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

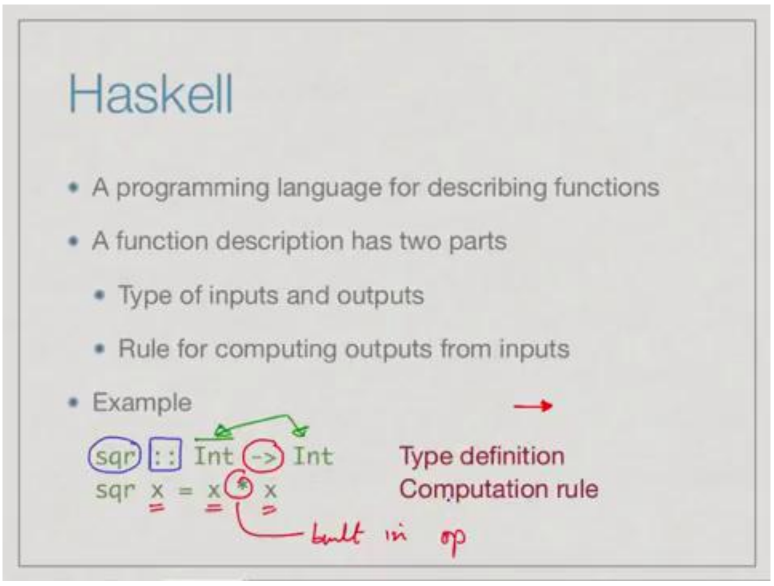
Module # 01

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 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)



Haskell

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

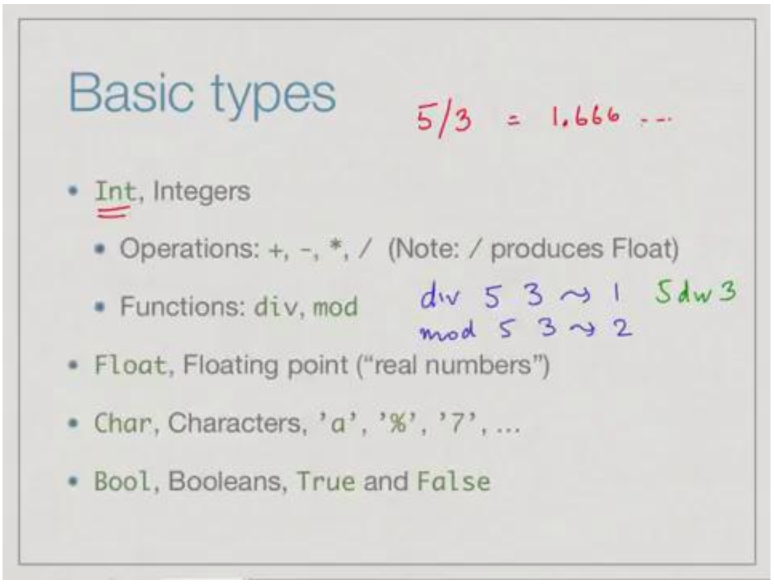
```
sqr :: Int -> Int        Type definition
sqr x = x * x            Computation rule
```

So, Haskell is basically a programming language for describing functions, in Haskell a function and 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 is the type definition.

As we will see Int is a built in type in Haskell, so Int stands for the integers which have the whole numbers including negative and positive numbers. And finally, this funny notation which is a minus followed by greater than is indicated of the mathematical arrow. So, this is two you will see in many Haskell syntactic forms in Haskell that we try to describe in characters something that is of familiar symbols from mathematics.

So, this says that square sqr is a function which takes integers as a 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 have given an input x then the output should be x times 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 the 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)



Basic types

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

$5/3 = 1.666 \ldots$
div 5 3 ⤳ 1
5 div 3
mod 5 3 ⤳ 2

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 I n t with a capital I and note that in Haskell variable names and types and all the upper case, lower case matters and all types by conventions start with the capital letter. So, capital I n t stands for the set of integers this is of course, the usual set of numbers 0 plus 1 minus 1 plus 2 minus 2 and so on.

And there are the usual arithmetic operations supported integers, addition which is denoted plus, subtraction, multiplication division. But, note the 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 bla, bla, bla 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 and divided by 3 is 2.

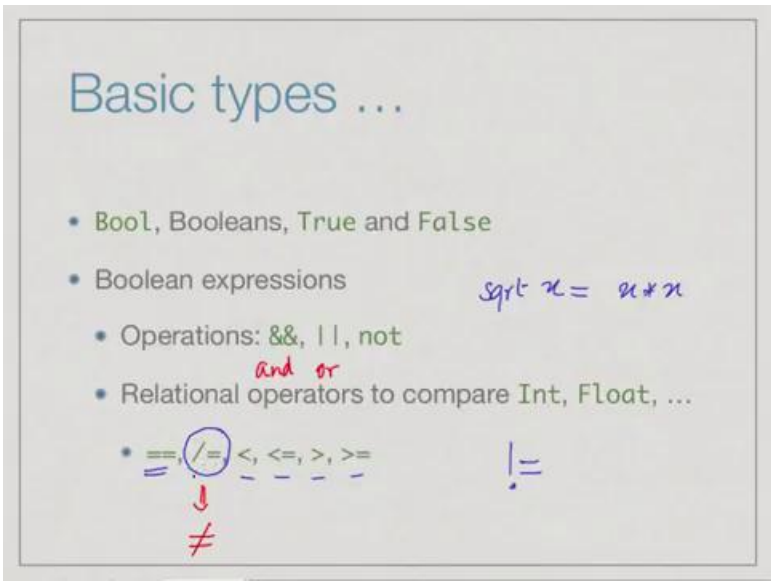
So, I should be careful not to write 5 div 3 there is a way to write it like this, but not the way I have written it here. So, you can take functions and treat them as operators and vice versa we looked at this later. But, for the moment just note that when it is a function you must write it as a function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

So, the other numbers rather then integers are of course, was so called real numbers and in Computer Science, they 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 position, but anyway that does not bother as 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 is a character and a character is written with single codes, provided with single code a, single code percent, single code 7. So, any character which we can type on the keyboard can be represents to a single code 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.

So, the Boolean type has two constant values in Haskell denoted True, T r u e with a capital T and false F a l s e with a capital F. So, Booleans are denoted True and False, so Booleans are not 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
  - Operations: &&, ||, not
  - Relational operators to compare Int, Float, ...
    - ==, /=, <, <=, >, >=

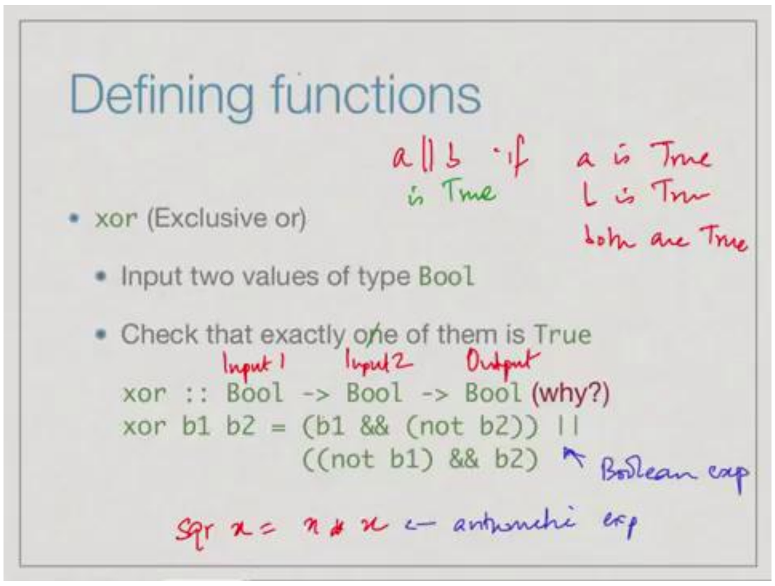
sqrt x = x*x
and
or
≠
!=

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 which is true provided both it is inputs are true, we have the Boolean OR which is true provided either one or both of it is inputs are true and then, we have the function which negates the value, not of true is false, not of false is true.

And notice that we use the word not, we do not uses single form and 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. So, equality because the single equal to in Haskell will be used for function definitions that we saw when we set square root of x is x star x. So, single equality is really equality in the sense of the function definition.

So, equality of values in many programming languages is denoted equal to equal to and then we have less than, less than or equal to greater than or equal to, mod is probably slightly unusual is the symbol for not equal to and some programming languages is written with exclamation not. But, Haskell as we mention before tries to mimic the symbols we actually use in mathematics. So, this is supposed to be a representation for the normal symbol equal to the slash across it which we use not equal to and we write mathematics by hand. So, just remember that slash equal to is the symbol for not equal to as not exclamation not.

(Refer Slide Time: 06:58)



Defining functions

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

```
xor :: Bool -> Bool -> Bool (why?)
xor b1 b2 = (b1 && (not b2)) ||
            ((not b1) && b2)
```

a || b if
is True
a is True
b is True
both are True
Input 1
Input 2
Output
Boolean exp
sqr x = x * x ← arithmetic 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, so we have a OR b is true, if a is true, b is true or both. So, this is the so called inclusive all, so this is true if either one or both of the values are true, the exclusive OR is the function which requires precisely one of the values to be true. So, you take two inputs of type Bool and checks that exactly one of them is true.

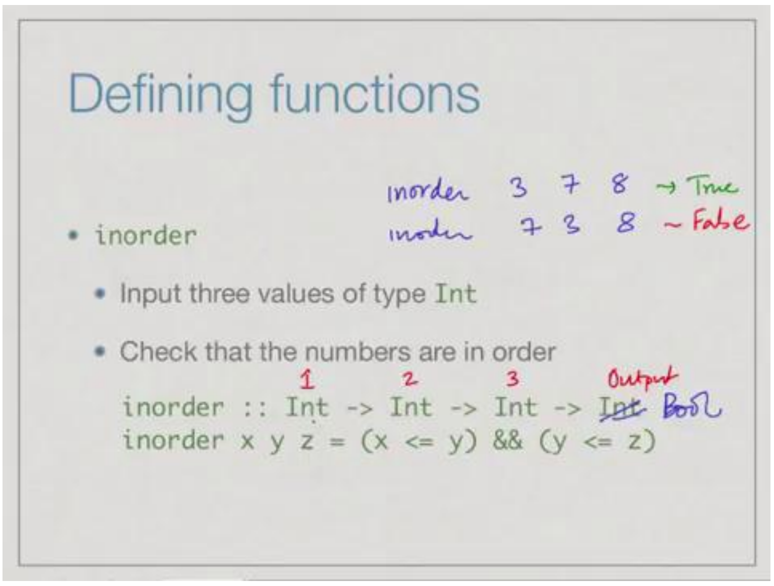
So, now, here is our first surprise that how do we write a function of the type of a function two input values, well just take it for a granted right now, that were just right the types of the value in suggestion followed by the output. So, this is the input 1, this is the input 2 and the last one is the output, why this is the case will become clearly 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 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 Boolean and produce the Boolean is Bool arrow Bool arrow Bool and what is the rule for this, well if I have given two input value b 1 and b 2 then either I want b 1 to be true and b 2 to be false, but b 2 is false provided not b 2 is true.

So, I want b 1 and not b 2 to be true or I want not b 1 and b 2, so I want b 1 to be false and b 2 to the false, so this is the exclusive OR function.

So, this is very similar to our earlier function which says square of x is x times x. So, here we use an arithmetic expression and here we are using a Boolean expression. So, in principle there is no difference between a function which takes numeric value and using numeric expression and a function will takes Boolean inputs and takes a Boolean expression, this form is very similar, it just as we have to be clear about what values of function are manipulate.

(Refer Slide Time: 09:25)



Defining functions

- inorder
  - Input three values of type Int
  - Check that the numbers are in order

```
inorder :: Int -> Int -> Int -> ~~Int~~ Bool
inorder x y z = (x <= y) && (y <= z)
```

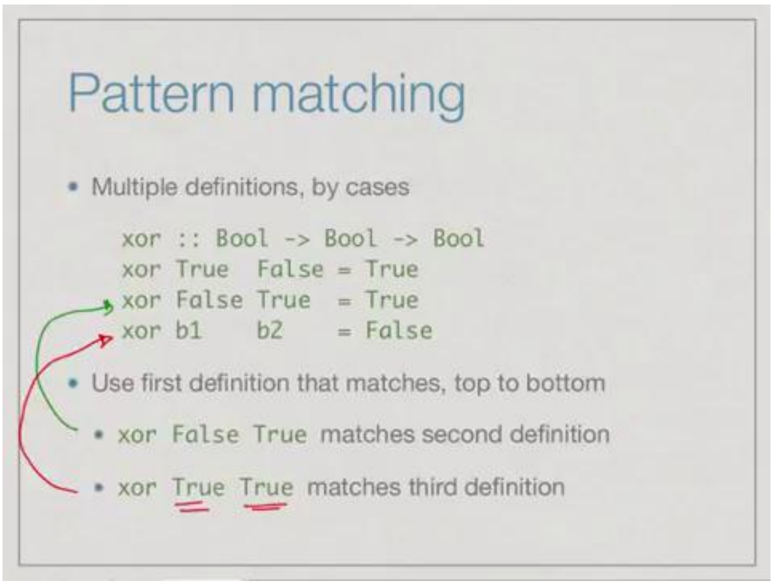
inorder 3 7 8 → True
inorder 7 3 8 ~ False
1
2
3
Output

So, let us look at another function now it mixes the two. So, supposing we want to take three integers and just check that the three integers are in orders. So, we want to say something like in order of say 3, 7, 8 this should be true, but if I am on the other hand if I say in order of 7, 3, 8 we should be false. So, we have 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 an Int, now given our earlier discussion about how we write the 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 arrow Int arrow Int arrow Int, so the first three Ints corresponds to first three inputs in that sequence and the last one is the output type. So, it says takes x, y and z let these three be the input values, then check that x is less than equal to y and y is less than equal to z. So, we 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 it will apply this relational operator to the first two input, relational operator second pair of inputs and use a Boolean operator to combine them. So, it produce a Boolean expression from three integers. So, this is the operate function which mixes types from the input to the output, so it converts integer inputs to Boolean outputs.

(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

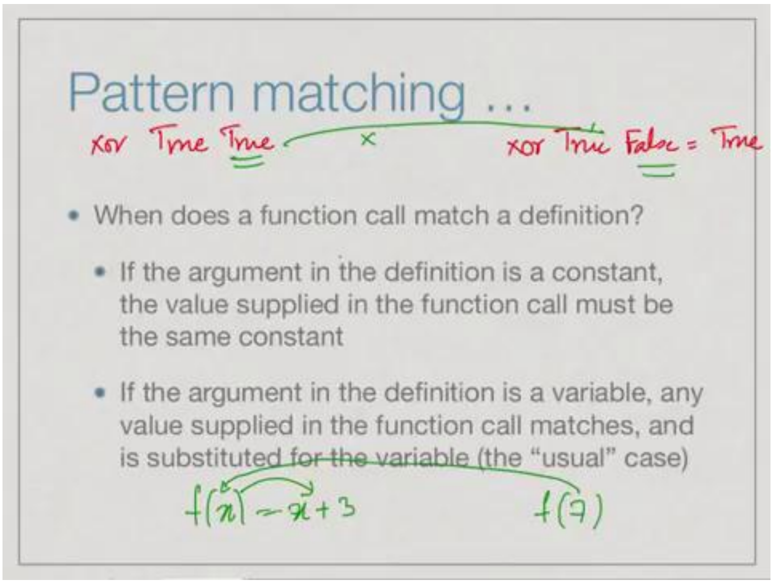
So, now let us illustrate more complicated ways of defining functions using the function that we saw before. So, recall that exclusive OR is the function, which tries to check that exactly one of it is inputs is true, now in the case of Boolean values we only have two possibilities for the inputs, they 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 let we have b 1 b 2 and we have the output, so we can enumerate these things you can say this is true, this is false, this is true, this is true, this is false, this is false, this is false, this is false, this is true these are 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 we say XOR of true and false is true.

Similarly, if this is the case you say XOR of false and 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 given input to this function, we look at the input and see whether matches the definition. So, if I give XOR of false and 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 is 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 true 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 called to match a definition that is quit straight forward.

(Refer Slide Time: 13:11)



Pattern matching ...

- 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)

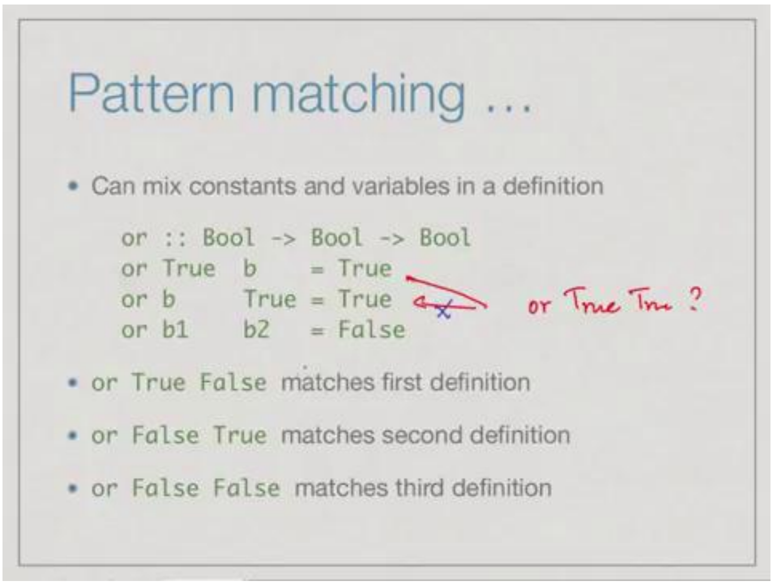
xor True True
×
xor True False = True
f(x) = x + 3
f(7)

So, if the argument in the definition is a constant, so supposing I write XOR true false equal to 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 called matches the definition only if the same constant appears in the function. On the other hand, as we saw before if we have b 1 or b 2 then if the definition is a variable then any value matches that call and then you substituted as well.

So, this is what we normally use when we have function, if we write f of x equal to something and then we write f of 7 then 7 is substituted for x and then everywhere in this if I write x plus 3 then 7 becomes the value here as well. So, I will get 10, so this is how the normally write function, 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 and Haskell is that I can give this constant patterns and say that if my variable is of a fixed type, fixed value then that pattern will be match and that definition could be matched. So, we will see more examples 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 True True ?
or b1    b2   = False
```

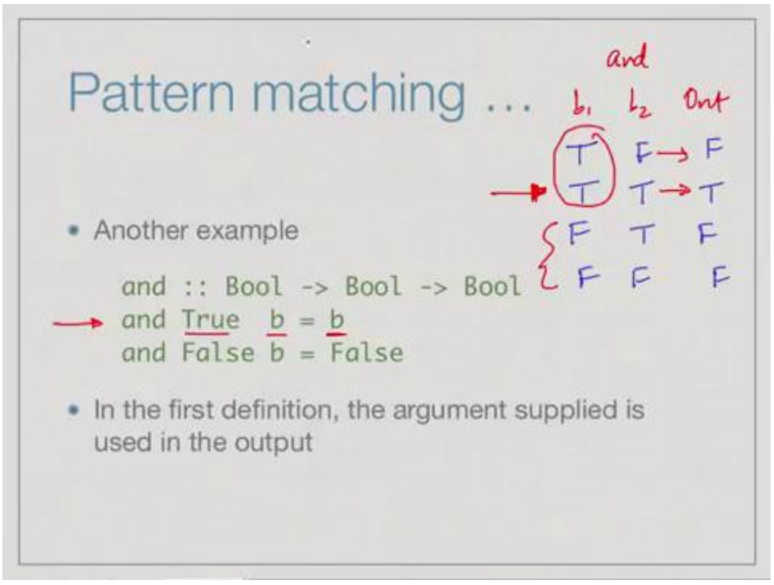
- `or True False` matches first definition
- `or False True` matches second definition
- `or False False` matches third definition

So, here is the definition of the build in function OR, but we are writing it ourselves. So, remember that OR for 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 this is here now is that I mixing constants and variable, it is says the first argument is definitely true and whatever second argument may be the output is true. So, that says that I am either in this case or I am in this case.

Now, if I have the second argument true then whatever the first argument maybe the answer is true. So, now, I am in this case or in this case and finally, if neither of these thing holes then the first argument is not true and the second argument is not true, then I must be in this case, so the output must be false. So, here are true false matches the first definition OR false true matches the second definition OR false, false matches in third definition.

And what about OR true, true well OR true, true matches both this definition and this 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 with matches multiple definition will match the first one from the top with succeeds.

(Refer Slide Time: 16:34)



Pattern matching ...

- Another example

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

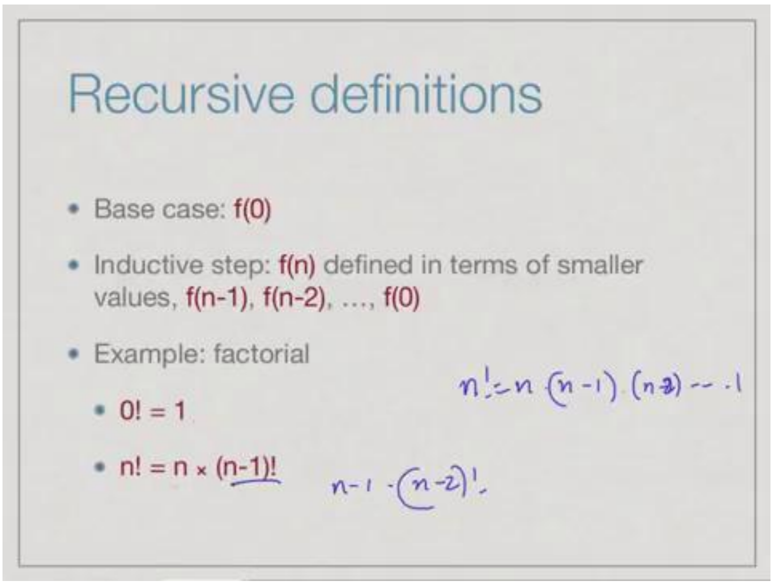
- In the first definition, the argument supplied is used in the output

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 pick the first input is true and the second false and the output is false, if it is true and true that it is true, if it is false and true and it is false and if it is false and false and this false. So, basically this is the only case in which and will give true which is both inputs are true, but notice that in the first input is true then the output is exactly the second input, in the second input is false the output is false, second 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 in directly as the output, if it is false it is false, if it is true it is true. So, that is capturing these two rule and finally, these two rules both have this common fact that the first argument is false. So, we say the first argument is false no matter what the other argument is the output is false.

So, here we are mixing first of all Constance 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 our case, in the our case we explicitly computed a constant of 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), ..., f(0)
- Example: factorial
  - 0! = 1
  - n! = n × $\underline{(n-1)!}$

$n! = n\,(n-1)\,(n-2) \cdots 1$

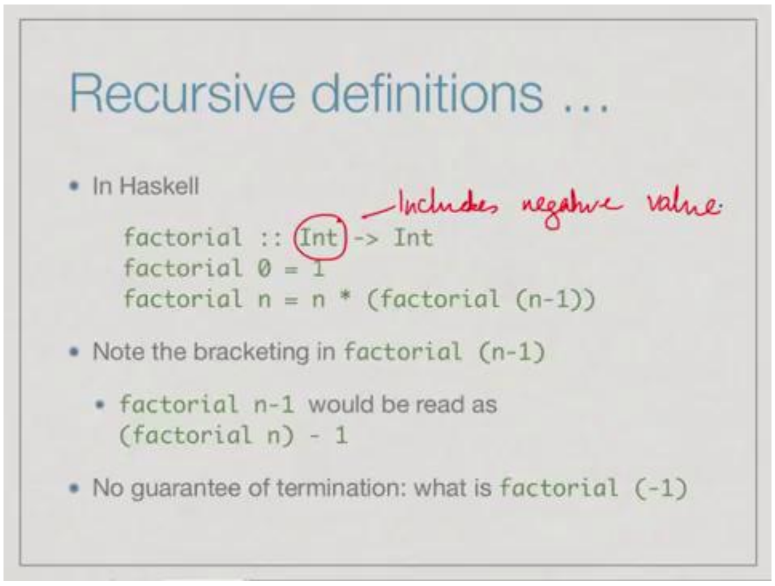
$n-1 \cdot (n-2)!$

So, as we saw in the beginning when we are defining functions we typically need to right functions which apply operations are compose functions and 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, recursively define function we specify a base case, so for example, if it is for the Bool numbers we would define the value for f of 0 and then inductively for f of n we would write an expression involving smaller values of the arguments, so f of n minus 1, f of n minus 2 down to f of 0.

So, you can define it will terms are any smaller value assuming that those have been inductively computed already. So, this is the meaning of a recursive or an inductively definition and one of the most standard examples use to illustrate recursive and inductive definitions is a factorial function. So, by definition 0 factorial is 1 and n factorial this n times m minus 1 and of course, if you unravel this, this in term becomes n minus 1 times n minus 2 factorial and so on.

So, we will get the familiar expression that n factorial is n into minus 1 into n minus 2 up to 0, n factorial is the product of all the numbers from1 to n, but this is recursively capture by these two rules 0 factorial is 1, n factorial is n into n minus 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))
```

Includes negative value.

- 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)`

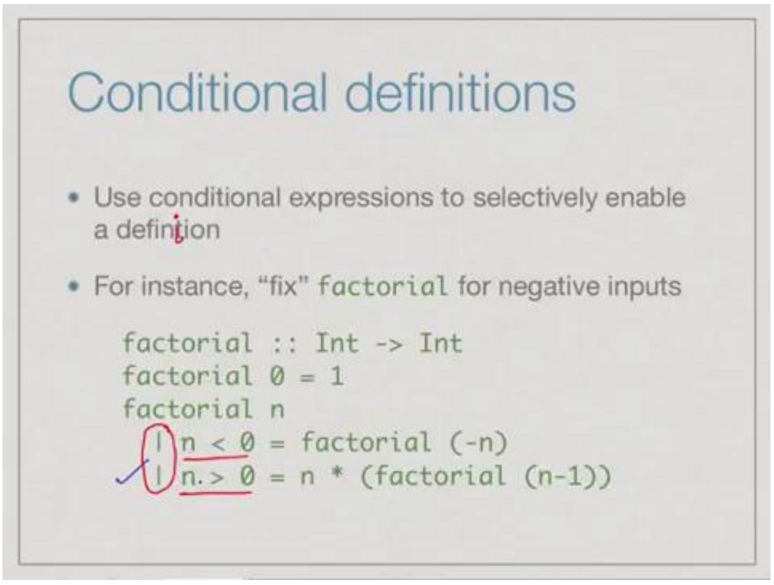
So, we say that factorial is a function which takes an integer and produces integers ((Refer Time: 19:48)) factorial of 0 is a 1. So, this is the base case if my input is already the base case I know the answer, if it is not that base case then I apply the rules saying I take n into factorial n minus 1. So, first note this expression, so we have been careful to put n minus 1 in brackets before providing it a factorial, because the way Haskell works if I just write factorial n minus 1 without brackets, then it will put the brackets around factorial n.

So, it will compute factorial n and then subtract 1 which is not we want. In fact, that would leave has into a kind of infinite regression of computation, because factorial n is then defined again in terms of factorial n. So, it will have to apply the same rule and we want get any answer. So, this is one thing to another and other thing to note is that Haskell does not guarantee you that fact that we have return a recursive definition means that it always works correctly all in works.

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 the 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 minus 1 factorial of minus 1 will say minus 1 is not 0. So, the first rule does not apply, so the second rule does not apply. So, it will be minus 1 times factorial of minus 1 minus 1 is minus 2 then again the rule does not applies to minus 2 we call minus 3 and so on.

So, this function as started works correctly for positive values of n, but because the type Int, so this type Int includes negative values Haskell's built in type Int include negative values and there is no Haskell type built in type which consist of only the non negative integers. So, if I write factorial and have to write it 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

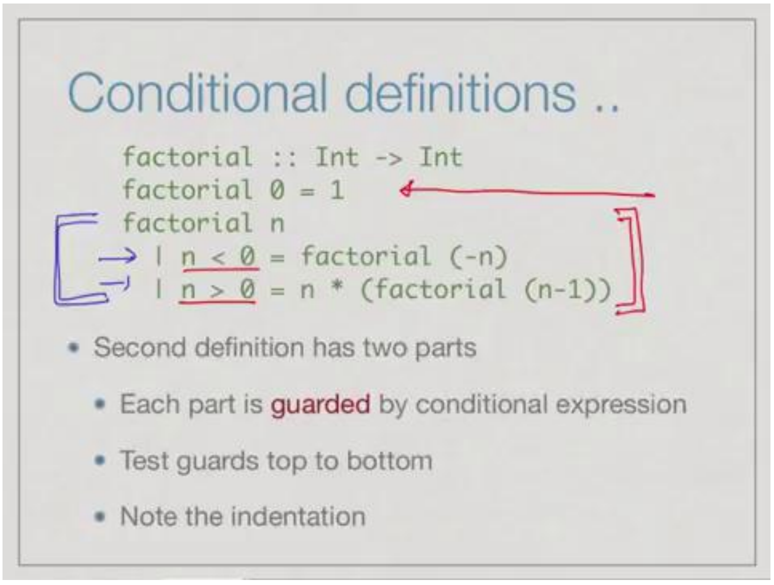
```
factorial :: Int -> Int
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 0 = n * (factorial (n-1))
```

So, we can fix this by checking if the input is negative, so we need to now extend are syntax for defining functions to use conditional expressions to say when a definition is enabled. So, for instance we have the base cases before the factorial of 0 is 1, now if it is not 0 it could be positive warning, the positive case is as before if n is greater than 0 then I want to say that the output is n times factorial of n minus 1, but what if n is less than 0 well of course, factorial of a negative number is not mathematically define, but for the sake of computation I can always make it work, reversing the sign of inputs.

So, if I ask to factorial of minus 6 then I will convert to factorial of 6 and give you that answer. So, at least computation the value will be produced in finite amount of time instead of going into a Int to Int condition. So, I say that if n is less than 0 then factorial of n is computed by taking factorial of minus n. So, minus 6 will become minus of minus 6 which is plus 6. So, what we have here is we have a conditional expression n less than 0 and so we have this equality remember is a function definition.

So, it says that factorial of n if n is less than 0 is factorial of minus n and if n is greater than 0 then it is n times factorial n minus 1. So, this vertical bar signifies options, so it says evaluate this condition if this condition is true use this definition; otherwise, go to the next condition 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))
```

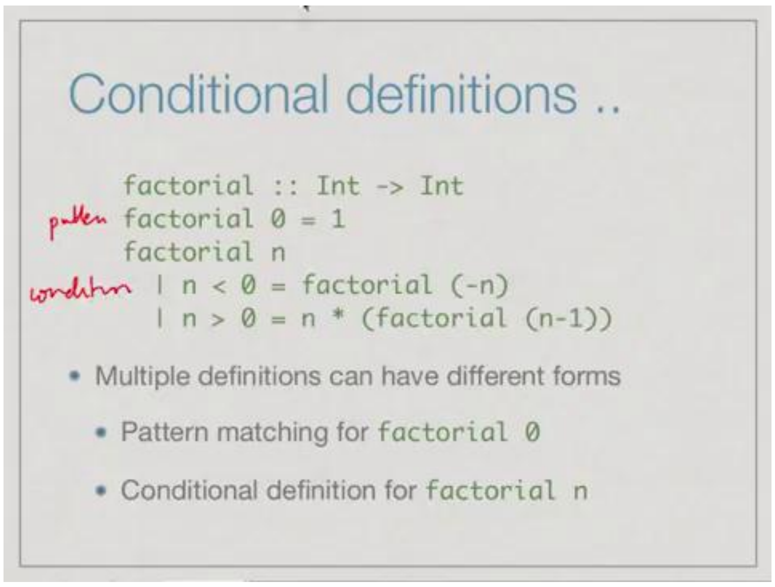
- Second definition has two parts
  - Each part is **guarded** by conditional expression
  - Test guards top to bottom
  - Note the indentation

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 guys saying can you use definition to what a value is only then you can go forward and we check that guards and top with bottom and we use the first guard that works. In this case only one of them either n is less than 0 if n is not 0 either must m is less than 0 or n is bigger than 0. Now, notice that n must be an integer, because we have already describe the 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 intended list.

So, technically this whole thing these three lines together constitutes one definition one definition with guards. So, there are over all 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 with the pattern is not match.

(Refer Slide Time: 25:07)



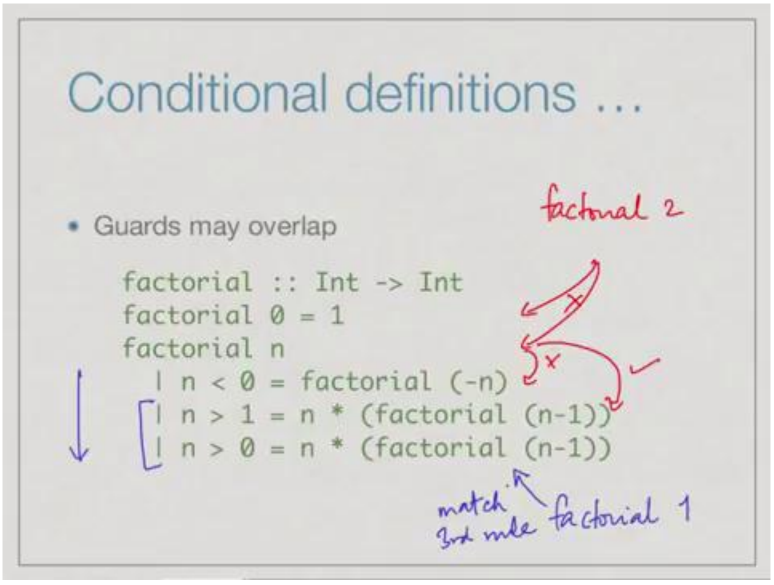
Conditional definitions ..

```
     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

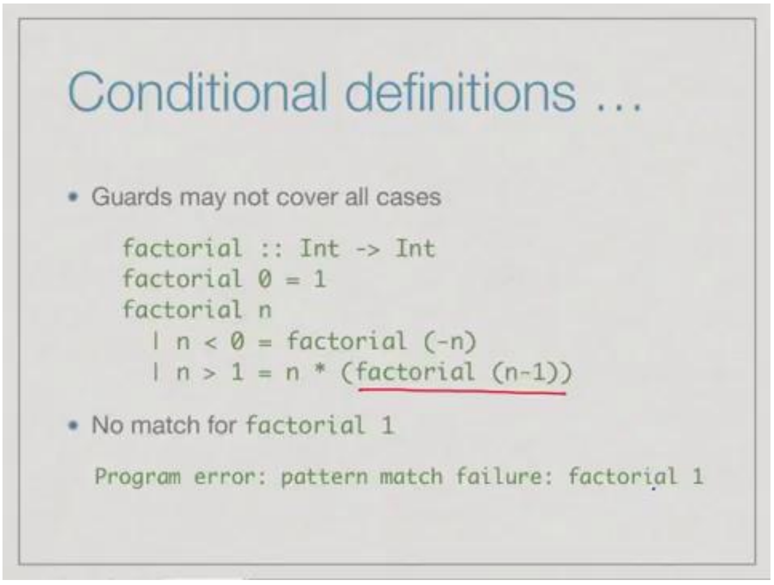
```
factorial :: Int -> Int
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 1 = n * (factorial (n-1))
  | n > 0 = n * (factorial (n-1))
```

factorial 2
match 3rd rule
factorial 1

Now, suppose we fiddle with the definition, this is not very meaningful computation, but I can just add I can split this second case n greater than 0 as first n greater than 1 and n greater than 0. A 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 match. So, do it matches a third rule it will not reach third.

So, the only value that will reset third rule is what, you are right the only value the 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 will. So, guards may over lap, there is no requirement that the guards must be exclusive that it 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
```

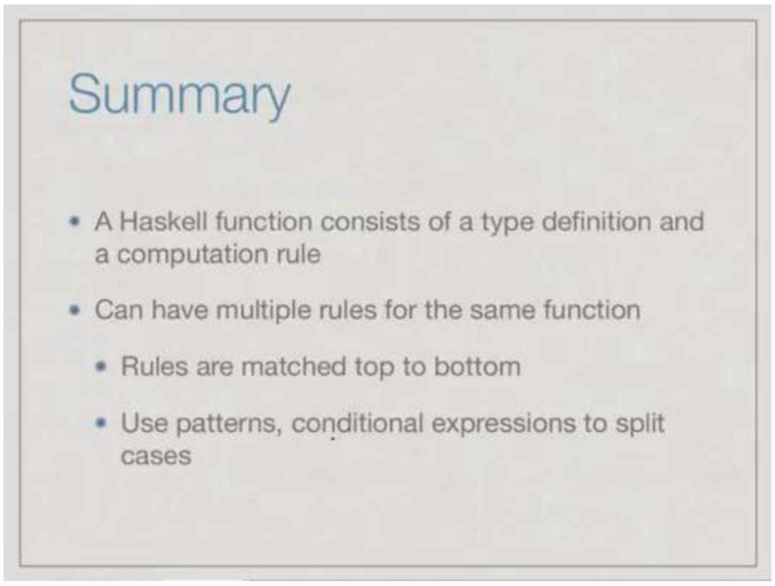
Now, the other thing is just as we saw before that fact that recursive definition of Haskell need not terminate, Haskell we not promise you that the value actually terminates and the function definition terminates and gives you are value. For instance, from we did not take care of negative input factorial of minus 1 was giving as a undominated computation. So, here supposing I write this definition and I have made a mistake, so I got n less than 1 and greater than 1.

So, now, if I look at the set of integers then I have this for the base case and I have everything and minus 1 on those. So, this is the definition and I have everything from to onwards power by this definition. So, now, I have a basically a problem because, I powered this, but this thing is not power. So, if say factorial of 1 if does not match the first case, it does not match either of the guards and now what will happens is that if I actually run this Haskell it will give me a error saying that no patterns matches.

So, the function definition can miss cases and in some situations when you miss cases some values make give you outputs which are correct 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 on 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 times factorial 2, 2 times factorial 1 and when I try to value factorial 1 and get this error message send that program a match failure.

So, not just if I provide factorial 1, but if we have provide factorial of anything even thought the first few times it may match eventually the recursive this computation will come down to factorial 1 and when it comes to factorial 1 and get some. 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 summaries a Haskell function consists of a type definition and computational rule. Now, these computational rules can be use to define for example, recursive functions we can have multiple rules for the same function and the rules are match top to bottom and the way we define rules for different input values is to use patterns of condition expressions. So, split the cases that have possible for the input values.