Function Programming in Haskell

Prof. Madhavan Mukund and S. P. Suresh

Chennai Mathematical Institute

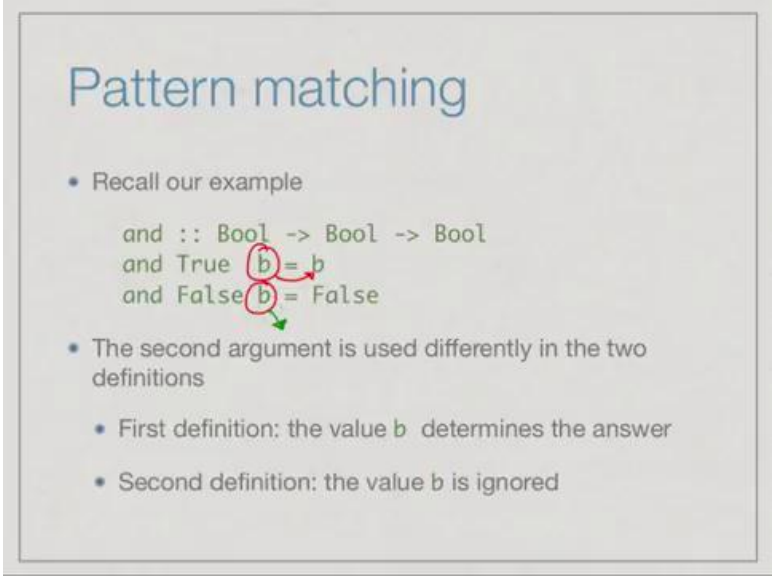
Module # 01

Lecture – 06

Haskell Examples

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

(Refer Slide Time: 00:10)



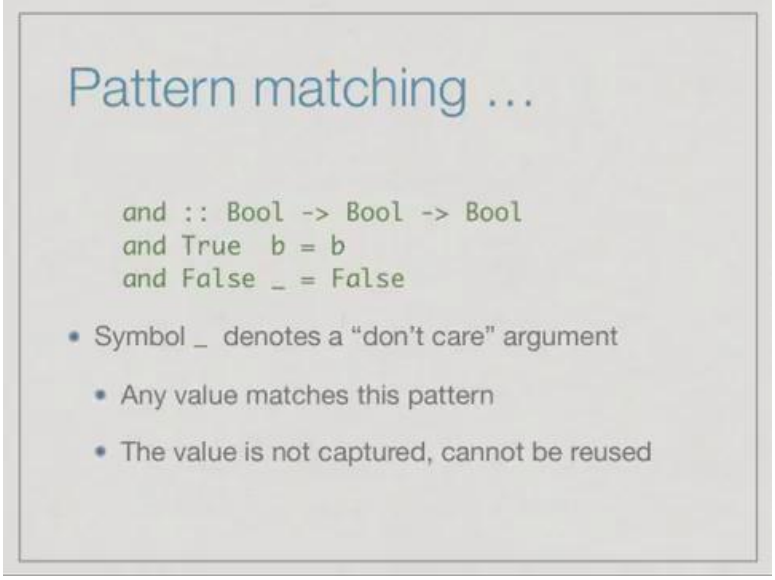
Before we do that, we will revisit a couple of the notions that we have seen for writing Haskell function and explain a few concepts that we did not do when we introduce them. So, we begin with pattern matching, so remember that in Haskell functions, we can use patterns to illustrate cases. So, here 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 is 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 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)

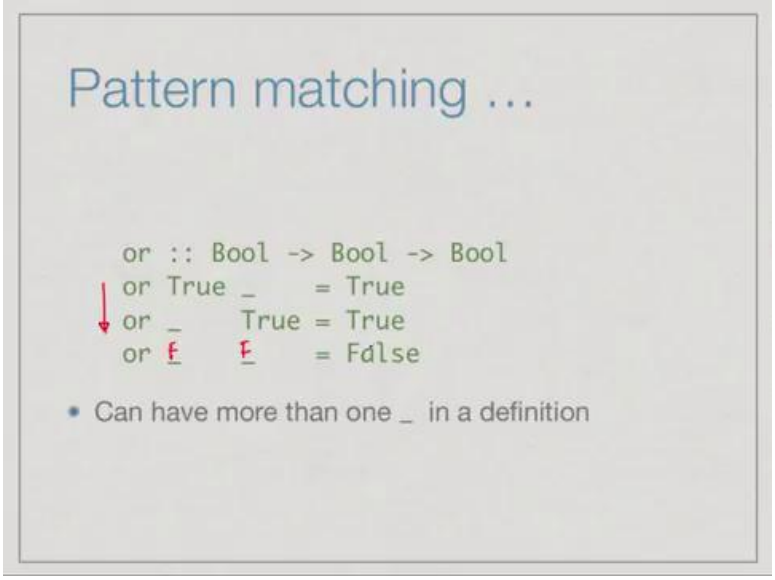


So, there is a special notation in Haskell, which allows us to specify a wild card. So, this symbol underscore denotes an argument with matches everything. So, it is like using the variable, remember a variable matches everything and it will be substituted by the value we provide. So, the difference between a wild card, which is the do not 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 capture 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 makes it also transparent that this function really does not care about second value. So, from the point of view reading the code is immediately

obvious that AND with false in the first argument does not require second argument give it answer.

(Refer Slide Time: 02:36)



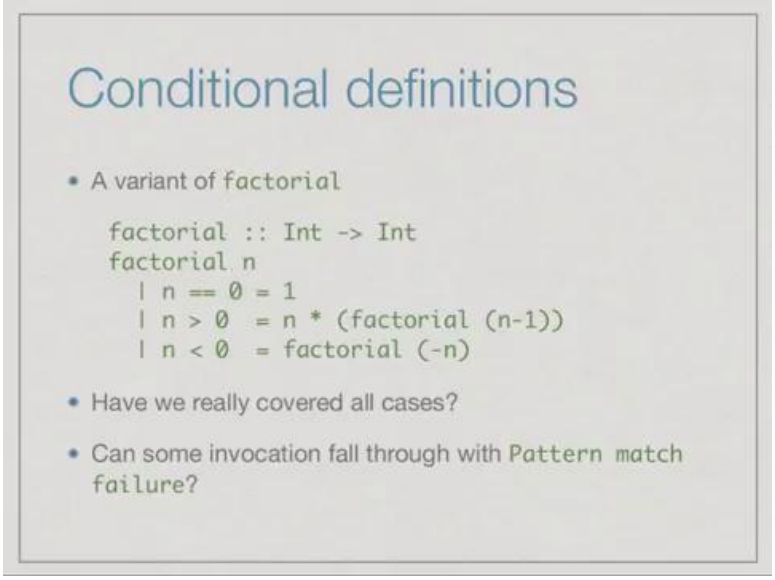
So, here is more extreme version of that, here is the version of or that we have written, but now with these wild cards. So, or says that if the first argument is true, it does not matter what the second argument is, the answer is true. Similarly, the second argument is true, it does not matter what is the first argument, 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 it is own would be useless, but because it comes after the first two definitions.

So, we already gone, because these two cases, we know that, this must be false in this must be false in this case, but it says is, if I reached here at this point, if I reach the third line and I have not the match 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, is really emphasizing the fact that underscore is not a name of a value.

We cannot write two arguments with the same name and the normal definition, because then they 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 on the hand side, which x we are going to be use, whereas, if you use underscore, you cannot

use the value in the 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)

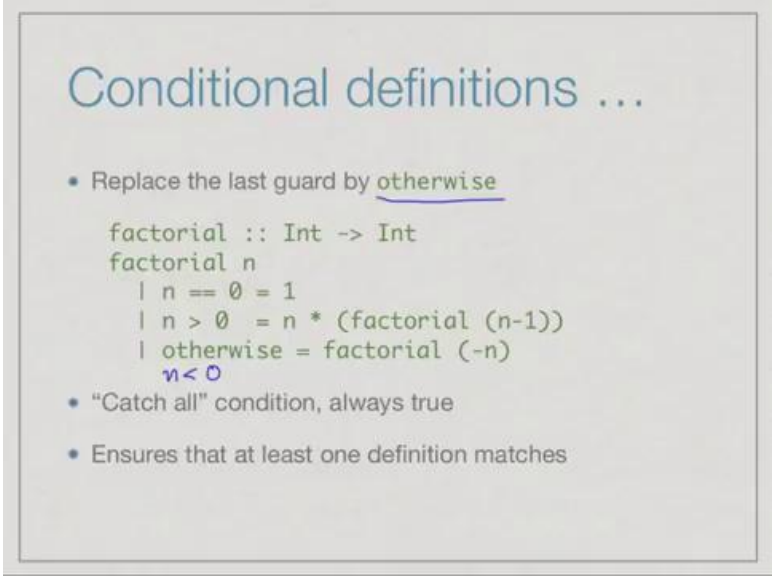


The other new concept that is useful to use is to do with conditional definitions. So, here is variation of our factorial function which handles negative numbers. Earlier, we are given factorial 0 as a pattern match, so we had return something like factorial of 0 is 1 as the separate case. And then,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

But, we can as well use the conditional to the give a completely conditional reference. So, this in a more conventional language, you will say something like, if n is 0, then the answer is 1, else if n is greater than 0, then the answer is n into factorial n minus 1. Else, if n is less than 0, then factorial is the value factorial minus 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, this here we can check that n must be 0 or greater than 0 or less than 0. But, if it for a more complicated situation, it may well be difficult to determine whether every input actually matches one of these things. So, have been really covered all the cases or is there a situation, where we are miss something in which case some inputs might give us some error about pattern match failures.

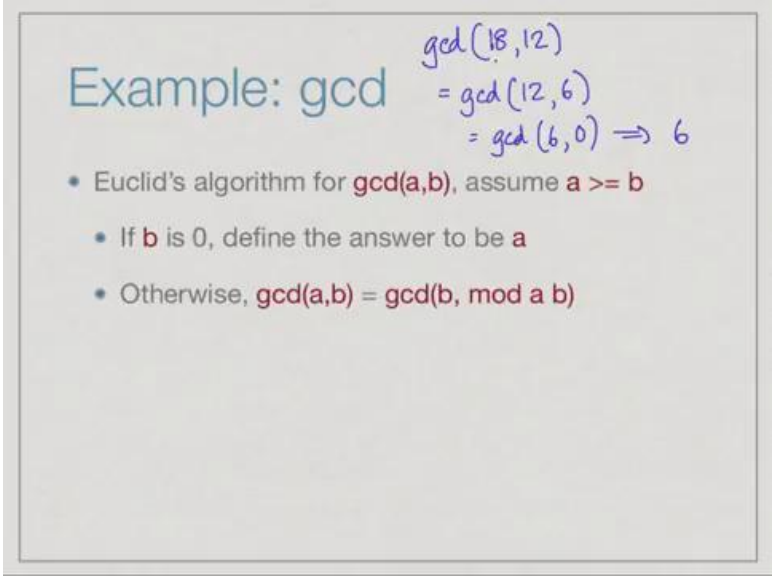
(Refer Slide Time: 05:09)



So, the equivalent of the else in a normal programming language, so normally we say if condition 1, then something else condition 2, then something and so on and finally, the last one can catch all else. So, here there is a catch all word is called otherwise. So, instead of saying explicitly, but the last cases n less than 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 towards in Haskell, which is a condition, which is always true. So, if we reach the otherwise condition in a sequence of guards, then back guard will always evaluate to true and whatever definition this provided is that guard will match. So, this ensures that every in location will match at least one of the definitions in a condition. So, we have these two new things, we have this do not care patterns and otherwise and we will use these to simplify some of the code that we will develop in the examples this types.

(Refer Slide Time: 06:08)

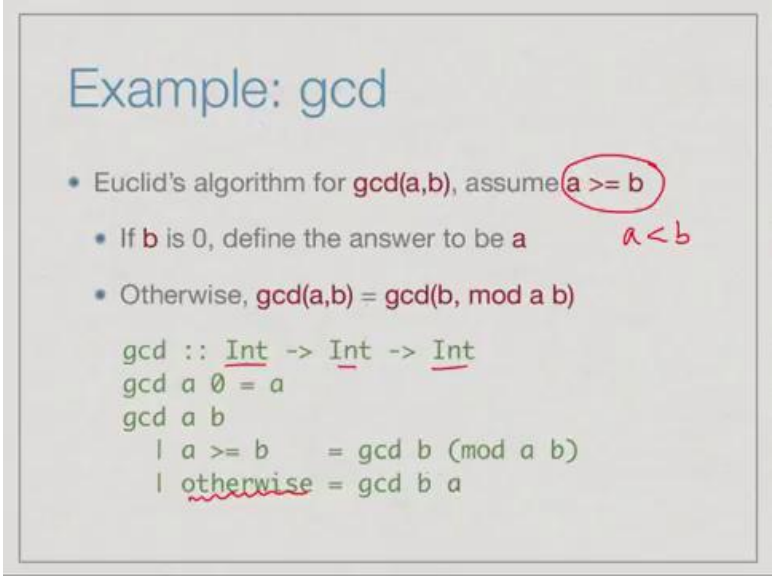


So, let us look at a first example, the first example is the very traditional example and one which illustrate is many Interesting concepts and programming in all languages. So, this is the GCD algorithm. So, remember this GCD of a and b is the greatest common device, it is the largest number the divides both a and b and since one definitely divides both a and b, this is always well defined, it is at least 1. So, if the GCD is 1, then there set to be co prime.

So, assuming that a is the bigger of the two numbers, then what Euclid propose for following algorithm this that we keep replacing the GCD of a, b by the GCD of the smaller number and the remainder of the larger number, divide by a smaller number. So, for instance, if we start with the GCD of say 18 and 12, then this will say replaces by GCD of this smaller number.

And the remainder if I divide 18 by 12, I get 6 as a reminder; because it is 1, remainder 6, then I will get GCD of 6 and the remainder of 12 divided 6 is 0 and then, we should be the base case and this will say GCD's 6, if b is 0, the answer is a. So, this is the Euclid algorithm, we will not worry about it is correctness, but it is useful for you to think about why this works. But, it keeps reducing GCD to smaller and smaller number in your guarantee that it will terminate. In fact, it is an efficient algorithm; it is it turns out to be proportional to the number of digits. So, it is actually a linear algorithm in the size of it is input.

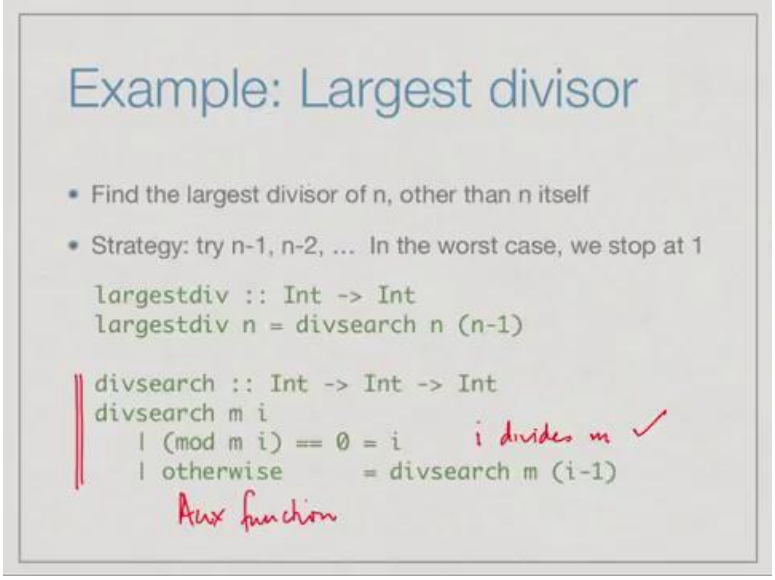
(Refer Slide Time: 07:42)



So, this is a very simple function to code and Haskell. So, GCD first of all takes two Integers in produces an Integers; that is the type. So, the base case is easy, GCD of a and 0 for any a is just the value a itself. Then, if we assume as we have done here; that a is greater than b, then we take the GCD of the smaller number and we have the mod function which gives as the remainder.

So, it is GCD of a and b, if b is not already 0 is given by GCD of b and the remainder of a divided by b and in case it turns out that this assumption is false. So, we have actually have a less than b, then we just reversal. So, here we are using this new word that we have discovered 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 in the remainder will be smaller than b, all successive calls to GCD will have this property. So, leave a first call which may be wrong which case be reverse. So, this is like are earlier case will be set factorial of negative number can be fixed by taking the negation of the end.

(Refer Slide Time: 08:48)



So, here is a 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 one divides n. So, one is certainly a candidate just like a GCD, we know that 1 is a GCD, so 1 will always work. So, here we can just iteratively try n minus 1, n minus 2.

So, we go back we want to largest 1. So, we go back in reverse order from n, we start with n minus 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 just trying for each i from n minus 1, n minus 2, down to 1, if i divides n, then we stop and report that. So, this loop can be simulated using the simple recursive call, so we say that, so the operative things actually the second function.

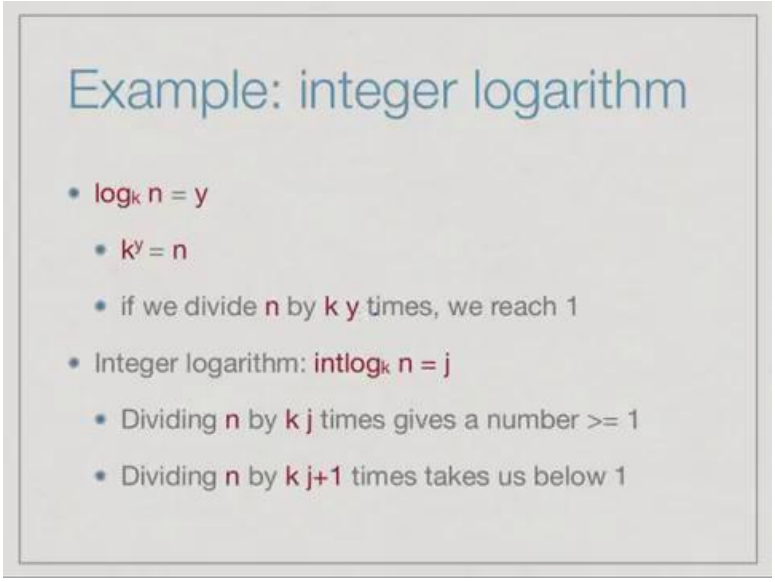
So, here is an example where the function we want to is expressive terms of other function. So, the largest divisor of n can be obtained by searching for devices of n starting from n minus 1. How do we search, well, if we get an answer, if the second argument divides the first argument, so if m divided by i is 0, then we are found a argument, 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 minus 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 and what order you write them, it does not have to be before or after and these functions can call each other.

So, we have largest divisor the function we want, which calls this divisor search and divisor search is an Interesting thing about the divisor search is a kind of recursive version of what you normally call is looks, this looking at i and if it is not i, it is going to i minus 1.

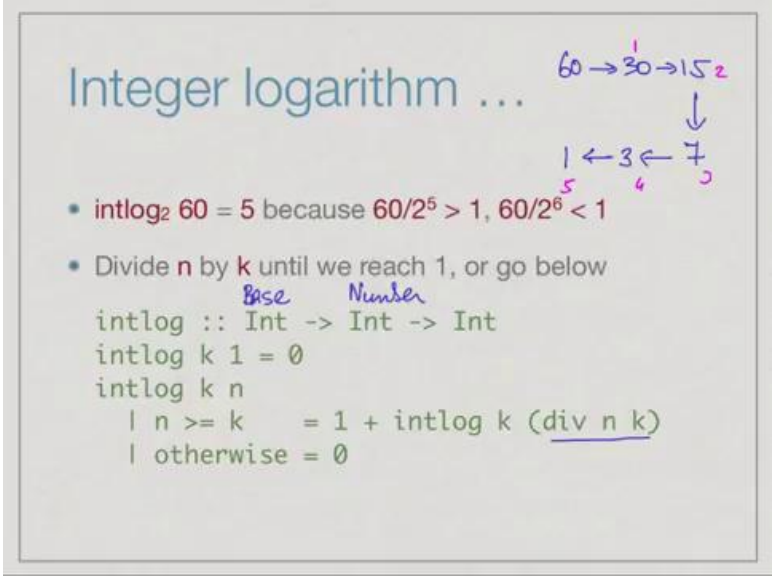
(Refer Slide Time: 11:02)



So, we know what the logarithm is, logarithm tells us what power we need to get back to the number. So, the log to the base k of n is y provided k to the y is n and other 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 be do this until we reach 1. So, usually the logarithm is some fractional number.

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

(Refer Slide Time: 11:49)



So, as an example, if you take the base 2 and we take the number 16, then the Integer log is 5, because if we divide 5 times, then we stay above 1, 2 to the 5 by the way equal to 32. So, 60 divided by 32 is more than 1, 2 to the 6 is 64. So, 60 divided by 2 to the 6 less than 1. So, therefore, by our definition 5 is the largest number of time we can divide by 2 and stay above 1.

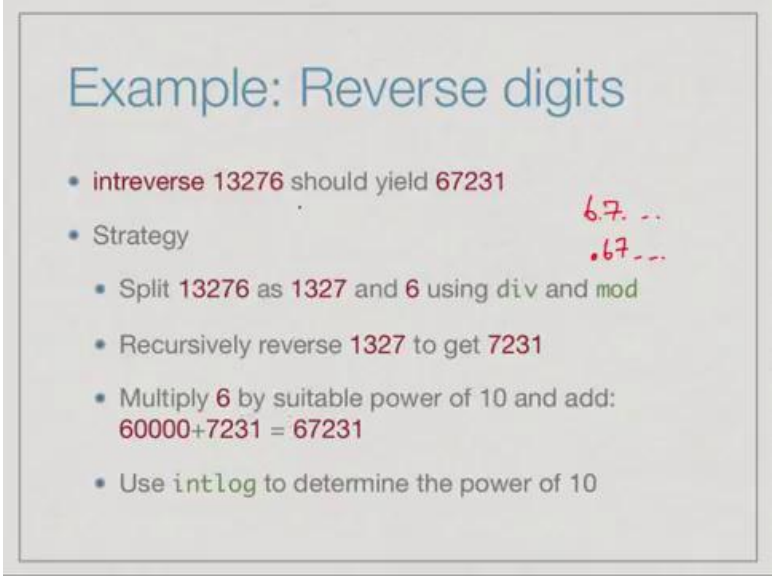
So, this gives as a strategy to compute the Integer logarithm, we start with n, we keep dividing by the base k until we reach 1 or go below. If you reach 1 exactly, then we get the exact logarithm. So, for example, if we are done Integer logarithm is 64, then we got 6 and this is exactly, because 64 divide by 2 to the 6 is equal to 1. But, because we start with something, which was half that we got something, which went below 1, so either we reach 1 or we go below.

So, here is a function, it takes here now we should we clear that, this is the base and this is the number. So, the first argument to Integer log, because of the way we it we write Int log of the base of n. So, k is a base, 1 is a number. So, anything rise to 0 is 1. So, if we get the number 1 as a input, then the base we respect to base case 0 by definition, otherwise provider we can divided.

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

our case, we go to 30 here, then we go to 15 and now, if we divide once more we 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 otherwise. So, we went on 1, 2, 3, 4, 5 times and at 6 time, you will go below 1 and therefore, we will see. So, this is the integer logarithm function, it just does this divide by n dividing n by k, until we reach one form go below.

(Refer Slide Time: 14:13)



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

So, if we give it 13276, it should give us 67231. So, how would be 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 it is remainder with respect to 10, we get 6. So, using div and mod, we can split this number in to the last digit and the rest. Now, using this style that we are use to in Haskell, we will do something inductive.

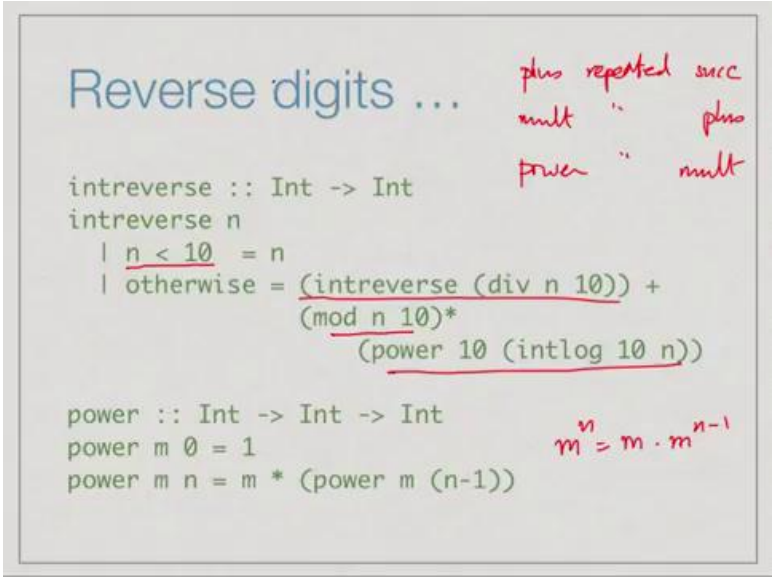
So, we will reverse the first part, this is the smaller number. So, you have the induction is on the length of the number, earlier are induction you stay on the argument itself, we do n in term n minus 1, here the induction on the length of the number. So, we will see that

for things like list and sequences, the length is something we can 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 we need to stick 6 in front of it. So, how do get 6 in front of it, we have to shift 6 to the left which means we have 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 with 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, if this as 1 less than number of digits, if we divide this by 10, 4 times, we get 6.7 something and if you divided by 5 times, we get the 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 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)

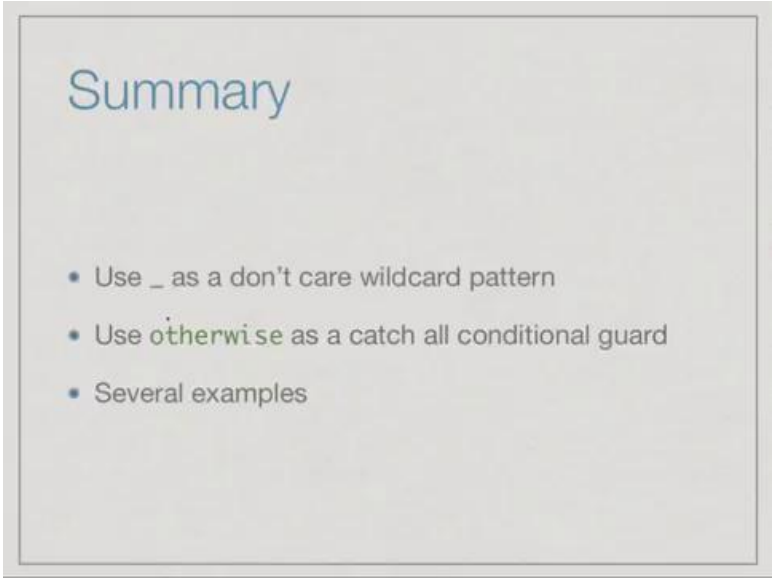


So, here is the function. So, it says that, I want to reverse in integer, if the number is single digit number, if it is less than 10 at the nothing. Otherwise, I recursively reverse the first k minus 1 digits, then I take the last digit and multiplied by suitable power. Suitable power is the integer logarithm to the base 10 of the original log and how do I

define power, when we saw in the first few examples that plus is repeated successor, mult is repeated plus.

So, power or exponentiation is repeated multiplication, where say 2 to the power 7 it means 2 times 2 times 2 times to 7 times. So, this exactly we wrote plus and mult, we can write a power function with says anything to the power 0 is 1 and if something to the power n is m times m to the power n minus 1. This is the n to the power m is equal to m times m to the power n minus 1. So, using this function and the Int log function that we wrote last time, we can compute the reverse any set of digits.

(Refer Slide Time: 18:00)



So, what we are 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 do not care pattern, which allows us to specify arguments, which are used in evaluating the function. So, it also makes simple gives as two option, the one is not use a variable where we do not need it, the second it makes it more transparent that a function is independent of the particular algorithm.

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 defiantly match this one. So, this ensures that, we do not have any pattern match failures.