

Introduction To Haskell Programming

Prof. 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 is titled "Haskell" in a large, light blue font. Below the title, there are four bullet points:

- 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 "Example" bullet point, there is a code snippet with handwritten annotations:

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

Handwritten annotations include:

- A green arrow pointing from the `sqr` variable to the `Int` type in the type signature.
- A green arrow pointing from the `Int` type in the type signature to the `Int` type in the function body.
- A red circle around the `->` symbol in the type signature.
- A red arrow pointing from the text "Type definition" to the `sqr :: Int -> Int` line.
- A red arrow pointing from the text "Computation rule" to the `sqr x = x * x` line.
- A red circle around the `*` operator in the function body.
- A red arrow pointing from the text "built in op" to the `*` operator.

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 is titled "Basic types" in blue. It contains a bulleted list of Haskell types and operations, with handwritten examples in red and green.

- Int, Integers
 - Operations: +, -, *, / (Note: / produces Float)
 - Functions: `div`, `mod` $5/3 = 1.666 \dots$
 $\text{div } 5 \ 3 \rightsquigarrow 1$ $\text{mod } 5 \ 3 \rightsquigarrow 2$
- 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)

The slide is titled "Basic types ...". It contains a bulleted list of Haskell basic types and operators. The first bullet is "Bool, Booleans, True and False". The second bullet is "Boolean expressions", with a handwritten note "sqrt x = x * x" to its right. The third bullet is "Operations: &&, ||, not", with handwritten notes "and or" above the ampersand and pipe symbols. The fourth bullet is "Relational operators to compare Int, Float, ...". Below this bullet, there is a list of operators: "==" (double equals), a circled "/" (not equal), "<", "<=", ">", ">=", and "!=" (exclamation mark equals). A red arrow points from the circled "/" down to a red handwritten "≠" symbol.

- Bool, Booleans, True and False
- Boolean expressions $\text{sqrt } x = x * x$
- Operations: &&, ||, not
and or
- Relational operators to compare Int, Float, ...
• ==, /=, <, <=, >, >=, !=
↓
≠

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

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

a || b - if a is True, b is True, both are True

Input 1 Input 2 Output

```
xor :: Bool -> Bool -> Bool (why?)  
xor b1 b2 = (b1 && (not b2)) ||  
            ((not b1) && b2) 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 \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 not $b2$ is true. So, I want $b1$ and not $b2$ to be true or I want 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 $\text{sqr } x = x * x$. 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 which 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`
 - Input three values of type `Int`
 - Check that the numbers are in order

$\text{inorder } 3 \ 7 \ 8 \rightarrow \text{True}$
 $\text{inorder } 7 \ 3 \ 8 \rightarrow \text{False}$

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

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 '`<=`' to the first two inputs, relational operator '`<=`' 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 b1, b2 and we have the output, so we can enumerate these things. You can say b1 is True, b2 is False, b1 is True, b2 is True, b1 is False, b2 is False, b1 is False, b2 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 b1 and b2 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 ...

xor True True \times xor True False = 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 b1    b2  = False
```

or True True ?

- 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 ...

and

b_1	b_2	Out
T	F	F
T	T	T
F	T	F
F	F	F

• 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 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, $f(n-1)$, $f(n-2)$, ..., $f(0)$
- Example: factorial
 - $0! = 1$
 - $n! = n \times (n-1)!$

Handwritten notes on the slide:

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

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

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))
```

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 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 what 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 won't 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 makes sense if n is positive, because if n is positive then I come down to 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))
```

Handwritten notes:

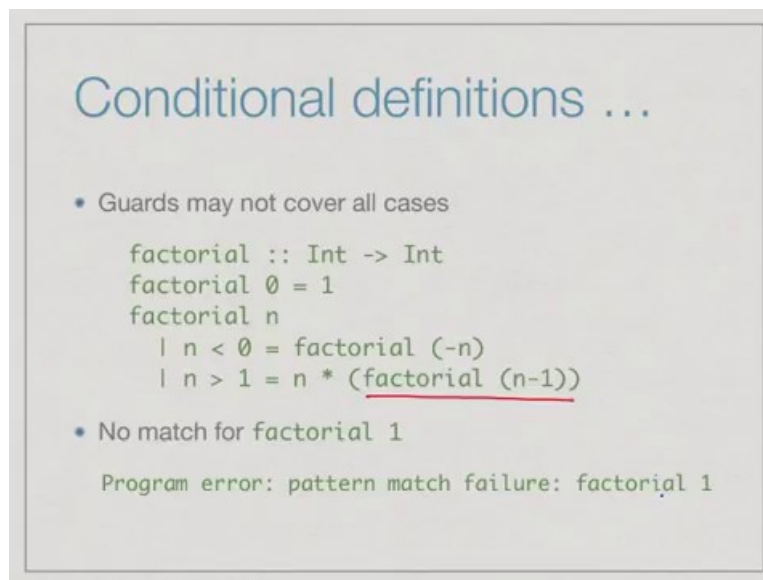
- factorial 2* (with arrows pointing to the `n > 1` and `n > 0` guards)
- match 3rd rule factorial 1* (with an arrow pointing to the `n > 0` guard)

Now, suppose we fiddle with this definition, this is not very meaningful computation, but I can just I can split this second case $n > 0$ 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

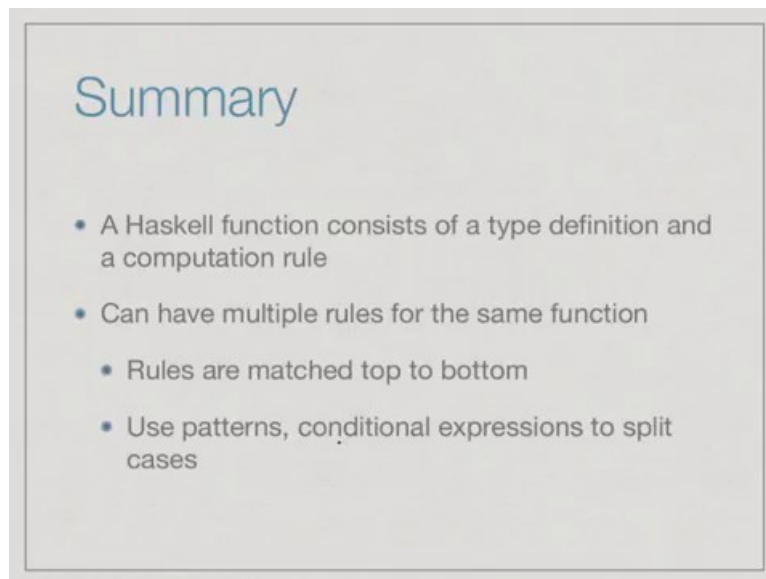
Now, the other thing is, just as we saw before the fact that recursive definition of Haskell need not terminate, Haskell need not promise you that the function definition actually terminates and gives you a value. For instance, when we did not take care of negative input factorial of -1 was giving us 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 a 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 $3 * \text{factorial } 2$, $2 * \text{factorial } 1$ 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.