

Introduction To Haskell Programming

Prof. S. P. Suresh

Chennai Mathematical Institute

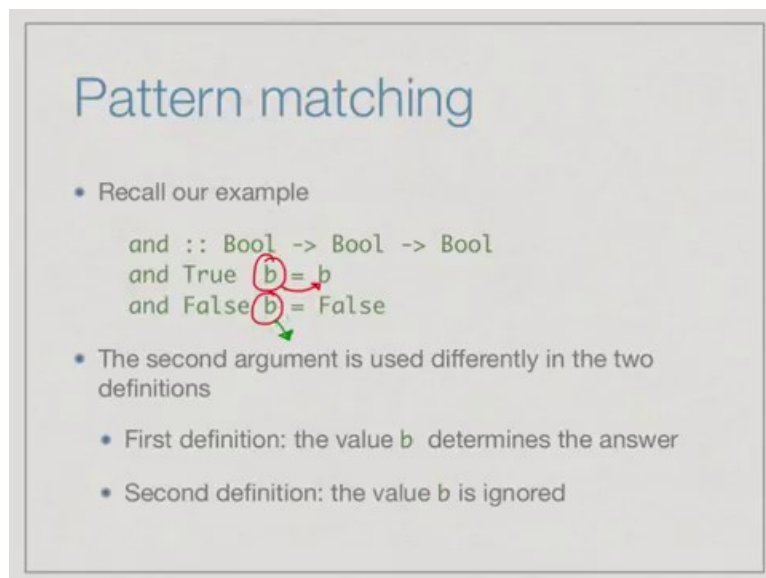
Module # 01

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)



The slide is titled "Pattern matching" in a large, light blue font. Below the title, there is a bulleted list. The first bullet point says "Recall our example". Below this, there is Haskell code: `and :: Bool -> Bool -> Bool`, `and True b = b`, and `and False b = False`. A red circle is drawn around the variable `b` in both `and True b = b` and `and False b = False`. A red arrow points from the `b` in the first line to the `b` in the second line. Below the code, there are two more bullet points. The first says "The second argument is used differently in the two definitions". The second bullet point has two sub-bullets: "First definition: the value `b` determines the answer" and "Second definition: the value `b` is ignored".

- 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

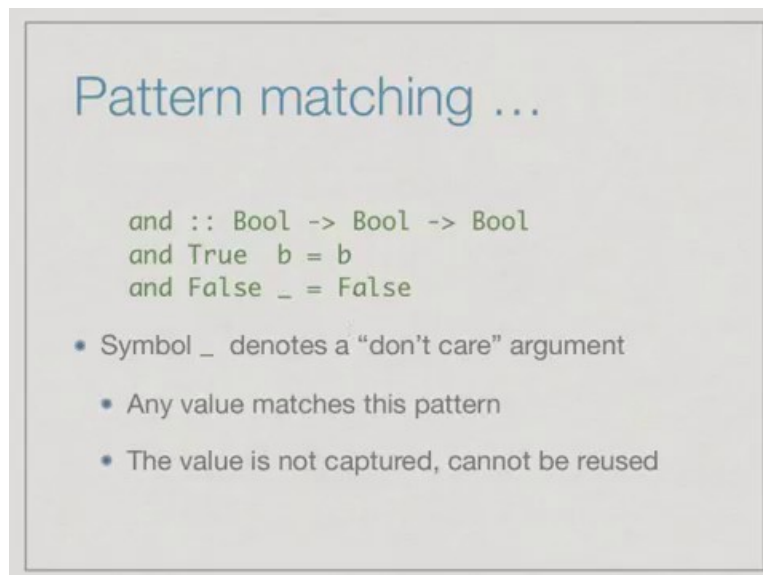
Before we do that, we will revisit a couple of the notions that we have seen for writing Haskell functions and explain a few concepts that we did not do when we introduced them. So, we begin with pattern matching. Remember that in Haskell functions, we can use patterns to illustrate cases. So, here is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

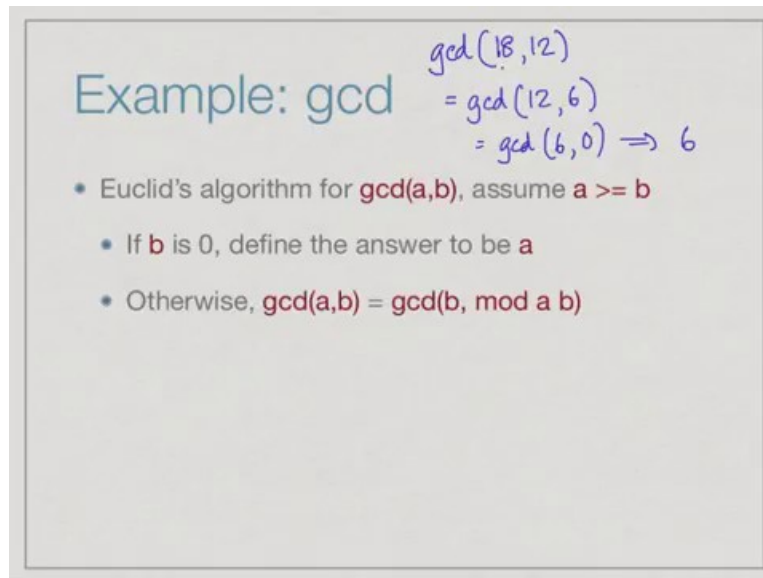
```
factorial :: Int -> Int
factorial n
  | n == 0 = 1
  | n > 0  = n * (factorial (n-1))
  | otherwise = factorial (-n)
  n < 0
```

- “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, $\text{gcd}(a, b) = \text{gcd}(b, \text{mod } a \text{ } b)$

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 remainder 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, $\text{gcd}(a,b) = \text{gcd}(b, \text{mod } a \text{ } b)$

```
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 $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$. 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 gcd of 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 $n-1, n-2, \dots$. 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 $n-1, n-2$.

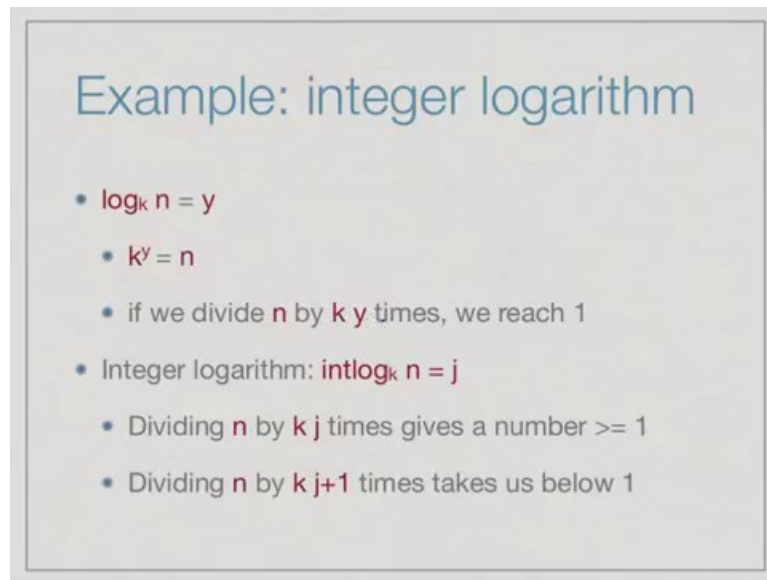
So, we go back we want the largest one. So, we go back in reverse order from n , we start with $n-1$, because we do not want n itself and we check for each of these. So, it is just a normal programming, this would just be a loop, we are just trying for each i from $n-1, n-2$, down to 1, if i divides 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 ≥ 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 ...

$60 \rightarrow 30 \rightarrow 15 \rightarrow 7 \rightarrow 3 \rightarrow 1$

$1 \leftarrow 3 \leftarrow 7$

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

```
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, 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 $\text{intlog } k \ 1$, because of the way we write it as $\text{intlog base number}$, 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 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 ...

plus repeated succ
mult " plus
power " mult

```

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))

```

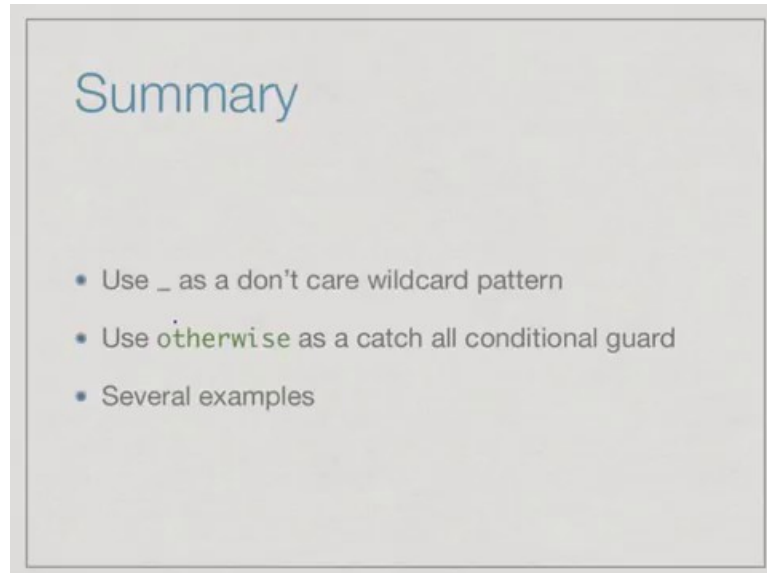
$m^n = m \cdot m^{n-1}$

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 $2 \cdot 2 \cdot 2$ 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 mn. This is equal to $m \cdot mn$.

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)



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.