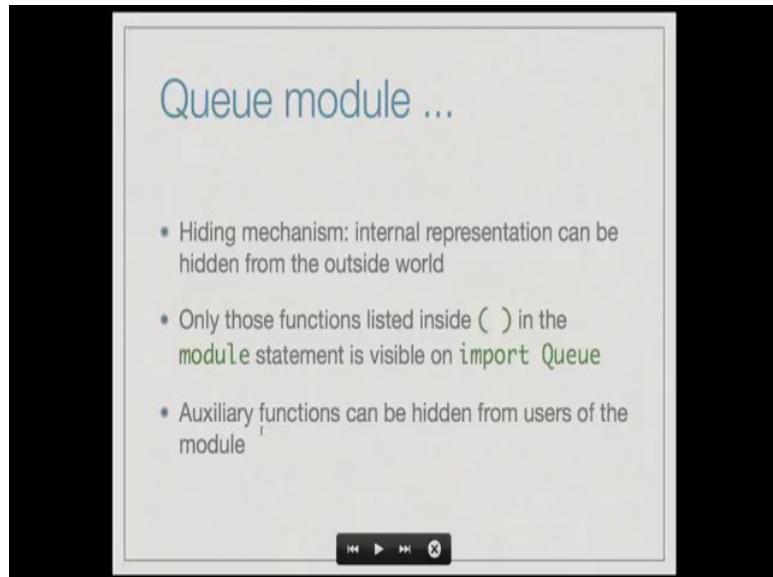
```
module Queue(NuQu(..), empty, isEmpty,
enqueue, dequeue, show) where
data Queue a = NuQu [a] [a]
empty = ...
isEmpty = ...
enqueue = ...
```
- \* We can use the Queue module in another file in the same directory by adding the line `import Queue`

At the bottom right of the slide area is a small control bar with four icons: back, forward, and close.

Here is an example of a Queue module, the Queue module is defined in the file Queue.hs. The first line is as follows, `module Queue( NuQu(..), empty, isEmpty, enqueue, dequeue, show) where` data Queue a = NuQu [a] [a]... module Queue within parenthesis the names of many functions, ‘where’ and then you give the definition of functions and data types as usual., module Queue, within parenthesis, I have given the name of the constructor NuQu, notice the special syntax you have to say NuQu followed by parenthesis and within parenthesis you put two dots.

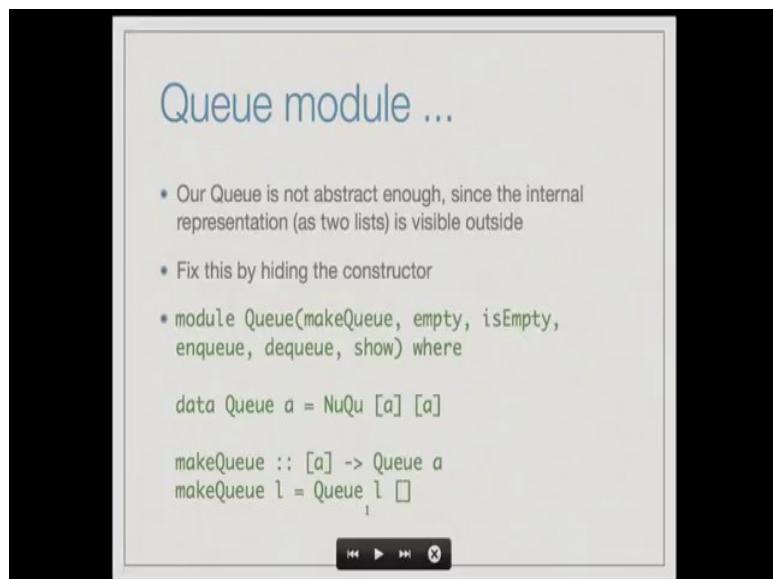
And then, these are some other functions that I want to be visible outside this file, empty, isEmpty, enqueue, dequeue, show etc. The definitions of empty, isEmpty, enqueue, etc were already been presented in the last lecture. We can use the Queue module in another file in the same directory by adding the line `import Queue`.

(Refer Slide Time: 01:34)



A module acts as a hiding mechanism, the internal representation of the data types can be hidden from the outside world. Only those functions listed inside the parenthesis in the module statement are visible to another file, when it does not import Queue. This means that auxiliary functions can be hidden from users of the module.

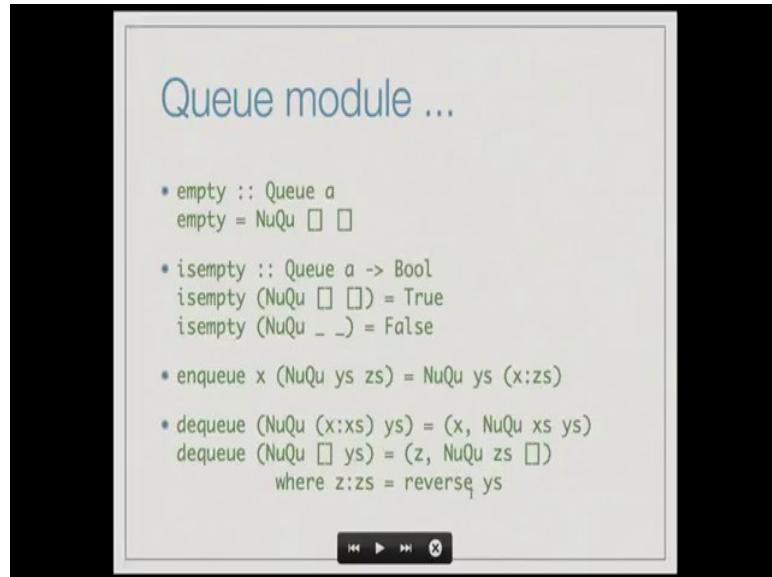
(Refer Slide Time: 02:03)



Let us look our Queue module in more detail, our Queue is not abstract enough. Since, the internal representation is visible outside, recall that the queue is represented in terms of two lists, one half of the list and the reverse of the rest of the list. We can fix this by hiding the constructor, in this new declaration the NuQu function is not exported. Instead we have a makeQueue function, which takes any list of elements of type a, [a] and produces a Queue a.

The new declaration is as follows, module Queue(makeQueue, empty, isEmpty, enqueue, dequeue, show) where data Queue a = NuQu [a] [a] and we have a new function makeQueue, which takes a lists of objects of type a and returns the Queue a, makeQueue of 1 is nothing but, Queue 1 empty list.

(Refer Slide Time: 03:10)



The other functions are as before, empty just returns a Queue, which is empty NuQu [] [], isempty will check whether a queue is empty, isempty (NuQu [] []) = True, isempty NuQu of anything else is false : isempty (NuQu \_ \_) = False.  
enqueue x ( NuQu ys and zs) = NuQu ys (x:zs) .

Recall that we add x to the head of the second list, dequeue (NuQu (x:xs) ys) is nothing but, (x, NuQu xs ys), recall that in a dequeue we remove the element from the head of the first list.

And if the first list is empty, then we reverse the second list into the first list and extract the first element. So, dequeue (NuQu [] ys) = ( z, NuQu zs [] ) where z:zs = reverse ys.

(Refer Slide Time: 04:21)

The slide has a light gray background with a dark gray header bar. The title 'Queue module ...' is centered in the header. Below the title is a bulleted list of points. The last point contains Haskell code for the 'show' instance of the 'Queue' type class.

- \* One can add instance declarations inside a module
- \* 

```
instance (Show a) => Show (Queue a) where
    show (NuQu xs ys) =
        show "[" ++ printElems (xs ++ reverse ys)
        ++ "]")
```
- \* 

```
printElems :: (Show a) => [a] -> String
printElems [] = ""
printElems [x] = show x
printElems (x:xs) = show x ++ "->" ++
    printElems xs
```

One can also add instance declarations inside a module, here we see that as before we are declaring Queue a to be an instance of the type class Show provided a itself is a member of the type class show. instance(Show a) implies Show(Queue a) where the show function on NuQu xs and ys is just show of some fancy characters “[“ ++ print all the elements of the queue, printElems (xs ++ reverse ys) ++ “]”].

So, in our representation you have a brace followed by a square bracket followed by all the elements listed in order, followed by closing the brackets and the brace. PrintElems is a function from list a to string, [a] -> String, provided a is of type Show, a belongs to the type class Show. PrintElems of empty list [] is the empty string “”, printElems of the singleton x is nothing but show x, printElems of x:xs is nothing but, show x ++ “->” ++ printElems xs.

in this case we have an arrow (Refer Slide Time: 05:44)

## Using the Queue module

- One uses the Queue module by adding `import Queue` at the start of a file (before defining any functions)
- The constructor `NuQu` and the function `printElems` are not available outside of `Queue.hs`
- One creates new queues by invoking the `makeQueue` function  
`newq = makeQueue [1..100]`

So, one uses the Queue module by adding `import queue` at the start of a file, before defining any functions in that file. When we do this, the constructor `NuQu` and the function `printElems` are not available outside of `Queue.hs`. The constructor `NuQu` we are not making available externally, because we want to hide the internal representation and `printElems` is just some helper function that we need in order to define `show`.

So, therefore, there is no need for it to be visible outside the queue module. One creates new queues by invoking the `makeQueue` function, for instance you might say `newq = makeQueue [1..100]`.

(Refer Slide Time: 06:32)

## A Stack module

```
* module Stack(Stack(), empty, push, pop, isEmpty,
  show) where

  Stack a = Stack [a]

  empty = Stack []

  push x (Stack xs) = Stack (x:xs)

  pop (Stack (x:xs)) = (x, Stack xs)

  isEmpty (Stack []) = True
  isEmpty (Stack _) = False
```

Here is another example a stack module, module `stack`, `stack` there are to be two dots here, `stack...` Here is, this is the data constructor and we have `empty`, `push`, `pop`, `isEmpty`, `show`,

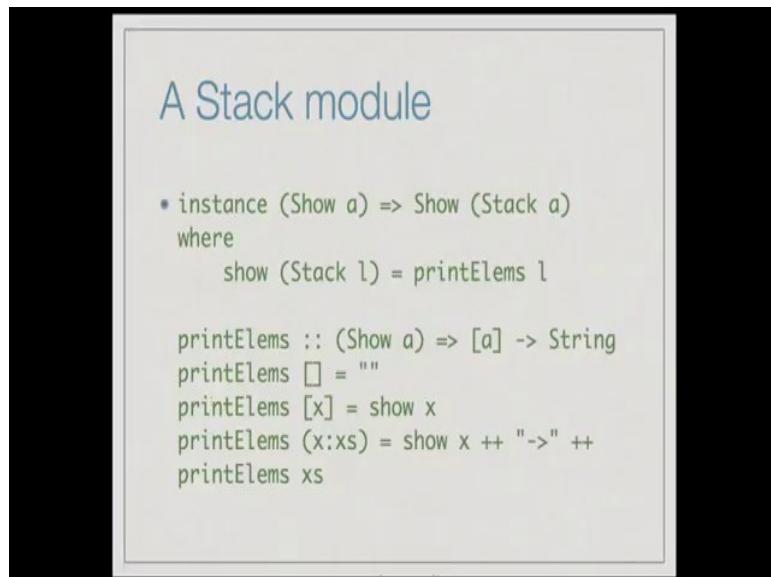
etc. We are exporting all these functions and the definitions are as before.

Stack a = Stack [a], Stack on the right hand side is the value constructor and then you have a list a. The empty , push and pop functions are as defined below.

empty = Stack [], push x (Stack xs) = Stack (x:xs), pop (Stack (x : xs)) =(x, Stack xs)

isempty checks whether a stack is empty, if on stack empty list, isempty(Stack []) = True, it will return true, for a stack anything else , isempty (Stack \_) = False , it will return false.

(Refer Slide Time: 07:26)



Similar, to queue we can also add instance declaration inside the stack module.

instance (Show a) => Show (Stack a), where show(Stack l) = printElems l, just like in the queue example. Here again we are printing the elements with arrows in between, this is exactly the same as the printElems function that we had earlier.

(Refer Slide Time: 07:52)

The slide has a light gray background with a dark gray header bar. The title 'Postfix expressions' is centered in the header. Below the title, there are four bullet points:

- A postfix expression is an arithmetic where the operator appears after the operands
- No parentheses required in a postfix expression
- $3 \ 5 \ 8 \ * \ + = (3 \ (5 \ 8 \ *)) \ + = 43$
- $2 \ 3 \ + \ 7 \ 2 \ - = ((2 \ 3 \ +) \ (7 \ 2 \ +)) \ - = -4$

At the bottom right of the slide area, there is a small navigation bar with icons for back, forward, and close.

Let us say an extended example of using stacks, in this example we will try to compute the value of postfix expressions. A postfix expression is an arithmetic expression, where the operator appears after the operands, importantly no parenthesis are required in a postfix expression. Here, is an example  $3 \ 5 \ 8 \ * \ +$  is a postfix expression and the way to interpret it as that this star occurs after two numbers.

So, we have to take this as a sub expression this is the sub expression  $5 \ 8 \ *$ , which stands for  $5*8$ . Then, we have a  $+$  following a string of numbers and operators, this is to be interpreted as the plus operator applied on the previous two expressions, the expression previous to  $+$  is  $5 \ 8 \ *$  and the expression previous that is  $3$ . So, this whole thing stands for  $3+ (5 * 8) = 43$ , here is another example  $2 \ 3 \ + \ 7 \ 2 \ -$ , so this  $+$  occurs after two numbers.

So, this is to be taken as one expression,  $2 \ 3 \ +$  which stands for  $2+3$ .  $7 \ 2 \ -$  this stands for  $7+2$ , which is  $9$  and then there is a minus this stands for the there are two expressions preceding the minus. So, it is the first expression minus the second expression. So,  $2 \ 3 \ +$  is an expression  $7 \ 2 \ -$  is an expression and this expression followed by that expression minus is another expression and it stands for  $(2+3) - (7+2)$  namely  $-4$ .

(Refer Slide Time: 09:55)

The slide has a light gray background with a dark border. At the top, the title "Postfix expressions" is written in a blue font. Below the title is a bulleted list of rules:

- \* Every bracket-free expression can be converted uniquely to a bracketed one
- \* Scan from the left
  - \* If it is a number, it is a standalone expression
  - \* If it is an operator, bracket it with the previous two expressions
- \*  $3 \ 5 \ 8 \ * \ + = (3 \ (5 \ 8 \ *)) \ + = 43$
- \*  $2 \ 3 \ + \ 7 \ 2 \ + \ - = ((2 \ 3 \ +) \ (7 \ 2 \ +)) \ - = -4$

Every bracket free expression can be converted uniquely to a bracketed one that is the specialty of postfix expressions, the way to do it is to scan from left, if it is a number then it is a standalone expression, if it is an operator you bracket it with the previous two expressions. So, 3 is a standalone expression, 5 is a standalone expression, 8 is a standalone expression you encounter a \* and this creates a new expression, which is got by applying by bracketing this with the previous two expressions namely 5 and 8.

So,  $(5 \ 8 \ *)$  becomes a new expression. When you reach + you have two expressions before it, namely 3 and  $(5 \ 8 \ *)$ , so you bracket this plus along with this two. Similarly, in  $2 \ 3 \ + \ 7 \ 2 \ + \ -$ , 2 is a standalone expression, 3 is a standalone expression, when you encounter the plus it creates a new expression, which is the previous two expressions bracketed along with plus. So, you get  $(2 \ 3 \ +)$  you move on, 7 is a standalone expression, 2 is a standalone expression. When you encounter the plus  $(7 \ 2 \ +)$  becomes an expression, when you encounter the minus here it is to be grouped with the previous two expressions, which is  $(2 \ 3 \ +)$  and  $(7 \ 2 \ +)$ . So, ultimately you get a value of -4.

(Refer Slide Time: 11:18)

Evaluating postfix expressions

- Follow the bracketing algorithm and use a stack
- Scan from the left
  - If it is a number, push it onto the stack
  - If it is an operator
    - remove the top two elements of the stack
    - apply the operator on them
    - push the result onto the stack

We shall now consider, how we can automatically evaluate postfix expressions. You can just follow in the bracketing algorithm and use a stack, here is how it proceeds. Scan from the left. if it is a number it is a standalone expression, so push it on to the stack, if it is an operator, we have to apply the operator to the previous two expressions and the strategy we follow is that we push all expressions on to the stack.

So, we remove the top two elements of the stack, which correspond to the two expressions that this operator has to be bracketed with, you apply the operator on the two expressions and push the result onto the stack. This is the overall strategy. Now, we will have to use a stack module and implement this algorithm.

(Refer Slide Time: 12:15)

A calculator program

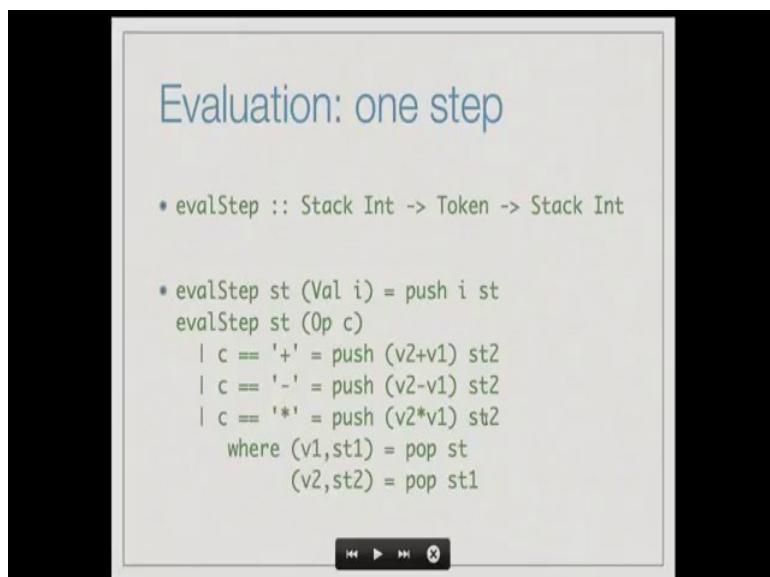
- A postfix expression is a list of integers and operators
- We represent it as a list of tokens
- import Stack

```
data Token = Val Int | Op Char  
type Expr = [Token]
```

Here is how we would program this calculator, to calculate the value of postfix expressions. A postfix expression is given as a list of integers and operators. So, in this a list of a mixed type, we need to create a new data type, we will call these tokens. A token is either an integer or an operator, here is the definition. `data Token = Val Int | Op Char`, it is either an integer given by the data constructor Val or it is an operator.

And we denote the fact that is an operator by using the data constructor Op, if it is an integer it has a parameter, which is of type Int. If it is an operator it has a parameter, which is of type Char. And, importantly we need to import stack before we program this calculator and we can define a type synonym for list of tokens, type Expr = [Token]. So, an expression for us is just a list of tokens.

(Refer Slide Time: 13:27)



Recall that our strategy for evaluating an expression is to read the expressions from left to right and when we encounter a number to push it on to a stack and when we encounter an operator to apply the operator on the top two elements of the stack. This leads us to the function evalStep, which is one step in this computation. evalStep is a function that takes a stack of integers and a token as input and produces a stack of integers as output. evalStep st (Val i) = push i st.

Recall that Val is a data constructor that indicates that the token that we have encountered is an integer and the integer is given by i, which we push on to the stack. evalStep st (Op c) recall that Op is a data constructor that indicates that the token that we have encountered is a operator. And, the operator happens to be c and for simplicity we will assume that c is either

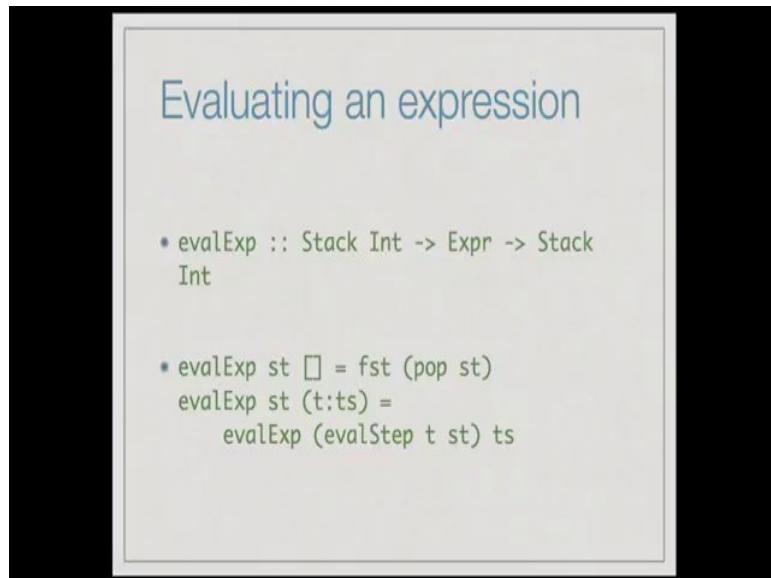
+ or - or \*. the definition is as follows evalstep st (Op c) is push (v2+v1) on to st2, if c == '+'.

Here v1 is the value on top of the stack given by pop st and v2 is the value that is the second from the top of the stack, which is given by pop st1. recall that pop is a function that returns a pair, the top element to the stack and the stack that you obtain from removing the top element from the stack. So, st1 is, what you get if you remove the top element from st and st2 is, what you get if you remove the top element from st1. So, v1 and v2 are the top two elements in the stack and st2 is the stack that you obtain after you are remove the top two elements.

So, you remove the top two elements from the stack, add them and push them on to the stack, so push( $v_2 + v_1$ ) on to s2. If c is minus then you would push ( $v_2 - v_1$ ) on to the stack, recall that if you encounter something of the form 3 5 -, what you intend is 3 - 5 and when you encounter 3, 3 would have been pushed on to the stack and when you encounter 5, 5 would have been pushed on top of 3.

So, the second element from the top is v2 and you have to subtract the top element from the second element of the stack. So, you have to push ( $v_2 - v_1$ ) onto st 2, if c equals \* then you do push of ( $v_2 * v_1$ ) onto st2.

(Refer Slide Time: 16:23)

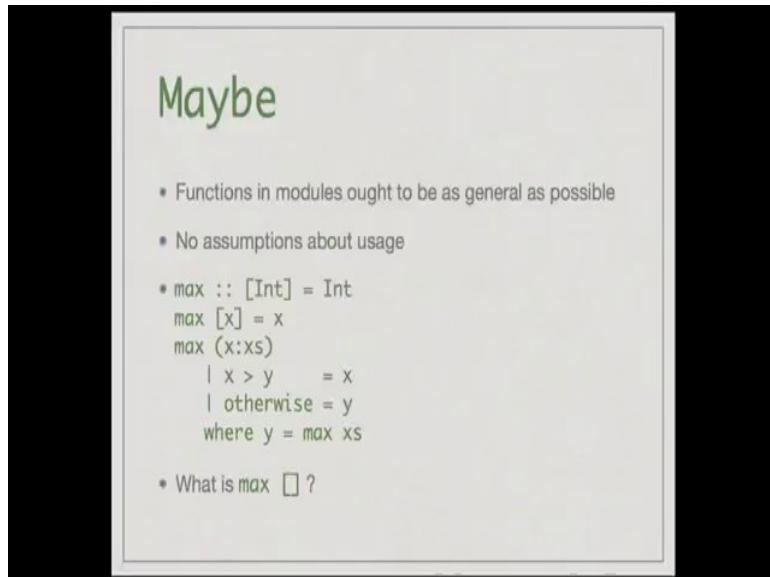


Once you have defined evalStep, we can actually evaluate an expression. `evalExp :: Stack Int -> Expr -> Stack Int`, `evalExp` is a function from Stack of Int to Expr to Stack of int. So, it is a function that takes two arguments as input, a stack of integers and an expression and it produces a stack of integers, typically the stack of integers that an expression takes is the empty stack. So, `evalExp st []` this denotes that there is nothing to be done with the

expression is `pop st`, but recall that `pop st` returns a the top of the stack and the remainder of the stack.

And first of `pop st` will be the top element of the stack. `evalExp st (t:ts)`, `t` is a token followed by another bunch of tokens, `evalExp (evalStep t st) ts`. So, this `evalStep` of `t st` modifies this state of the stack `st` by taking into consideration one token `t`, and what we do is we evaluate the rest of the expression namely `ts` on the modified stack. So, this is how we define the calculator module based on the stack module.

(Refer Slide Time: 18:07)



After having seen a non-trivial example, we shall consider some more features on modules. One thing to remember in modules is that the functions that we provide in modules ought to be as general as possible. We should not make any assumptions about how the functions would be used by other modules that import this module. Here is an example, consider the function `max` which tries to find the maximum of a list of integers. `max :: [Int] = Int`. `max` is a function from list of `Int` to `Int`.

So, `max` of the singleton list `[x]` is just `x`. `max` of `(x:xs)` is `x` if `x > y`, otherwise is equal to `y`, where `y` is nothing but, `max` of `xs`. So, this `max` should not be confused with the built in function `max`, which takes two integers as arguments and produces the larger of the two integers. we use the name `max` here for simplicity. But, in this definition what is the case that we have to use for `max` of empty list, `max []` ?

(Refer Slide Time: 19:28)

The slide has a title 'Maybe' at the top. Below it, under the heading '\* Option 1:', there is a code snippet:

```
* max :: [Int] = Int
max [] = -1i
max [x] = x
max (x:xs)
| x > y     = x
| otherwise = y
where y = max xs
```

Below the code, there is a note: '\* -1 is a default, works if the input list contains only nonnegative integers'. At the bottom right of the slide area, there are navigation icons.

One option is to define max of empty list to be some default value like -1, -1 is a default value and it works if the input list contains only non-negative integers. Then, this -1 serves to indicate that the list that was input was empty. But, what if the input list could also contain negative numbers, then this option does not work you might have to provide a different default.

(Refer Slide Time: 20:03)

The slide has a title 'Maybe' at the top. Below it, under the heading '\* Option 2:', there is a code snippet:

```
* max :: [Int] = Int
max [] = error "Empty list"
max [x] = x
max (x:xs)
| x > y     = x
| otherwise = y
where y = max xs
```

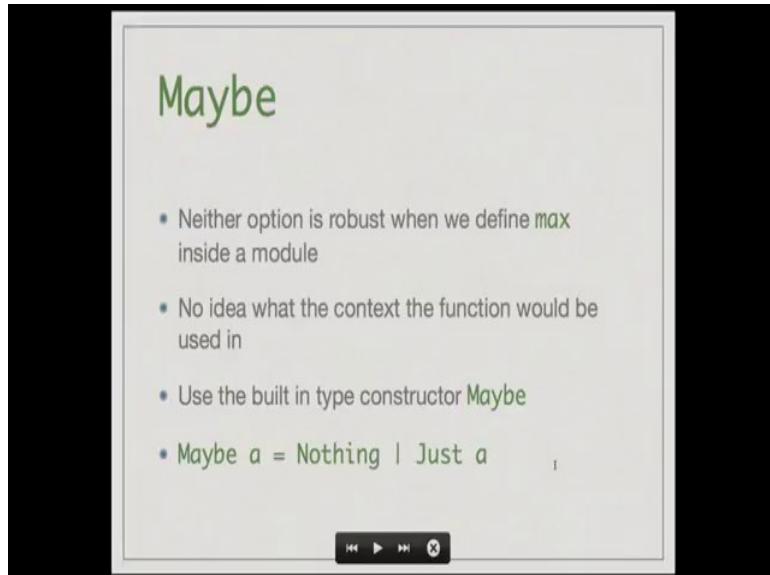
Below the code, there is a note: '\* error :: [Char] -> a is a function that prints the error message supplied and causes an exception'. Underneath that, there is another note: '\* Aborts execution'.

Another option which is slightly more robust is to define max of empty list to be error "Empty list". error is a function that is of this type String -> a, any type a, whatsoever. It is a function that takes an error message as a parameter and it just causes an exception. it prints the error message out on screen and aborts the execution. So, this is slightly more robust, than

what we had earlier, because if we default to -1, then if your input list does contain negative numbers, then you do not know whether a -1 was returned.

And because the list was empty or because, -1 was actually the maximum of the list. But even this is not very desirable. Because, the error function actually aborts the execution.

(Refer Slide Time: 21:11)



We want something that works on most inputs and it also does not abort execution. The point is that we have no idea what the context is in which the function would be used. In these cases one can actually use the built in type constructor `Maybe`. `Maybe` is defined as follows. `Maybe a = Nothing | Just a`. So, no matter what type `a` is, for instance if `a` is `Int` we can build a type `Maybe Int`, which is given by nothing or just `Int`.

(Refer Slide Time: 22:03)

The slide has a light gray background with a dark border. At the top center, the word "Maybe" is written in a large, green, sans-serif font. Below it is a code block in a smaller, monospaced green font.

```
* max :: [Int] = Maybe Int    -- inside a module
max [] = Nothing
max [x] = Just x
max (x:xs)
| x > y    = Just x
| otherwise = Just y
where Just y = max xs

* printmax :: [Int] -> String  -- outside the module
printmax l = case (max l) of
    Nothing -> show "Empty list"
    Just x -> show "Maximum = " ++ show x
```

Here is how we use it. Here we define max to be a function from list of Int to Maybe Int, this is inside the module. max [] = Nothing; max [x] = Just x; max (x:xs) is if max of xs is Just y, and x> y, then max (x:xs) is Just x, otherwise it is Just y. This is how we would define this max function to return a Maybe Int rather than just an Int and outside the module we might have a routine like printmax, which takes a list of integers as input and produces a String as output. printmax of l is in case (max l ) of Nothing then show “Empty list”.

In case it is Just x then you show “Maximum = “ ++ show x. this is a scenario, where we leave it to the user of the module to use the function in the appropriate manner. So, therefore, we should neither return a default value assuming the type of usage of the function nor should we abort. Here we see that even in the case when the list is empty the print max routine would like to print something meaningful rather than just abort.

(Refer Slide Time: 23:36)

Maybe

- Consider a table datatype that stores key-value pairs
- type Key = Int  
type Value = String  
type Table = [(Key, Value)]
- myLookup :: Key -> Table -> Maybe Value
- looks up the value corresponding to key in table, if key occurs in table

◀ ▶ ⌂ X

Here is another example of the use of Maybe. consider a table data type that stores (Key, Value) pairs. type Key is just integer, let us say and type Value is just String, let us say. So, type Table is just list of (Key, Value) pairs. We want to define a function myLookup, which is a function of type Key -> Table -> Maybe value. What it does is to look up the value corresponding to the key in table, if key occurs in table. if key occurs in table it will print the corresponding value otherwise it will return nothing.

(Refer Slide Time: 24:16)

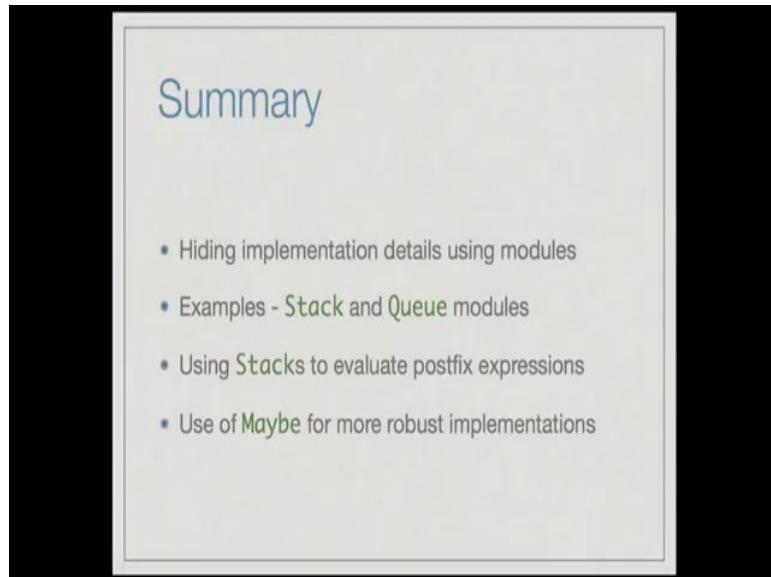
Maybe

- \* myLookup :: Key -> Table -> Maybe Value
  - myLookup k [] = Nothing
  - myLookup k ((k1,v1):kvs)
    - | k == k1 = Just v1
    - | otherwise = myLookup k kvs
- \* Built-in function
  - lookup :: Eq a =>  
a -> [(a,b)] -> Maybe b
- \* More robust than returning error or some default value on absence of key

Here is the definition. myLookup of k empty list is Nothing. myLookup of k and ((k1,v1):kvs) is if k equals k1 you return just v1. Otherwise you do a myLookup of k in kvs in the remainder of the table. Actually, this behavior is given by the built in function lookup

whose type is `Eq a` implies `a -> [(a,b)] -> Maybe b`. this is more robust than returning error or some default value on the absence of the key.

(Refer Slide Time: 25:02)



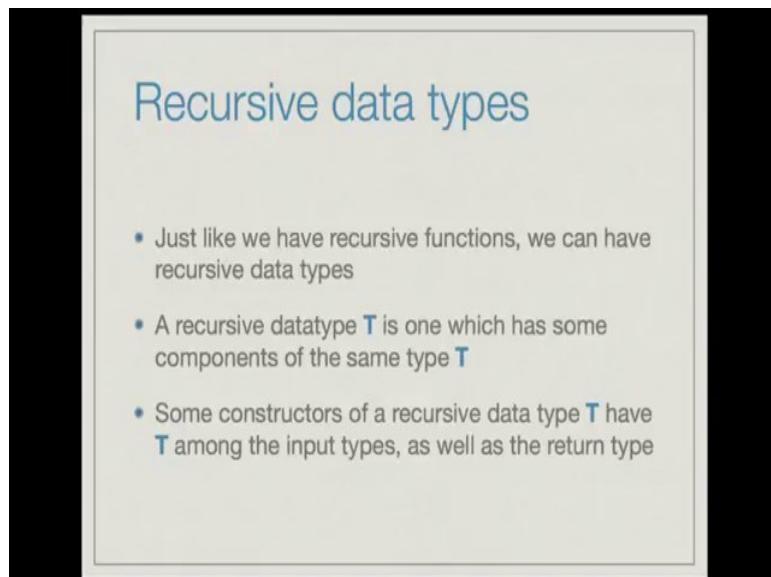
To summarize, we have introduced modules and shown how we can hide implementation details using modules. We have seen the examples of Stack and Queue modules , we have seen how to use the Stack module to do some non trivial computation namely to evaluate postfix expressions. We have also shown the use of Maybe for more robust implementations especially, when we define functions inside the modules.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

**Module # 06**  
**Lecture – 01**  
**Recursive Data Types**

Welcome to week six of the NPTEL course on Functional Programming in Haskell. In this lecture, we shall be introducing Recursive Data Types.

(Refer Slide Time: 00:11)



**Recursive data types**

- Just like we have recursive functions, we can have recursive data types
- A recursive datatype  $T$  is one which has some components of the same type  $T$
- Some constructors of a recursive data type  $T$  have  $T$  among the input types, as well as the return type

Just like we have recursive functions in Haskell, we can also have recursive data types. A recursive data type  $T$  is one which has some components that are of the same type  $T$ , this means that some constructors of a recursive data type  $T$  have  $T$  among the input types as well as the return type.

(Refer Slide Time: 00:35)

The slide has a light gray background with a dark gray header bar. The title 'First example: Nat' is in blue at the top left. Below the title is a list of bullet points in green text:

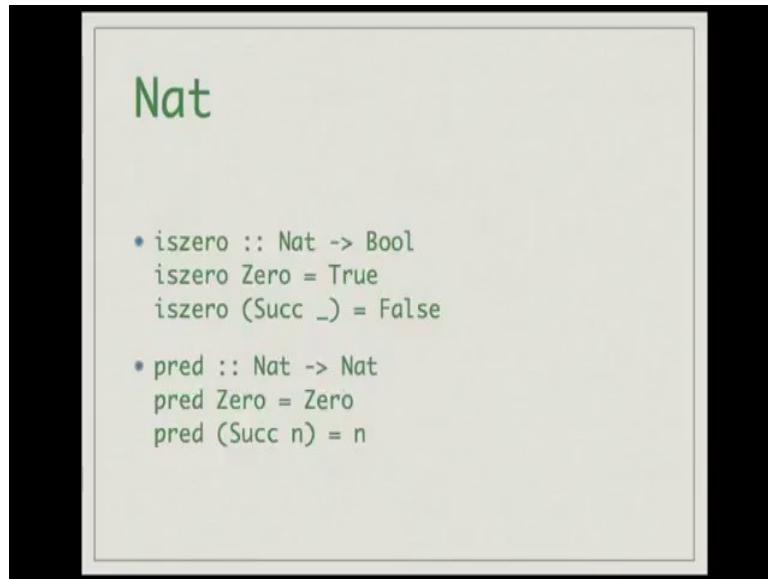
- Simplest example is `Nat`
- `data Nat = Zero | Succ Nat`
- `Zero :: Nat`
- `Succ :: Nat -> Nat`

At the bottom right of the slide area, there is a small black navigation bar with four icons: back, forward, first, and last.

Here is a simple example of a recursive data type that of natural numbers. We know that Haskell offers a built-in type of Int for integers but these represent both positive as well as negative numbers. In this example, we want to represent the non negative numbers that is 0, 1, 2, 3, etcetera; here is how we could do it. Data Nat = Zero | Succ Nat, this is the data declaration. We see that Nat is the type constructor or the name of the type and Zero and Succ are the value constructors.

Zero is a value constructor that takes no argument at all whereas, Succ is a value constructor that takes a natural number as argument or an object of type Nat as argument and returns an object of type Nat. As you can see, Zero is of type Nat; Succ is a function from Nat -> Nat.

(Refer Slide Time: 01:47)



We could now define functions on the type Nat by using pattern matching as usual. For instance, here is a function that checks whether an object of type Nat is Zero or not. isZero is the function from Nat  $\rightarrow$  Bool, isZero Zero = True , isZero of Zero is True. isZero (Succ \_) = False , isZero of Succ of anything is False.

Here is a function that computes the predecessor of a natural number, by convention the predecessor of Zero is taken to be Zero, since we do not allow non negative numbers. So, pred is a function from Nat  $\rightarrow$  Nat, pred Zero = Zero, pred (Succ n) =n, we could have other similar definitions.

(Refer Slide Time: 02:37)

The slide has a title 'Nat' at the top. Below it, there are two bullet points defining functions:

- `plus :: Nat -> Nat -> Nat`  
`plus m Zero = m`  
`plus m (Succ n) = Succ (plus m n)`
- `mult :: Nat -> Nat -> Nat`  
`mult m Zero = Zero`  
`mult m (Succ n) = plus ((mult m n) m)`

At the bottom right of the slide area, there is a small navigation bar with icons for back, forward, and exit.

For instance, here is a definition of the plus function which takes two Nat as inputs and produces a Nat as an output. This function is defined using recursion. `plus m Zero = m`, `plus m (Succ n) = Succ (plus m n)`. Recall that `plus m n` returns a Nat as a result and `Succ`, recall is the constructor that takes a Nat and produces a Nat. So, `Succ (plus m n)` produces an object of type Nat.

Here is the multiplication function defined recursively, `mult m Zero = Zero`, `mult m (Succ n) = plus ((mult m n) m)`. This is just saying that  $m^*(n+1)$  is the same as  $m^*n + m$ .

(Refer Slide Time: 03:31)

The slide has a title 'Second example: List' at the top. Below it, there are three bullet points defining a recursive data type:

- Recursive data types can also be polymorphic
- `List a = Nil | Cons a (List a)`
- This is the built-in type `[a]`

At the bottom right of the slide area, there is a small navigation bar with icons for back, forward, and exit.

Here is a second example, another simple example of a recursive data type. This is the example of list, this example also shows that recursive data types can be polymorphic. Here is the data declaration, List a = Nil | Cons a (List a). Recall again, that Nil and Cons are the value constructors and List is the type constructor. This is just the built-in type List a, that is provided by Haskell, but we are providing this definition just as an example.

(Refer Slide Time: 04:12)

The slide has a title 'List' in green. Below it is a bulleted list:

- Functions are defined as usual using pattern matching
- `head :: List a -> a`  
`head (Cons x _) = x`
- This causes an exception on `head Nil`
- You can have your preferred behaviour
- `head :: List a -> Maybe a`  
`head Nil = Nothing`  
`head (Cons x _) = Just x`

At the bottom right of the slide is a navigation bar with icons for back, forward, and close.

The functions are defined as usual using pattern matching, `head :: List a -> a`, `head` is a function that takes a list of type `a` as input and produces an object of type `a` as output, `head (Cons x _ ) =x`. Notice that this function fails when you invoke `head` on the list `Nil`, this causes an exception on `head Nil`, you can have your own preferred behavior. For instance, `head` is a function from `List a` to `Maybe a` written as `head :: List a -> Maybe a`, `head Nil = Nothing`. instead of not providing any definition for `head Nil`, you define `head Nil` to be `nothing` and you define `head (Cons x _ )` to be `Just x`.

(Refer Slide Time: 05:02)

Binary trees

- A binary tree data structure is defined as follows:
  - The empty tree is a binary tree
  - A node containing an element with left and right subtrees is a binary tree
- `data BTree a = Empty | Node (BTree a) a (BTree a)`

Navigation icons: back, forward, search, close.

Here is a third and may be more challenging example that of binary trees. A binary tree data structure is defined as follows, the empty tree is a binary tree and a Node containing an element with left and right subtree is also a binary tree. So, the definition is as follows, data BTree a = Empty | Node (BTree a) a (BTree a). So, again recall that Empty is a function which does not take any inputs and whose output is an object of type BTree a. Node is a function with type signature BTree a -> a -> BTree a -> BTree a, which is the tree that is returned.

(Refer Slide Time: 05:54)

Binary trees

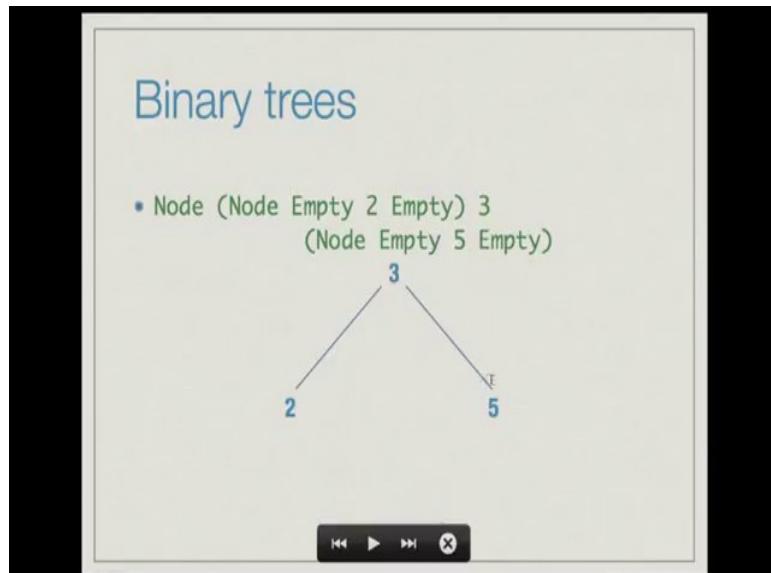
- `Empty :: BTree a`  
`Node :: BTree a -> a -> BTree a`  
`-> BTree a`
- `Node (Node Empty 2 Empty) 3`  
`(Node Empty 5 Empty)`
- `Node (Node Empty 4 Empty) 6`  
`(Node (Node Empty 2 Empty) 3`  
`(Node Empty 5 Empty))`

Navigation icons: back, forward, search, close.

Now, that is explained here, Empty is a function with type BTREE a, Node is a function of type BTREE a -> a -> BTREE a -> BTREE a. Here is an example of a binary tree: Node ( left subtree ) and the Node whose value is 3 and there is a (right sub tree). The left sub tree is recursively given by (Node Empty 2 Empty) : Node and a left sub tree of it and value at the root of the left sub tree, which is 2 and the right sub tree of the left sub tree.

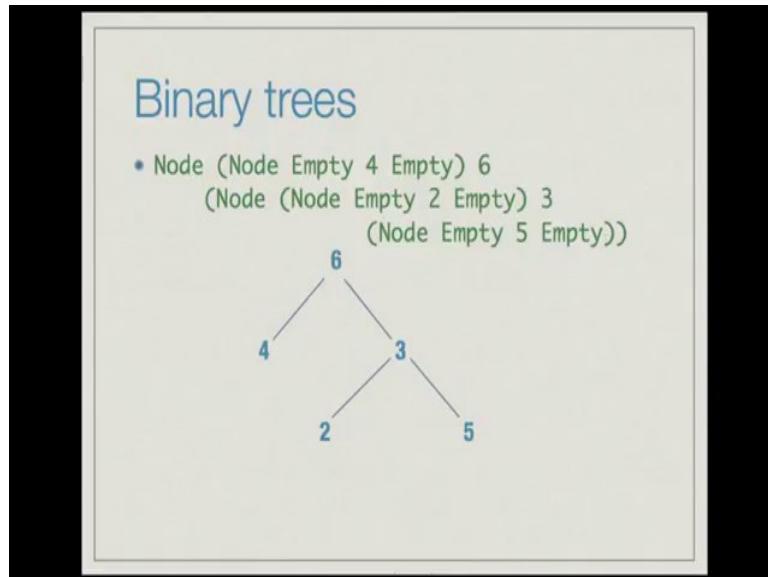
In this case, the left sub tree and the right sub tree are empty; similarly the right sub tree of the bigger tree with the root 3 is ( Node Empty 5 Empty ). Here is another example, Node of some left sub tree which is given by (Node Empty 4 Empty) and the root is 6 and the right sub tree itself is a non trivial tree. Node of some left sub tree and the root value being 3 and its right sub tree being (Node Empty 5 Empty).

(Refer Slide Time: 07:09)



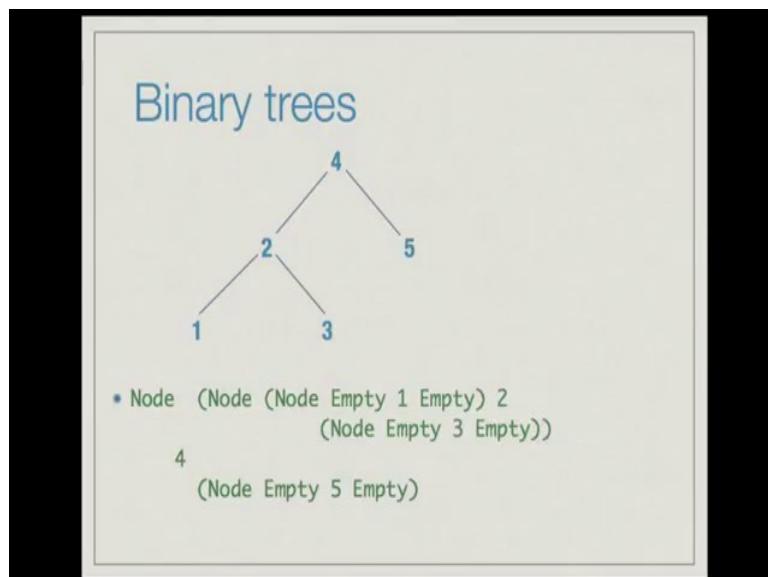
This tree would be defined as this, Node ( Node Empty 2 Empty ) which defines this left sub tree, 3 which defines the root; (Node Empty 5 Empty), which defines this right subtree which consists only of a single Node.

(Refer Slide Time: 07:33)



Here is another example, you have a binary tree with 6 as root, 4 as the only Node in the left subtree and the right subtree consisting of 3 as the root, 2 as the only Node in its left subtree and 5 as the only Node in its right subtree. This would be defined as follows, Node ( Node Empty 4 Empty) 6 (Node (Node Empty 2 Empty) 3 (Node Empty 5 Empty)), the structure should be fairly simple now.

(Refer Slide Time: 08:07)



Here is another example, this tree is Node of a fairly large left sub tree, a non trivial left subtree and the root value being 4, which is declared here and a right subtree which has only

a single Node. The right subtree is (Node Empty 5 Empty), the left sub tree itself is a tree which is of the form Node 2 and a left sub tree which says (Node Empty 1 Empty) and a right subtree which says (Node Empty 3 Empty). This is how you represent binary trees.

(Refer Slide Time: 08:48)

The slide has a light gray background with a dark gray header bar. The title 'Functions on binary trees' is centered in the header in a blue font. Below the title, there is a bulleted list and some code. The bullet points are:

- `size` - Number of nodes in a tree
- `size :: BTree a -> Int`

Below the second bullet point, there are two lines of code:

```
size Empty = 0
size (Node tl x tr) = size tl + 1
                      + size tr
```

At the bottom of the slide, there is a navigation bar with four icons: a double-left arrow, a single-right arrow, a double-right arrow, and a close (X) button.

Now, you can define functions on binary trees as usual by using pattern matching. Here is the simple function on binary trees, the function `size` which returns the number of Nodes in a tree. `Size` is a function with signature `BTree a -> Int`, `size` of `Empty` equals 0, `size` of `(Node tl x tr)` where `tl` represents the left subtree, `tr` represents the right subtree, `x` denotes the value at the Node, equals `size tl + 1 + size tr`, the size of the left subtree plus 1, because the root is also a Node plus the size of the right subtree.

(Refer Slide Time: 09:36)

## Functions on binary trees

- height - Longest path from root to leaf
- height :: BTrie a -> Int  
height Empty = 0  
height (Node tl x tr) = 1 +  
max (height tl) (height tr)

Here is another simple function on binary trees, height which gives the longest path from root to leaf. height is a function with signature Btree a -> Int, height of Empty equals 0, because there are no Nodes at all, the root is the same as the leaf. Height of (Node tl x tr ) equals 1 plus max of height tl and height tr. If the left sub tree had height 4 and the right sub tree had height 3, then the tree itself would have height 5, because 5 is 1 + max (4,3).

(Refer Slide Time: 10:20)

reflect - Reflect the tree on the “vertical axis”

```
graph TD; A((4)) --> B((2)); A --> C((5)); B --> D((1)); B --> E((3)); C --> F((5));
```

```
graph TD; A((4)) --> G((5)); A --> H((2)); G --> I((3)); G --> J((1));
```

Here is a function which reflects the tree on the vertical axis, for instance you start out with 4, left sub tree having 2, 1, 3; right subtree having just the Node 5. If you reflect it, the left sub

tree will now have a single Node 5 and the right sub tree will have the reflection of the left sub tree here. So, you will get 2, 3, 1 instead of 2, 1, 3.

(Refer Slide Time: 10:51)

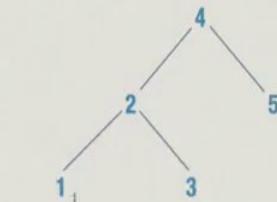
## Functions on binary trees

- `reflect` - Reflect the tree on the “vertical axis”
- `reflect :: BTree a -> BTree a`  
`height Empty = Empty`  
`height (Node tl x tr) = Node`      `1`  
                        `(reflect tr)`  
                        `x`  
                        `(reflect tl)`

How would you define this? Reflect a function of signature `BTree a -> BTree a`, reflect of `Empty` equals `Empty`, reflect of `(Node tl x tr)` is `Node (reflect tr) x (reflect tl)`. Notice that you form the Node with the reflection of the right sub tree on the left and the reflection of the left sub tree on the right.

(Refer Slide Time: 11:18)

## Functions on binary trees



- `levels` - List nodes level by level, and from left to right within each level
- `levels` of the above tree - `[4,2,5,1,3]`

Here is another function, we want to list all the Nodes in a tree level by level and from left to right within each level. So, in this tree there are three levels, this is one level, this is another level and this is the third level and you want to list elements from the first level and then, from the second level and then from the third level and within each level from left to right. So, levels on the above tree would yield [4, 2, 5, 1, 3].

(Refer Slide Time: 11:56)

So, levels is a function which has signature  $\text{BTree } a \rightarrow \text{list } a$  and it is defined as follows,  $\text{levels } t = \text{concat } (\text{myLevels } t)$ . Recall that the concat function takes a list of lists of type  $a$ ,  $[[a]]$  and produces a list of type  $a$ , so myLevels  $t$  have to be a function that returns a list of list of type  $a$ . myLevels is a function with signature  $\text{BTree } a \rightarrow [[a]]$ . This list is supposed to represent, the first list in this list represents all the Nodes in the first level, the second list inside this list represents all the Nodes in the second level, etcetera.

So, myLevels of Empty is nothing but, the empty list because there are no levels in an empty tree, myLevels of  $(\text{Node } t1 \ x \ t2)$  is the list that you get by prepending the list containing  $x$ , because  $x$  is the only Node in the top level. So, the list consisting of  $x$  should be the first list in this list of lists and following that, you have a bunch of lists that you get from myLevels  $t1$  and you have a bunch of lists that you get from myLevels  $t2$ , you join the lists at the appropriate levels. So, the definition of myLevels  $(\text{Node } t1 \ x \ t2)$  is the list consisting of  $x$  coming in front of join,  $[x] : \text{join } (\text{myLevels } t1) (\text{myLevels } t2)$ .

(Refer Slide Time: 13:56)

The slide has a light gray background with a dark gray border. At the top, the title 'Functions on binary trees' is written in blue. Below the title, there is a bullet point followed by a series of Haskell-like code definitions:

- join :: [[a]] -> [[a]] -> [[a]]  
join [] yss = yss  
join xss [] = xss  
join (xs:xss) (ys:yss) = (xs ++ ys):  
                                  join xss yss

The code for join is given here, join is a function which has signature list of list of a arrow list of list of a arrow list of list of a,  $[[a]] \rightarrow [[a]] \rightarrow [[a]]$ . Now, the function join is very similar to zip with of ++ applied to xss and yss, but there is a slight difference in this case. So, let us look at the definition in a bit more detail, join [] yss equals yss; in the case of zip with this would have been empty list; join xss [] equals xss.

These two cases represent the situations, where the left sub tree has fewer levels than the right sub tree and this represents the case where the left sub tree has more levels than the right sub tree. Now, join of (xs:xss) (ys:yss) is just (xs ++ ys) coming in front of join xss yss. so this completes the definition of levels.

(Refer Slide Time: 15:12)

## Showing trees

- ```
data BTree a = Empty
           | Node (BTree a) a (BTree a)
deriving (Eq, Show)
```
- Default show of trees is very hard to parse
- ```
show (Node (Node Empty 2 Empty) 3 (Node
Empty 5 Empty)) = "Node (Node Empty 2
Empty) 3 (Node Empty 5 Empty)"
```

Let us look at how we can display trees, so a simple way of displaying tree is to say deriving Eq and Show, is to say deriving show in the data type declaration. But, the default show method on trees is very hard to parse, for instance show of this complicated tree is just the same representation given inside codes. As you can see, this is not very easy to read.

(Refer Slide Time: 15:52)

## A prettier show

- We want a better layout
- ```
tree1 = Node (Node Nil 4 Nil) 6 (Node (Node Nil
2 Nil) 3 (Node Nil 5 Nil))
```
- Typing tree1 in ghci should give us (each node on a line,
and 2n spaces before each node at level n)  

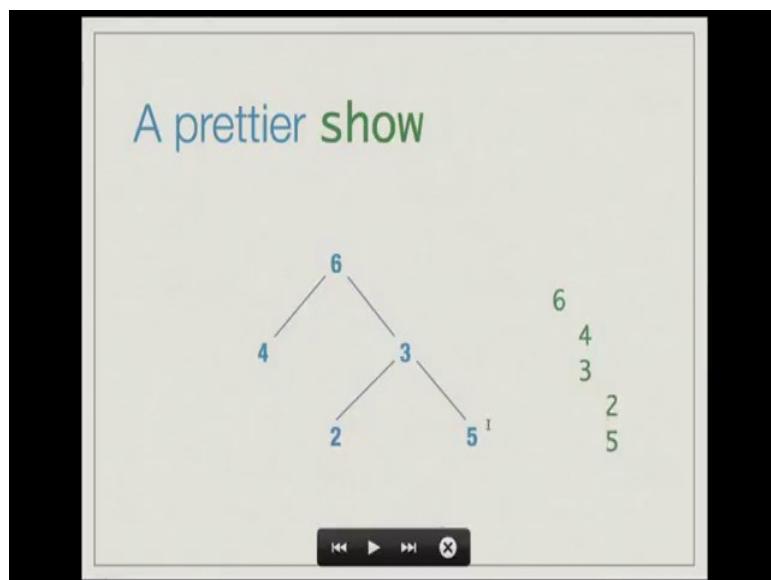
```
6
  4
  3
    2
    5
```

We may want a prettier show with a better layout, for instance suppose we define tree1 to be this tree, Node with a root 6 and left sub tree consisting of a single Node 4 and right sub tree consisting of a root 3 with left child 2 and right child 5. Then, typing tree1 in ghci should

give us this, this is the desired behavior that we want. In this, we see that each Node is printed on a line, is printed on a line and there are  $2n$  spaces before each Node at level n.

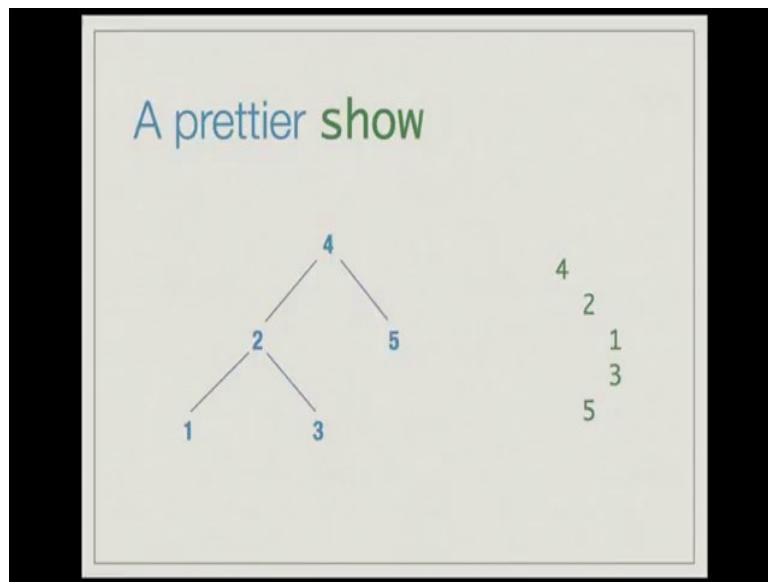
Here are assuming that the root is in level 0. So, 4 and 3 which are Nodes at level 2 have two spaces before them and 2 and 5 which are Nodes at level 3 have four spaces before them. And this layout makes the structure of the tree clear, 6 is the root with two children 4 and 3, 4 does not have any children, whereas 3 has children 2 and 5.

(Refer Slide Time: 17:07)



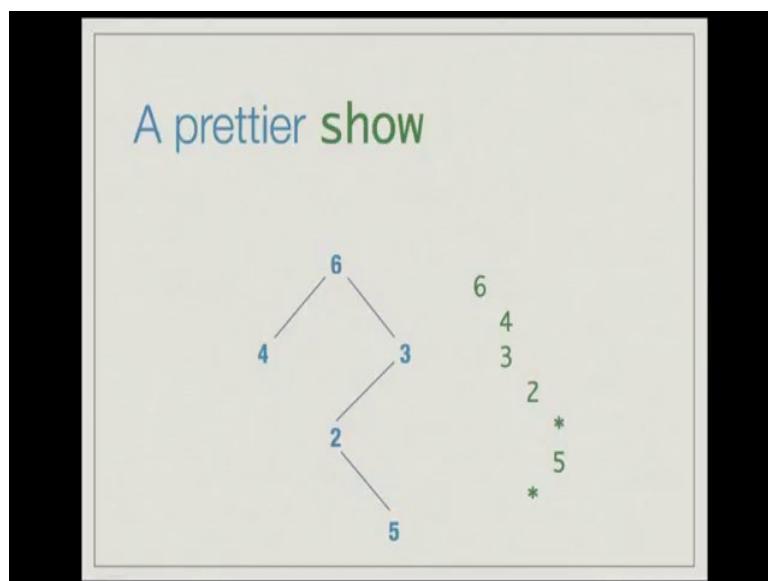
You can see that the printed layout mirrors the structure of the tree and makes it pretty clear, 6 is the root, 4 and 3 are the children, 4 does not have any children, 3 has two children 2 and 5.

(Refer Slide Time: 17:27)



Here is another tree, 4 with left sub tree consisting of 2, 1 and 3 and the right sub tree being a single Node 5. We want this to be laid out in this fashion, 4 with two children 2 and 5 and the children of 2 immediately displayed on the lines following 2, 1 and 3, 5 would displayed after that. But, you can look at which column 5 appears in and determine whose child it is, 5 is the child of 4 because 5 appears in column 3, whereas 4 appears in column 4.

(Refer Slide Time: 18:07)

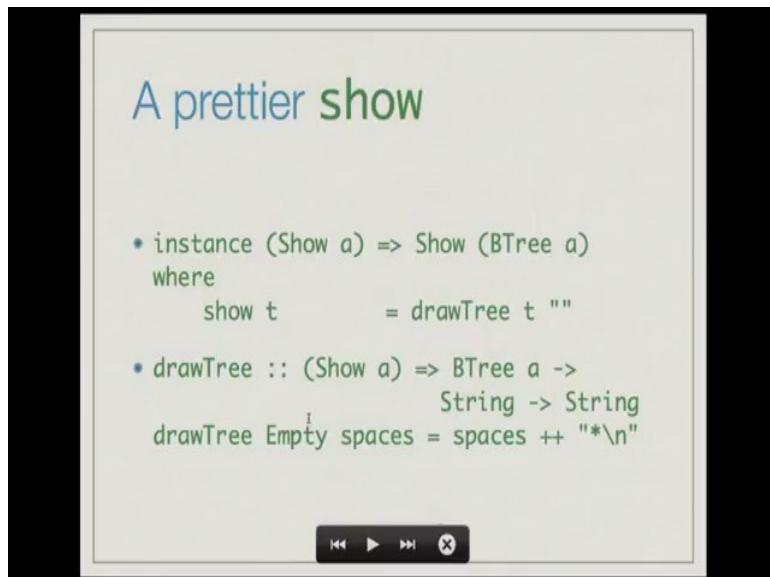


Here is another tree, it is a tree with root 6, left sub tree consisting of a single Node 4, right subtree having a Node 3 whose left child is 2 and whose right child is 5. This is laid out as

follows, 6 followed by its two children 4 and 3, followed by its left child which is 2 whose children are displayed here as \* and 5, this is because this Node has only one child and the other child is empty and we use star \* to denote which child is empty, in case there is only one child.

In case both children are empty as in the example of this Node 4, we do not display anything. But, if one subtree is empty and the other subtree is non empty, then we display a star corresponding to the sub tree that is empty. So, 2 has left sub tree which is empty and right sub which consists of a single Node 5 and 3 itself has this as the left sub tree, 2 \* 5 and its right sub tree is empty which is denoted by \*. Now, how do we define this show function?

(Refer Slide Time: 19:23)



We defined it as follows using an instance declaration, instance (Show) a implies Show (BTree a), where Show t, t is the tree here, is just drawTree t "", "" is empty string. drawTree will draw the layout of the tree, where the second parameter which is the string tells how many spaces to insert before drawing the current sub tree. So, drawTree is a function with signature (Show a) => implies BTree a -> String -> String, this string is the number of spaces that have to be displayed before the current sub tree is displayed and this string is actually the output, the layout of the tree.

drawTree Empty spaces = spaces ++ "\*\n" which denotes the new line character. We will see that you will encounter this case only when this is the sole empty subtree of a Node.

(Refer Slide Time: 20:47)

A prettier show

```
• instance (Show a) => Show (BTree a) where
    show t      = drawTree t ""

• drawTree (Node Nil x Nil) spaces
    = spaces ++ show x ++ "\n"
drawTree (Node tl x tr) spaces
    = spaces++ show x ++ "\n"
    ++ drawTree tl (' ':':':spaces)
    ++ drawTree tr (' ':':':spaces)
```

Navigation icons: back, forward, search, close.

The other cases when you have a Node x both of whose subtrees are empty, (Node Nil x Nil), drawTree (Node Nil x Nil) spaces is nothing but, spaces ++ show x ++ “\n”. drawTree of (Node tl x tr) spaces, where tl is the left subtree and tr is the right subtree, spaces is nothing but, spaces ++ show x ++ “\n”. Remember that we always add this much, this many spaces before displaying the current subtree, plus you append that with drawTree of tl.

But, when you draw the left sub tree you have to add two more spaces, so which you do by adding two spaces from the beginning of this list. And then, you concatenate that with drawTree of tr which is the right subtree, again giving two spaces adding two more spaces in the beginning of laying out the right subtree.

(Refer Slide Time: 21:58)

The slide has a light gray background with a dark gray border. The title 'Summary' is at the top in a blue font. Below it is a bulleted list in black font:

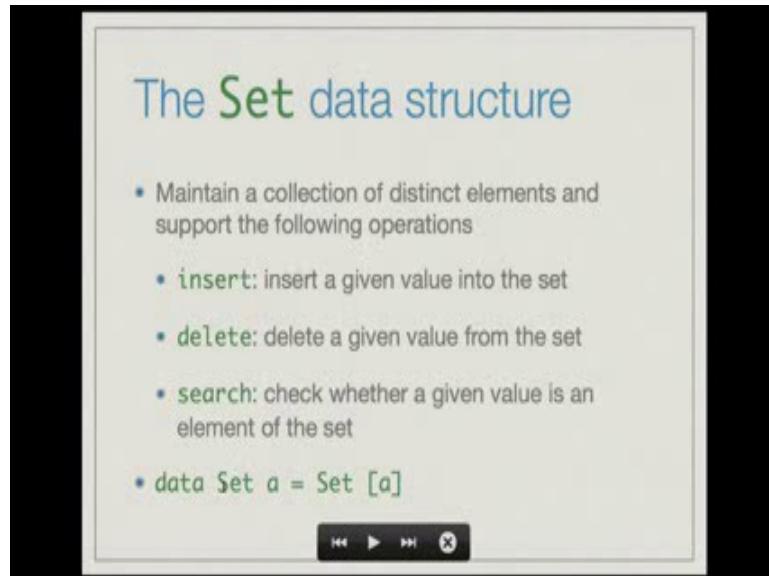
- Recursive datatypes are an important concept in Haskell
- A recursive datatype **T** is one which has some components of the same type **T**
- Two canonical and important examples of recursive datatypes -- Lists and trees

So in summary, recursive data types have been introduced in this lecture, they are a very important concept in Haskell. A recursive data type T is one which has some of its components of the same type T and we have seen two canonical and important examples of recursive data types, lists and trees. In the next lecture, we will see more on trees.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P Suresh**  
**Mathematical Institute Chennai**

**Module # 06**  
**Lecture – 02**  
**Binary Search Trees**

(Refer Slide Time: 00:08)

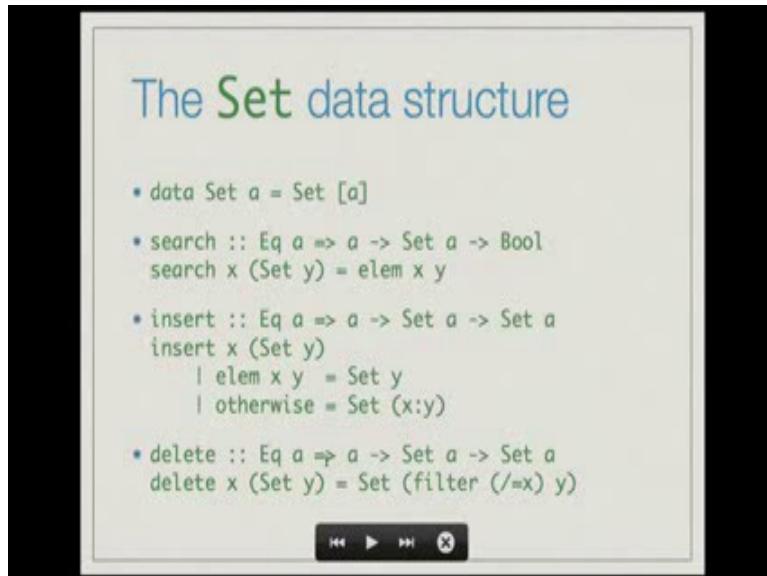


The Set data structure

- Maintain a collection of distinct elements and support the following operations
  - `insert`: insert a given value into the set
  - `delete`: delete a given value from the set
  - `search`: check whether a given value is an element of the set
- `data Set a = Set [a]`

In this lecture, we shall introduce a very important data structure the binary search trees. Imagine that we want to implement a Set data structure. It is defined as follows. You want to maintain a collection of distinct elements and support the following operations. Insert - it inserts a given value into the set; delete which deletes a given value from the set and search which checks whether a given value is an element of the set or not. Here is one way of defining this. `data Set a = Set [a]`. Recall again that the `Set` on the left hand side is the type constructor, the `Set` on the right hand side is the value constructor. The name of the new type that we have defined is `Set a` and it has a single constructor `Set` which takes list `a`, `[a]` as a argument and outputs an object of type `Set a`.

(Refer Slide Time: 01:10)



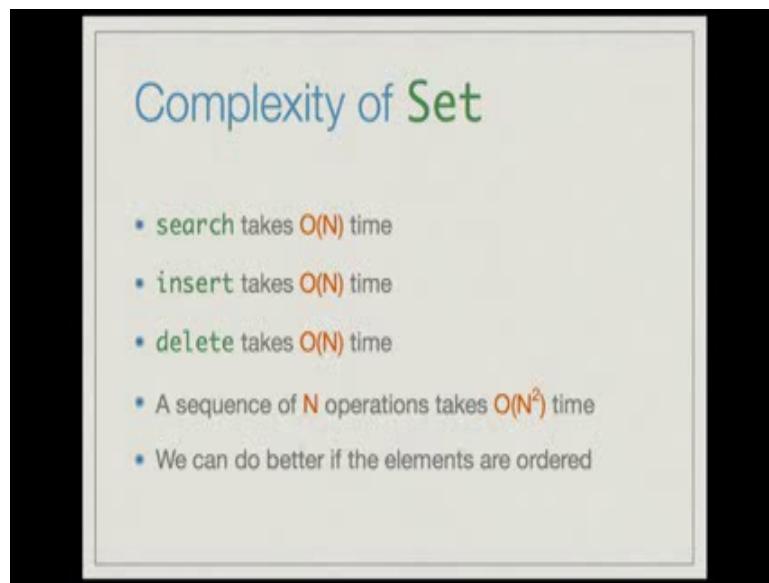
Here is how we might implement the three functions that we described earlier. Search is the function with the following signature. `Eq a => a -> Set a -> bool`. Search of `x` and `Set y`, searching whether `x` belongs to `Set y`, recall here that `y` is a list consisting of elements of type `a`, so it is the constructor. It is just searching whether `x` is an element of `y`, it is given by the primitive function, the built in function `elem` which checks whether `x` is an element of the list `y`. So, `search x (Set y) = elem x y`, is the implementation of search.

Insert can be implemented as follows. Insert as a function with signature `Eq a => a -> Set a -> Set a`. This is the original set and this is the set that we get after the element has been inserted. Insert of `x` inside `Set y`, there are two cases; if `x` is an element of `y`, you just return `Set y`; this is because we want to maintain a set of distinct elements. If `x` is already present, we don't have to do anything. Otherwise, you insert `x` in `y`, this you do by the simple interface of attaching `x` to the front of the list `y`. So in the otherwise case, you define it to be `Set (x:y)`.

Here is how you might ((Refer Time: 02:49)) define delete. Delete is the function with signature `Eq a -> a -> Set a -> Set a`. Delete of `x` from `Set y` is nothing but the `Set` which is gotten by applying `filter (/=x)` on `y`. Recall that `filter` is a function that applies a predicate to a list and returns all the elements of the list that satisfy the predicate. The predicate that we are giving here is this funny looking object, this is the so called section in Haskell. Recall that `/=` is an operator,

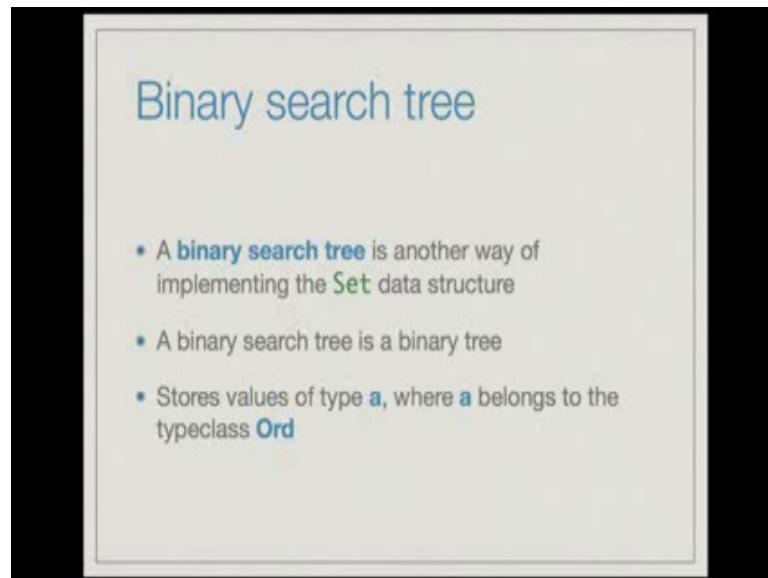
it is a binary operator in Haskell. So whenever you have a binary operator, you can use it as a function by enclosing it in parenthesis; furthermore, you can use it as not just a binary function but a unary function where one of the arguments have been fixed. In this case, we have fixed the argument  $x$ , and you want to check whether  $x$  is not equal to any of the elements of  $y$ . So, this section is gotten by fixing the right argument and making this as unary function. So, this is the predicate that you are applying to each element in the list  $y$ . So, this function will return all those elements of  $y$  that are not equal to  $x$ , which means that it will delete  $x$  from  $y$ .

(Refer Slide Time: 04:30)



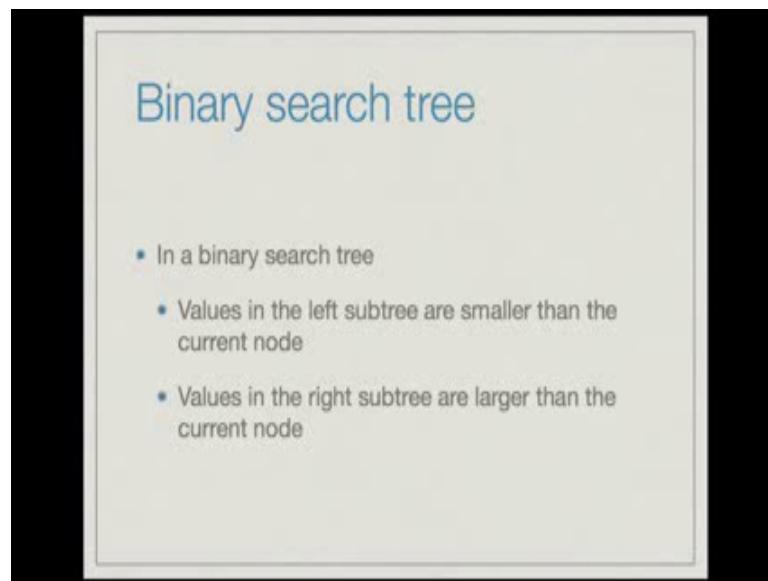
Let us look at the running time of these operations for the above implementations. Search takes  $O(N)$  time, insert takes  $O(N)$  time and delete takes  $O(N)$  time. Search takes  $O(N)$  time, because the built in function `elem` takes  $O(N)$  time, where  $N$  is the length of the list. Insert also takes  $O(N)$  time, because first of all you have to make a search in the list. Delete takes  $O(N)$  time, because `filter` takes  $O(N)$  time. So a sequence of  $N$  operations takes  $O(N^2)$  times, and we might want to do better than this. We can do better than this if the elements are ordered, but storing these elements as a sorted list is not the way to go, we need a much better data structure.

(Refer Slide Time: 05:37)



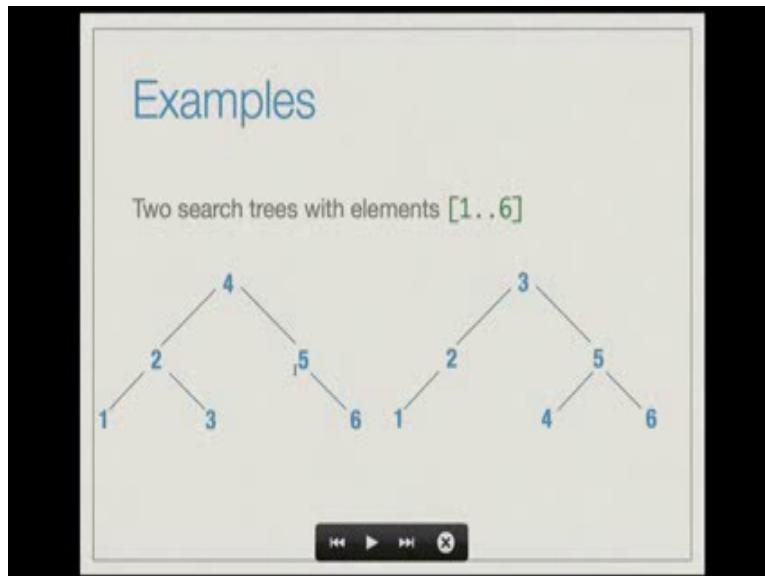
And that brings us to a binary search tree. A binary search tree is another way of implementing the Set data structure. A binary search tree is just a binary tree that we discussed in the last lecture. It stores values of type a, where a belongs to the type class Ord. So a binary search tree makes sense only for values that can be ordered.

(Refer Slide Time: 06:08)



The crucial property of the binary search tree is that at any node in a binary search tree, values in the left subtree are smaller than the current node. And values in the right subtree are larger than the current node. And this holds true for every node in the binary search tree.

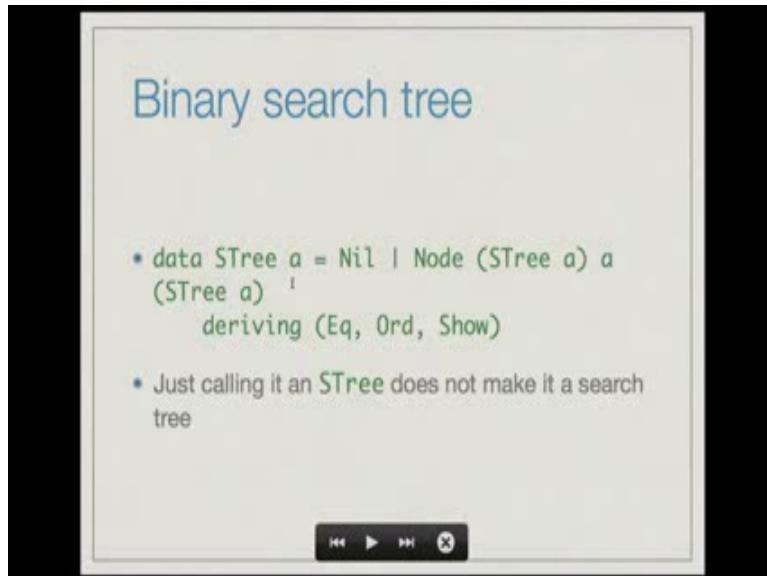
(Refer Slide Time: 06:28)



Here are two examples of binary search trees, which consist of the elements [1..6]. Here is one tree, this tree has root 4 and left subtree consisting of the node 1,2 and 3, right subtree storing the value of 5 and 6. Notice how the binary search tree property holds true in this tree. Every node in the left subtree has a smaller value than the root, so the left subtree has value as 1, 2 and 3, the root has value 4. Every node in the right subtree namely 5 and 6 has value greater than 4. And recursively in this left subtree, the same property holds; 2 is greater than one and 2 is smaller than 3. Similarly here, 5 is smaller than 6.

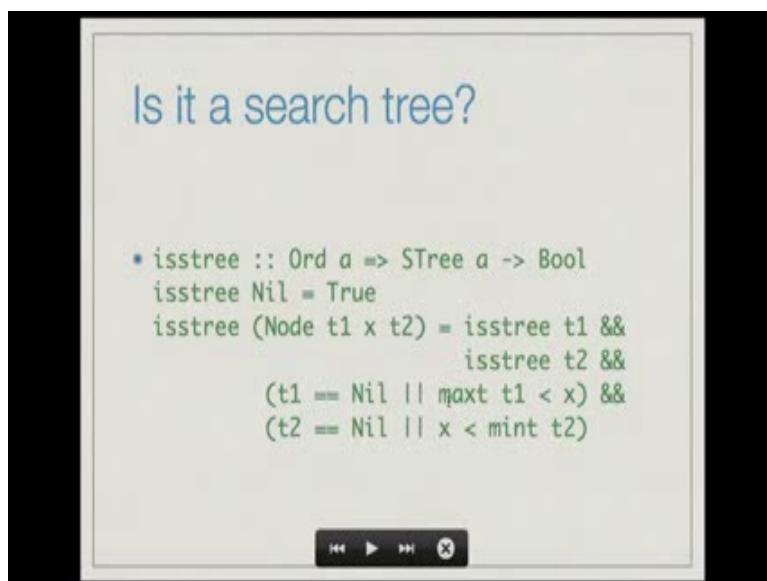
Here is the other tree which represents the same elements. This tree has root 3, and the left subtree has the nodes 2 and 1. In the left subtree 1 is to the left of 2 as it ought to be; and the right subtree has the nodes 4, 5 and 6 and this itself satisfies the search tree property because 5 is the root whose left child is smaller than 5 and whose right child has the value that is greater than 5.

(Refer Slide Time: 07:53)



We define binary search tree as follows. `data STree a = Nil | Node (STree a) a (STree a)`. This is exactly similar to the definition of binary trees that we had earlier, and we derive Eq, Ord, Show etcetera. We might create our own instance of the show method or we can just use the default show function. Just calling this `STree` does not make it a search tree. We could have objects of type `STree` which completely violate the search tree property.

(Refer Slide Time: 08:40)



So here is the function that takes a search tree as input and tells whether it is indeed a search tree or not. It takes a binary tree as input and tells whether it is indeed a search tree or not. So this is the function `isstree` which signature is `Ord a => STree a -> Bool`. `isstree` of `Nil` is just `True`, because the empty tree obviously satisfies the binary search tree property. `isstree` of `(Node t1 x t2)` is nothing but `isstree` of `t1`, the left subtree should recursively satisfy the search tree property, the right subtree also should recursively satisfy the search tree property. So `isstree t1 && isstree t2` and you have to check whether all the values in the left subtree are smaller than `x`. One way to check it, is to check whether the max value in the left subtree `maxt` of `t1` is less than `x`, and we also need to check whether all the values in the right subtree are greater than `x`, and one way of checking it, is to check whether `x` is less than the minimum value in the right subtree `mint` of `t2`.

But we had an extra clause here, because the `maxt` function that we will define later is defined only for non-empty trees. So you say either `t1 == Nil` or `maxt t1 < x`, and either `t2 == Nil` but in case it is not null, check if `x < mint t2`, `x` is less than the minimum value in the tree `t2`. This is the definition of is search tree.

(Refer Slide Time: 10:25)

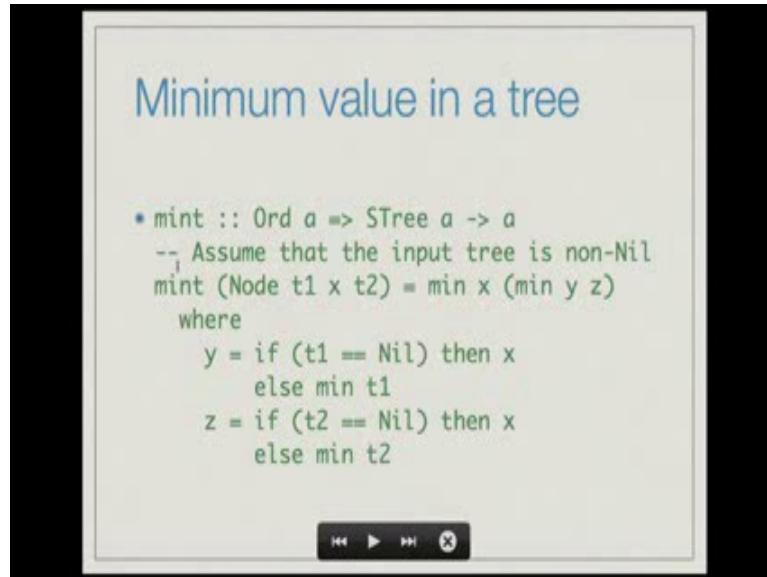
### Maximum value in a tree

```
* maxt :: Ord a => STree a -> a
-- Assume that the input tree is non-Nil
maxt (Node t1 x t2) = max x (max y z)
where
  y = if (t1 == Nil) then x
      else maxt t1
  z = if (t2 == Nil) then x
      else maxt t2
```

The maximum value in a tree is computed as follows. `maxt` is a function with signature `Ord a => STree a -> a`. In this function, we assume that the input tree is non-nil. `maxt` of `Node t1 x t2` is nothing but `max` of `x` and (`max` of `y z`). Here `y` is supposedly the maximum of `t1`, and `z` is

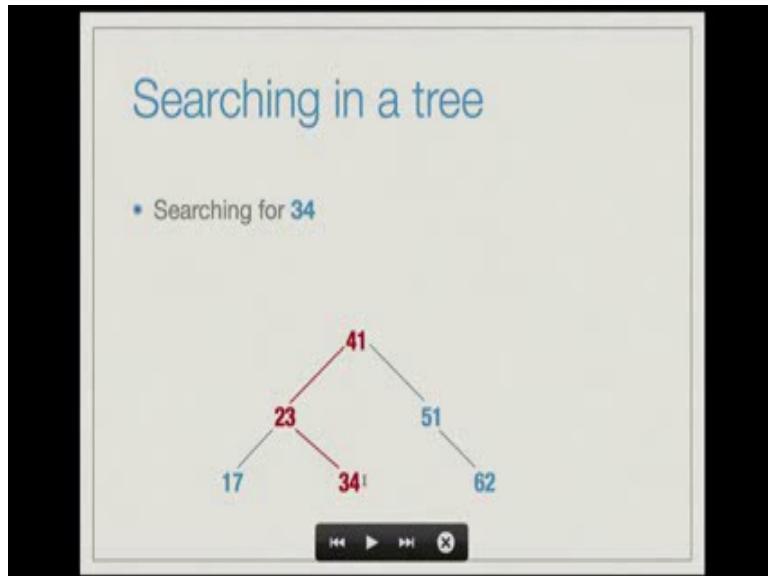
supposed to be the maximum of t2, but t1 could be empty. Therefore, the definition of y is slightly more complicated. The y is defined to be just x incase t1 is nil; otherwise, it's defined to be maxt of t1 recursively. And z is defined to be just x incase t2 is nil; otherwise, it's defined to be maxt of t2.

(Refer Slide Time: 11:24)



Symmetrically, you can define the minimum value in a tree also. Mint which is the function with signature  $\text{Ord } a \Rightarrow \text{STree } a \rightarrow a$ . Again we assume that the input tree is non-nil. Mint of  $(\text{Node } t1 \ x \ t2)$  equals  $\min(x, (\min(y, z)))$ , the minimum of these three values, where y is x incase t1 is Nil otherwise it is mint of t1. And z is x incase t2 is Nil (refer time: 12:00) otherwise it's mint of t2.

(Refer Slide Time: 12:05)



Now let us look at an important function on tree, namely searching for an element in a tree. Let say we want to search for the element 34 in this tree; this tree consist of the nodes 41, 23, 17, 34, 51, 62. As you can see this is the search tree, because it satisfies the search tree property. Every value in the left subtree is smaller than the root; and every value in the right subtree is greater than the value of the root. And recursively this property holds for both the left subtree and the right subtree. So searching starts at the root of the tree. We check whether 34 is same as the value 41; in this case, we see that 34 is not equal to 41, but we also see that 34 is smaller than 41. And therefore, if the value 34 is represented in this tree, it has to lie in the left sub tree, because the value we have searching for is smaller than 41 and it cannot lie in the right subtree, therefore we go left.

And we check whether 34 is equal to the value in this node; value in this node is 23 which is smaller than the 34. And now we know that if 34 were to be present in this tree, it has to be in the right subtree of this node, because to the left of this node, all node, all values are smaller than 23, and therefore and then smaller than 34. So from here, we go right. And here we see that the value is indeed 34 which is what we were seeking for, so we stop at this point.

(Refer Slide Time: 13:55)

## Searching in a tree

- Searching for 49

```
graph TD; 41 --- 23; 41 --- 51; 23 --- 17; 23 --- 34; 51 --- 62; 51 --- X
```

Let us say we are searching for 49 in the same tree. Now 49 again we start the search at the root. Now, 49 is greater than 41, so it has to lie in the right subtree, so we go right. The value that we have to examine now is 51 and 51 is greater than 49, so 49 if it exists at all has to lie to the left of 51, but there is no element to the left of 51. So this search ends in a failure and we say that 49 is not present in the tree.

(Refer Slide Time: 14:32)

## Searching in a tree

- Searching for value  $v$  in a search tree
- If the tree is empty, report **No**
- If the tree is nonempty
  - If  $v$  is the value at the root, report **Yes**
  - If  $v$  is smaller than the value at the root, search in left subtree (which could be empty)
  - If  $v$  is larger than the value at the root, search in right subtree (which could be empty)

So here is the strategy for searching for an element in a tree. We are searching for a value v in a search tree. If the tree is empty, we report No. If the tree is nonempty then we do a case analysis. If the v is equal to the value at the root, we report yes. If v is smaller than the value at the root, we search in the left subtree, which could be empty, in which case we would report No. If v is larger than the value at the root, we search in the right subtree; again this could be empty, in which case we would report No.

(Refer Slide Time: 15:09)

**Searching in a tree**

- `search :: Ord a => STree a -> a -> Bool`

```

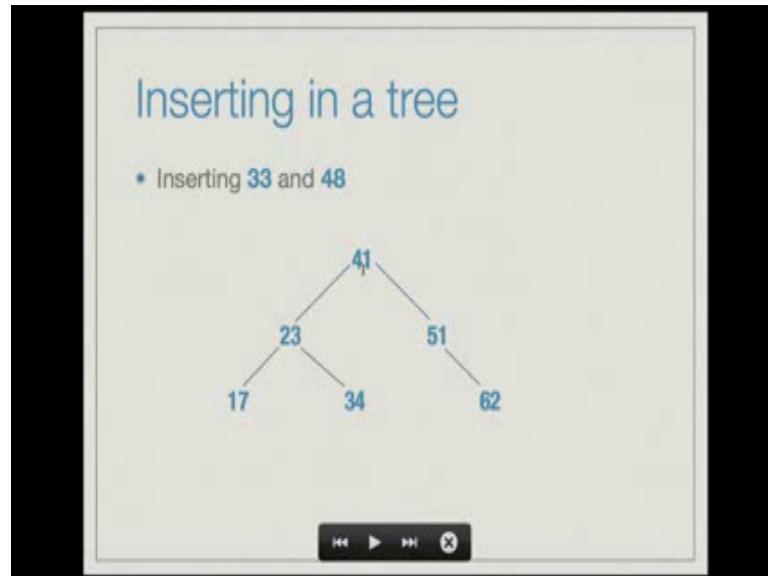
search Nil v           = False
search (Node tl x tr) v
| x == v              = True
| v < x               = search tl v
| otherwise            = search tr v

```
- Worst case: running time proportional to length of the longest path from root to a leaf (**height**)

◀ ▶ ⏪ ⏩ X

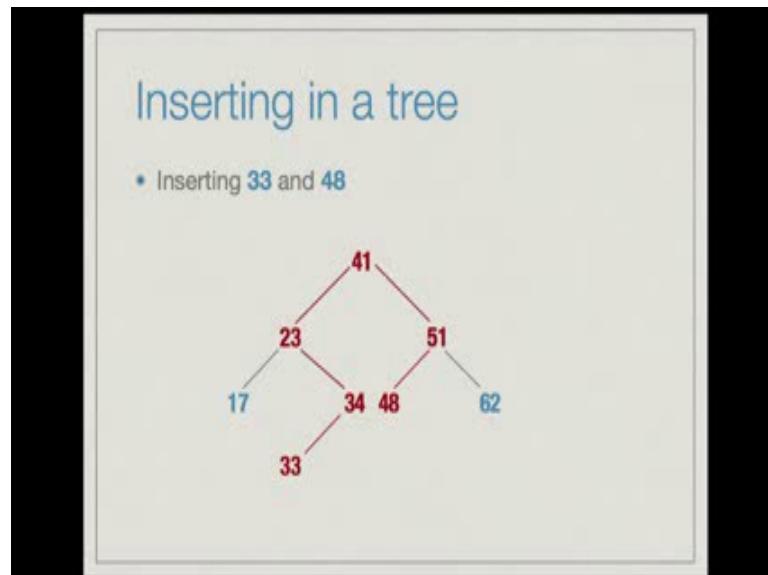
Here is the code that realizes the above strategy. Search which is the function with signature `Ord a => STree a -> a -> Bool`. `search Nil v`, if we are searching for the value v in the empty tree, then the output is false. `search (Node tl x tr) v`, this is the tree with left subtree tl, right subtree tr and the value at the root equals x. Here there are three cases, incase x is equal to v then we say True; incase v is smaller than x then we search in the left subtree – `search tl v`. Otherwise, we search in the right subtree – `search tr v`. The worst case running time of this procedure is proportional to the length of the longest path from root to a leaf, because we start at root and keep descending the tree till we reach a leaf or we reach the value that we are seeking for, but we are talking about the worst case. And the length of the longest path from root to a leaf is nothing but the height of the tree. So in the worst case, the running time of search is proportional to the height of the tree.

(Refer Slide Time: 16:27)



Here is the other important function that we wanted to look, that we wanted to implement inserting an element in a tree. Let us consider the example of inserting 33 and 48 into this tree; the same tree that we considered earlier.

(Refer Slide Time: 16:59)

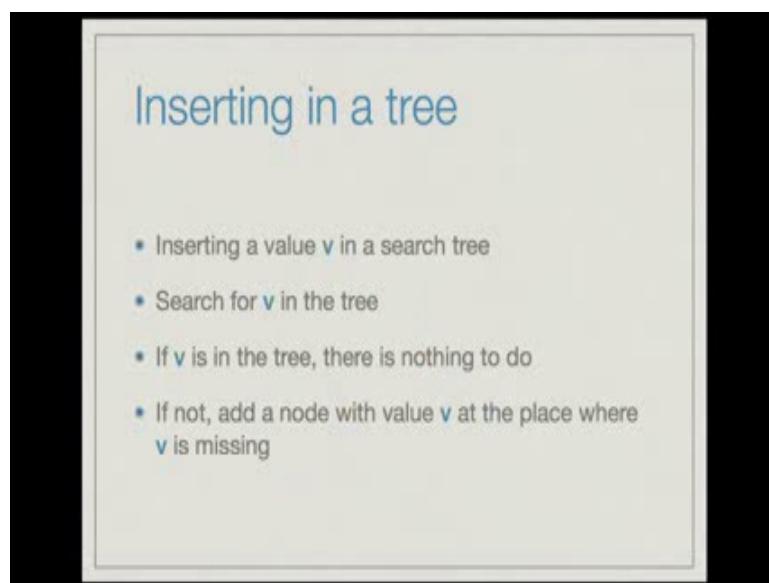


To insert 33 and 48, we first need to determine the location where 33 should go. So we start to search for thirty-three in the list, in the tree. We start at the root and compare the value thirty-

three with the value at the node which is 41,  $33 < 41$ , so we descend along the left subtree. Now 33 is greater than 23, so it should if it were present in the tree, it should lie in the right subtree, so we descend right. Now we see a node 34;  $33 < 34$ , but there is nothing to the left of 34. If 33 were present in the tree, in the subtree starting from 34, it should be present at the left of 34, but there is nothing to the left of 34. So this tells us that the element 33 is not present in the tree, and we can add it. And the most natural place to add it is, in fact the only correct place to add it is as the left child of 34, so we add it there.

Now let us consider 48. We again check with 41; 48 is greater than 41, so we descend on the right subtree. 51 is greater than 48, so we have to descend down the left subtree, but there is nothing on the left subtree. So this again tells us that 48 is not present in the tree, so we add it as the left child of 51.

(Refer Slide Time: 18:18)



So this is the overall strategy for insert. To insert a value v in a search tree, you search for v in the tree. If v is in the tree, there is nothing to do. If not, add a node with value v at the place where v is missing. And the place where v is missing is found by the search routine.

(Refer Slide Time: 18:41)

Inserting in a tree

- If the tree is empty, create a node with value  $v$  and empty subtrees
- If the tree is nonempty
  - If  $v$  is the value at the root, exit
  - If  $v$  is smaller than the value at the root, insert  $v$  in left subtree (which could be empty)
  - If  $v$  is larger than the value at the root, insert  $v$  in right subtree (which could be empty)

If the tree is empty, you create a node with value  $v$  and empty subtrees. If the tree is nonempty, if  $v$  is the value at the root, you exit. If  $v$  is smaller than the value at the root, you insert  $v$  in the left subtree; if  $v$  is larger than the value at the root, you insert  $v$  in the right subtree.

(Refer Slide Time: 19:01)

Inserting in a tree

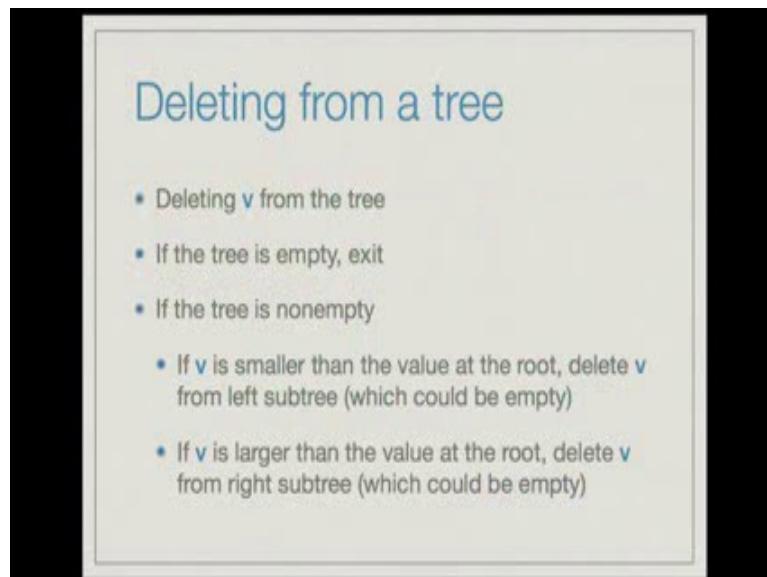
- $\text{insert} :: \text{Ord } a \Rightarrow \text{STree } a \rightarrow a \rightarrow \text{STree } a$ 

```
insert Nil v = Node Nil v Nil
insert (Node tl x tr) v
| x == v = Node tl x tr
| v < x = Node (insert tl v) x tr
| otherwise = Node tl x (insert tr v)
```
- Worst case: running time proportional to length of the longest path from root to a leaf (**height**)

Here is code that realizes the above strategy. Insert which is a function with signature  $\text{Ord } a \Rightarrow \text{STree } a \rightarrow a \rightarrow \text{STree } a$ .  $\text{insert Nil } v = \text{Node Nil } v \text{ Nil}$ . This is the node with root  $v$  and two

empty subtrees. This is what happens when we insert  $v$  into the empty tree. Insert ( $\text{Node } tl \ x \ tr$ )  $v$ , we are inserting  $v$  into a tree with root  $x$  at left subtree  $tl$  and right subtree  $tr$ . If  $x$  is the same as  $v$  then we just return that tree ( $\text{Node } tl \ x \ tr$ ), because there is nothing to do. If  $v$  is smaller than  $x$  then we go down the left subtree,  $\text{Node}(\text{insert } tl \ v) \ x \ tr$ . Notice that we modify the left subtree by inserting  $v$  into  $tl$  and then we make that new tree as the left subtree of the overall tree. So you first compute  $\text{insert } tl \ v$  and then make that the left tree of this tree which is given by  $\text{Node}(\text{insert } tl \ v) \ x \ tr$ . Otherwise, which means that  $v$  is greater than  $x$ , you insert  $v$  in the right subtree;  $\text{Node} tl \ x (\text{insert } tr \ v)$ . Again in the worst case, the running time of this routine is proportional to the length of the longest path from root to leaf or the height of the tree.

(Refer Slide Time: 20:29)



Let us now consider deletion from a tree. If you want to delete  $v$  from the tree, if the tree is empty, we can just exit, because there is nothing to delete. If the tree is nonempty, then if  $v$  is smaller than the value at the root then you delete  $v$  from the left subtree as usual, because  $v$  is evidently not the value at the root, because  $v$  is smaller than value at the root. And  $v$  cannot lie on the right subtree, therefore  $v$  has to lie in the left subtree, and we descend down the left subtree and delete  $v$ . If  $v$  is larger than the value at the root, we similarly delete  $v$  from the right subtree.

(Refer Slide Time: 21:21)

## Deleting from a tree

- What if  $v$  is the value at the root?  $v = y$

- What value should replace  $y$ ?

The important case.. The important case is when  $v$  is equal to the value at the root. For example, let's consider the tree with root  $y$ , left subtree consisting of the node  $x$  and its subtree is  $t_1$  and  $t_2$ . Right subtree being a node  $z$  with left subtree  $t_3$  and right subtree  $t_4$ . Now if we want to delete  $v$ , we have to remove  $y$  from this tree. But if you remove  $y$ , what node should be replaced  $y$ ,  $x$  or  $z$  or something else? One can easily see that it is not possible to blindly push  $x$  or  $z$  up the tree.

(Refer Slide Time: 22:17)

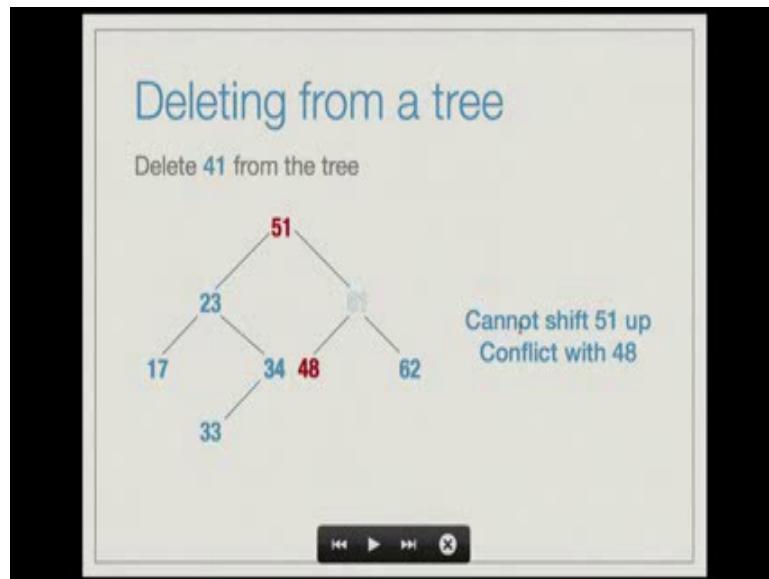
## Deleting from a tree

Delete 41 from the tree

Navigation icons: back, forward, search, etc.

This is illustrated in the following example. Suppose we want to delete 41 from this tree; this is the tree that we considered earlier after we have inserted 33 into that tree.

(Refer Slide Time: 22:34)



Suppose we want to delete 41 from this tree, we cannot shift 23 up, because there is a conflict with the value 34. 34 earlier, it was in the right subtree of a node having the value 23, but now it is part of left subtree of the root and the root has value 23. And any node in the left subtree has to be smaller than the value at the root. And if we push 23 up the tree to the root then there is a conflict between 23 and 34.

Can we therefore push 51 to the root? We cannot shift 51 up to the root, because there is a conflict between 51 and 48. Again 48 lies in the right subtree of the node which contains the value 51. And according to the search tree property, 48 has to be greater than 51, but 48 is smaller than 51. So there is a conflict again.

(Refer Slide Time: 23:36)

## Deleting from a tree

- What if  $v = y$ ?
- Cannot blindly push  $x$  or  $z$  up the tree
- Move up a value that is larger than the left and smaller than the right
- Either maximum value in left subtree, or minimum value in right subtree

How do we resolve this conflict. As we saw earlier we cannot blindly push  $x$  or  $z$  up this tree. The solution is to move up a value to the root that is larger than the left subtree and smaller than the right subtree, which means either the maximum value in the left subtree, which will lie somewhere here, or the minimum value in the right subtree, which will lie somewhere here. So you should pick a value which lies somewhere inside  $t_2$  and replace  $y$  with that or we can pick a value which lies somewhere within  $t_3$  the smallest value in  $t_3$  and replace  $y$  with that.

(Refer Slide Time: 24:44)

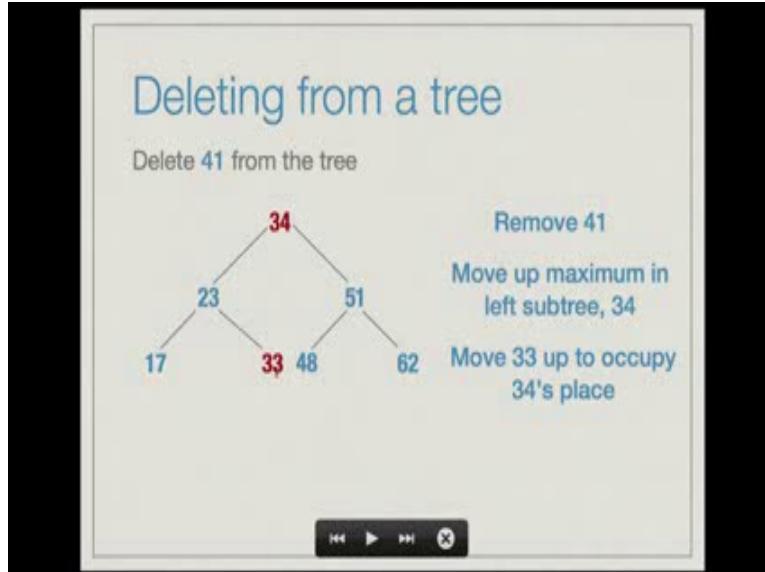
## Deleting from a tree

Delete 41 from the tree

```
graph TD; 34 --> 23; 34 --> 51; 23 --> 17; 23 --> 34; 34 --> 33; 34 --> 48; 51 --> 62;
```

To consider the same example again, suppose we want to delete 41 from this tree. We remove 41, and we find the maximum value in the left subtree which is 34, and we move 34 up to the root, but now this leaves a hole at the position where 34 was.

(Refer Slide Time: 25:05)



One can argue that 34 cannot have a right subtree. If 34 had a right subtree then there would be a value that is larger than 34 in the left subtree of the original tree, therefore 34 will not have a right subtree, can only have a left subtree. We can push the left subtree up to occupy the place of 34. In this case, the left subtree consists of a single node 33, so we move 33 up to occupy 34's position. (Refer Slide Time: 25:32)

## Deleting the maximum value

- Keep going right till you reach a node whose right subtree is empty
- Remove the node
- Replace the node by its left subtree

[Navigation icons: back, forward, search, etc.]

So here is how we delete the maximum value. Keep going right till you reach a node whose right subtree is empty. From x, we keep going right till we see y here and then we see a z here, whose right subtree is empty. At this point, we know that we have found the maximum value in the tree, whose root is x. Now we remove this node. Since it does not have a right subtree, we do not need to worry about that. We just push tz in the place of z, this is how we delete the maximum value of a tree.

(Refer Slide Time: 26:14)

## Deleting the maximum value

```

• deletemax :: Ord a => STree a -> (a, STree a)

    -- At the rightmost node
    deletemax (Node tl x Nil) = (x, tl)

    -- Always descend right
    deletemax (Node tl x tr) = (y, Node tl x ty)
        where (y, ty) = deletemax tr

• deletemax returns the maximum value and the modified
tree
  
```

[Navigation icons: back, forward, search, etc.]

Here is the code, deletemax is a function whose signature is  $\text{Ord } a \Rightarrow \text{STree } a \rightarrow (\text{a , STree } a)$ . Deletemax as you see returns both the maximum value of the tree as well as the modified tree after having deleted the maximum value. There are two cases to consider. Deletemax of (Node tl x Nil) is the first case, this is the case where we are already at the right most node then we know that x is the maximum value, so we return x and we have to return the modified tree after deleting x. In this case, x is the root, we have deleted x, there is nothing to do, so tl is the tree that we get after deleting x. The other case is when there is a non null right subtree. Deletemax of (Node tl x tr), where tr is the right subtree which is not equal to Nil is (y ,Node tl x ty) where y is the maximum value in tr and ty is the tree that you get after deleting y from tr. So deletemax of (Node tl x tr )is the pair (y , node tl x ty), where (y , ty) equals deletemax of tr.

(Refer Slide Time: 27:43)

## Deleting from the tree

```

• delete :: Ord a => STree a -> a -> STree a
  delete Nil v = Nil

  delete (Node tl x tr) v
    | v < x      = Node (delete tl v) x tr
    | v > x      = Node tl x (delete tr v)
    | otherwise   = if (tl = Nil) then tr
                    else (Node ty y tr)
                    where (y, ty) = deletemax tl

  • Worst case: running time proportional to length of the
    longest path from root to a leaf (height)

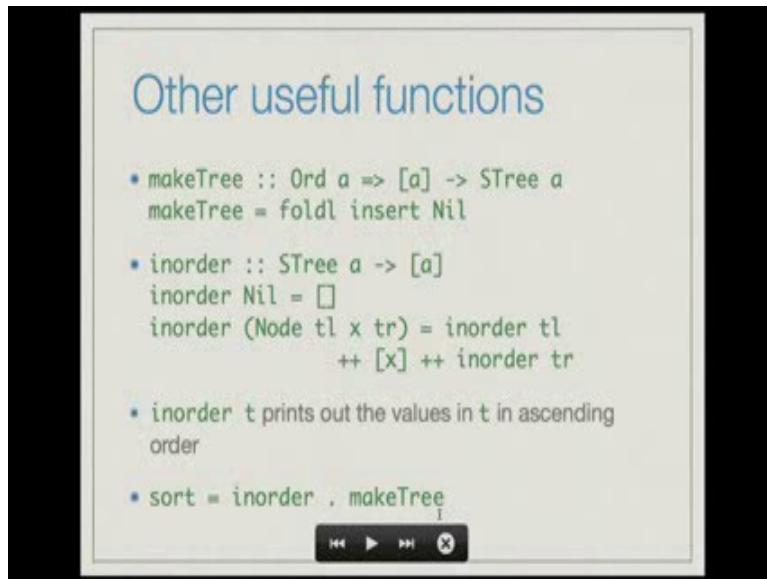
```

Having defined how to delete the largest value in a subtree, let's return to the original task which was to delete an element from a tree. Delete is a function whose signature is  $\text{Ord } a \Rightarrow \text{STree } a \rightarrow a \rightarrow \text{STree } a$ . Delete Nil v = Nil. If you want to delete v from the empty tree, you do not have to do anything.  $\text{delete } (\text{Node tl x tr}) v$ , this is a non trivial tree. In case, v is smaller than x, then you delete v from tl from the left subtree; so in case, v is smaller than x, the result is Node ( delete tl v) x tr. In case v is larger than x, the result is Node tl x (delete tr v). Otherwise this is the case when v is equal to x; if tl is the empty tree, then you have a tree whose root is x, whose right

subtree is tr and whose left subtree is empty and you are deleting x, there is nothing to do, you just return tr.

Else you have a tree, where x is the root, right subtree is tr and left subtree is tl, you delete the maximum from tl and that will give you (y,ty). The maximum element is y, ty is the tree that you get after deleting y from tl; and the new tree that you form is nothing but (Node ty y tr). Recall that our strategy is to find the maximum element in the left subtree and replace x by that value, so y is the maximum element in the left subtree, we replace x by y. So y becomes the new root, ty is the new left subtree and tr is the right subtree as usual. Again in the worst case, this code runs in time proportional to the height of the tree.

(Refer Slide Time: 29:46)

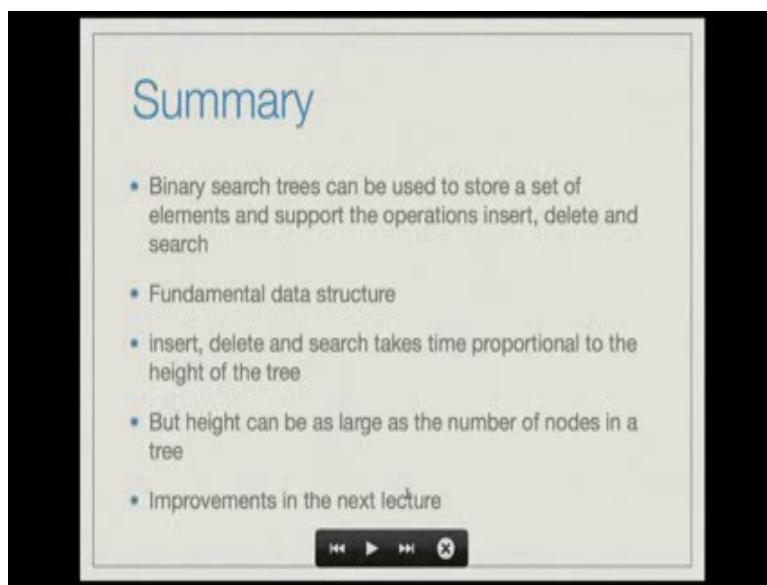


Here are some other useful functions. makeTree which makes a tree from a list of elements; the signature is  $\text{Ord } a \Rightarrow [a] \rightarrow \text{STree } a$ . makeTree is nothing but foldl insert Nil. Recall that foldl is the function that takes a function as the first argument, an initial value as the second argument and then applies this function repeatedly on a list that is given as input by using this as an initial value. So the initial value that we give is the empty Tree denoted by nil. So what this will do is it will first insert the first element of the list into the empty tree. Then it will insert the second element of the list into the resulting tree, the third element into that tree etcetera. So this will

achieve the effect of inserting all the elements in the given list into the tree, into the empty tree, so it forms a tree with all these elements.

inorder is another useful function. inorder of Nil is nothing but empty list. Inorder of (Node t l r) is to recursively print inorder the left subtree and then print the root and then print inorder the right subtree. This is called inorder, because the root is placed in between the inorder of the left subtree and inorder of the right subtree. You can observe that inorder t prints out the values in the tree t in ascending order. And therefore, one way to sort a list of integers or a list of any values of type Ord a is to makeTree with that list and then apply the function inorder on that tree, which we define as inorder.makeTree. The dot syntax in Haskell allows to compose two functions. So sort is the composition of inorder and makeTree.

(Refer Slide Time: 31:58)



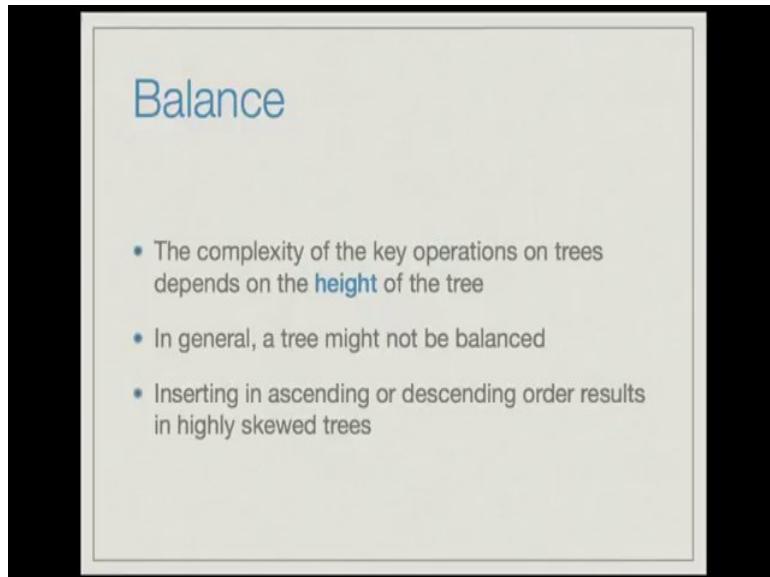
So in summary binary search trees can be used to store a set of elements and support the operations insert, delete and search. It is the fundamental data structure that is widely studied. Insert, delete and search takes time proportional to the height of the tree. But the height of the tree can be as large as the number of nodes in a tree. So we will see how to improve this in the next lecture.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

**Module # 06**  
**Lecture – 03**  
**Balanced Search Tree**

In this lecture, we shall study AVL Trees, which is an example of a Balanced Search Tree.

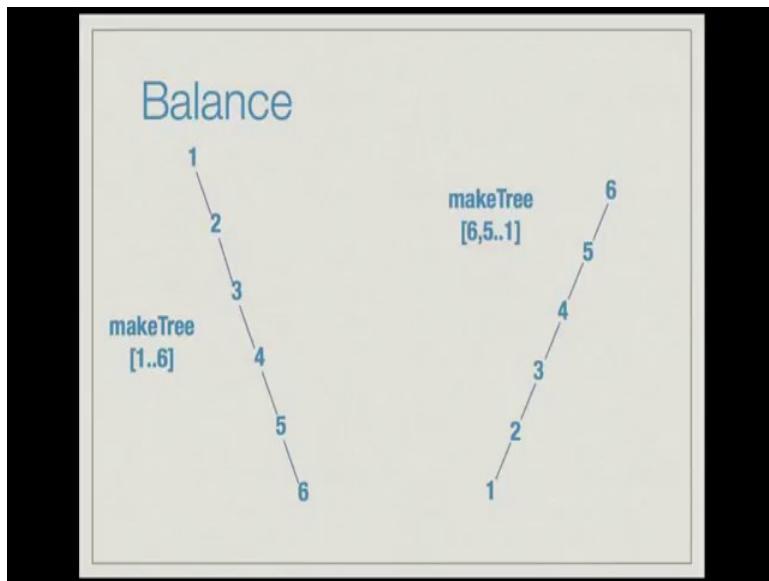
(Refer Slide Time: 00:11)



Recall that in the previous lecture, we studied implementations of the set data type, which maintains a set of elements and supports the operations search, insert and delete. We saw two implementations, one which was to store the elements as a list and the other, which was to store the elements as a binary search tree. The list implementation was considered inefficient, because each search takes time proportional to the length of the list, in other words the number of elements in the set.

So,  $N$  operations can take up to  $O(N^2)$  time. On the other hand, the binary search tree data structure supports insert, search and delete in time proportional to the height of the tree. But, in general, a tree might not be balanced, for instance, inserting elements in ascending or descending order results in highly skewed trees.

(Refer Slide Time: 01:25)



Here are two examples, if we consider `makeTree` of 1 to 6, recall that `makeTree` was a function that starts with the Nil tree and inserts all these elements in the list one after the other. So, `makeTree` of `[1..6]` will create a tree that looks like this; it is skewed to the right. `makeTree` of `[6,5..1]` inserts elements in descending order into the tree and this tree is skewed to the left. It will start with 6, the next element to be inserted is 5, which will go to the left of 6.

The next element to be inserted is 4, which will go to the left of 6 and to the left of 5 and so on. So, in general, it is possible that a tree has height as much as the number of elements on the tree. This means that the operations on a binary search tree might take up to  $O(N)$  for search, insert and delete. So, a sequence of  $n$  operations might still take up to  $O(N^2)$  time.

So, where is the advantage in using a binary search tree? The answer is that we can manage to balance a tree, so that it is of small height. This brings us to the concept of a balanced search tree.

(Refer Slide Time: 02:52)

## Balanced search trees

- Ideally, for each node, the left and right subtrees differ in size by at most 1
  - Height is guaranteed to be at most  $\log N + 1$ , where  $N$  is the size of the tree
  - When size is 1, height is also  $1 = \log 1 + 1$
  - When size is  $N > 1$ , subtrees are of size at most  $N/2$
  - Height is  $1 + (\log N/2 + 1) = 1 + (\log N - 1 + 1)$   
$$= \log N + 1$$



Ideally, we want that for each node, the left and right subtrees differ in size by at most 1. If we maintain this property, then the height of the tree is guaranteed to be at most  $\log N + 1$ , where  $N$  is the size of the tree. The proof is as follows, when the size of the tree is 1, the height is also 1, which is the same as  $\log 1 + 1$ . When the size of the tree is  $N$  greater than 1, then observe that both subtrees are of size at most  $N/2$ , because if each subtree is of size greater than  $N/2$ , then the tree itself would be of size greater than  $N$ .

And now, inductively we can see that the height of each subtree is  $\log N/2 + 1$ , which is the same as  $\log N - 1 + 1$ , which is the same as  $\log N$ . And therefore, the height of the tree itself which is  $1 + (\text{maximum of the height of the two subtrees})$  is  $1 + \log N$ .

(Refer Slide Time: 04:05)

## Balanced search trees

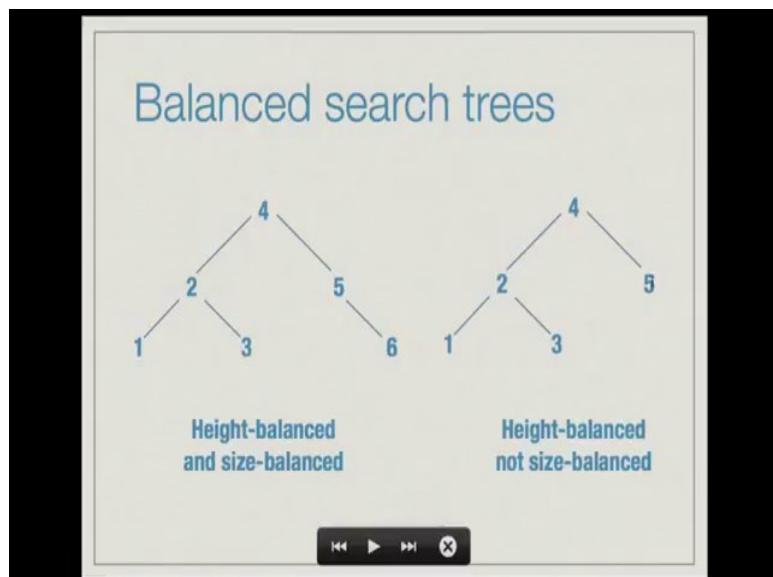
- Not easy to maintain size balance
- Maintain height balance instead
- At any node
  - The left and right subtrees differ in height by at most 1
    - Somewhat easier to maintain: use tree rotations
  - AVL trees (Adelson-Velskii, Landis)
  - Height is still  $O(\log N)$



But, the trouble is that it is not easy to maintain size balance, what we shall ought to do is to maintain height balance instead. This is maintained as follows, at any node we maintain the invariant that the left and right subtrees differ in height by at most 1. In the earlier case, the left and right subtrees differed in size by at most 1. But, now we will maintain the somewhat easier to maintain property of the left and right subtrees differing in height by at most 1.

We maintain this property by using tree rotations, which we will describe later. These trees are called AVL trees, named after the inventors of this data structure Adelson Velskii and Landis. One can prove that, if we maintain this property, the height of the tree is order  $\log N$ , where  $N$  is the size of the tree.

(Refer Slide Time: 05:15)



Here are some examples, on the left, you see a tree that is both height balanced and size balanced. We can check it as follows, the tree is of size 6, there are three nodes in the left subtree; that is the root node here and there are two nodes in the right subtree and you see that the difference in size of left and right subtrees is 1. There are three nodes in the left subtree; there are two nodes in the right subtree. If we look at the left subtree, the difference in size of the left and right subtree of this tree is 0.

Because, the left subtree here is just the single node 1, the right subtree is just the single node 3 and they are of the same size. So, the left part is size balanced. So, if you consider the right subtree of the original tree, it is this tree, 5, with the right child 6 and with no left child. Here, you see that the left subtree is of size 0 and the right subtree is of size 1.

So, in this tree, we maintain size balance throughout, it is also true the height balance is maintained, because on the left, if you take the overall tree, the left subtree is of height 2, and the right subtree is of height 2. If you take this tree, the left subtree is of height 1, the right subtree is also of height 1. So, if you take this subtree the left subtree is of height 0 and the right subtree is of height 1. So, this tree maintains both height balance and size balance.

The tree on the right is height balanced, but it is; however, not size balanced. So, if you take the overall tree, the height of the left subtree is 2, because there is one node here and it has two children, the height of the right subtree is 1, so the difference is 1. If you consider this subtree, there are no children, so it is trivially height balanced, if you consider this subtree, it is also height balanced, because the height of the left subtree is 1 and height of this subtree is 1. So, both the left and right subtrees of this tree are height balanced, but it is not size balanced, because the left subtree has three nodes and the right subtree has only one node.

(Refer Slide Time: 07:52)

**Height balanced trees**

- For a height-balanced tree of size  $N$ , the height is at most  $2 \log N$
- Let  $S(h)$  be the size of the smallest height-balanced tree of height  $h$
- **Claim:** For  $h \geq 1$ ,  $S(h) \geq 2^{h/2}$
- $S(1) = 1 = 2^{1/2}$
- $S(2) = 2 = 2^{2/2}$

◀ ▶ ⏪ ⏩ ✕

Before, we move on to the implementation of a height balanced tree, let us look at the bounds. We claim that for a height balanced tree of size  $N$ , the height is at most twice  $\log N$ , the proof is as follows. Let  $S(h)$  be the size of the smallest height balanced tree of height  $h$ , we claim that for  $h \geq 1$ ,  $S(h) \geq 2^{h/2}$ ,  $S(1) = 1 = 2^{1/2}$ .

Here, we take  $1/2$  to the 0,  $S(2)$  equals 2, which is  $2^{2/2}$ ,  $S(2) = 2$ , because the smallest height balanced tree of height 2 that we can create is a node and one child. So, it is the two node tree.

(Refer Slide Time: 08:45)

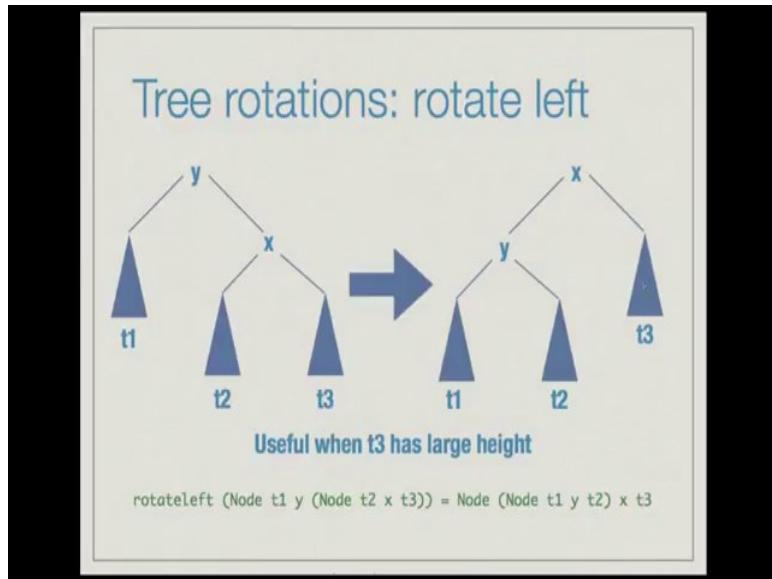
## Height balanced trees

- **Claim:** For  $h \geq 1$ ,  $S(h) \geq 2^{h/2}$
- If a tree has height  $h$ , then one of the subtrees is of height  $h-1$  and the other has height at least  $h-2$
- $$\begin{aligned} S(h) &= 1 + S(h-1) + S(h-2) \geq S(h-2) + S(h-2) \\ &= 2^{(h-2)/2} + 2^{(h-2)/2} \\ &= 2^{(h-2)/2+1} = 2^{h/2} \end{aligned}$$
- A height-balanced tree with  $N$  nodes has height at most  $2 \log N$

In general, if a tree has height  $h$ , then one of its subtrees is of height exactly  $h-1$ , and the other has height at least  $h-2$ . Now, the height of a tree is defined to be  $1 + \max$  of the height of the two subtrees. So, one of the subtrees definitely has to have height exactly  $h - 1$  and the other subtree cannot have height smaller than  $h-2$ , because that would violate the height balance property.

Now,  $S(h)$  in general is  $1 + S(h-1) + S(h-2)$ , which is greater than or equal to this loose bound  $S(h-2) + S(h-2)$ , which by induction hypothesis is  $2(h-2)/2 + 2(h-2)/2$ . This is nothing but, twice  $2(h-2)/2$ , which is  $2(h-2)/2 + 1$ , this simplifies to  $2h/2$ . So, overall we have  $S(h) \geq 2h/2$ , therefore, a height balanced tree with  $N$  nodes has height at most  $2 \log N$ .

(Refer Slide Time: 10:06)



Having established the mathematical bounds, let us look at how to implement a height balanced tree. If you recall, we mentioned that, we maintain height balance by using tree rotations, there are two types of tree notations, rotate right and rotate left, rotate right is defined as follows. We have a tree with root  $x$  and a left subtree with its own root  $y$  and subtrees  $t_1$  and  $t_2$ , the right subtree of the bigger tree is  $t_3$ .

Now, we rotate this to the right by which we mean that will push this  $y$  up to the root and pull the  $x$  down, by pulling the  $x$  down, we mean that we will make  $x$  the right child of  $y$ . But, if so, what do we do with the original right child of  $y$ , we make that the left child of  $x$ , recall that  $y$  was earlier the left child of  $x$ , but now once we pull  $x$  down,  $x$  does not have a left sub child, left child, so we can make  $t_2$  to the left child of  $x$ .

In doing this, we still maintain the binary search tree property, notice that, every node in  $t_1$  is smaller than  $y$ ,  $t_1$  appears to the left of  $y$ ,  $y$  appears to the left of  $x$ ,  $y$  is smaller than  $x$ . But, now  $x$  appears to the right of  $y$  and clearly  $y$  is smaller than  $x$  or in other words,  $x$  is greater than  $y$ ,  $t_2$  appears.. every node in  $t_2$  appears to the right of  $y$ , so it is greater than  $y$ . But, on the other hand, it appears in the left subtree rooted at  $x$ , so it is smaller than  $x$ .

If you look at the new configuration  $t_2$  appears to the left of  $x$ , so every node in  $t_2$  is smaller than  $x$ , but  $t_2$  is part of the right subtree of  $y$ , therefore, every node in  $t_2$  is greater than  $y$ . So, the search tree property, if it was maintained earlier, it would also be maintained after the

rotate. This operation is useful when  $t_1$  has large height, as you can see now earlier the subtree to the left of  $x$  had height  $1 + \text{height of } t_1$ .

Now, the left subtree of the root has only height  $t_1$ , so in this manner, we can reduce the height of the left subtree of the overall tree. The rotate right operation can be implemented in Haskell as follows: rotate right of Node (Node  $t_1$   $y$   $t_2$ ) we just state this pattern here,  $x$  which is the root,  $t_3$  which is the right subtree equals we just write the pattern for this tree. Node  $t_1$ , which is the left subtree,  $y$  which is the root and Node  $t_2$   $x$   $t_3$ , which is the right subtree.

The symmetric operation is rotate left, where the original tree has root  $y$  and right child  $x$  with its own subtrees  $t_2$  and  $t_3$ , the left child of the original tree is  $t_1$ . Now, in case  $t_3$  has large height, we might want to push  $t_3$  up towards the root of the tree, which is what we are doing. We are achieving this by pushing  $x$  towards the root of the tree and pulling  $y$  down. So, when we push  $x$  to the root of tree and pull  $y$  down,  $y$  becomes the left child of  $x$ , the previous left child of  $x$  namely the subtree  $t_2$  now becomes the right child of  $y$ , because now earlier the right child of  $y$  was  $x$ .

Now, the right child of  $y$  has been freed up, so we can let  $t_2$  occupy that position. Again, we can check that, if the original tree satisfies the search tree property after the rotate left also, the tree will continue to satisfy the search tree property. This operation is useful when  $t_3$  has a large height and it is exactly the inverse of rotate right and the Haskell description is again very simple. Rotate left of this pattern, which is specified by Node  $t_1$   $y$  (Node  $t_2$   $x$   $t_3$ ) equals node of left subtree which is (Node  $t_1$   $y$   $t_2$ ) and  $x$   $t_3$ , where  $x$  is the root and  $t_3$  is the right subtree.

(Refer Slide Time: 14:55)

The slide has a light gray background with a dark gray border. The title 'Height balanced trees' is at the top in a blue font. Below the title is a list of four bullet points in black text.

- Assume tree is currently balanced
- Each insert or delete creates an imbalance
- Fix imbalance using a rebalance function
- We need to compute height of a tree (and subtrees) to check for imbalance

Having seen the operations rotate right and rotate left, we will now see, how to use those two operations, assume that the tree is currently balanced, each insert or delete operation performed on the tree might create an imbalance. What we do is to fix the imbalance using a rebalance function, which involves rotate rights and rotate lefts. Before, we actually describe the rebalance function; we will take care of some preliminaries.

We need to compute height of a tree and its subtrees to check for imbalance. Recall that, we are implementing a height balanced tree and imbalance means that, we have a node, such that, one subtree and other subtree differ in height by at least 2.

(Refer Slide Time: 15:54)

## Height balanced trees

- We need to compute height of a tree (and subtrees) to check for imbalance
- $\text{height Nil} = 0$   
 $\text{height } (\text{Node tl } x \text{ tr}) = 1 + \max(\text{height tl}, \text{height tr})$
- This takes  $O(N)$  time
- Save effort by storing height at each node



So, we need to compute the height of a tree and its subtrees to check for imbalance, usually height would be defined as follows, height of Nil = 0, height (Node tl x tr) = 1 + max (height tl) (height tr). But, unfortunately this definition takes O(N) time to compute height and we would not like to spend O(N) time for performing an operation, which would be performed multiple times during each insert and delete. We can save this effort by storing the height at each node.

(Refer Slide Time: 16:41)

## AVL trees

- ```
data AVLTree a = Nil
  | Node (AVLTree a) a Int (AVLTree a)
```
- $\text{height :: AVLTree a} \rightarrow \text{Int}$   
 $\text{height Nil} = 0$   
 $\text{height } (\text{Node tl } x \text{ h tr}) = h$
- We also need a measure of how skewed a tree is: its `slope`
- ```
slope :: AVLTree a -> Int
slope Nil = 0
slope (Node tl x h tr) = height tl - height tr
```

So, the new structure is as follows. `data AVLTree a = Nil | Node (AVLTree a) a Int` (`AVLTree a`) where the first (`AVLTree a`) stands for the left subtree, `a` which stands for the value at the node, `Int` which stands for the height, `AVLTree a`, which stands for the right subtree. Now, the height function is a constant time operation, which is defined as follows, `height Nil = 0, height (Node tl x h tr) = h..`

We also need a measure of how skewed a tree is, namely its slope and we define it as follows, slope of `Nil` is `0`, slope of `(Node tl x h tr)` is nothing but, `height of tl - height of tr`. We use this to determine whether a tree is still balanced or not, if slope is greater than or equal to `2`, then r greater than or less than or equal to minus `2`, then we know that the tree has an imbalance.

(Refer Slide Time: 17:53)

AVL trees: rotates

- Since we store the height at each node, we need to adjust it after each operation
- `rotateright :: AVLTree a -> AVLTree a`  
`rotateright (Node (Node tll y hl tlr) x h tr) =`  
`Node tll y nh (Node tlr x`  
`nhr tr)`  
where  
`nhr = 1 + max (height tlr) (height tr)`  
`nh = 1 + max (height tll) nhr`
- Constant time operation

Let us now give implementations of rotate right and rotate left with the new data declaration. The wrinkle is that, since we store the height at each node, everytime we touch the tree, we need to adjust the height. So, this is the definition of rotate right: `rotateright (Node (Node tll y hl tlr) x h tr) = Node tll y nh (Node tlr x nhr tr)` where `tll` is the left subtree of the left subtree, `tlr` is the right subtree of the left subtree, `hl` is the height of the left subtree, `x` which is the root, `h` which is the height of the tree itself and `tr`, which is the right subtree of the original tree, equals `Node tll y nh (Node tlr x nhr tr)`.

This definition is exactly the same as the definition earlier except that we need to recompute the heights, `nh` is the new height of the overall tree. And `nhr` is the new height of the right subtree, `nhr` clearly is  $1 + \max(\text{height tlr}, \text{height tr})$ , `nh` is  $1 + \max(\text{height tll})$  and

nhr , because for the overall tree nhr is the height of the right subtree and height of tll is the height of the left subtree. Clearly, this is the constant time operation, because we modify one pattern to another pattern and in computing the new heights, we are just comparing a constant number of heights against each other.

(Refer Slide Time: 19:38)

AVL trees: rotates

- Since we store the height at each node, we need to adjust it after each operation
- `rotateleft :: AVLTree a -> AVLTree a`  
`rotateleft (Node tl y h (Node trl x hr trr)) =`  
                                                          `Node (Node tl y nhl trl)`  
`x nh trr`  
where  
`nhl = 1 + max (height tl) (height trl)`  
`nh = 1 + max nhl (height trr)`
- Constant time operation

Here is the symmetric definition for rotate left; we just repeat the earlier definition with the appropriate heights, inserted. And in the result, we have the new heights nhl and nh, where again nhl is defined as  $1 + \max(\text{height of } \text{tl}) \text{ and } (\text{height of } \text{trl})$ , nh is defined as  $1 + \max(\text{nhl} \text{ and } (\text{height of } \text{trr}))$ . This is again a constant time operation.

(Refer Slide Time: 20:12)

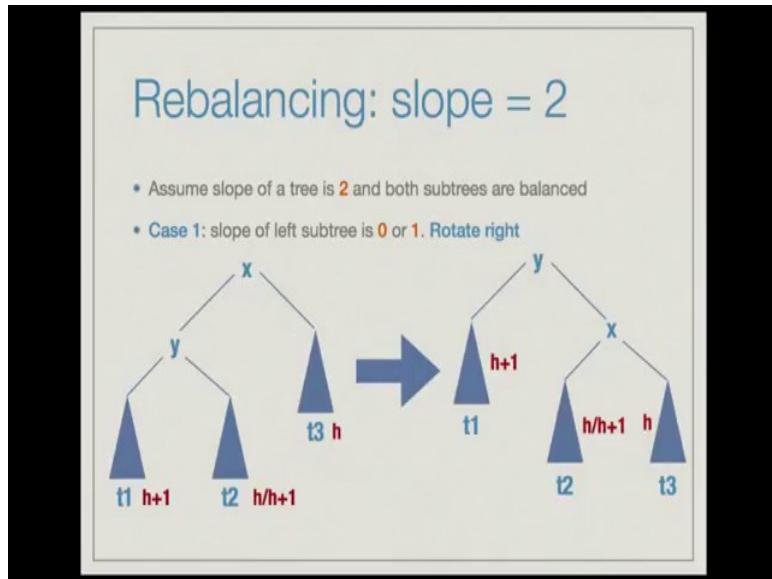
The slide has a light gray background with a dark gray header bar. The title 'Rebalancing trees' is in blue at the top left. Below it is a bulleted list of five items. At the bottom right is a small navigation bar with icons for back, forward, and exit.

- Recall: slope (Node tl x h tr) = height tl - height tr
- In a height balanced tree, slope is -1, 0, or 1
- After an insert or delete, slope can be -2, -1, 0, 1, or 2
- Violations happen only at nodes visited by operation  
    <sup>i</sup>
- We rebalance each node on the path visited by operation

Having looked at the rotates, let us now consider how to rebalance trees, this is the most crucial function in the implementation of AVL trees. Recall that slope of a tree here is the height of the left subtree minus height of the right subtree. In a height balanced tree slope is -1 or 0 or 1, but after an insert or delete, it can happen that the slope is -2 or +2. A slope of -2 or +2 constitutes a violation, it cannot be less than -2 or greater than +2, because we are just inserting one node or deleting one node.

Now, the violations can happen only at nodes that are visited by an operation, recall that to insert or delete, we need to start at the root and search for the node that we need to insert or delete and in doing so, we will be traversing a path from root to that node. Only those nodes along the path will be affected. So, what we need to do is to rebalance each node on the path visited by the operation.

(Refer Slide Time: 21:31)



There are many cases to consider, let us first consider the case where the slope is +2, which means, that the height of the left subtree is exactly two more than the height of the right subtree. The case where slope is -2 is symmetrical to this and we will not be elaborating that, so let us assume that, the slope of a tree is +2 and both subtrees are balanced.

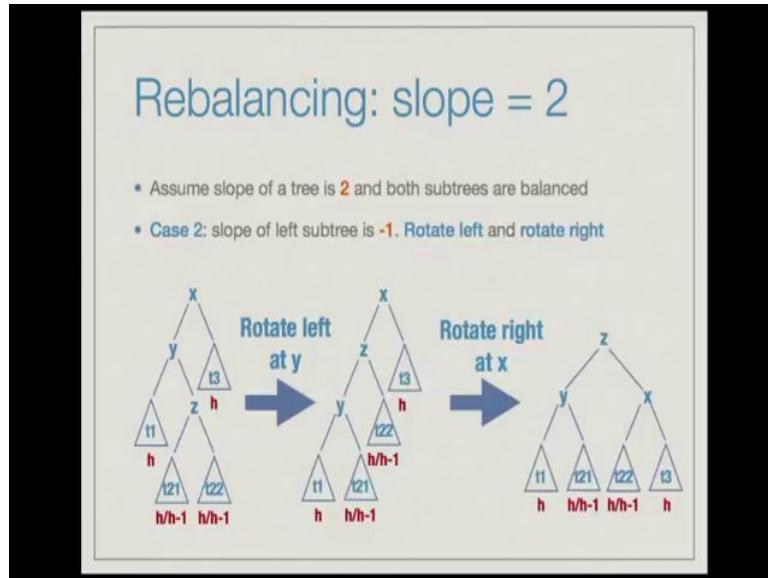
Here, two cases might arise, let us look at the first case, the first case is when slope of left subtree is 0 or 1. Here, is the scenario, we have a root  $x$ , whose left child is  $y$ , which is the root of a tree consisting of  $y$ ,  $t_1$  and  $t_2$ . Now, we said that this slope of the overall tree is 2, therefore the height of the tree rooted at  $y$  is  $h+2$  and height of  $t_3$  is  $h$ . If the height of the tree rooted at  $y$  is  $h+2$ , it means that one of the children is of height  $h+1$  and the other child is of height  $h$  or  $h+1$ .

Since, we say that the slope of the left subtree is 0 or 1, therefore it has to be the case that the height of  $t_1$  is at least as much as the height of  $t_2$ . If the height of  $t_1$  were smaller than the height of  $t_2$ , then the slope of the tree rooted at  $y$  would actually be -1. Since, we say that the slope of the left subtree is 0 or 1, it has to be the case that  $t_1$  is of height of  $h+1$  and  $t_2$  is of height either  $h$  or  $h+1$ , this is the scenario.

In which case, all we need to do is rotate right, this brings  $y$  up to the root, it brings  $h$  down to be the right child of  $y$  and it shifts  $t_2$  from being the right child of  $y$  to the left child of  $x$ , everything else is unchanged. Let us look at the slopes of the various subtrees before and after the operation, earlier the slope of  $x$  was 2 and slope of the left subtree was 0 or 1.

Now, the slope of the new tree is height of left subtree, which is  $h + 1$  - height of the right subtree, which is either  $1 + h$  or  $1 + h + 1$ . So, the height of the right subtree is either  $h+1$  or  $h+2$ . So, in doing this rotate right, we have managed to bring this slope from 2 to either 0 or -1. This is for the root node, for the right subtree, the slope is just +1. So, the new tree that we get is height balanced.

(Refer Slide Time: 24:49)



Here is the next case, this is the case when the slope of the tree is 2 and both subtrees are balanced, but the slope of the left subtree is -1. So, the tree looks as follows, we have the root x, whose left child is y, which has subtrees t1 and t2, but expanded t2 into it is root z and the two subtrees, t21 and t22, the right subtree of x is t3. Since, this slope of the tree is 2, the height of the left subtree rooted at y is  $h + 2$  and the height of t3 is  $h$ .

But, the slope of the left subtree is -1, which means that the height of t1 is  $h$  and the height of the subtree rooted at z is  $h+1$ . For the height of the subtree rooted at z to be  $h + 1$ , it has to be the case that either t21 or t22 is of height  $h$  and the other is of height  $h-1$ . So, this is the scenario, now we achieve a height balanced tree in two steps, in the first step, we rotate left at y, so what happens is that, we just concentrate on the left subtree of the original tree.

This portion and we rotate left at y, which means that, we pull z up, we push y down that is what we have done. But, when you pull z up and push y down, you make y the left child of z, which means that t21 needs a place to go and it goes as the right subtree of y, which is what happens here. You have t1 choose the original left subtree of y, still being the left subtree of

y, but you have t21, which will originally be to the left of z, now to the right of y, z has moved up and become the parent of y and t22 is as before the right subtree of the tree rooted at z, we only changed this portion.

Now, what we have achieved is to balance the subtree rooted at y. So, if you consider this subtree rooted at y, the slope is either 1 or 0, but the subtree rooted at z might have an imbalance and the imbalance will occur in case t22 is of height h-1. Because, the left subtree rooted at y, now has height h+1, because t1 is of height h, but the right subtree has height only if the right subtree has height only h-1, then there will be an imbalance of z.

And this imbalance would be caused by its slope being 2. To fix this situation we do a simple trick which is to do a rotate right at x. If you do a rotate right at x, what happens is that x goes down as the right child of z, z becomes the new root, t22 becomes the left child of x, this is the situation. We have z as the root y, as the left child of z and t1 and t21 being the subtrees of the tree rooted at y as before as here. But, we have done a rotate right at x, so z is the root, x is the right child and t22 and t3 are children or subtrees of the tree rooted at x.

Now, let us compute the various heights. height of t1 has not changed yet. height of t21 is either h or h-1. height of t22 is either h or h-1, t3 is of height h. now, height of the subtree rooted at y is h+1 and height of the subtree rooted at x is also h+1. So, we have got a tree whose slope is 0 and balance has been restored.

(Refer Slide Time: 29:07)

The slide has a light gray background with a dark gray header bar. The title 'Rebalancing: slope = -2' is centered in the header in a blue font. Below the title is a list of bullet points in black font:

- Symmetric to the **slope = 2** case
- Two subcases:
  - slope of right subtree is **0 or -1**
  - slope of right subtree is **1**
- Handled symmetrically

At the bottom of the slide is a dark gray footer bar with white icons for navigation: back, forward, and exit.

Rebalancing for the case when slope equals -2 is symmetric to the slope equals 2 case. We have two sub cases, one where the slope of the right subtree of the root is either 0 or -1 and one where the slope of the right subtree is 1 and the cases are handled symmetrically.

(Refer Slide Time: 29:35)

The slide has a title 'The rebalance function' in blue. Below it is a code block in Haskell:

```

* rebalance :: AVLTree a -> AVLTree a
rebalance (Node tl x h tr)
| abs (st) < 2           = Node tl x h tr
| st == 2 && stl /= -1 = rotateright (Node tl x h tr)
| st == 2 && stl == -1 = rotateright (Node
                                (rotateleft tl) x h tr)
| st == -2 && str /= 1 = rotateleft (Node tl x h tr)
| st == -2 && str == 1 = rotateleft (Node tl x h
                                (rotateright tr))
where
  st          = slope (Node tl x h tr)
  stl         = slope tl
  str         = slope tr

```

\* Constant time operation

So, here is a Haskell implementation of the rebalance function, rebalance is a function from which takes an input AVLTree a and produces an output of type AVLTree a. Rebalance of (Node tl x h tr) is in the case, where the absolute value of st, obs(st), is less than 2, st is the slope of the tree, slope of Node tl x h tr. Just for references tl is the slope of the left subtree, tl, str is the slope of the right subtree tr.

So, in the case where the absolute value of st is less than 2 which means that st is either -1 or 0 or 1, then you just return the tree as it is. In case when st is equal to 2 and stl is not equal to -1, which means that stl is either 0 or 1, this is the first subcase we looked at, then you just do a rotate right at the root x. In case the slope is 2 and the slope of the left subtree is -1, then you first do a rotate left of the left subtree and then, you do a rotate right of the overall tree.

And the cases for the slope being -2 are symmetric, if the slope of the tree is -2 and slope of the right subtree is not equal to 1, then you do a rotate left at the root. If the slope is -2 and the slope of the right subtree is +1, then you first do a rotate right of the right subtree and then, you do a rotate left at the root. It is left as an exercise for the reader to work out the symmetric cases and ensure themselves of the correctness.

Notice that, rebalance is a constant time operation, because we just do either a rotate right or a rotate left or a rotate right and a rotate left. Recall that, rotate left and rotate right are themselves constant time operations and apart from this, we just compute the slope, which is again a constant time operation.

(Refer Slide Time: 31:56)

The slide has a title 'Searching in a tree' at the top. Below the title is a bullet point containing a Haskell-like code snippet for a search function:

```
• search :: Ord a => AVLTree a -> a -> Bool
  search Nil v      = False
  search (Node tl x h tr) v
    | x == v        = True
    | v < x         = search tl v
    | otherwise      = search tr v
```

Below the code snippet is another bullet point:

- Time taken: proportional to height ( $= 2 \log N$ )

At the bottom of the slide is a navigation bar with icons for back, forward, and exit.

Now, let us consider the key functions on an AVL tree, namely search, insert and delete. Search is as usual, search of Nil v is false, search of (Node tl x h tr); we are searching for a value v in a tree given by this pattern. If x is the same as v, then you return true, if v is smaller than x, then you have to descend down the left subtree. So, you do a search tl, search tlv, otherwise you do a search trv and the time taken is proportional to the height of the tree. But, now we have proved that in a height balanced tree, the height is  $2 \log N$ , where N is the size of the tree. So, therefore search takes  $O(\log N)$  time.

(Refer Slide Time: 32:46)

## Inserting in a tree

```
* insert :: Ord a => AVLTree a -> a -> AVLTree a
insert Nil v          = Node Nil v 1 Nil
insert (Node tl x h tr) v
| x == v            = Node tl x h tr
| v < x             = rebalance (Node ntl x nhl tr)
| otherwise          = rebalance (Node tl x nhr ntr)
where
  ntl      = insert tl v
  ntr      = insert tr v
  nhl      = 1 + max (height ntl) (height tr)
  nhr      = 1 + max (height tl) (height ntr)

• Time taken: proportional to height (= 2 log N)
```

More important is insert and delete, here we see how to use rebalance, insert is a function which has the signature `Ord a => AVLTree a -> a -> AVLTree a` which are in the order of input tree, value to be inserted and the output tree . Insert v in the Nil tree is nothing but, `Node Nil v 1 Nil`, notice that, we enter the height of the tree, when we create the single node tree. Insert `(Node tl x h tr) v` is as usual, if  $x$  is equal to  $v$ , then you do not need to do anything. So, you return the tree itself. If  $v < x$ , then we have to insert  $v$  in the left subtree.

So, we do a rebalance after we insert  $v$  into the left subtree. So, we do a rebalance of `(Node ntl x nhl tr)` where  $ntl$  is `insert tl v`,  $nhl$  is the new height of the left subtree, which we calculate by  $1 + \max(\text{height } ntl \text{ and } \text{height } tr)$ . otherwise, this is the case when  $v > x$ , in this case, we have to insert  $v$  in the right subtree. So, we insert  $v$  in the right subtree and get  $ntr$ .  $ntr$  is `insert of tr v` and  $nhr$  is  $1 + \max(\text{height } tl \text{ and } \text{height } ntr)$  and then, we do a rebalance.

Again, the time taken is proportional to height and we do a rebalance at each node; that is touched on the way to insert, but rebalance is a constant time operation and height is proportional to  $\log N$ . So, therefore, insert again is an operation that takes time proportional to  $\log N$ .

(Refer Slide Time: 35:07)

The slide has a light gray background with a dark border. At the top, the title 'Deleting from a tree' is centered in a blue font. Below the title is a code block in Haskell. The code defines a function 'delete' with type `Ord a => AVLTree a -> a -> AVLTree a`. It handles three cases based on the value `v` relative to the root `x`: if `v < x`, it rebalances the left subtree; if `v > x`, it rebalances the right subtree; otherwise, it checks if the left subtree is Nil, and if so, returns the right subtree, otherwise, it rebalances the entire tree. Below the main function, there is a 'where' clause defining helper functions: `y` and `ty` are set to `deletemax tl`; `ntl` is `delete tl v`; `ntr` is `delete tr v`; `nhl` is `1 + max (height ntl) (height tr)`; `nhr` is `1 + max (height tl) (height ntr)`; and `hyr` is `1 + max (height ty) (height tr)`. A note at the bottom states: 'Time taken: proportional to height (= 2 log N), assuming deletemax behaves well'. At the bottom of the slide is a navigation bar with icons for back, forward, and search.

```
* delete :: Ord a => AVLTree a -> a -> AVLTree a
  delete Nil v          = Nil
  delete (Node tl x h tr) v
    | v < x           = rebalance (Node ntl x nhl tr)
    | v > x           = rebalance (Node tl x nhr ntr)
    | otherwise        = if (tl == Nil) then tr else
                           rebalance (Node ty y hyr tr)

  where
    (y, ty)           = deletemax tl
    ntl               = delete tl v
    ntr               = delete tr v
    nhl               = 1 + max (height ntl) (height tr)
    nhr               = 1 + max (height tl) (height ntr)
    hyr               = 1 + max (height ty) (height tr)

  • Time taken: proportional to height (= 2 log N), assuming deletemax behaves well
```

Here is `delete`, `delete Nil v` is just `Nil`, `delete v` from a tree of the form `(Node tl x h tr)`, again we consider three cases, if `v` is smaller than `x`, then we delete `v` from the left subtree and get `ntl`, `ntl` is `delete tl v`, the new height of the left subtree is `nhl`. This is computed as usual  $1 + \max(\text{height } ntl) (\text{height } tr)$  and we take this tree and rebalance it. And the case where  $v > x$ , we delete `v` from the right subtree and we rebalance the overall tree.

The crucial case is, when `v` is equal to `x` in which case, we do deletion as in the binary search tree case, we first check if the left subtree is `Nil`, if that is the case, then we just replace the tree `(Node tl x h tr)` by just `tr`. Because, we are deleting `x` and there is no left subtree of `x`, so `tl` is `Nil` and `tr` is the right subtree of `x`, `tr` can take the place of `x`, so this is what we return.

Otherwise, we consider the maximum element of the left subtree, which is `y`, which is gotten by `deletemax` of `tl`, this `deletemax` is exactly as we saw earlier and `ty` is the left subtree after the maximum element has been deleted. And we just replace the node `x` by `y` here and `tl` is replaced by `ty` and `hyr` is the new height and we rebalance this tree. Again, the time taken by this operation is proportional to height which is  $2 \log N$  assuming `deletemax` behaves well.

(Refer Slide Time: 37:14)

The slide title is **deletemax**. The content is as follows:

- `deletemax :: AVLTree a -> (a, AVLTree a)`  
`deletemax (Node tl x h Nil) = (x, tl)`  
-- At the rightmost node  
`deletemax (Node tl x h tr) =`  
`(y, rebalance (Node tl x nh ty))`  
-- Always descend right  
where  
`(y, ty) = deletemax tr`  
`nh = 1 + max (height tl) (height ty)`
- Time taken: proportional to height (= **2 log N**)

At the bottom of the slide is a navigation bar with icons for back, forward, and exit.

But, delete max does indeed behave well and the implementation is just like earlier. Modulo adjusting the heights, again time taken is proportional to  $2 \log N$ .

(Refer Slide Time: 37:38)

The slide title is **Summary**. The content is as follows:

- Each operation (insert, delete, search) on an AVL tree takes  $O(\log N)$  time
- A sequence of  $N$  operations takes  $O(N \log N)$  time
- Fundamental, but non-trivial data structure
- Excellent example of the power of Haskell
- Mathematical definitions transcribed almost directly to code

In summary, we have defined AVL trees which supports a set data structure, supporting the operations insert, delete and search, each of which take  $O(\log N)$  time, where  $N$  is the size of the set. Therefore, a sequence of  $N$  operations takes  $O(N \log N)$  time, this is the fundamental, but non-trivial data structure and is an excellent example of the power of Haskell. As you

may have seen during the course of this extended example, the mathematical definitions pertaining to height balanced trees could be transcribed almost directly to Haskell code.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

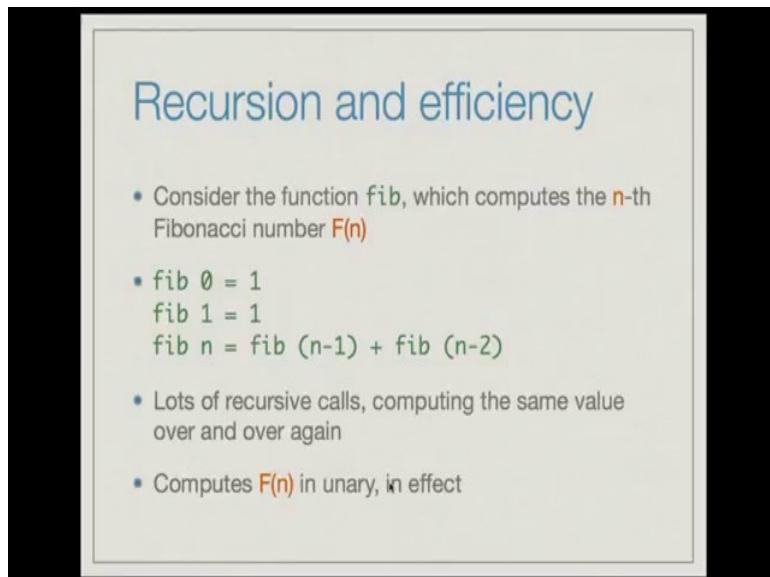
**Module # 07**

**Lecture - 01**

**Arrays**

In this lecture, we shall look at a new topic namely arrays in Haskell.

(Refer Slide Time: 00:07)



The slide has a light gray background with a dark gray border. The title 'Recursion and efficiency' is at the top in a blue font. Below it is a bulleted list:

- Consider the function `fib`, which computes the  $n$ -th Fibonacci number  $F(n)$
- $\text{fib } 0 = 1$   
 $\text{fib } 1 = 1$   
 $\text{fib } n = \text{fib } (n-1) + \text{fib } (n-2)$
- Lots of recursive calls, computing the same value over and over again
- Computes  $F(n)$  in unary, in effect

Arrays are generally used to make programs more efficient, we will illustrate this with the following example. Consider the function `fib`, which computes the  $n$ th Fibonacci number  $F(n)$ . `fib 0` equals 1, `fib 1` equals 1, `fib n` equals `fib (n-1) + fib (n-2)`. You can see that this program is a straightforward translation of the mathematical definition of the Fibonacci series.

The program is quite simple, but the problem with it is that there are lots of recursive calls computing the same value over and over again. For instance, `fib 3` makes two calls, one to `fib 2` and another to `fib 1`. `fib 2` in turn makes a recursive call to `fib 1` and `fib 0`. So, you see that already `fib 1` is being called twice, if you consider a call like `fib 5`, then you can imagine that there are more calls to `fib 1`, which in a `fib` recomputes the same value again and again. In fact, one can see that this program computes  $F(n)$  unary in effect.

(Refer Slide Time: 01:27)

The slide has a light gray background with a dark border. At the top, the title "Recursion and efficiency" is centered in a blue font. Below the title, there is a bulleted list of text and code snippets:

- Let  $G(n)$  be the number of recursive calls to `fib 0` in the computation of `fib n`, for  $n > 1$
- $G(2) = 1$       - one call to `fib 0`  
 $G(3) = 1$       - one call to `fib 0`
- **Claim:**  $G(n) = F(n-2)$   
**Proof:**  
True for  $n = 2$  and  $n = 3$ .  
For  $n > 3$ ,  $G(n) = G(n-1) + G(n-2)$ , since there is one call to `fib (n-1)` and one to `fib (n-2)`.  
But  $G(n-1) = F(n-3)$  and  $G(n-2) = F(n-4)$ , by induction hypothesis.  
Thus  $G(n) = F(n-3) + F(n-4) = F(n-2)$ .

To make this formal, let  $G(n)$  be the number of recursive calls to `fib 0` in the computation of `fib n` for  $n > 1$  it is easy to see that  $G(2)$  equals 1, because `fib 2` = `fib 1` + `fib 0`. So, there is one recursive call to `fib 0`, it is also easy to see that  $G$  of 3 equals 1, because `fib 3` = `fib 2` + `fib 1`, `fib 2` makes a recursive call to `fib 1` and another recursive call it `fib 0`, `fib 1` is defined directly, so there is one recursive call to `fib 0`.

We claim that  $G(n)$  equals  $F(n-2)$ , for a proof we see that the statement is true for  $n = 2$  and  $n = 3$  for  $n > 3$ ,  $G(n)$  equals  $G(n) + (n-1) + G(n-2)$ . Because there is one call to `fib n-1` and one call to `fib n-2` in `fib n` and there are  $G(n-1)$  calls to `fib 0` in `fib n-1` and  $G(n-2)$  calls to `fib 0` in `fib n-2`. But, by induction hypothesis we know that  $G(n-1)$  equals  $F(n-3)$  and  $G(n-2)$  equals  $F(n-4)$ .

Therefore,  $G(n)$  equals  $F(n-3) + F(n-4)$ , which is  $F(n-2)$ , which is a large number, which grows exponentially in the size of  $n$ . Thus, we see that there are exponentially many calls to `fib 0` in a computation of `fib n`.

(Refer Slide Time: 03:09)

## Recursion and efficiency

- How do we fix this?
- Store the computed values ([in an array](#)) and use them
- In a language like **C**, we would have this code:

```
int fibs[n];
fibs[0] = fibs[1] = 1; i = 2;
while (i <= n) {
    fibs[i] = fibs[i-1] + fibs[i-2];
    i++;
}
return fibs[n];
```

How do you fix this situation? One easy way to do this in other languages, is to just store the computed values in an array and use the values from the array rather than re computing them again and again. In a language like C for instance, we would have the following code, we have an array of fibs which store all the Fibonacci numbers from  $f_0$  to  $f_{n-1}$ . You initialize the array by saying  $fibs[0] = fibs[1] = 1$ , then you fill in entries in the array from index 2 till index  $n-1$ .

So, you initialize  $i = 2$ , while  $i$  is less than  $n$ , you just compute  $fibs[i]$  to be  $fibs[i-1] + fibs[i-2]$ . This computation does not involve a recursive call, rather it just picks up two values from the array, adds them and stores it in the new value of the array. Finally, you return  $fibs$  of  $n$ , which is the  $n$ th entry in the array; this program actually takes time proportional to  $n$  rather than proportional to  $2n$  as in the earlier case.

(Refer Slide Time: 04:40)

## Recursion and efficiency

- We can simplify this even more, since only the last two elements of the fibs array are needed

```
• int prev = 1, curr = 1, i = 2;
  int temp;
  while (i <= n) {
    temp = prev;
    prev = curr;
    curr = temp + prev;
    i++;
  }
  return curr;
```

We can simplify this program even more by observing that only the last two elements of the fibs array are ever needed. So, you can have two variables, previous and current storing the last two values of the fibs array, then you run a loop from for i equals 2 to n, where you move the current value to the previous, to the value previous and you move the sum of the two variables to current. In this way you just keep track of the last two entries of the fibs array and finally, you return the value of the current variable.

(Refer Slide Time: 05:26)

## Recursion and efficiency

- Linear-time Fibonacci in Haskell. [Laziness to the rescue!](#)

```
• fastfib n = fibs !! n
  fibs :: [Integer]
  fibs = 1 : 1 : zipWith (+) fibs (tail fibs)

  • 1:1:zipWith (+) [1,1,...] [1,...]
    => 1:1:(1+1):zipWith (+) [1,2,...] [2,...]
    => 1:1:2:(1+2):zipWith (+) [2,3,...] [3,...]
    => 1:1:2:3:(2+3):zipWith (+) [3,5,...] [5,...]
    => 1:1:2:3:5:...
    => ...
```

We can also program a linear time Fibonacci function in Haskell by using the power of laziness. Here is the function `fastfib n`, which just builds the Fibonacci series in a list and extracts the nth entry, `fibs` is a function with signature list of integer and it is given by this

program fibs = 1 : 1 : zipWith(+) fibs( tail fibs). The way the computation unfolds is as follows, this is the expression 1:1: zipWith (+) fibs (tail of fibs).

But, fibs we know now is 1,1,... some other entries, tail of fibs is 1,... some other entries. So, now, zipWith will add this 1 with this 1 and do a zipWith of + on the tail of the two lists. So, it will be the tail of this list, but we now know that the second entry in this list is 2, because that has been computed. So, you will have 1:1 :2: zipWith using (+) [1,2,...] and the tail of this list which is [2,...]

This in turn will give rise to 1:1:2:3: zipWith (+) on the two lists, [2,3,...] and its tail, which is [3,...] and so on. You see that you finally, end up with the list 1:1:2:3:5 etcetera, which is an infinite list that contains all the Fibonacci numbers extracting the nth element gives you the nth Fibonacci number.

(Refer Slide Time: 07:31)

### Another example: lcss

- Given two strings `str1` and `str2`, find the length of the longest common subsequence of `str1` and `str2`
- `lcss "agcat" "gact" = 3`
  - "gat" is the subsequence
- `lcss "abracadabra" "bacarrat" = 6`
  - "bacara" is the subsequence

That is fine, but now let us consider the another example; this is the example of computing the longest common sub sequence of two strings, str1 and str2. We in fact, want to just compute the length of the longest common sub sequence of str1 and str2. For instance lcss “agcat” “gact” = 3, because “gat” is a subsequence of “agact” and “gat” is also a sub sequence of “gact”, that is the longest common subsequence, lcss of “abracadabra” and “bacarrat” equals 6, because “bacara” is a common sub sequence and that is the longest sub sequence b a c a r a, here also you see b a c a r a.

(Refer Slide Time: 08:27)

Another example: lcss

- $\text{lcss}("") = 0$
- $\text{lcss}(_) = 0$
- $\text{lcss}(c:cs)(d:ds)$ 
  - |  $c == d = 1 + \text{lcss}(cs, ds)$
  - | otherwise =  $\max(\text{lcss}(c:cs, ds), \text{lcss}(c:cs, (d:ds)))$
- $\text{lcss}(cs, ds)$  takes time  $\geq 2^n$ , when  $cs$  and  $ds$  are of length  $n$
- Similar problem to fib, same recursive call made multiple times
- Store the computed values for efficiency

How do you program this? Well,  $\text{lcss}("") = 0$ .  $\text{lcss}$  of the empty string with anything is 0, because the empty string does not have any subsequence. Or you could say the empty string has only itself as a subsequence and the empty string is always a subsequence of any other sequence.  $\text{lcss}(_) = 0$ .  $\text{lcss}$  of anything else and the empty string is also 0,  $\text{lcss}$  of two non empty strings, which are given by  $(c:cs)$  and  $(d:ds)$ , is computed as follows.

In the case that  $c$  is equal to  $d$ , then you know that any subsequence any common subsequence of  $cs$  and  $ds$  can be extended by adding  $c$  to the front and that will give you a subsequence of length one longer. So, if you have something to be the longest common subsequence of  $cs$  and  $ds$ , which is computed by  $\text{lcss}(cs, ds)$ , we can always add 1 to it, you can always add one letter to the front namely  $c$  and get a common subsequence whose length is 1 longer than the length of  $\text{lcss}(cs, ds)$ .

Otherwise this is the case when  $c$  is not equal to  $d$ , then it is clear that the first letter is not the same. So, therefore, in the longest common subsequence of  $(c:cs)$  and  $(d:ds)$  is either the longest common subsequence of  $(c:cs)$  with  $ds$  or the longest common subsequence of  $cs$  with  $(d:ds)$  which is exactly, what we are doing here. In the case that  $c$  is not equal to  $d$ , the longest common subsequence, the length of the longest common subsequence of these two lists is the same as  $\max(\text{lcss}(c:cs, ds), \text{lcss}(cs, (d:ds)))$ .

One can prove that  $\text{lcss}$  of  $cs$  and  $ds$ , which are two strings takes time at least  $2n$ , where  $cs$  and  $ds$  are of length  $n$ . There is a similar problem to fib, in that the same recursive call is

made multiple times with the same arguments. So, the easiest way out is to store the computed values for efficiency, store the values computed and extract the values later rather than re computing them.

(Refer Slide Time: 10:57)

The slide has a light gray background with a dark gray border. At the top left, the title 'Linear-time sort' is written in a blue sans-serif font. Below the title, there is a bulleted list of five items, each preceded by a small black square bullet point:

- Given a list of  $n$  integers, each between 0 and 9999, sort the list
- Easy to do with arrays
- Count the number of occurrences of each  $j \in \{0, \dots, 9999\}$  in the list, storing in an array `counts`
- Output `count[j]` copies of  $j$ ,  $j$  ranging from 0 to 9999

Here is another example, which is that of linear time sort, we already know that programs like merge sort take  $O(n \log n)$  time. And one can even prove a lower bound for sorting that to sort a list, to sort an array of length  $n$  you need  $n \log n$  time, but these are for algorithms which are based on comparison. In linear time sort, we will follow a different idea under a crucial assumption, we have given a list of  $n$  integers; such that each integer is between 0 and 9999 so.

So each of the integers lie between a pre specified range and now, we want to sort this list. Suppose the list were stored in arrays, what we could do is to count the number of occurrences of each number between 0 and 9999 in this list and store it in a different array, which stores all the counts. Now, we just need to output  $\text{count}[j]$  copies of  $j$ , where  $j$  ranges from 0 to 9999 that will actually produce the sorted array.

(Refer Slide Time: 12:20)

## Linear-time sort

```
• // Input - int arr[n];
int counts[10000], output[n];
for (j = 0; j < 10000; j++)
    counts[j] = 0;
for (i = 0; i < n; i++)
    counts[arr[i]]++;
last = 0;
for (j = 0; j < 10000; j++)
    for (i = 0; i < counts[j]; i++)
        output[last] = j, last++;
```

- This works in time  $O(n+10000)$  time

Here is the program that realizes it, you have an array counts which holds 10000 entries, entries ranging from counts 0 to counts 9999. You have an input array which is of size n, you have an output array which is of size n, you first fill in the counts array with the appropriate, you first initialize the counts array to 0. And then, you walk through the input array for i equals 0 till n-1, you look at each array entry and then update the appropriate counts entry.

So, if  $\text{arr}[i]$  equals 500 let say,  $\text{counts}[500]$  will be incremented by 1. Now, the final part is just to go through the counts array and output so many copies of j, for j ranging from 0 to 9999, you look at  $\text{counts}[j]$  and output, so many copies of j, this will give you the sorted version of the original array. This algorithm works in time  $O(n+10000)$ , so if n is much larger than 10000, we have actually managed to sort an array of size n in linear time.

(Refer Slide Time: 13:48)

## Arrays in Haskell

- Lists store a collection of elements
- Accessing the  $i$ -th element takes  $i$  steps
- Would be useful to access any element in constant time
- **Arrays** in Haskell offer this feature
- The module **Data.Array** has to be imported to use arrays

To achieve all this in Haskell, we need to actually use arrays. We know that lists store a collection of elements, the crucial point about lists is that accessing the  $i$ th element takes  $i$  steps. It would be useful to access any element in constant time and this feature is offered by arrays in Haskell. To use arrays, you need to import the module **Data.Array**.

(Refer Slide Time: 14:23)

## Arrays in Haskell

- `import Data.Array`  
`myArray :: Array Int Char`
- The **indices** of the array come from **Int**  
The **values** stored in the array come from **Char**
- `myArray = listArray (0,2) ['a','b','c']`

| Index | 0   | 1   | 2   |
|-------|-----|-----|-----|
| Value | 'a' | 'b' | 'c' |

Here is how we use this import **Data.Array** and then, you want to declare an array, let say `myArray`, the type of an array is given like this, for instance `myArray` is of type `Array Int Char`. Here the indices of the array come from `Int` and the values stored in the array come from `Char`. For instance, if you say that `myArray` equals `listArray (0,2) ['a','b','c']`, then it produces this array. This is an array with three indices 0 1 and 2 and the values a at index 0, b

at index 1 and c at index 2. This notation says that we have to create an array from the given list with indices lying between 0 and 2.

(Refer Slide Time: 15:31)

Creating arrays: listArray

- `listArray :: Ix i => (i,i) -> [e] -> Array i e`
- Ix is the class of all `Index` types, those that can be used as indices in arrays
  - If Ix a, x and y are of type a and  $x < y$ , then the range of values between x and y is defined and finite
- Ix includes `Int`, `Char`, `(Int, Int)`, `(Int, Int, Char)` etc. but not `Float` or `[Int]`
- The first argument of `listArray` specifies the smallest and largest index of the array
- The second argument is the list of values to be stored in the array

We look at `listArray` in more detail now. `listArray` has type `Ix i` implies the pair `(i,i) -> [e] -> Array i e`. `Ix` is the type class of all index types, which are those types that can be used as indices in array. If `Ix a` holds and  $x$  and  $y$  are of type  $a$  and  $x < y$ , then the range of values between  $x$  and  $y$  is defined and finite. This is the property of the type class `Ix`. For instance `Ix` includes the types `Int`, `Char`, `(Int, Int)`, the triple `(Int, Int, Char)` etc, but not `Float` or `[Int]`.

Because, if you take two floating point values  $x$  and  $y$  and let, say  $x < y$ , then the range of values between  $x$  and  $y$  is not finite. Similarly, if you take the list, let us say the list consisting of the single element 1 and the list consisting of the single element 2 there are infinitely many lists that lie in between these two. The list consisting of `[1,2]` , the list `[1,1,2]`, the list `[1,1,1,2]` etc. All lie in between the singleton list 1 and the singleton list 2, therefore, the list `Int` cannot be used as an index type.

The first argument of the `listArray` function specifies the smallest and largest index of the array (`i.i`), the second argument is the list of values to be stored in the array. And finally, the output is an array whose index type is `i` and whose value type is known.

(Refer Slide Time: 17:29)

## Creating arrays: listArray

```
* listArray (1,1) [100..199]
array (1,1) [(1,100)] ,  

* listArray ('m','p') [0,2..]
array ('m','p') [(‘m’,0),(‘n’,2),(‘o’,4),(‘p’,6)]  

* listArray ('b','a') [1..]
array ('b','a') []  

* listArray (0,4) [100..]
array (0,4) [(0,100),(1,101),(2,102),(3,103),(4,104)]  

* listArray (1,3) ['a','b']
array (1,3) [(1,’a’),(2,’b’),(3,*** Exception:  

(Array.!): undefined array element]
```

For example, listArray applied to the pair (1,1) and the list [100...199] will give you the following array, array (1,1), which is the range of the indices, the indices range from 1 to 1, which means that there is only one index. And the list itself, the array itself consists of one entry with index 1 and value 100. The values in the array are filled in the order they are presented in the list, so therefore 100 is associated with the index 1 from and not say 129.

Here is another example. listArray the pair ('m','p') where m and p are the characters and, where you want to store values from [0,2..], which is the infinite list of all the even positive integers will give you the following array. Array with the index bounds m and p and the entries of the array, there are 4 entries in the array, in the index m stores the value 0 ,index n stores the value 2, index 0 stores the value 4, index p stores the value 6. Here is another example, listArray ('b','a') and values from [1..], which is the infinite list will give you the empty array.

This is because, ‘a’, the character ‘a’ is actually less than the character ‘b’. So, therefore, there cannot be any index. The prerequisite for an array to have at least one entry is that the upper bound of the index should be actually greater than or equal to the lower bound of the index. Here is another example listArray(0,4). [100..], this will be array with bounds for the index 0 and 4 and entries being (0 , 100), (1,101), (2,102),(3,103) and (4, 104).

Here is another example listArray (1,3), which tells that there are three indices 1 2 and 3 and the list itself has only two elements ['a','b'] this will actually produce an exception. Because

Haskell tries to fill a value corresponding to index 3, but it cannot find any element, so it gives an exception saying undefined array element.

(Refer Slide Time: 20:24)

Creating arrays: listArray

- The value at index i of array arr is accessed using arr!i (unlike !! for list access)
- arr!i returns an exception if no value has been defined for index i
- myArr = listArray (1,3) ['a','b','c']
- myArr ! 4  
\*\*\* Exception: Ix{Integer}.index: Index (4)  
out of range ((1,3))

The value at index i of an array is accessed using the single exclamation mark like so, arr ! i (unlike the double exclamation mark for list access). So, arr ! i returns the ith value in the array, but it returns an exception if no value has been defined for index i which is why in the earlier slide, we saw that we got an exception for creating an array with three index values, but one with only two values.

Suppose, we created an array, myArray using listArray (1,3), ['a','b','c'], now if you try to access the fourth element in the array, myArr ! 4 will again get an exception saying that index 4 is out of range.

(Refer Slide Time: 21:25)

## Creating arrays: listArray

- Haskell arrays are **lazy**: the whole array need not be defined before some elements are accessed
- For example, we can fill in locations 0 and 1 of `arr`, and define `arr!i` in terms of `arr!(i-1)` and `arr!(i-2)`, for  $i \geq 2$
- `listArray` takes time proportional to the range of indices

An important point to note is that Haskell arrays are lazy; the whole array need not be defined before some elements are accessed. For example, we can fill in locations 0 and 1 of the array and define the  $i$ th element of the array in terms of the  $i$  minus first element and  $i$  minus second element. This is exactly the reminiscent of the array version of the computing the Fibonacci series in c, another point to note is that `listArray` takes time proportional to range of the indices. So, if there are  $k$  indices the call to `listArray` takes time  $O(k)$ .

(Refer Slide Time: 22:15)

## Creating arrays: array

- `array :: Ix i => (i, i) -> [(i, e)] -> Array i e`  
Creates an array from an associative list
- The associative list need not be in ascending order of indices  
`myArray = array (0,2) [(1,"one"),(0,"zero"),(2,"two")]`
- The associative list may also omit elements  
`array (0,2) [(0,"abc"), (2,"xyz")]`
- `array` also takes time proportional to the range of indices

There is another way to create arrays, which is to use the function `array` whose type is as follows  $\text{Ix } i \Rightarrow (i, i) \rightarrow [(i, e)] \rightarrow \text{Array } i \text{ e}$ . So, instead of producing a list of only values,

which was a list of elements of type e, we are now providing an associative list and creating an array out of this.

The associative list need not be in ascending order of the indices, for instance you could create an array as follows myArray = array (0,2) [(1,"one"), (0,"zero"),(2,"two")]. But, notice that the elements of the associative list are not presented in ascending order of the indices. The associative list may also omit elements, so you have array (0,2)[(0,"abc"), (2,"XYZ")] there is no entry for 1, this call also takes time proportional to the range of the indices.

(Refer Slide Time: 23:37)

More on indices

- Any type a belonging to the type class `Ix` must provide the functions
  - `range :: (a,a) -> [a]`
  - `index :: (a,a) -> a -> Int`
  - `inRange :: (a,a) -> a -> Bool`
  - `rangeSize :: (a,a) -> Int`

Let us look at indices in a little more detail. any type a belonging to the type class `Ix` must provide the following functions. A range, which is a function from pairs of the form `(a,a) -> [a]`, index, which is a function with signature `(a,a) -> a -> Int`. function `inRange`, which has signature `(a,a) -> a -> Bool` and the `rangeSize`, which has signature pair `(a,a) -> Int`.

(Refer Slide Time: 24:12)

## More on indices

- \* range :: (a,a) -> [a]  
range gives the list of indices in the subrange defined by the bounding pair
- \* range (1,2) = [1,2]  
range ('m','p') = "mnop"  
range ('z','a') = ""
- \* index :: (a,a) -> a -> Int  
The position of a subscript in the subrange
- \* index (-50,60) (-50) = 0  
index (-50,60) 35 = 85  
index ('m','p') 'o' = 2  
index ('m','p') 'a'  
\*\*\* Exception: Ix[Char].index: Index ('a') out of range  
(('m','p'))

The range function, range :: (a,a) -> [a], gives the list of indices in the sub range defined by the bounding pair, the first a is a lower bound and the second a is upper bound and this list a gives all the elements that lie between the first element here and the second element here. For instance range (1,2) is the list [1,2]. range('m','p') is the string “mnop”. Range ('z', 'a') is the empty string, because a is smaller than z.

Index gives the position of the subscript in the subrange, for instance index of -50 in the subrange defined by (-50,60) is 0, index of 35 in the sub range defined by (-50,60) is 85. Because, you go from 50 to 0 and then from 1 to 35, index of o the character ‘o’ in the range defined by the character ‘m’ and character ‘p’ is 2, because you go m is 0 n is 1 and the o is 2. Index of a, the character a in the range defined by ('m','p') will give an exception, because ‘a’ is out of the range ('m','p').

(Refer Slide Time: 25:38)

## More on indices

- `inRange :: (a,a) -> a -> Bool`  
Returns True if the given subscript lies in the range defined by the bounding pair
  - `inRange (-50,60) (-50) = True`
  - `inRange (-50,60) 35 = True`
  - `inRange ('m','p') 'o' = True`
  - `inRange ('m','p') 'a' = False`
- `rangeSize :: (a,a) -> Int`  
The size of the subrange defined by the bounding pair
  - `rangeSize (-50,60) = 111`
  - `rangeSize ('m','p') = 4`
  - `rangeSize (50,0) = 0`

`inRange` is a function that checks whether a given element lies in the range or not. for instance `inRange` of `(-50,60)` applied on `(-50)` will be true. You can also check that `o` is inside the range defined by `m` and `p`, but `a` is not inside the range defined by `m` and `p`, so `inRange` returns false. `rangeSize` just gives you the size of the range defined by the lower bound and the upper bound, for instance `rangeSize` applied to `(-50,60)` will give you 111. Range size applied to the character range `('m','p')` will give you 4. `rangeSize` defined by `(50,0)` will give you 0, because 0 was strictly less than 50.

(Refer Slide Time: 26:29)

## Functions on arrays

- `(!) :: Ix i => Array i e -> i -> e`  
The value at the given index in an array
- `bounds :: Ix i => Array i e -> (i,i)`  
The bounds with which an array was constructed
- `indices :: Ix i => Array i e -> [i]`  
The list of indices of an array in ascending order
- `elems :: Ix i => Array i e -> [e]`  
The list of elements of an array in index order
- `assocs :: Ix i => Array i e -> [(i,e)]`  
The list of associations of an array in index order

Here are some more function on arrays. `(!)` the exclamation mark which we mentioned earlier is to access an array entry, it has signature `Ix i` implies `Array i e -> i -> e`, so it gives the value

at a given index in an array. Bounds is a function that takes Array i e as an argument and returns the lower and upper bounds of the indices. The bounds with which the array was originally constructed. Indices is a function that lists the indices of an array in ascending order rather than providing just a pair, it produces the list of all indices.

Elems is the function that provides produces the list of all elements of an array in index order. Assocs is a function that lists all associations of array in the ascending order of index, its signature is  $\text{Ix } i \Rightarrow \text{Array } i \ e \ -> [(i, e)]$ .

(Refer Slide Time: 27:42)

Fibonacci using arrays

```
* fib :: Int -> Integer
fib n = fibA!n

fibA :: Array Int Integer
fibA = listArray (0,n) [f i | i <- [0..n]]
where
    f 0 = 1
    f 1 = 1
    f i = fibA!(i-1) + fibA!(i-2)

• The fibA array is used even before it is completely defined, thanks to Haskell's laziness
• Works in O(n) time
```

Let us now, get back to our original example that of computing Fibonacci numbers, but now using arrays. Fib is a function with signature  $\text{Int} \rightarrow \text{Integer}$ . fib of n equals  $\text{fibA}!n$ , which is accessing the nth element of the array fibA, fibA is  $\text{Array Int Integer}$  the indices are from  $\text{Int}$  and the values are from  $\text{Integer}$ . And now we use the fact that Haskell arrays are lazy. fibA is defined to be  $\text{listArray} (0,n)$ , where the ith entry is given by the function f i.

You create an array out of the list consisting of all values of f i. f i for i ranging from 0 to n. f is defined as follows f of 0 is 1. f of 1 is 1. f of i is not  $f(i-1) + f(i-2)$ , as we would have done in a recursive program. But,  $\text{fibA}!(i-1) + \text{fibA}!(i-2)$ , we are just referring to the array and picking the elements from the array the i minus first element and the i minus second element. In case the value at fibA has not be defined yet, Haskell will lazily make a call to f (i-1).

But, the first time it will make a call to f, it will fill the entry in the array, but the second time the called to `f(i-1)` is made it will just pick out the entry from the array. The `fibA` array is used even before it is completely defined, thanks to Haskell's laziness and this program works in  $O(n)$  time, because all it needs is to just fill n indices , by referring to previous entries in the array.

(Refer Slide Time: 29:43)

**lcss using arrays**

- We restate the recursive lcss in terms of indices

```

lcss :: String -> String -> Int
lcss str1 str2 = lcss' 0 0
  where
    m = length str1 - 1
    n = length str2 - 1
    lcss' i j
      | i > m || j > n = 0
      | str1!!i == str2!!j = 1 + lcss' (i+1) (j+1)
      | otherwise          = max (lcss' i (j+1))
                                (lcss' (i+1) j)

```

Here is how we can compute lcss using arrays, we restate , we first restate the recursive lcss in the terms of indices. lcss is a function with signature `String -> String -> Int`. But, we will work with the version `lcss'`, which works as `Int -> Int -> Int`. `lcss str1` and `str2` equals `lcss' 0 0`, where `lcss'` computes the length of the longest common sub sequence of drop `i` `str1` drop `j` `str2`.

And since, drop 0 of `str1` is equal to `str1` and drop 0 of `str2` is equal to `str2`, computing `lcss prime of 0, 0` achieves, what you want to achieve, which is to compute lcss of `str1` and `str2`. Let `n` be the length of `str1 -1` and `n` be the string length of `str2 -1`. these are the last indices, if you will of `str1` and `str2`. Now, `lcss prime of ij` is defined as defined as follows. if `i` is greater than `m` or `j` is greater than `n` the value is 0.

Otherwise if the character at the `i`th location of `str1` is the same as character at the `j`th location of `str2`, the result is  $1 + \text{lcss}' \text{ of } (i+1) (j+1)$ . If, the `i`th character of `str1` is not equal to the `j`th character of `str2`, then as earlier we compute the result using  $\max(\text{lcss}' \text{ i } (j+1))$  and  $(\text{lcss}' \text{ (i+1) } j)$

(i+1) j). Now, we have restated the original lcss function in terms of recursion on indices, this we will try to transcribe directly into an array based program.

(Refer Slide Tim-e: 32:07)

## lcss using arrays

```

• lcss :: String -> String -> Int
lcss str1 str2 = lcssA!(0,0)
  where
    m = length str1
    n = length str2
    lcssa = array ((0,0),(m,n))
      [((i,j),f i j) | i <- [0..m],j <- [0..n] ]
      f i j
        | i >= m || j >= n     = 0
        | str1!!i == str2!!j   = 1 + lcssa ! ((i+1),(j+1))
        | otherwise             = max (lcssa ! (i,(j+1)))
                                (lcssa ! ((i+1),j))
  • lcssa is a two-dimensional array. Indices are of type (Int,Int)
  • Drawback?? The repeated use of (!! ) in accessing str1 and str2
  • Solution? Turn the strings to arrays!

```

Here is lcss using arrays, lcss of str1 str2 is the 0 0'th entry of the array lcssA. the array lcssA is a two dimensional array whose indices range from (0, 0) to (n,n). The entry of the array at (i,j) is supposed to be the longest common sub sequence of str1 starting from index i and str2 starting from index j or in other words drop i of str1 and drop j of str2. In creating lcssA we use the array function rather than the listArray function, because it is easier to provide the values of the array in terms of an associative list.

So, we define lcssA as array with range (0, 0) to (n,n) and entry is of the form the part (i,j) comma the value f applied to i and j. So, this is the index and this is a value and the index and value is given as a pair, where i ranges from 0 to n and j ranges from 0 to n. In a two dimensional array of this form the indices are ordered as follows (0,0) followed by (0,1) followed by (0,2) all the way to (0,n), then comes (1,0), (1,1), (1,2) etc, all the way to (1,n), then (2,0), (2,1), (2,2) etc.

Now, f is defined as follows f on i and j is very similar to the index based recursive function that was described earlier, if i is greater than or equal to m or if j is greater than or equal to n the value is 0. Otherwise; if check if the ith element of string1 is the same as the jth element of str2.

In that case the result is  $1 + \text{lcssA}!((i+1),(j+1))$ . Notice, that here instead of making a recursive call to f we just refer to the corresponding element in the array at the appropriate index. Otherwise, if  $\text{str1}[i]$  is not equal to the  $j$ th element of  $\text{str2}$ , we define f of ij to be max of ( $\text{lcssA}!(i, j + 1)$ ) and ( $\text{lcssA}!(i+1,j)$ ) this is the function.

One minor drawback here is that we repeatedly use the exclamation, exclamation in accessing  $\text{str1}$  and  $\text{str2}$  here. Recall that we said that accessing an element of an array can be done in constant time whereas accessing the  $i$ th element of a list takes time order of order  $i$ , the solution would be to turn the strings themselves into arrays, which is done as follows.

(Refer Slide Time: 35:32)

```


• lcss :: String -> String -> Int
lcss str1 str2 = lcssA!(0,0)
where
  m = length str1
  n = length str2
  ar1 = listArray (0,m) str1
  ar2 = listArray (0,n) str2
  lcssA = array ((0,0),(m,n))
    [((i,j),f i j) | i <- [0..m], j <- [0..n]]
    f i j
      | i > m || j > n = 0
      | ar1!i == ar2!j = 1 + lcssA ! ((i+1),(j+1))
      | otherwise = max (lcssA ! (i,(j+1)))
                    (lcssA ! ((i+1),j))
  

```

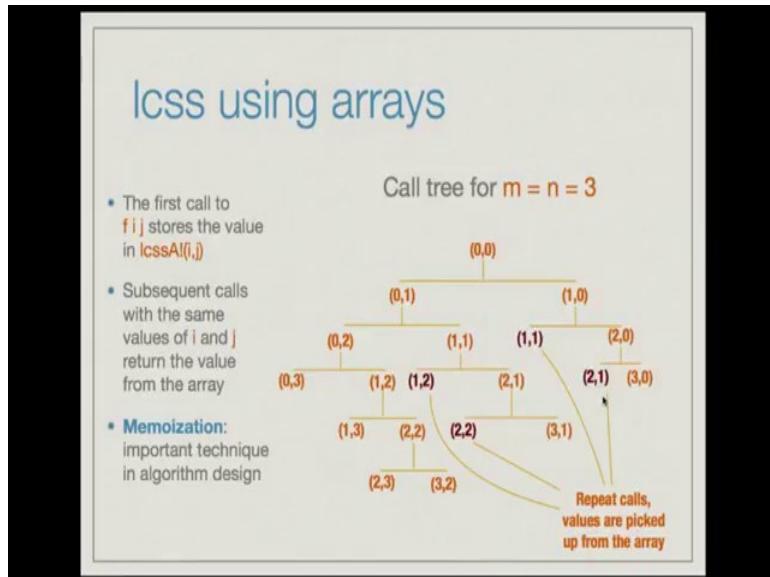
\* This program runs in time  $O(mn)$

Here is the version, which works on arrays rather than strings. lcss of str1 and str2 is lcssA!(0,0). Where instead of using str1 and str2 in the description, of f , we use ar1 and ar2, which are 2 arrays. ar1 is got by listArray (0 ,m) str1, where m is the length of str. ar2 is listArray (0,n) str2, where n is the length of str2. LcssA is itself defined as usual array with indices ranging from 0 to 0 to m comma n and entries of the form (i,j), f applied to i and j.

Where i ranges from 0 to m and j ranges from 0 to n. f is now defined in terms of ar1 and ar2. f of i j equals 0 if  $i > m$  or  $j > n$ . Otherwise; if ar1 at position i is equal to ar2 at position j, then you define it to be  $1 + \text{lcssA}!((i+1),(j+1))$ . Otherwise; you define it as usual. the difference here is that we access the array rather than the string repeatedly, this program one can check runs in time  $O(mn)$ .

Because, all we need do is fill in elements in the array and the number of indices of this array is  $(m+1)*(n+1)$ . So, the program runs in time  $O(m)$  which is a vast improvement over  $2^n$  that we had earlier.

(Refer Slide Time: 37:24)



Let us look at the computation in a little more detail, let us look at the call tree for the case when  $m = n = 3$ , the call tree for lcss. So, the first call to  $f[i,j]$  is we are considering  $f$  of 3 3 the first call to  $f$  of  $i,j$  stores the value in the array and subsequent calls with the same values  $i$  and  $j$  return the value from the array rather than making the recursive call and this technique is called memoization, it is an important technique in algorithm design.

Where you translate a recursive algorithm to a more efficient version by storing the values and referring to them later. So, initially there is a call made to  $f$  of  $(0, 0)$ . This points calls to  $f$  of  $(0, 1)$  and  $f$  of  $(1, 0)$  possibly it might also span a call to  $f$  of  $m = 1$ , but that occurs here anywhere.  $F$  of  $(0, 1)$  might span calls to  $f$  of  $(0, 2)$  and  $(1, 1)$ .  $(0, 2)$  spans calls to  $(0, 3)$   $(1, 2)$  this in turn leads to calls.  $(0, 3)$  terminates, because 3, if you recall is greater than or equal to  $n$ , which is 3.

Therefore, this call terminates by giving the value 0 and that will be stored in the array.  $(1, 2)$  gives rise to calls to  $(1, 3)$  and  $(2, 2)$ .  $(2, 2)$  makes call to  $(2, 3)$  and  $(3, 2)$ . These two calls terminate, because here 3 is greater than  $n$ . Now, once this returns we will have to process calls to  $(1, 1)$ ,  $(1, 1)$  gives rise to calls to  $(1, 2)$ ,  $(2, 1)$  but  $(1, 2)$  here is a repeat call. So, the values are picked up from the array rather than leading to a recursive call.

So, there won't be any, this whole tree under (1,2) will not be repeated. (1,1) also gives rise a call to (2,1), which in turn gives rise to calls to (2,2) and (3,1) but (2,2) is a repeat cal. It occurs earlier here already. So, its values are picked up from the array and this tree is not repeated under, (3,1) terminates, because 3 is greater than or equal to n and so on, you see that there are two more repeat calls here. So, in this manner we see that we never have to spend time more than O(n) and this example also illustrates, how exactly lazy arrays work.

(Refer Slide Time: 40:13)

## Creating arrays: accumArray

```

• accumArray
  :: Ix i
  => (e -> a -> e)  - accumulating function
  -> e                  - initial entry (at each index)
  -> (i,i)               - bounds of the array
  -> [(i,a)]             - association list
  -> Array i e           - array

• accumArray (+) 0 ('a', 'd')
  [('a',2),('b',3),('a',2),('c',4)]
  array ('a', 'd') [('a',4),('b',3),('c',4),('d',0)]

• accumArray (+) 0 (1,3)
  [(1,1),(2,1),(2,1),(1,1),(3,1),(2,1)]
  array (1,3) [(1,2),(2,3),(3,1)]

```

Here is another way to create an array, which is to use the function called accumArray, accumArray accumulates values into array positions and it works as follows. Its signature is  $\text{Ix } i \Rightarrow (e \rightarrow a \rightarrow e)$ , which is an accumulating function. this is similar to the kinds of functions you would provide as arguments to foldr or foldl. In particular this is the kind of function that you would provide as an argument to foldl, e which is an initial entry that will be placed in each index of the array (i,i) this provides the bounds of the array.

And an association list which is a list of pairs of (i,a), with all this it produces an array Array i e. Lets look at how it works, accumArray with function plus (+) and initial value 0 with indices ranging from the character 'a' to the character 'b' and with the elements coming from the associative list given here,  $[(\text{'a'},2), (\text{'b'},3), (\text{'a'},2), (\text{'c'},4)]$  produces the following array. An array with indices ranging from 'a' to 'd' and entries  $[(\text{'a'},4), (\text{'b'},3), (\text{'c'},4), (\text{'d'},0)]$ .

How do you explain the entries? Well 4 is 2+2, 3 is just 3, the 4 here is just 4 and the 0 here, is the initial value that was placed at the index d. So, you see that accumArray does the

following it places the initial value at all entries and then, whenever it encounters a particular index it adds the corresponding value it adds the value that is encountered to the corresponding entry in the array. In this case it adds, because the function that was provided is plus, if it is star it would have multiplied.

So, you see that using this function you accumulate all the values associated with this particular index in the list into the array at the appropriate index. Look at another example accumArray with function (+) and initial value 0 and the bounds (1,3) and the list provided by [(1,1), (2,1),(2,1),(1,1),(3,1),(2,1)] produces the following array, array bounds (1,3) and entries being [(1,2), (2,3), (3,1)].

Here we are just counting the number of repetitions of each element 1 occurs twice, here and here and we keep track of the count by initializing the values 0 at array entry 1. And then, using the function plus to add 1 every time, we encounter 1. Similarly, whenever we encounter 2 we add 1 to the accumulator. Since 2 occurs thrice here, here and here and the final value associated to index 2 in the array is 3, 3 occurs once, so that finally, you will have one as the value at the index 3.

(Refer Slide Time: 43:50)

Creating arrays: accumArray

- `accumArray`  
     $:: \text{Ix } i$   
     $\Rightarrow (e \rightarrow a \rightarrow e) \rightarrow e \rightarrow (i, i) \rightarrow [(i, a)]$   
     $\rightarrow \text{Array } i \text{ e}$
- `accumArray f e (l,u) list` creates an array with indices  $l..u$ , in time proportional to  $u-l$ , provided `f` can be computed in constant time
- For a particular  $i$  between  $l$  and  $u$ , if  $(i, a_1), (i, a_2), \dots, (i, a_n)$  are all the elements with index  $i$  appearing in `list`, the value for  $i$  in the array is  $f \dots (f \ (f \ e \ a_1) \ a_2) \dots \ a_n$
- The entry at index  $i$  thus **accumulates** (using `f`) all the  $a_i$  associated with  $i$  in `list`

`accumArray f e (l,u) list`.  $f$  is the function  $e$  is the initial entry  $(l,u)$  is the lower bound on the indices and upper bound on the indices and the list is the associated list. It creates an array with indices  $l..u$  in time proportional to  $u - l$  which is important provided  $f$  can be computed

in constant time. So, for a particular  $i$  between 1 and  $u$ , if  $(i,a_1), (i,a_2), \dots (i,a_n)$  are all the elements with index  $i$  appearing in this list.

The value for  $i$  in the array is  $f(...(f(f\ e\ a_1)\ a_2) \dots) \ a_n$ .  $f$  applied to  $e$  and  $a_1$ .  $f$  applied to the first result and  $a_2$ ,  $f$  applied to the second result and  $a_3$ , etcetera and finally,  $f$  applied to that result and  $a_n$ . So, it is a foldl version applied with all the values corresponding to  $i$  that occurs in the association list. So, the entry at index  $i$  thus accumulates using the function  $f$  all the  $a_i$  values that are associated with  $i$  in the list.

(Refer Slide Time: 45:10)

```

Sorting with accumArray
* [2,3,4,1,2,5,7,8,1,3,1]
  ↪ zip [2,3,4,1,2,5,7,8,1,3,1] [1,1,1,1,1,1,1,...]
  = [(2,1),(3,1),(4,1),(1,1),(2,1),(5,1),(7,1),(8,1),(1,1),(3,1),(1,1)]
  [Recall that iterate f x = [x, f x, f (f x), f (f (f x)), ...].
  So iterate id 1 = [1,1,1,1,1,...]]
  ↪ array (1,8) [(1,3),(2,2),(3,2),(4,1),(5,1),(6,0),(7,1),(8,1)]
    - counts number of repetitions of each entry
  ↪ [(1,3),(2,2),(3,2),(4,1),(5,1),(6,0),(7,1),(8,1)]
  ↪ replicate 3 1 ++ replicate 2 2 ++ replicate 2 3 ++ replicate 1 4 ++
  replicate 1 5 ++ replicate 0 6 ++ replicate 1 7 ++ replicate 1 8
  = [1,1,1]+[2,2]+[3,3]+[4]+[5]+[7]+[8]
  = [1,1,1,2,2,3,3,4,5,7,8]

```

We can use this to sort in linear time as follows, suppose we are given the list 2, 3, 4, 1, 2, 5, 7, 8, 1, 3, 1. We first create another list an association list as follows by zipping it with 1 repeated infinitely often that will produce a list of pairs where the first element of each pair is from the original list and the second element is always 1.

How do you produce this list [1,1,1,...], recall that iterate is a function, which behaves as follows: iterate  $f x$  equals the list  $[x, f x, f (f x), f (f (f x)) \dots]$ . So, you can start with  $x$  being 1 and  $f$  being the identity function and will get you see that iterate id 1 equals the infinite list consisting only of 1s. From this association list will produce on array, array (1 8), the 1 here is the minimum value on the original list and 8 is the maximum value on the original list.

So, we have the bounds of the array and you are produced, this array this is an accumArray therefore, this array stores the count of the number of repetitions of each entry. From this

array you go to the list [(1,3),(2,2),(3,2),...]. You can obtain this list from the array by using the function assocs. Once you have this list, you replicate 1 thrice and that is done using the function replicate, replicate n i gives you the list consisting of the value i repeated n times.

So, replicate 3 1 gives of the list [1,1,1]. When you replicate 2 twice, replicate 2 ,2 .replicate of 2 3 that is, because you want to replicate the value 3 twice. replicate 1 4 ++ replicate 1 5 etcetera. That will produce this list [1, 1,1] ++ [2, 2] ++ [3,3] ++ etcetera. and this will finally, produce this list which is the sorted version of the original list.

Accumarray works in time proportional to the length of the association list and assocs also works in time proportional to the length of the list produced. Therefore, this algorithm works in linear time or order n plus max minus min of the elements in the array, if you prefer of the elements in the list.

(Refer Slide Time: 48:05)

### Sorting with accumArray

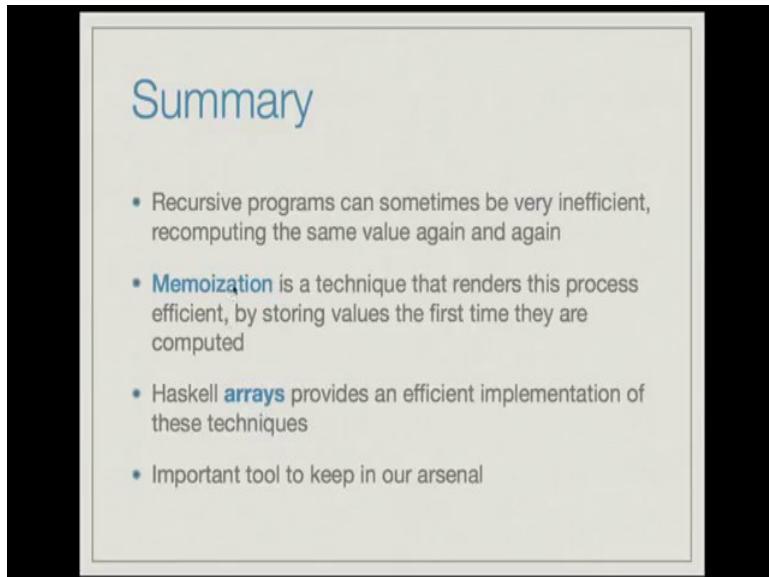
```
* counts :: [Int] -> [(Int,Int)]
counts xs = assocs (
    accumArray (+) 0 (l,u) (zip xs ones)
)
where
    ones = iterate id 1
    l    = minimum xs
    u    = maximum xs

* arraysort :: [Int] -> [Int]
arraysort xs = concat [replicate n i | (i,n) <- ys]
where
    ys  = counts xs
```

Finally, here is the code for linear time sorting with accumArray , you start with the function count it takes the list of integer and produces the list of pairs of integers this is the associated list of counts, counts of xs is assocs of accumArray with (+) 0 (l,u) on the association list zip xs ones. Ones, if as you recall is iterate id 1, which produces the infinite list consisting of 1, 1, 1, etcetera, l is the minimum in the original list u is the maximum in the original list with these as indices we produce this array of counts and extract the association list that is embedded in the array by using the function assocs,

Arry sort takes ys which is counts of xs and for each entry (i,n) <- ys, it does replicate n i and finally, concatenates this list using the function concat. So, finally, you get a sorted version of the original list.

(Refer Slide Time: 49:16)



The slide has a light gray background with a dark gray border. At the top left, the word "Summary" is written in blue. Below it is a bulleted list in black text:

- Recursive programs can sometimes be very inefficient, recomputing the same value again and again
- **Memoization** is a technique that renders this process efficient, by storing values the first time they are computed
- Haskell **arrays** provides an efficient implementation of these techniques
- Important tool to keep in our arsenal

In summary, recursive program can sometimes be very inefficient, re computing the same value again and again; this is illustrated in the fib function as well as in the less function, memoization is an important technique that renders this process efficient sometimes by storing values the first time they are computed and referring to the store values rather than re computing in the subsequent times when they are needed. Haskell arrays provide an efficient implementation of these techniques and it is an important tool to keep in our arsenal.

**Functional Programming in Haskell**  
**Prof. Madhavan Mukund and S. P. Suresh**  
**Chennai Mathematical Institute**

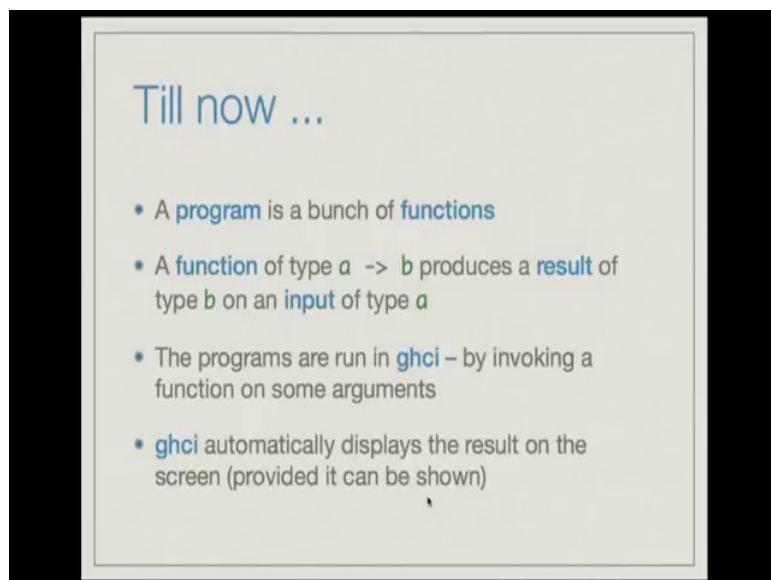
**Module # 07**

**Lecture - 02**

**Input / Output**

In this lecture, we shall study Input Output in Haskell.

(Refer Slide Time: 00:08)



Till now, the view we have taken is that a program is a bunch of functions, a function of type **a -> b** produces a result of type **b** on an input of type **a**. The programs are run in **ghci** by invoking a function on some arguments, **ghci** automatically displays the result on the screen provided the result can be shown or in other words, provided the result is a type that belongs to the type class **Show**.

(Refer Slide Time: 00:46)

## User interaction

- Can we execute programs outside `ghci`?
- How do we let the programs interact with users?
  - Accept user inputs midway through a program execution
  - Print output and diagnostics on screen / to a file
- Can interaction with the outside world be achieved without violating the spirit of Haskell?

But, this is a limited form of interaction with the user, we would like a slightly better user interaction model like in other programming languages. So, the questions we have are , can we execute programs outside ghci, how do you let the programs interact with users that is accept user inputs midway through a program execution. Print output and diagnostics on a screen or to a file, can interaction with the outside world be achieved without violating the spirit of Haskell. So, these are some of the questions that we considered in this lecture.

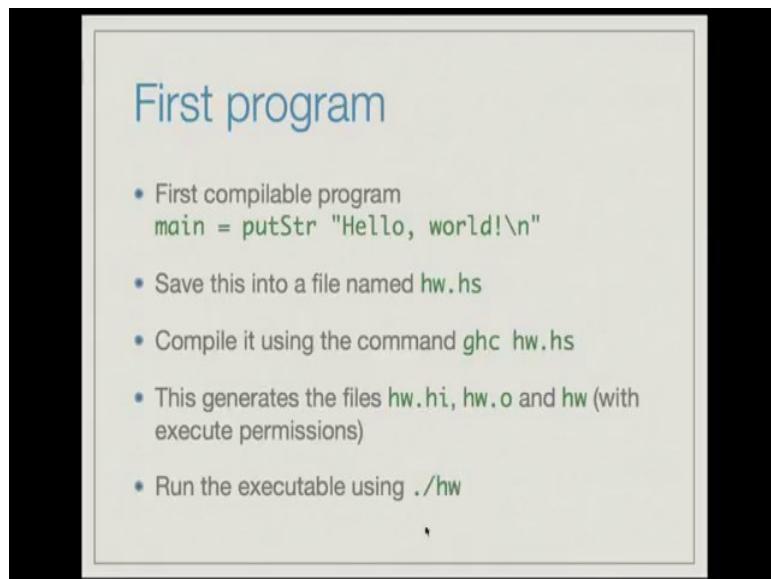
(Refer Slide Time: 01:25)

## Standalone programs and `main`

- Execution of a Haskell program starts with the function `main`
- Every standalone Haskell program should have a `main` function

We shall first consider standalone programs, execution of a Haskell program starts with the function main, this is not something we have seen till now. Till now, our program was just a bunch of functions and ghci automatically interpreted any function that we entered in it. But, if you have to write a standalone program, execution has to start at some place and that is the function main, every standalone Haskell program should have a main function.

(Refer Slide Time: 02:03)



Here is an example program, in fact this is the simplest compilable program that we can write, `main = putStrLn "Hello, world! \n"`. So, this as the name suggests puts a string on the screen, so the way we run this is to save this in to a file named `hw.hs`, `hw` standing for Hello world. We compile this file using the command `ghc hw.hs`. Compiling this generates the following files, `hw_hi`, `hw_o` and `hw`, the file of interest was the `hw` without any extension.

If you see this on a Unix terminal, we will see that the permissions for these two are read write, but `hw` has execute permissions or in other words `7 5 5`. We can run the executable using `./hw`, dot denotes the current directory, dot slash `hw` means that the executable files `hw` can be found in the current directory. If you add the path, where your Haskell executable resides in your path environment, you can just invoke the functions using `hw`, but for now in the examples we will just use dot slash programming.

(Refer Slide Time: 03:45)



GHC is the Glasgow Haskell Compiler, ghci that we have been using till now is the interactive version of the compiler, one can view ghci as an interpreter or a play ground in which you test your programs. If the program is intended for use by others, then it is usually written as a standalone program, compiled using ghc and shipped. Compiled versions of programs run much faster and use much less memory as compared to running them in ghci.

You can check the various options that ghc offers by typing `ghc --help` in the terminal, you can use `ghc --show-options` to know all the options that you can provide, all the compiler options that you may provide to ghc, but this is a huge list. If you want to know more about ghc and ghci, you can consult the GHC Manual at this url which is the part of the official Haskell page. So, we have learnt how to write a simple program which stands alone and runs on its own, compile it and run it.

(Refer Slide Time: 05:05)

Hello, world!

- `main = putStrLn "Hello, world!\n"`
- `putStrLn` str prints the string str on screen
- Clearly `putStrLn` is of type `String -> b`, for some type b
- The return value is not used at all, so perhaps it returns nothing of significance
- The type `()`, which consists of a single value, also denoted by `Unit`, can be used to model “nothing”
- So is its type `String -> ()`?

Let us study the program in more detail, `main = putStrLn "Hello, world! \n"`, `putStrLn` is a function, the behavior is that `putStrLn` prints the string str on screen. So, clearly `putStrLn` is of type `String -> b` for some b, because the input is a string, but you notice that in this main program the return value is not used at all. So, perhaps we can say that `putStrLn` does not return anything of significance, the type `()` or empty as it is called denotes nothing or it can be used to model nothing. This type empty is denoted like an empty tuple and it consists of a single value, which is also denoted by an empty tuple, so the question is, is the type of `putStrLn`, `String -> ()` ?.

(Refer Slide Time: 06:25)

Hello, world!

- Is `putStrLn` of type `String -> ()`?
- But it does not return the value `()`!
- And how do we account for the **side effect** of printing something on screen?
- `ghci> :t putStrLn`  
`putStrLn :: String -> IO ()`
- `ghci> :t putStrLn "Hello, world!"`  
`putStrLn "Hello, world!" :: IO ()`

But, we notice that `putStr` is not an expression that returns a value and more over, it has a side effect, which is that of printing something on screen. So how do we account for the side effect? So, if you actually type `:t putStr` in `ghci`, you will see that `ghci` says what the type of `putStr` is, the type is `String -> IO ()`. If you check what the type of `putStr "Hello world!"` is, you will see that, the type is `IO ()`.

(Refer Slide Time: 07:13)

The slide has a title 'IO a' at the top left. Below the title is a list of four bullet points:

- `IO` is a type constructor, just like `List` or `BTree` or `AVLTree` that we encountered in previous lectures
- `IO a` is a type whenever `a` is a type
- Recall that the value constructors and internal structure of `List`, `BTree` etc. are visible
- The internal structure and constructors of `IO` are not visible to the user

So, what is this `IO`? `IO` is a type constructor, just like some other type constructors we have encountered in previous lectures like `List` or `BTree` or `AVLTree`, etc. So, therefore, `IO a` is a type whenever `a` is a type, but there is a distinction. Recall that the value constructors and internal structure of user defined data types like `List`, `BTree` etc are visible. But, the internal structure and constructors of `IO` are not visible to the user. In other words the user cannot do any kind of pattern matching based on the constructors of `IO`.

(Refer Slide Time: 07:57)

**IO a**

- One can understand IO as follows:  
`data IO a = IO (RealWorld -> (RealWorld,a))`
- So an object of `IO a` is a **function** which takes as input the **current state of the real world**, and produces a **new state of the real world** and a value of type `a`
- In other words, objects of `IO a` constitute both a value of type `a` and a **side effect** (the change in state of the world)

One way to understand IO is as follows ,its declaration can be thought of as `data IO a = IO (RealWorld -> (RealWorld,a))` , the IO on the left is the type constructor, the IO on the right is the value constructor. And the values are of type `RealWorld -> (RealWorld,a)`. RealWorld here is not an actual Haskell type, but it is just one way we can understand what IO means. Assume that there is a type, which represents all states of the real world.

So, we can think of IO as taking as input the current state of the real world and producing a new state of real world due to side effects and also producing the value of type `a`. In other words object of `IO a` constitute both a value of type `a` and a side effect namely the change in state of the world.

(Refer Slide Time: 09:06)

## IO a and actions

- Technically, an object of type IO a is not a function but an **IO action**
- An IO action **produces a side effect** when its **value** is extracted
- Any function that produces a side effect will have return type IO a

Technically an object of type IO a is not usually referred to as a function, but as an IO action. This is an important distinction. An IO action produces a side effect when its value is extracted. Any function that produces a side effect will have return type IO a.

(Refer Slide Time: 09:32)

## putStr and main

- `putStr :: String -> IO ()`
- `putStr` takes a string as argument and returns (), producing a side effect when the return value is extracted
- The side effect is that of printing on screen the string provided as argument
- `main :: IO ()`
- `main` is always of type IO a

So, lets get back to put string. `putStr` has a type `String -> IO ()`. `putStr` takes a String as the argument and returns an empty tuple (). And in the process of producing the empty tuple as output it also produces a side effect when the return value is extracted. The side effect in the

case of `putStr` is that of printing on screen the string that is provided as argument. `main` as you can see is of type `IO` empty. `main` is always of type `IO a` for some `a`.

(Refer Slide Time: 10:15)

## Side effects

- Kind of side effects
  - Printing on screen
  - Reading a user input from the terminal
  - Opening / closing a file
  - Changing a directory
  - Writing into a file
  - **Launching a missile**

Now, we talked about side effects in the course of executing an action, what kind of side effects can happen. Examples are printing on screen, reading the user input from the terminal, opening or closing a file, changing the directory, writing into a file etc or maybe launching a missile, driving a truck etcetera.

(Refer Slide Time: 10:45)

## putStr and putStrLn

- `putStr "Hello world!"` prints the string on the screen
- `putStrLn "Hello world!"` prints the string and a newline ('\n') on the screen
- `putStrLn str` is equivalent to  
`putStr (str ++ "\n")`

Now, there is a close variant of putStr namely putStrLn, you can read it as put String line. putStr “Hello world!” prints the string on screen, whereas putStrLn “Hello world!” prints the string and appends a new line on the screen. So, putStrLn str is equivalent to putStr (str ++ “\\n”)

(Refer Slide Time: 11:17)

## Chaining actions

- We use the command `do` to chain multiple actions
- `main = do`  
 `putStrLn "Hello!"`  
 `putStrLn "What's your name?"`
- `do` makes the actions take effect in **sequential order**, one after the other
- Indentation is important

Stand alone actions are not of much use unless you can perform a lot of actions and Haskell provides a way to chain actions. We use the command `do` to chain multiple actions. For example, you could say

```
main = do  
    putStrLn "Hello!"  
    putStrLn "what's your name?"
```

`do` makes the actions take effect in sequential order one after the other in the order presented in the program text. Here the indentation is important in the `do` command, but since the indentation is sometimes hard to keep track of...

(Refer Slide Time: 12:00)

The slide has a light gray background with a dark gray border. The title 'Chaining actions' is at the top in a blue font. Below it is a bulleted list:

- Alternative, friendlier syntax

```
main = do {  
    putStrLn "Hello!";  
    putStrLn "What's your name?";  
}
```
- Actions can occur inside let, where etc.
- main = do {act1; act2;}  
where  

```
act1 = putStrLn "Hello, "  
act2 = putStrLn "world!"
```

Haskell offers an alternative friendlier syntax, which is to use braces and semi colons.

```
main = do {  
    putStrLn "Hello!";  
    putStrLn "what's your name?";  
}
```

And these actions can also occur inside let, where etc. For instance,

```
main = do {act1; act2;}  
where  
    act1 = putStrLn "Hello, ";  
    act2 = putStrLn "world!"
```

So, you can define local actions.

(Refer Slide Time: 12:42)

## More actions

- `print :: Show a => a -> IO ()`  
Output a value of any printable type to the standard output (screen), and adds a newline
- `putChar :: Char -> IO ()`  
Writes the Char argument to the screen
- `getLine :: IO String`  
Read a line from the standard input and return it as a string
- The side effect of `getLine` is the consumption of a line of input, and the return value is a string
- `getChar :: IO Char`  
Read the next character from the standard input

Here are some more actions, print, its signature is `Show a => a -> IO ()`, this outputs a value of any printable type to the standard output, which is the screen and adds a new line. PutChar is a function from `Char -> IO ()`. It writes the Char argument that is provided to it on the screen. getLine is of type `IO String`, it reads the lines from the standard input and returns it as a `String`, the side effect of `getLine` is the consumption of line of input rather than the production of a line of ouput and the return value of `getLine` is a `String`. getChar is a function that reads the next character from the standard input.

(Refer Slide Time: 13:38)

## Binding

- `getLine` is of type `IO String`, but is there a way to use the return value?
- We need to bind the return value to an object of type `String` and use it elsewhere
- The syntax for binding is `<-`
- `main = do {  
 putStrLn "Please type your name!";  
 n <- getLine;  
 putStrLn ("Hello, " ++ n);  
}`

We saw that getLine is of type IO string, but is there a way to use the value that is returned by getLine.

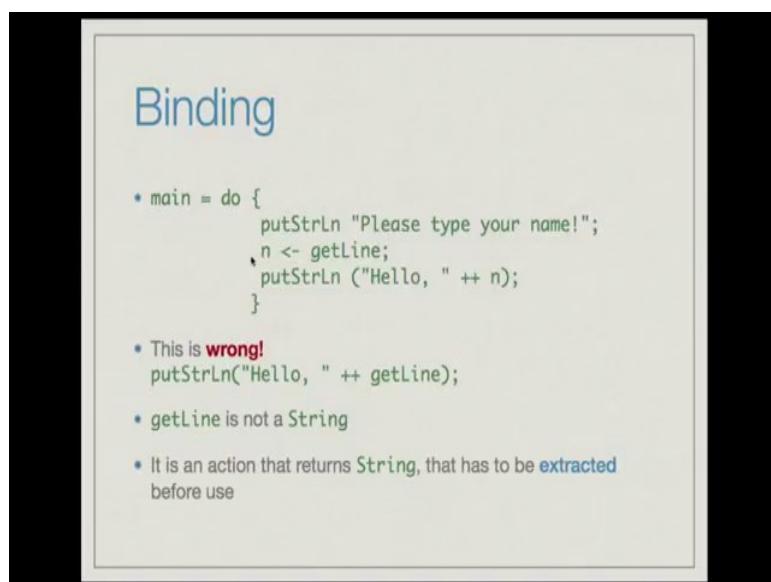
In other words we need to bind the return value of getLine to an object of type string and perhaps use it elsewhere. Haskell provides the following syntax for binding and the syntax is reminiscent of assignment in some other languages, which is just to say <- , it is like a left arrow for instance you could do this.

```
main = do {  
    putStrLn "Please enter your name!" ;  
    n <- getLine;  
    putStrLn ("Hello, " ++ n);  
}
```

n bound by get line or in other words the output of get line is bound to n.

And then I can use it here, in the following action, which is putStrLn ("Hello, " ++ n); This has the effect of first asking the user for the name, waiting for the user to input her name and press the enter key and then printing Hello followed by her name on the screen.

(Refer Slide Time: 14:56)



Please note that this is wrong. putStrLn ("Hello, " ++ getLine); this is because ++ is a so called pure Haskell function or operator and its arguments are list a and list a and the output is list a, the arguments are not of type IO a. Therefore, you cannot use getLine in the context

of a ++, you should always bind it to some name and then use that name. `getLine` is not a String. It is an action that returns String that has to be extracted before the use. The extraction is the binding that happens here through `<-`.

(Refer Slide Time: 15:47)

The slide has a light gray background with a dark gray border. The title 'Functions vs. Actions' is at the top in blue. Below the title is a bulleted list of four items:

- The functions we have seen till now (that are free of side effects) are called **pure functions**
- Their type gives all the information we need about them
- Invoking a function on the same arguments always yields the same result
- The order of evaluation of the subcomputations does not matter – Haskell utilizes this in applying its **lazy strategy**

At this point we need to look a little closer at the distinction between functions and actions. A function that takes an integer as the argument and returns an integer as a result has type `Int -> Int`. An action that has a side effect in addition to consuming an integer and producing an integer has type `Int -> IO Int`. This distinction that Haskell maintains is in contrast to languages like C or Java, where the type signature of both functions that have side effects and functions that do not have side effects are just `Int -> Int`.

And in general any function is assumed to potentially produce a side effect , any function can produce a side effect there is nothing in the language itself that prevents functions from producing side effects. Haskell enforces this distinction between pure functions and actions. The functions that you have seen till now that are free of side effects are called pure function. Their type gives all the information we need about them. Invoking the function on the same arguments always yields the same result. And the order of evaluation of sub computations does not matter. Haskell utilizes this to great effect in applying its lazy strategy.

(Refer Slide Time: 17:20)

## Functions vs. Actions

- The presence of `IO` in the type indicates that actions potentially have side effects
- External state is changed
- Order of computation is important – `sequencing`

In contrast to functions, actions usually have side effects and Haskell maintains this distinction by designating their output types with an `IO`. The presence of `IO` in the type indicates that actions potentially have side effects, external state is usually changed and the order of computation is important as in actions inside a `do` command. The actions take place in a sequence, so sequencing is something that is inherent to actions.

(Refer Slide Time: 18:03)

## Functions vs. Actions

- Performing the same action on the same arguments twice might have different results
- ```
greetUser :: String -> IO ()  
greetUser greeting = do {  
    putStrLn "Please enter your name";  
    name <- getLine;  
    putStrLn ("Hi " ++ name ++ ". " ++ greeting);  
}
```
- ```
main = do {greetUser "Welcome!";  
           greetUser "Welcome!"  
}
```
- The two actions print different things on the screen, depending on the name that is input by the user

Moreover, performing the same action on the same arguments twice might have different results. For instance consider the following action `greetUser`, which is of type `String -> IO ()`.

greetUser greeting equals do. Inside the do block we have putStrLn “please enter your name”; which binds the value of getLine. The return value of getLine is bound to name. ((ReferTime18:38)). Then, we put another string on the screen putStrLn (“Hi” ++ name ++ ”,” ++ greeting);

So, this greeting is something provided as an input. In main we can do the following greetUser “Welcome!”; greetUser “Welcome”; You see that the greetUser function is being called with the exact same argument twice, but the two actions might print different things on this screen depending on the name that is input by the user at this point. Because, the greetUser action itself has a way to get input from the user and type the input back to the user namely the user's name. So, this shows that performing the same action on the same arguments twice might have different results and this is the fundamental difference between actions and functions.

(Refer Slide Time: 19:44)

The slide has a light gray background with a dark gray border. The title 'Combining pure functions and IO actions' is centered at the top in a blue font. Below the title is a bulleted list of five items, each preceded by a black dot:

- Haskell type system allows us to combine pure functions and actions in a safe manner
- No mechanism to execute an action inside a pure function, even though pure functions can be used as subroutines inside actions
- IO is performed by an action only if it is executed from within another action
- `main` is where all the action begins

One can combine pure functions and actions, but in a limited manner. We can use pure functions as subroutines and IO actions, but not the other way round. The Haskell type system allows us to combine pure functions and actions in a safe manner. No mechanism exists to execute an action inside a pure function even though pure functions can be used as subroutines inside actions. IO is performed by an action only if it is executed from within another action and `main` is, where all the action begins. So, `main` embeds some actions inside it and each of those actions might be a do block with further actions inside.

(Refer Slide Time: 20:31)

The slide has a light gray background with a dark gray border. At the top, the title 'IO example' is centered in a blue font. Below the title is a list of bullet points and code. The code is written in Haskell:

```
• Read a line and print it out as many times as its length
• main = do {
    inp <- getLine;
    printOften (length inp) inp;
}
printOften :: Int -> String -> IO ()
printOften 1 str = putStrLn str
printOften n str = do {
    putStrLn str;
    printOften (n-1) str;
}

• What if the user inputs the empty string?
```

Let us look at a few examples, here is one example, which is to read a line and print it out as many times as the length of the string that is input on the first line. Here is the program.

```
main = do {
    inp <- getLine;
    printOften ( length inp) inp;
}

printOften :: Int -> String -> IO ()
printOften 1 str = putStrLn str
printOften n str = do {
    putStrLn str;
    printOften (n-1) str;
}
```

main equals do get a line and bind it to inp. inp stands for input. it is a variable that stores values of types String. Call the function printOften to print inp length inp times, if inp is of length n, then printOften input will be called and it will be printed n times.

Thus achieved as follows. printOften is a function from Int -> String -> IO () .

printOften 1 str is just putStrLn str;

printOften n str is a recursive function. Inside a do block you first put putStrLn str; and then you printOften (n-1) str. So, this is an example of a non trivial

interaction with the user, but what if the user inputs the empty string. Notices that, if length of the input equals 0 there is no case here the catches it.

(Refer Slide Time: 22:03)

The slide has a title 'return' in green at the top. Below the title is a bulleted list of seven items, each starting with a blue bullet point:

- What if the user inputs the empty string?
- How do we define `printOften 0 str`?
- Can we just define it to be `()`?
- But then the output type would be `()`, not `IO ()`
- Need a way to promote `()` to an object of type `IO ()`
- Achieved by the `return` function
- If `v` is a value of type `a`, `return v` is of type `IO a`

What if the user inputs the empty string, to handle this we need to define `printOften 0 str`. recall that the return value of `printOften` is `IO ()`. `printOften 0 str` - there should not be anything printed on the screen. So, can we just define it to be empty, but the output type then would be empty and not `IO empty`. So, you would get a type mismatch this means that we need a way to promote the empty tuple to an object of type `IO empty`. This is precisely achieved by the `return` function. The `return` function or the `return` action takes `v`, which is the value of type `a` and produces an action of type `IO a`, if `v` is a value of type `a`, `return v` is of type `IO a`.

(Refer Slide Time: 23:06)

## IO example, fixed

- Read a line and print it out as many times as its length

```
• main = do {
    inp <- getLine;
    printOften (length inp) inp;
}
printOften :: Int -> String -> IO ()
printOften 0 str = return ()
printOften n str = do {
    putStrLn str;
    printOften (n-1) str;
}
```

With this we can fix the earlier example as follows. main equals do get a line and bind it input print often length input input the crucial case is this printOften 0 str = return (). This matches the type properly and it also handles this case in the case of not printing anything on the screen. Notice that here, there is no side effect as such, but still we designate this as type IO empty. So, this illustrates the fact that if an object is of type IO a it need not necessarily produce a side effect it only indicates the potential to produce a side effect whereas a pure function, which is not, whose type is not embedded to IO can never produce a side effect.

(Refer Slide Time: 24:13)

## Another example

- Repeat an IO action n times

```
ntimes :: Int -> IO () -> IO ()
ntimes 0 a = return ()
ntimes n a = do {
    a;
    ntimes (n-1) a;
}

• Read and print 100 lines
main = ntimes 100 act
where
    act = do {
        inp <- getLine;
        putStrLn inp;
    }
```

Here is another example ntimes, which takes an integer and an action and repeat the action n times. This is the generalization of, what we did in the previous example, where we printed inp out to the screen some n times, where n is the length of the input. So, ntimes 0 a is just return empty ,ntimes n a is do a followed by ntimes (n-1) a. ntimes (n-1) a which is to do the action a (n-1) times. Now, we can read and print 100 lines as follows main = ntimes 100 act where act equals this block, which reads an input from the user and prints out that input. getLine binded to inp and putStrLn inp. This action is repeated 100 times by main.

(Refer Slide Time: 25:15)

## Reading other types

- The function `readLn` reads the value of any type `a` that is instance of the typeclass `Read`
- `readLn :: Read a => IO a`
- All basic types (`Int`, `Bool`, `Char`, ...) are instances of `Read`
- Basic type constructors also preserve readability – `[Int]`, `(Int, Char, Bool)`, etc are also instances of `Read`
- Syntax to read an integer  
`inp <- readLn :: IO Int`

Strings are not the only things that can be read, we can read other values, but these other values have to belong to a type a that is an instance of the type class read. For this we use the function `readLn`, which is denoted `readLn` whose type is `Read a => IO a`. All the basic types `Int`, `Bool`, `Char` etc are instances of `Read`. Therefore, you can use `readLn` to read Integers, Booleans Characters etc the basic type constructors also preserve readability.

So, for instance list `[Int]`, the triple `( Int, Char, Bool)` etc are also instances of `Read`. Here is the syntax to read an integer: `inp <- readLn :: IO Int`, and bind the result to `inp`. So, since `readLn` is supposed to cater to the reading of any of these types that belongs to class `Read` you need to specify, which type you want the input to be.

(Refer Slide Time: 26:38)

The slide has a light gray background with a dark gray border. At the top, the title "Another IO example" is centered in a blue font. Below the title, there is a bulleted list and some Haskell code. The Haskell code defines a main function that reads a list of integers from input, reverses it, and prints the result. It also defines a readList function that reads integers until it finds -1, at which point it returns the list.

```
* Read a list of integers (one on each line and terminated by -1) into a list, and print the list

• main = do {
    ls <- readList [];
    putStrLn (show (reverse ls));
}

readList :: [Int] -> IO [Int]
readList l = do {
    inp <- readLn :: IO Int;
    if (inp == -1) then
        return l;
    else readList (inp:l);
}
```

So, here is an example, in this example we read a list of integers one on each line and finally, terminated by -1 into a list and print the list.

```
main = do {
    ls <- readList [];
    putStrLn (show (reverse ls));
}
```

The return value of readList is empty list. So, readList is a function you are providing the empty list as an input to read list. It is of type [Int] -> IO [Int]. So, it produces some side effects and its return value is a list of integers, which is what is bound to the ls here and then it so happens that readList reads the list in reverse order.

So, we show the reverse of ls and invoke putStrLn on it , in this way we read a bunch of integers one on each line and output them as a list. Here is the description of read list. read list l equals do readLn as an integer, this you denote by saying `inp <- readLn :: IO Int` and bind the return value to inp. If inp is -1, then, there is nothing more to do, so you just return l, which is the list that was provided as input else you have read the input you add the input to the list by saying `inp:l` you add it to head of the list and you proceed with the readList function presumably reading more input.

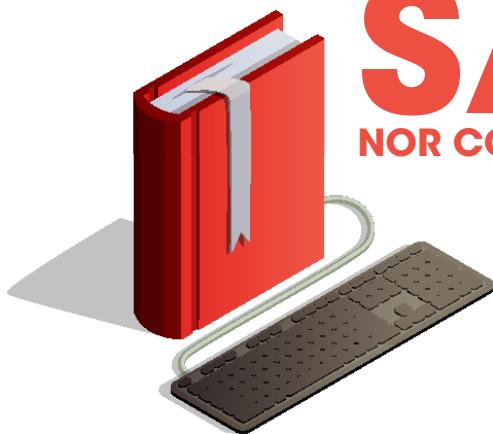
(Refer Slide Time: 28:27)

The slide has a black header and footer. The main content area is a light gray rectangle with a thin black border. The title 'Summary' is at the top in a blue font. Below it is a bulleted list in a smaller black font:

- Haskell has a clean separation of pure functions and actions with side effects
- Actions are used to interact with the real world and perform input/output
- `main` is an action where the computation begins
- `ghc` can be used to compile and run programs
- There is a lot more to explore in IO – only the most basic material is covered here

In summary Haskell has a clean separation of pure functions and actions with side effects, actions are used to interact with the real world and perform input output .`main` is the action where the computation begins, `ghc` can be used to compile and run programs. So, there is a lot more to explore in IO but only the most basic material is covered here which suffices for rudimentary interaction with users.

**THIS BOOK  
IS NOT FOR  
SALE  
NOR COMMERCIAL USE**



(044) 2257 5905/08



nptel.ac.in



swayam.gov.in