

Introduction To Haskell Programming

Prof. S P Suresh

Chennai Mathematical Institute

Module # 03

Lecture – 04

List comprehension

We have seen two important higher order functions on lists, 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, a 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 than just writing map and filter.

(Refer Slide Time: 00:37)

The slide is titled "New lists from old". It contains the following content:

- "Set comprehension"
- $M = \{x^2 \mid x \in L, \text{even}(x)\}$
- Generates a new set M from a given set L
- Haskell allows this almost verbatim
- $[x*x \mid x \leftarrow l, \text{is_even}(x)]$
- List comprehension, combines map and filter

Handwritten annotations include:

- Arrows mapping $x \in L$ to $x \leftarrow l$ and $\text{even}(x)$ to $\text{is_even}(x)$.
- A red arrow from x^2 to $x*x$.
- A legend on the right: $/=$ and $!=$ for \rightarrow and \in respectively.

So, we start with basic set theoretic notation. So, we have often seen this kind of notation to describe a set. So, here it says, given a set 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^2 , such that x belongs to L and x is even, written as $\{x^2 \mid x \in L, \text{even}(x)\}$. 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 in set theory, this notation is called set comprehension.

So, this is a technical term, defining sets in this way is defined to be using a technique called set comprehension and so this is just terminology from set theory. So, analogous to this, Haskell allows us to define lists using list comprehension. So, this notation is 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 you can type from the keyboard and now, for the element of, we use \leftarrow 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 \neq slash equal to for not equal to and now, we have already seen that, we use this \rightarrow to simulate the arrow for function. And now, we have \leftarrow , which is supposed to represent the element of operator. So, $[x*x \mid x \leftarrow L, \text{is_even}(x)]$ says, take the elements in L , check if they are even. So, $x \leftarrow L, \text{is_even}(x)$ is a filter and then, if they are even apply x^2 , so this is a map.

(Refer Slide Time: 02:30)

Examples

- Divisors of n

```
divisors n = [x | x <- [1..n],
                  (mod n x) == 0]
```

Handwritten example for $n=6$:

divisors 6

$[1, 2, 3, 4, 5, 6]$

✓ ✓ ✓ ✗ ✗ ✓

$[1, 2, 3, 6]$

So, here is an example. So, supposing we want to find the divisors of n , the divisors of n are those numbers that divide n without leaving any remainder. So, first of all, the first thing to note is the divisors 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 x is 0, so x divides n exactly, then we list it. So, this lists all the divisors of n .

So, if I take for example, divisors of say 6, then 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 $6 \bmod 3$ 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)

The slide is titled "Examples" in blue. It contains two bullet points with R code and handwritten annotations in blue.

- Divisors of n
`divisors n = [x | x <- [1..n],
 (mod n x) == 0]`
Handwritten: `divisors 7 = [1,7]`
- Primes below n
`primes n = [x | x <- [1..n],

(divisors x == [1,x])

]`
Handwritten: `divisors 1 = [1]` and `[1,1]` next to the `[1,x]` in the code.

Now, using this function `divisors`, we can classify whether a number is prime or not. So, primes below a given number `n`. So, a prime is a number, which is divisible by only itself and 1. So, if the number is prime, like a `divisors` of 7, you will get a list consisting of 1 and 7 and nothing else, because that is a definition of prime; that only two 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 `divisors` of `x` is exactly the list `[1,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 primes 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, `divisors` of 1 for the way we have defined it is going to be the list consisting of 1.

So, it will fail this test, because it is not `[1,1]`, which is what this test requires, it requires `divisors` of `x` to be list `[1,x]`. So, in order for 1 to be a prime it would have to produce a list of `divisors` of the form `[1,1]` but our function `divisors` will not do that, it will only check 1 number and produce that number. So, `divisors` of 1, we just get the list containing the single element 1. So, we are okay that 1 does not come out as prime, 2 on the other hand will give as `divisors` `[1,2]` and so on.

(Refer Slide Time: 04:58)

Examples ...

- Can use multiple generators
- Pairs of integers below 10

$[(x,y) \mid x \leftarrow [1..10], y \leftarrow [1..10]]$.

- Like nested loops, later generators move faster

$(1,1), (1,2), \dots, (1,10), (2,1), \dots, (2,10), \dots, (10,10)$

*for each x in [1..10]
for each y in [1..10]
(x,y)*

So far we have seen examples where you use only one generator but we can use more than one generator. So, what $[(x,y) \mid x \leftarrow [1..10], y \leftarrow [1..10]]$ 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,y) that are generated by combining these values of x and y. So, this is saying something like for each x in the range 1 to 10, for each y in 1 to 10 produce (x,y).

So, therefore, if I fix a value $x=1$, then it will generate every possible y, then for $x=2$ 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) .. (1,10). So, x is fixed as 1, y will run through 1 to 10, then x will move to 2, again y will move from 1 to 10 and finally, we will get of course, (10,10). So, when we have multiple generators, the generators to the right are executed or they run through for each element of the generators to the left.

(Refer Slide Time: 06:10)

Examples ...

$x^2 + y^2 = z^2$

- The set of Pythagorean triples below 100

```
[(x,y,z) | x <- [1..100],  
           y <- [1..100],  
           z <- [1..100],  
           x*x + y*y == z*z]
```

$[3, 4, 5]$
 $[4, 3, 5]$

So, here for example, this is a way to generate all pairs. So, remember Pythagoras's formula. So, if you want integers which satisfy the relation $x^2 + y^2 = z^2$ 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 $x*x + y*y == z*z$ and extract all of these.

So, this is using our list comprehension and multiple generators, we can generate all the Pythagorean integers. But If you see this, notice that $[3, 4, 5]$ is a Pythagorean triple. So, it will be generated by this when 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)

Examples ...

- The set of Pythagorean triples below 100

```
[(x,y,z) | x <- [1..100],  
           y <- [1..100],  
           z <- [1..100],  
           x*x + y*y == z*z]
```

- Oops, that produces duplicates.

$[3, 4, 5]$
 $[4, 3, 5]$

So, we get [3, 4, 5] and we also get [4, 3, 5]. So, supposing we do not want to list these separately, because these are essentially the same triple 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, z is generated, the values that we generate for y can depend on the current value for x.

(Refer Slide Time: 07:33)

Examples ...

- The set of Pythagorean triples below 100


```
[(x,y,z) | x <- [1..100],
            y <- [1..100],
            z <- [1..100],
            x*x + y*y == z*z]
```
- Oops, that produces duplicates.


```
[(x,y,z) | x <- [1..100],
            y <- [(x+1)..100],
            z <- [(y+1)..100],
            x*x + y*y == z*z]
```

$x \leq y < z$

[3, 4, 5]
[4

What we can say is that, we always want to generate these in the order x strictly less than y strictly less than z, $x < y < z$. So, x runs from 1 to 100, but for each value of x, I only check y is from x+1 onwards and I only check Z's from y+1 onwards. So, this gives us a set of triples without any duplicates, because now I will get [3, 4, 5], but once I set x to 4, I cannot generate 3, because y will start from 5.

(Refer Slide Time: 08:09)

Examples ...

$\text{concat}([[3,7], [], [4,6]])$

$[3, 7, 4, 6]$

- The built-in function `concat`

```
concat l = [x | y <- l, x <- y]
```

So, here is another function rewritten using list comprehension, remember the function `concat`, it dissolves brackets, if I said `concat` of say `[3 7]`, `[]` and `[4 6]`, then what this does is it produces from this, a list `[3, 7, 4, 6]`. So, we are taking a list of lists, and producing a single list by just collapsing all of them into a single list, and if is empty, it is just left out. So, this says 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 be first this and `y` will be this, `y` will be this.

And now, I take each `x` and `y`, so `x` now is this, then this, then this, then this, these are the values that `x` takes and output all of these as `1`'s. So, it takes every `y` in the inner list, takes every element of that list and extracts it as in algorithm. So, therefore, this dissolves one level of brackets and behaves exactly like `concat`.

(Refer Slide Time: 09:13)

Examples ... $[[], [3], [1, 8], [5, 7, 9]]$

0 1 2 3
x

- Given a list of lists, extract all even length non-empty lists

So, let us just look at a couple of more, slightly more exotic examples. So, supposing we want to take a list of lists and extract all the even length non-empty lists. So, if I take something like this, so this is the list of lists. Now, $[]$ has length 0, $[3]$ has length 1, $[6, 8]$ has length 2, $[5, 7, 9]$ has length 3, but I do not want this, I only want to extract 6, 8. So, I want to extract all the even length non-empty lists in a given list.

(Refer Slide Time: 09:54)

Examples ...

- Given a list of lists, extract all even length non-empty lists

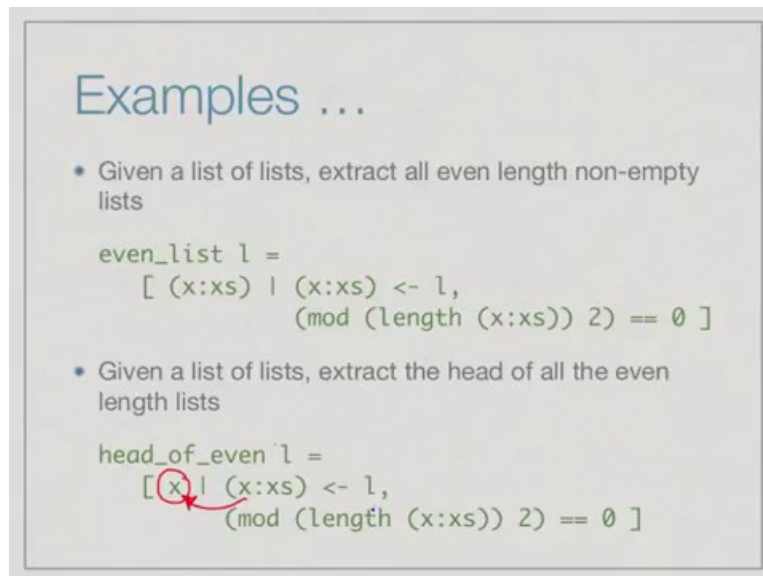
```
even_list l = pattern  
  [ (x:xs) | (x:xs) <- l,  
             (mod (length (x:xs)) 2) == 0 ]
```

So, the new thing here is that we want the filter to take the list of lists, check each element in the list and extract it provided its length is even. Remember, that length and 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 feature here is we

want non-empty list, the way we can do non-empty list in one shot 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 of `l`, which are of the form `x:xs`. So, in other words in our earlier example if we saw empty list as a given element in the list of lists, then this pattern wont match. So, we just skip it. So, for every non-empty list in `l`, check, if length is even and if so, then put out that list..

(Refer Slide Time: 10:58)



Examples ...

- Given a list of lists, extract all even length non-empty lists

```
even_list l =  
  [ (x:xs) | (x:xs) <- l,  
            (mod (length (x:xs)) 2) == 0 ]
```

- Given a list of lists, extract the head of all the even length lists

```
head_of_even l =  
  [ x | (x:xs) <- l,  
        (mod (length (x:xs)) 2) == 0 ]
```

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

But, now we don't take entire list, we use this pattern to extract only the head of it. So, the message from this example is that, when we write a generator it is not just a simple variable, we can actually use a pattern and use that pattern to generate elements of `l`.

(Refer Slide Time: 11:55)

Translating list comprehensions

- List comprehension can be rewritten using `map`, `filter` and `concat`
- A list comprehension has the form
$$[e \mid q_1, q_2, \dots, q_N]$$
where each q_j is either
 - a boolean **condition** b or
 - a **generator** $p \leftarrow l$, where p is a pattern and l is a list valued expression

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 in a 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 a version of filtering.

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

(Refer Slide Time: 12:47)

Translating ...

- A boolean condition acts as a filter.
$$[e \mid b, Q] = \text{if } b \text{ then } [e \mid Q] \text{ else } []$$
- Depends only on generators/qualifiers to its left

So, a boolean condition acts as a filter. So, if I have an expression form e , such that b followed by Q , then this is an expression which is allowed in Haskell, if you have not seen before. So, we can write.. this is an expression in Haskell, written using the conventional, if then else. So, 'if b then $[e \mid Q]$ else $[\]$ ' says if b is True then this is the output, otherwise this is the output. So, in other words for whatever list I am generating, if b holds then I continue to apply the remaining things, otherwise I just keep it.

So, for each element implicitly this applies to things which have been generated in the left. So, if the element from the left satisfies the condition, then I continue to process it using what remains, otherwise I drop it.

(Refer Slide Time: 13:33)

Translating ...

- Generator $p \leftarrow l$ produces a list of candidates
- Naive translation

$$[e \mid p \leftarrow l, Q] = \text{map } f \ l$$
 where

$$f \ p = [e \mid Q]$$

$$\times f \ _ = []$$

What about generators? So, generators produce lists of candidates. So, if I have a generator then I need to take each element of p and such that Q apply this e to it. So, I have mapped each element of p with this function. So, this is $\text{map } f \ l$, where f of p is e , such that Q and if p is not matched, so this is because it is a 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 match, if this is not a pattern I would not have this case, if it just said elements I will just have f of x is, such that Q , but since I have a pattern, it says if the pattern matches then do something, if the pattern does not match, skip it. Now, this is a naïve translation and we will see why. So far we have not used concat, we have used map and we have used a kind of filtering efforts.

(Refer Slide Time: 14:30)

Translating ...

```
* [n*n | n <- [1..7], mod n 2 == 0]
⇒ map f [1..7]
   where
     f n = [ n*n | mod n 2 == 0]
```

$[e | Q]$

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, check if they are even and then square it. Now the first thing that we have is a generator. So, we have to apply the map thing. So, it says map f to [1..7], where f of n is what remains; that is this and this. So, this is $[e | Q]$ and now, we have to inductively recover this. So, this is a property. So, this will be replaced by if then else.

(Refer Slide Time: 15:10)

Translating ...

```
* [n*n | n <- [1..7], mod n 2 == 0]
⇒ map f [1..7]
   where
     f n = [ n*n | mod n 2 == 0]

⇒ map f [1..7]
   where
     f n = if (mod n 2 == 0) then [n*n] else []

⇒ [ [], [4], [], [16], [], [36], []]
```

$2^2, 4^2, 6^2$
 $[4, 16, 36]$

So, it says map f to the list 1 to 7, where f of n is, if n is even, then n^2 , the list containing n^2 else the empty list 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 22 , 42 , 62.

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 spurious brackets. And this is precisely why we need the concat, we need to eliminate these brackets.

(Refer Slide Time: 15:52)

Translating ...

- Need an extra `concat` when translating `p <- l`
- Correct translation

```
[e | p <- l, Q] = concat map f l
  where
    f p = [e | Q]
    f _ = []
```

So, the correct translation of 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, the result of removing a generator from this expression is to concat the result of mapping f to the list.

(Refer Slide Time: 16:14)

Translating ...

- `[n*n | n <- [1..7], mod n 2 == 0]`

```
⇒ concat map f [1..7]
  where
    f n = [ n*n | mod n 2 == 0]

⇒ concat map f [1..7]
  where
    f n = if (mod n 2 == 0) then [n*n] else []

⇒ concat [[], [4], [], [16], [], [36], []]
```

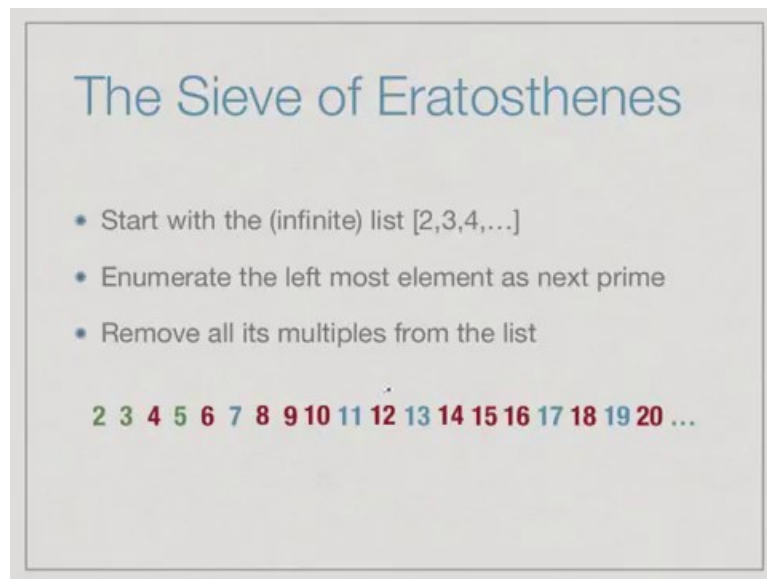
~~[1], [4], [9], [16], [25], [36], [49]~~

```
⇒ [4,16,36]
```

And now, if we do this everything works out fine for that example and you can check that it

works in general. So, we take the same list n squared, such that, n is from 1 to 7 and n is even. So, after one expansion we have this map, but now we have this concat in front of it 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 removes the empty list. So, I just get [4, 16, 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, the introductory video to the course. So, this is the sieve of Eratosthenes. So, what does the sieve of Eratosthenes do, it generates all the possible primes. So, the strategy is the following. you start with the list of all numbers beginning with 2, because a first prime is 2. So, the left most number in the list at any point is the next prime, once we enumerate a prime, we remove all its multiples from the list. because they are no longer candidates to be primes.

So, let us just see how it works. So, supposing we start with this infinite list of which we have written a 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 as a prime and now, we must remove all its multiples from the list. So, the first multiple of 2 is 4, the next is 6, 8 and so on. So, we go through this infinite list knocking off all those multiples. So, this gives us a resulting list in which the first number that is left is 3. So, 3 is now a prime.

So, we knock off its multiples. So, 6 is already knocked off, you knock off 9, 12 is already knockd off, you knock off 15, 18 is already knocked off, but we knock off 21 and so on. After knocking off multiples of 3, we have also got rid of 9 and 15 and of course, a lot of

numbers to the right which we do not see. So, now, the next prime is a 5 and then, we will knock off 10, 15, 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 we have to start with an infinite list and every time we pick out the first element, we have to remove all its multiples. So, that is again an infinite process because we have go through this infinite list and knock off all the multiples.

(Refer Slide Time: 18:48)

The Sieve of Eratosthenes

- In Haskell,

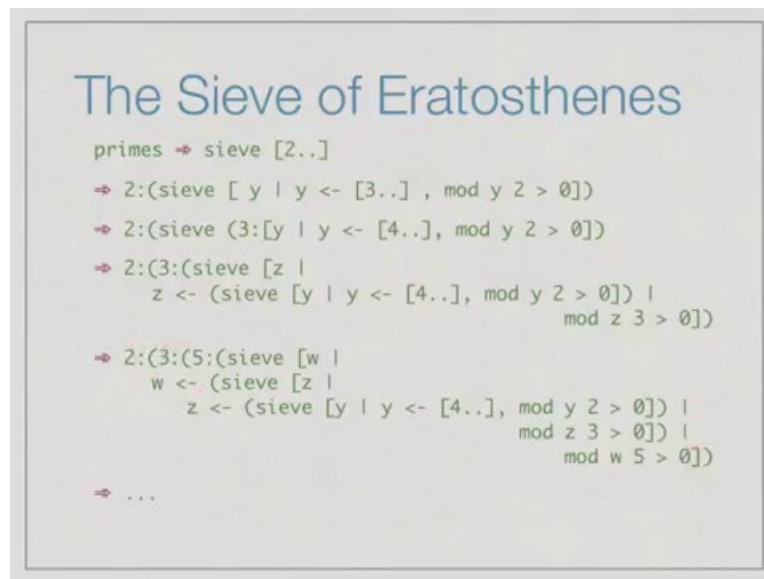
```
primes = sieve [2..]  
where  
  sieve (x:xs) =  
    x:sieve [y | y <- xs, mod y x > 0]
```

Handwritten notes: [2,3,4..] (circled x) (circled xs)

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 the lazy evaluation, this makes sense and result of applying sieve to a list is to extract the first element as a prime and then, remove all multiples of that element from the list.

So, if I take all this, a tail of the list for every y, I keep it only if does not get divided evenly by x and then, I recursively apply sieve to that. So, this is succinctly describing the sieve algorithm and says take the list 2 onwards and apply sieve to it where sieve extracts the first element, removes all this multiples from the tail and apply sieve recursively to that tail. So, if we look at the way that this evaluates it will actually make sense.

(Refer Slide Time: 19:47)



So, we say that set of primes is a result of applying the sieve to 2 onwards. So, sieve [2..] says take out the first element and apply sieve to [3..], such that elements don't divide by 2. Now, this in turn will say extract the first element of this, so first element of this happens to be 3. So, it will say apply sieve to 3 and the rest, so, this is just saying that, if I expand this inner list of this comprehension, then this produces 3 followed by [4..] in the same property.

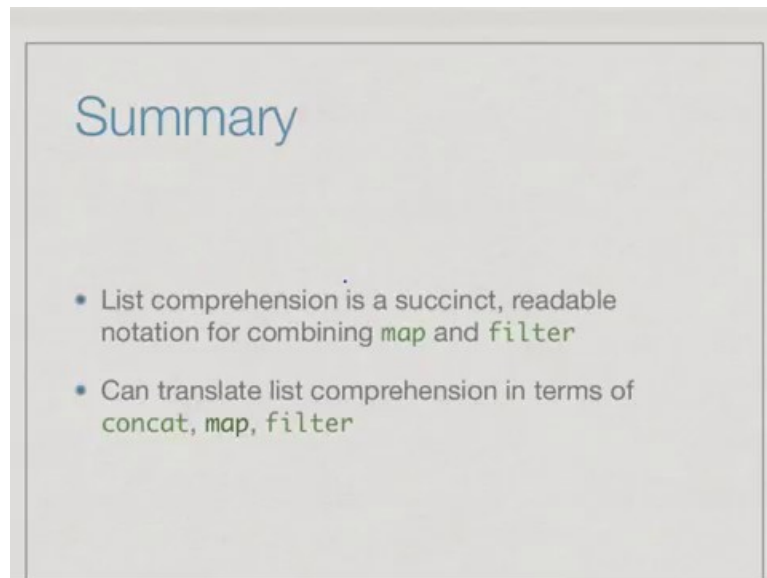
Having done this, I have got the first element, so sieve will extract it out and then, it will say apply sieve to the result of this original list, which we already, this is the list that is currently running. So, we take every element in that list and apply another condition. So, once again if we do this to nested list comprehension the first element that comes out will be 5. And so now, after we extract the 5, then it will say ok take the inner list that we are already working with, the 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 write this out for yourself in ghc and verify that, it does generate primes. So, this is not necessarily the most efficient way to generate primes. In fact, it is not the most efficient way to generate primes. But, it is certainly an interesting exercise, it says that a very direct implementation is possible, because of the combination of this list comprehension notation and lazy evaluation.

Of course, lazy evaluation is crucial otherwise we cannot go with infinite list at all and using lazy evaluation and using list 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 a new piece of notation in Haskell. It is merely what is called syntactic sugar. It is just an easy to read form of something that can be described directly using concat, map and filter.