Functional Programming in Haskell

Prof. Madhavan Mukund and S P Suresh

Chennai Mathematical Institute

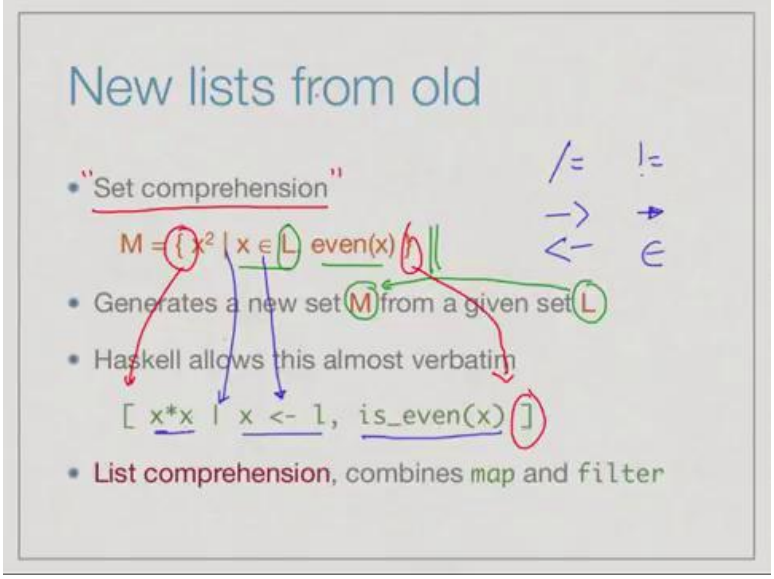
Module # 03

Lecture – 04

List comprehension

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

(Refer Slide Time: 00:37)

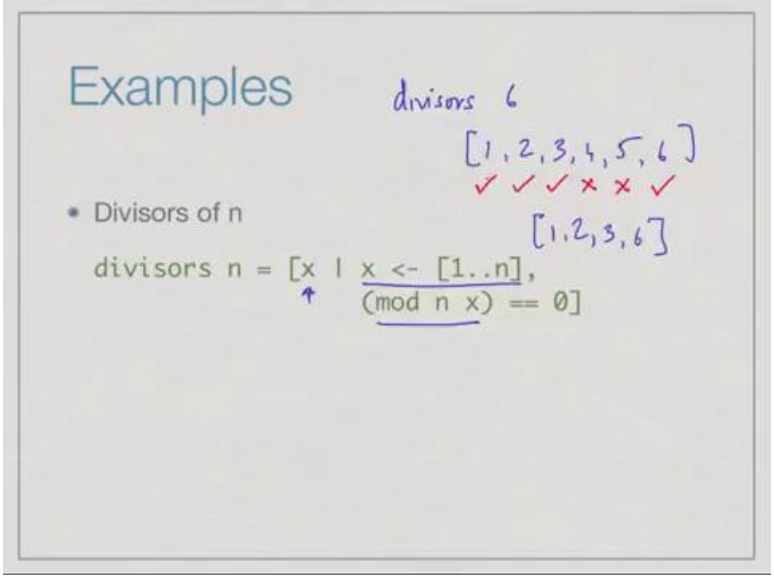


So, we start with basic set theoretic notation. So, we have often seeing this kind of notation to describe a set. So, here it says take given a set capital L, take all elements from the L which satisfy some condition, in this case, they are even and square these elements. So, this is set of all X squared, such that X belongs to L and X is even. So, effectively this takes a given set L and produces a new set M. So, it transforms a given set to a new set and this is in set theory, this notation is called set comprehension.

So, this is a technical term, so defining sets in this way is define to be using a techniques called set comprehension and so this is just terminology from set theory. So, analogues to this; Haskell allows us to define this using list comprehension. So, this notation with very similar to that, instead of a curly bracket we have a list square bracket and then, we have of course, the same vertical bar, which is the symbol from entire type on the keyboard and now, for the element of, we use thing which resembles elements of.

So, remember we said that Haskell tries to use notation, which looks like what we use in real life. So, we use slash equal to or not equal to and now, we have already seen that, we use this minus and greater than, simulate the arrow for function. And now, we have less than and minus, which is supposed to represent the element. So, this says, for a take the elements in L, check if they are even. So, this is a filter and then, if they are even apply X square, so this is a map.

(Refer Slide Time: 02:30)

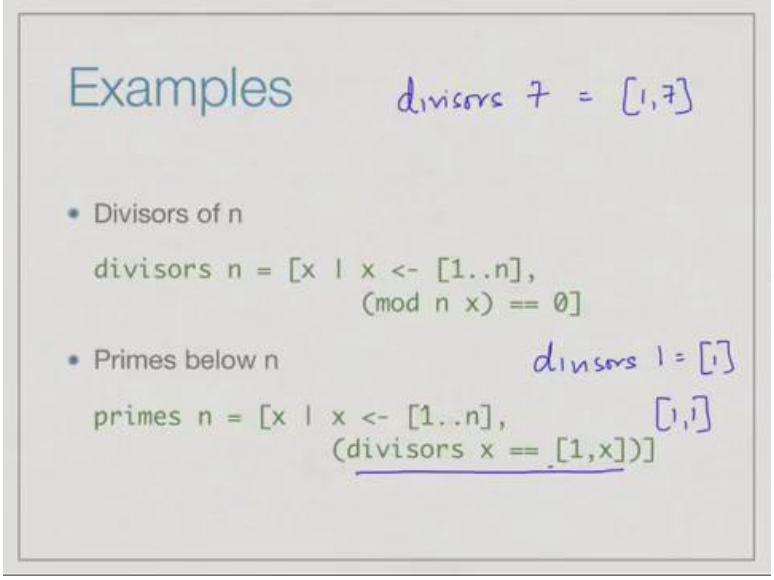


So, here is an example. So, supposing we want to find the divisor's of n, the divisor's of n or those numbers that divide n without leaving any reminder. So, the first of all, the first thing we notice the divisor's of n must be in the range 1 to n. So, we take all elements in the range 1 to n and if the remainder, when n is divided by that number is 0, so X divisor in exactly, then we listed. So, this list all the divisor's of n.

So, if I take for example, divisor's of say 6, when I will first generate 1, 2, 3, 4, 5, 6 are the possible candidates and then, applying this condition, it will say that 6 divided by 1 is okay, 6 divided by 2 is okay, 6 divided by 3 is okay, because 6 mod 3 is 0, 6 divided by 4

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

(Refer Slide Time: 03:30)

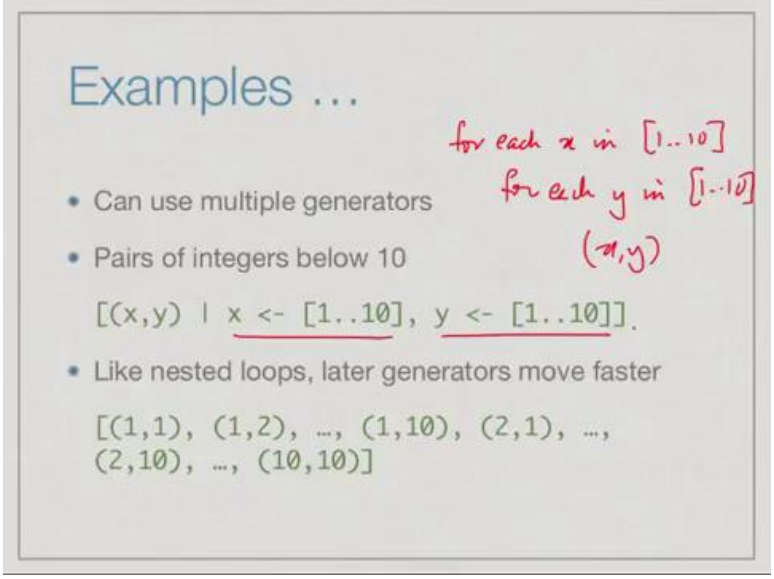


Now, using this function divisor's, we can classify whether a numbers prime or not. So, primes below a given number n. So, prime is a number, which is divisible by only itself and 1. So, if the number is prime, like a divisor's of 7, you will get a list consisting of 1 and 7 and nothing else, because that is a definition of prime; that would be 2 integers divide the number, the number itself and 1, it has no other non-trivial factors.

So, in order to check whether something is prime or not, we just check whether divisor's of X is exactly the list 1 comma X and then, we take all the numbers from 1 to n. Such that, this is true and we list them out, we get the price below the number in the range 1 to n, 2 to n. Because, notice that if I say 1, so 1 is not a prime, divisor's of 1 or the way we are defined it is going to be the list consisting of 1.

So, it will failed this test, because it is not 1 comma 1, which is what this test requires, it requires divisor of X to be list 1 comma X. So, in order for 1 to be a prime would have to produce a list in divisor from 1 comma 1 that are functions divisor's will not do that, it will only check 1 number and produce that number. So, divisor's of 1, we just fill the list containing the single element 1. So, we are okay that 1 does not comma to the n, 2 on the other hand will get this divisor's 1 comma 2 and so on.

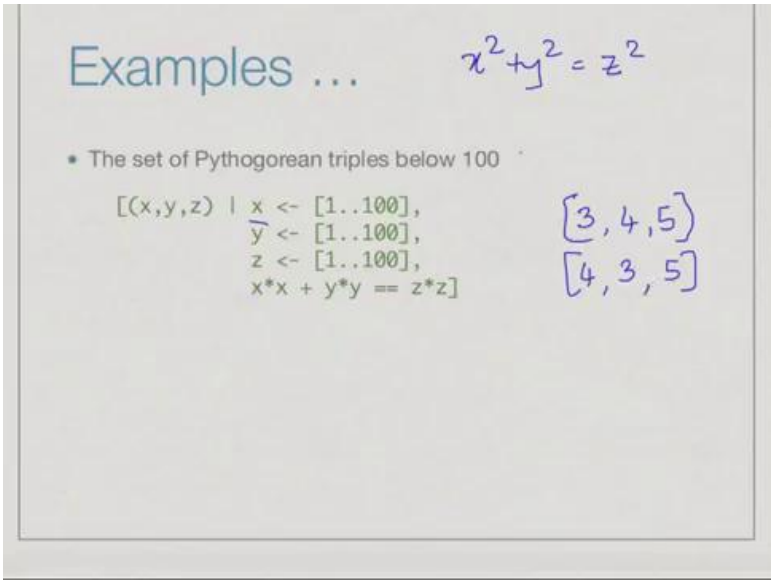
(Refer Slide Time: 04:58)



So, for we have seen examples where you use only one generator would we can use more than one generator. So, what this says is, let X run through the list 1 to 10, let Y run through the list 1 to 10 and construct the list of all pairs X comma Y that are generated by combining these values of X and Y. So, this is say something like for each X in the 1 to 10, for each Y in on 10 produce X comma Y.

So, therefore, if I fix a value X equal to 1, then it will generate every possible Y, then for X equal to 2 we generate every possible Y and so on. So, the later generators move fast. So, I have 1, 1; 1, 2; 1, 3; 1, 4 and 1, 10. So, X is fixed is 1, Y will 1 to 10, then X to move to 2, again Y will move comma 10 and finally, we will get of course, 10 comma 10. So, when we have multiple generators, they are generators to the right or executed are there run through for each element of the generators to the left.

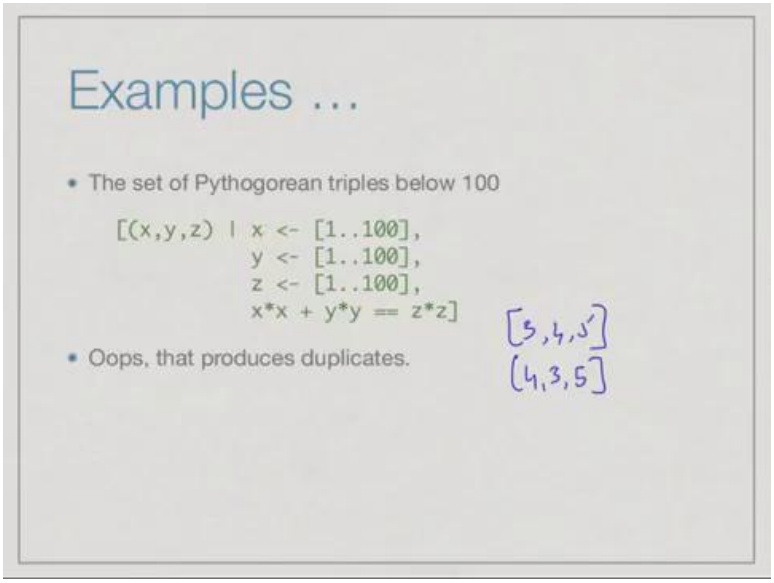
(Refer Slide Time: 06:10)



So, here for example, this is a way to generate all pairs. So, remember pair for to this is formula. So, if you want integers with satisfy the function X, the relation X squared plus Y squared equal to Z squared up to a certain upper limit. Then, we can run X from say 1 to 100, Y from 1 to 100, Z from 1 to 100 and check that X time X plus Y times Y is equal to Z times Z extract out this.

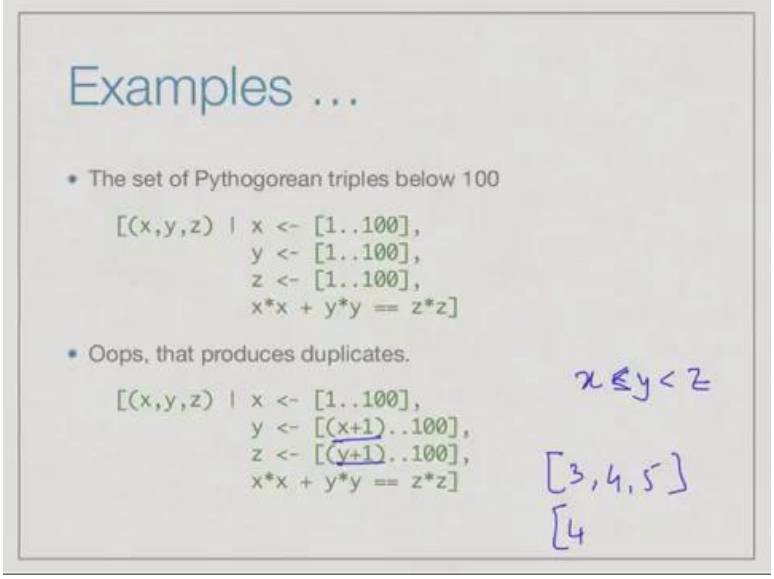
So, this is using are list comprehension and multiple generators, we can generate all the Pythagorean and so on. If you see this notice that 3, 4, 5 is a Pythagorean triple. So, it will be generated by this from we run, but then later on when X becomes 4, then we will also generate 4, 3, 5. So, this actually generates duplicates.

(Refer Slide Time: 07:06)



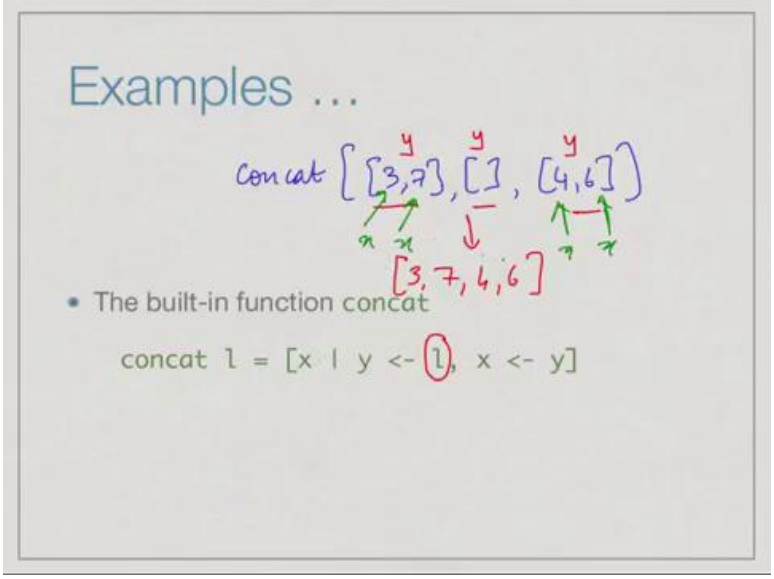
So, we get 3, 4, 5 and we also get 4, 3, 5. So, supposing we do not want to less these separately, because this are essential the same triple is just in different order, then we can be a little more careful in our generator. So, generators can actually just as we said that for every X, Y is generated for every Y is Z is generated, a values said we generated for Y can depend on the current value for X.

(Refer Slide Time: 07:33)



What we can say is that, we always want to generate these in the order X less than equal to strictly less than Y, strictly less than Z. So, X on to 1 to 100, but for each value of X, I only check Y is from X plus 1 and only check was Z's and Y plus 1. So, this gives us set of triples without any duplicates, because now I will get 3, 4, 5, but once I said X to 4, I cannot generate 3, because Y will start from 5.

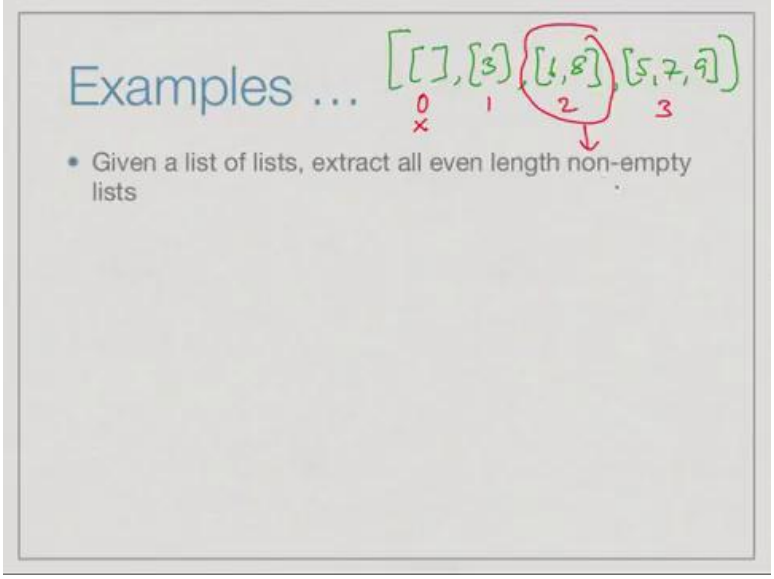
(Refer Slide Time: 08:09)



So, here is another function using rewritten using list comprehension, remember the function concat, it dissolves brackets, if has a concat of say 3, 7 into list and 4, 6, then what tells us produce from this, list 3, 7 4 6. So, we are taking the list of list, it producing single list by just collapsing all of them into single list, it is empty, just this is like vector. So, this as okay, concat of l, for you take each element in l, so I take this, then I take this, then I take this, so Y will we first this and Y will be first this, Y will be first this.

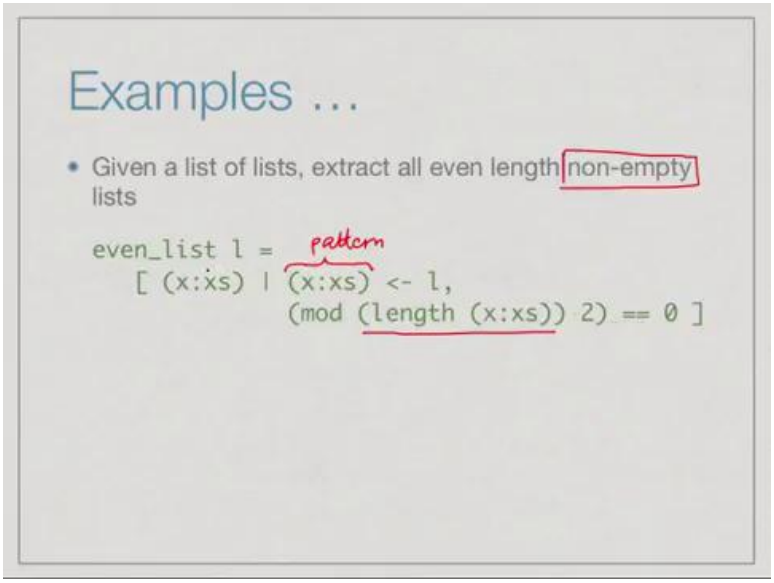
And now, I take each X and Y, so X now is this, then this, then this, then this, these are the values takes X, X and output all of this as 1's. So, it takes every Y in the inner list takes every element of that list and extracts does not algorithm. So, therefore, this disolves one level of brackets and behaves exactly like that kind.

(Refer Slide Time: 09:13)



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

(Refer Slide Time: 09:54)

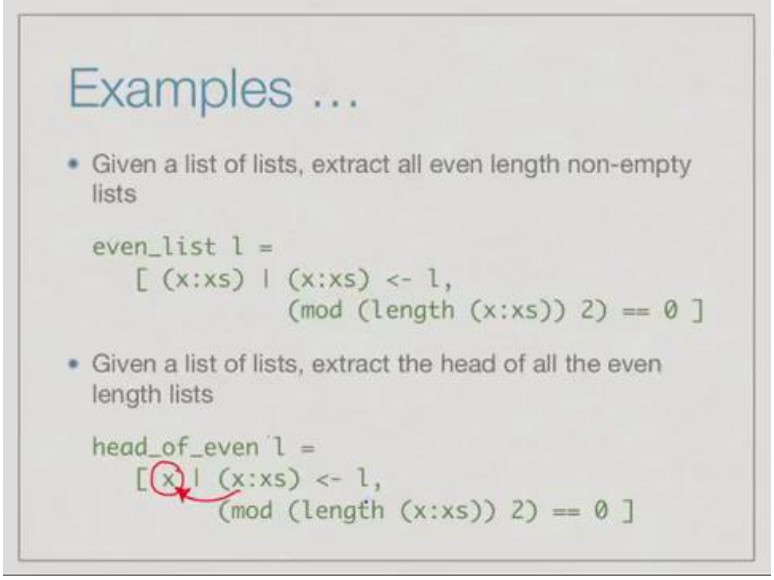


So, the new thing here is that; the filter that we want to take the list of list check each element in the list and extracted provided it is length is even. Remember, that length in anything be even is just checking, the remainder with respect to 2, we just want to take the length of the given list divided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

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

So, this tells do not take every element of l, only take those elements are l, which are of the form X colon X S. So, in other words in earlier example if we saw empty list does a given element in the list of list, then this pattern no match. So, it is just keep it. So, for every non-empty list in l, check, if is like this even and if so, then put out that X S.

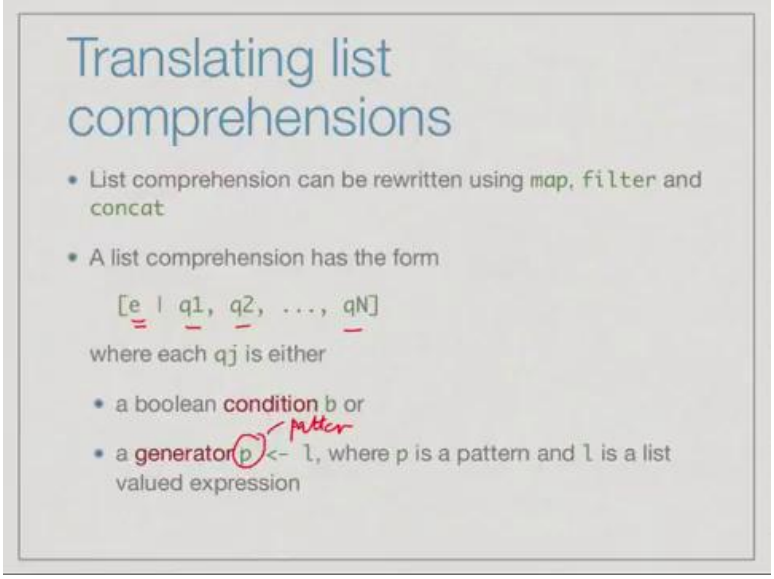
(Refer Slide Time: 10:58)



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

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

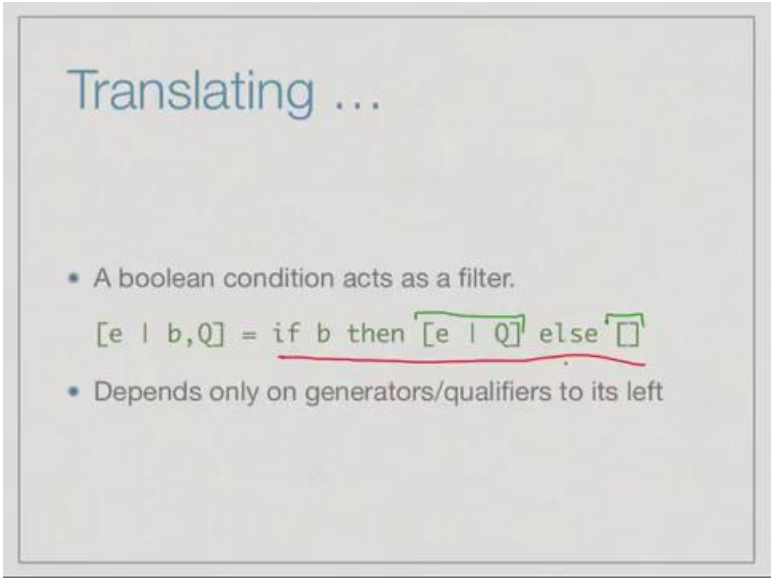
(Refer Slide Time: 11:55)



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

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

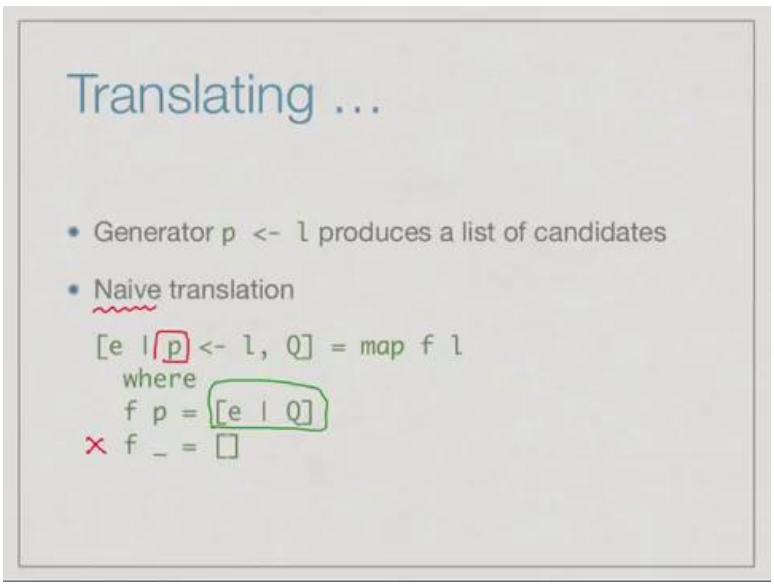
(Refer Slide Time: 12:47)



So, a boolean condition access a filter. So, if I have expression form e, such that b forward by Q, then this is an expression which is allowed in Haskell, if we are not seen before. So, we can write this is a expression Haskell, it will using the conventional, if there hence. So, it says if b is 2 and this is the output, otherwise this is the output. So, in other words for whatever list I am generating, if b wholes then I continue to apply the remaining things, otherwise I just keep it.

So, for each element in simplicity this apply to things have been generating in the left. So, if the element from the left satisfy the condition, then I continue to process using what remains, otherwise I got.

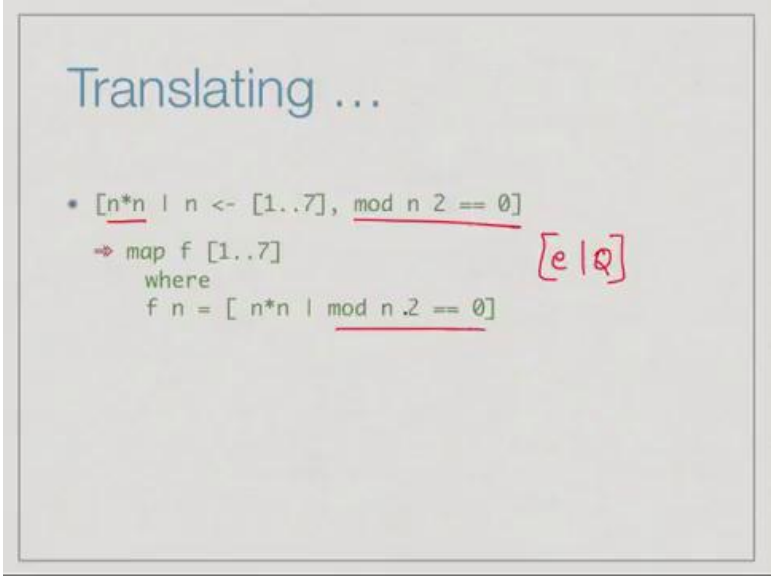
(Refer Slide Time: 13:33)



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

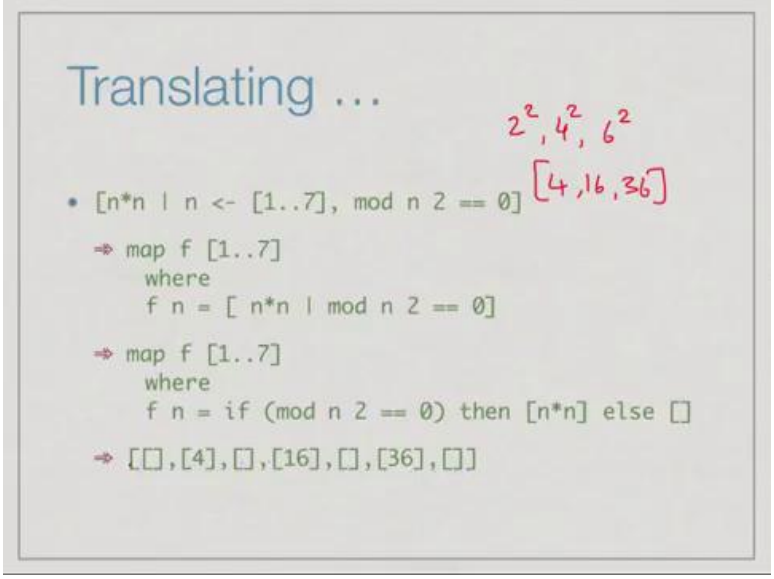
So, this is taking care of the fact this is the pattern. So, the pattern itself does not matter, if this not a pattern I would have this case, if a elements I will just have f of X this e, such that Q, but since I have a pattern, it says if the pattern matches then his nothing the pattern does not skip it. Now, this is a naïve translation and we will see why. So, for we have not use concat, we have use map and we are used kind of filtering efforts.

(Refer Slide Time: 14:30)



So, let us look at an example to see, why this goes off, so let us look at this simple example. So, supposing we want to square all the even numbers between 1 and 7, so we generate all the numbers from 1 to 7, so I keep there even and then square now the first thing that we have is a generated. So, we have to applied the map thing. So, it says map f to 1 to 7, where f of n is what remains; that is this and this. So, this is that e, Q, e such that and now, we are to inductively recover this. So, this is a property. So, this will be replace by if then else.

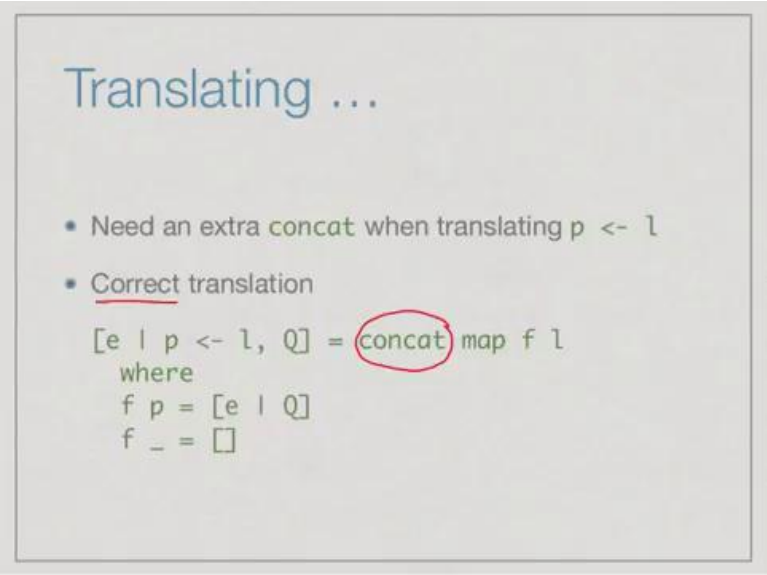
(Refer Slide Time: 15:10)



So, it says map f to 1 from to list 1 to 7, where f of n is, if n is even, then n square the list containing n square else their happens and now if I apply this to each element, then for 1 we get the empty list, for 2 we get the list containing 4 and so on. So, you will notice that, what you would expect from this list is, we will expect 2 squared, 4 squared and 6 squared.

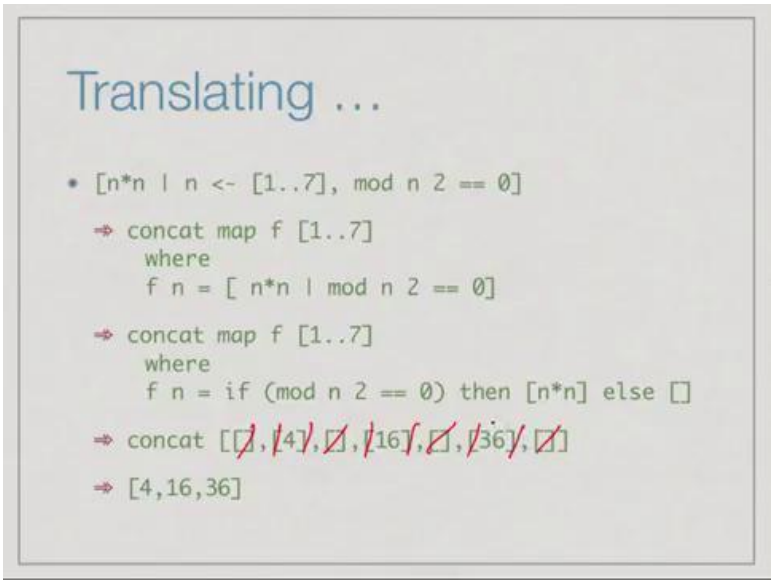
So, in other words we will expect the list 4, 16, 36 to be the output of this list comprehension, but we actually get this complicated expression with the lot of curious brackets. And this is precisely why we need the concat, we need to X eliminate these brackets.

(Refer Slide Time: 15:52)



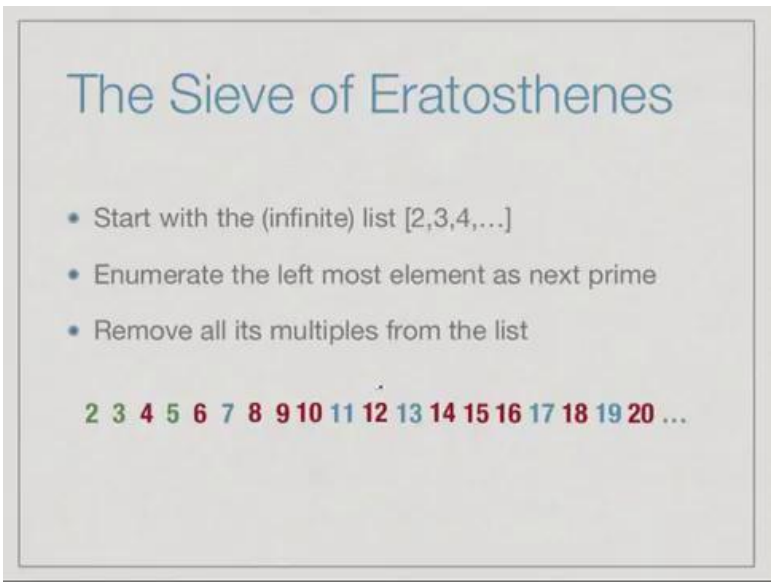
So, the correct translation of in the generator is to insert a concat. So, we do not just map f to l, we take the resulting output and we dissolve one level of brackets. So, result of moving a generator from this expression is to concat for result of mapping f to the list.

(Refer Slide Time: 16:14)



And now, if we do this everything works out find for that example and you can check it works in general. So, we take the same list n square, such that, n is 1 to 7 and n is 0. So, after one expansion we have this map, but now we have this concat of f. So, after second expression, we have this if, we still have this concat in front of it. So, now, we get concat of this earlier expression that we had and now the concat dissolves this brackets and we moves the empty list. So, I just get 4, 16 and 36, which is the expected output.

(Refer Slide Time: 16:49)



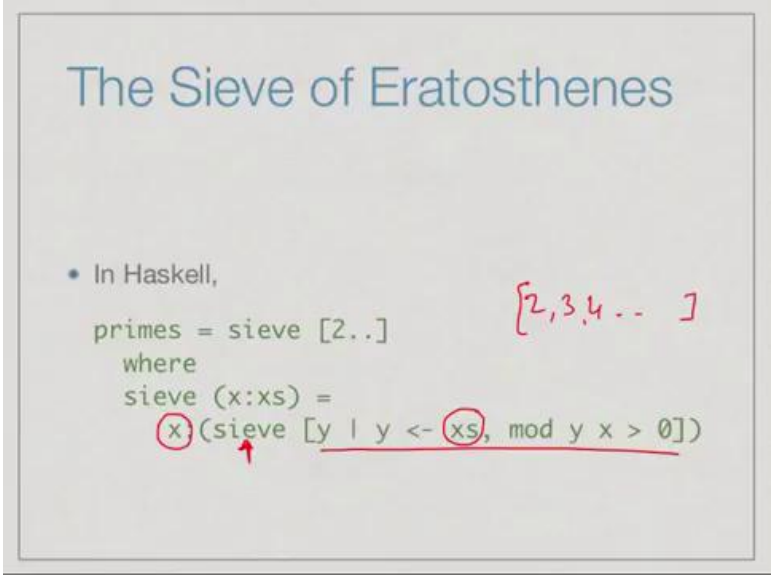
So, let us now look at the example that we had in the introduction to introductory video to the course. So, this is the sieve of Eratosthenes. So, what is the sieve of Eratosthenes, it generates all the possible primers. So, this strategy is a following start with the list of all numbers beginning with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

So, let us just see how it works. So, supposing we start with infinite list infinite list if you with in the finite prefix. So, we have up to 20. So, now, the first number in this list is the first prime namely 2. So, we mark 2 is a prime and now, we must remove all it is multiple symbol. So, the first multiple of 2 is 4, the next is 6, 8 and so on. So, we go through this infinite list marking of all the infinite list. So, this keeps as a resulting list in which the first number that is left is 3. So, 3 is now at time.

So, we knock of it is multiples. So, 6 is already knock off, if you knock of 9, 12 already knock off if you knock of 15, 18 is already knock off, but we important 21 and so on. After knocking of multiples of 3, we also got into 9 and 15 and of course, a lot of numbers to write which we do not see. So, now, the next prime is a 5 and then, we will knock of 10, 15, 25, 20, 25 and so on. So, this is the process by which we generate the primes.

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

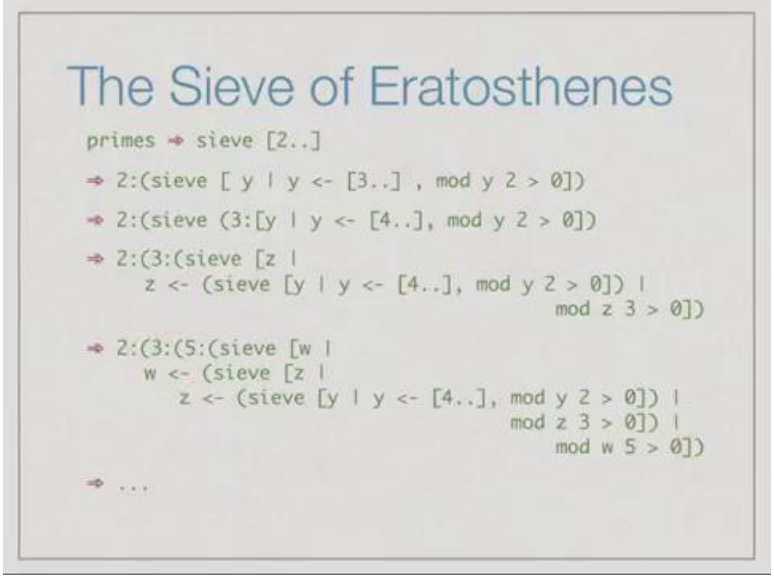
(Refer Slide Time: 18:48)



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

So, if I take all that this, a tail of the list for every Y, I keep it only if does not get divided even only for X and then, I recursively apply sieve to that. So, this is such singly describing the sieve Eratosthenes, take the list to onwards and apply sieve to it sieve extracts the first element, removes all this multiple from the tail and apply sieve recursively to that take. So, if we look at the way that this evaluates could actually makes sense.

(Refer Slide Time: 19:47)



So, we say that set of primes is a result of prime sieve to 2 onwards. So, sieve 2 onwards is take out the first element and apply sieve to 3 onwards, such that elements 1 divide 2. Now, this in turn will say expect the first element of this, so first element of this at this by 3. So, it will say applies if to 3 and the rest, s, this is just saying that, if I expand this inner list of this comprehension, when this produces 3 followed by 4 onwards in the same property.

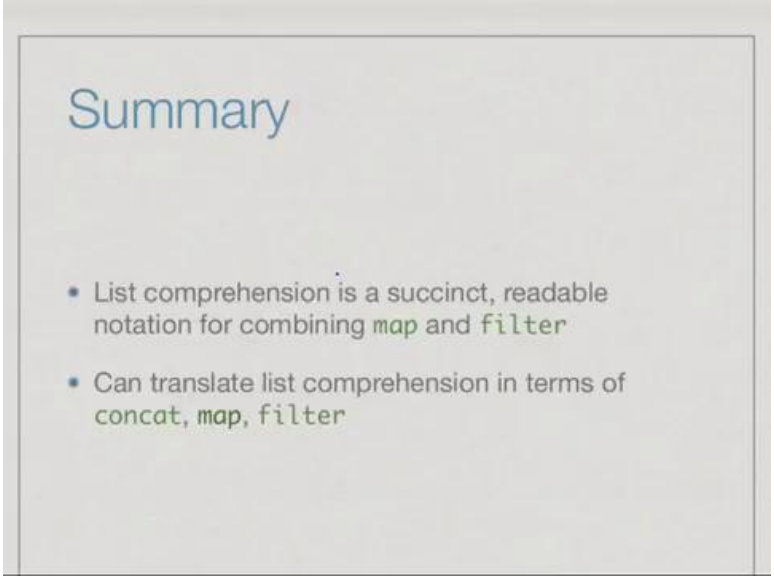
Having done this, I got the first element, so simple extract it out and then, it will say apply sieve 2, the result of this original list, which we already, this is the list that is currently running. So, we take every element in that list and apply another condition. So, one second, if we do this two nested list comprehension is a first element that conserve 5. And so now, after we extract the 5, then it will say take the inner list that we are already working with two list comprehensions and apply a third one.

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

Of course, lazy evaluation crucial other, we cannot go with infinitely set all and using lazy evaluation and using this comprehension, we can write a very succinct 2 or 3 line

implementation of a very basic algorithm. In such a way that, it is immediately obvious what is going on and it does the expected.

(Refer Slide Time: 21:57)



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