Formal Languages and Automata Theory

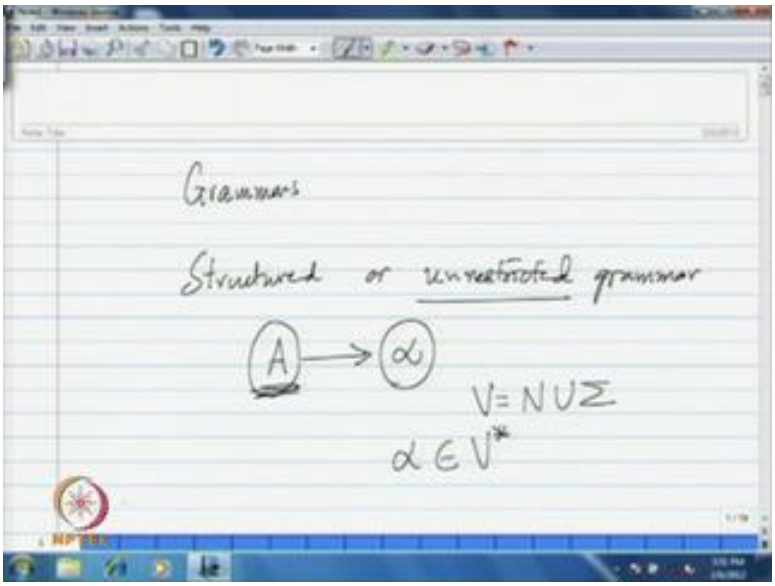
Prof. Dr. K.V. Krishna
Department of Mathematics
Indian Institute of Technology, Guwahati

Module - 12
Structured Grammars
Lecture - 1
Structured Grammars

So far we have discussed about computability with respect to turing machines. Now, like in case of finite automata or push down automata; we have an equivalent formulism with respect to grammars also. So, in case of push down automata we have called context free grammars; in case of finite automata that is a regular grammars or right linear grammars. So, we have discussed all those things and we have ascertained the equivalence between those classes. So, similarly in case of turing machines we have a class of grammars an equivalent set of grammars. So, here this is the most as you understand that turing machine is defined for the at most computability; that means, as a device which is useful to model the computation this is the model of computation.

So, grammar whatever that we are going to talk about that should also have you know facility to capture or to understand anything, which is which we wanted to compute. So, in this lecture I will talk about that more general type of grammars. So, we call this as because we have certain restrictions. So, far on the grammar in case of context free grammar or regular grammar this is the one, which is more in a relaxed fashion; that means, the more general one we call it as structured grammar or unrestricted grammar.

(Refer Slide Time: 01:56)



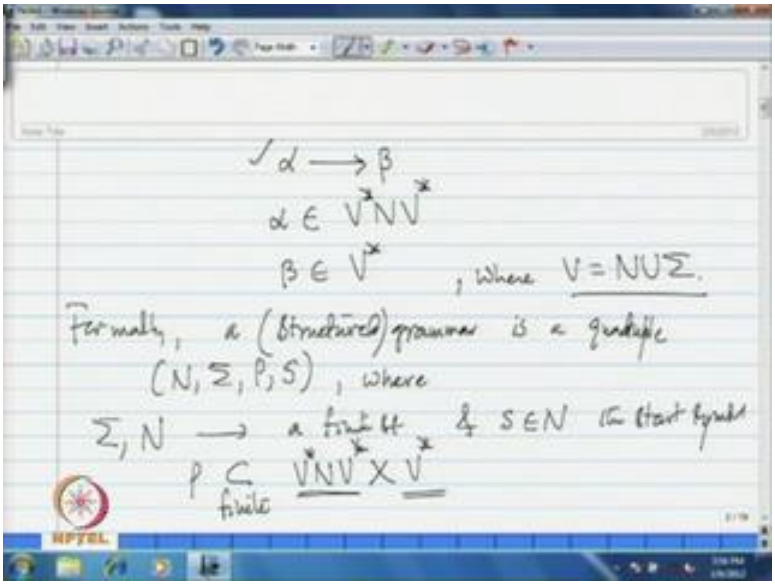
So, the grammars that I am going to talk about here are structured or unrestricted or unrestricted grammars. So, now, the question is what is that what is the meaning of unrestricting like if you look at if you remember in case of context free grammar a production rule A goes to alpha, where this A is a non-terminal symbol. This alpha is a mixture of terminals and non-terminals because if you write terminals by sigma; I wrote that as say V. So, alpha is an element of V star; as we you just recollect that this a non-terminal because some for every production rule left side some non-terminal has to their otherwise when you are applying a production rule because for a terminal.

We have already agreed upon that when you want to apply there has to be a non-terminal on the left hand side of each production rule. Now, we relax this condition that left side only one non-terminal symbol that was the restriction that we are maintaining so, we relax that condition. Now, what is the more general a situation the left side also you can have a mixture of terminals and non-terminals; right side also mixture of terminals and non-terminals. But you remember that if you want to apply a production rule at least one non-terminal has to be supposing if you arrive to a terminal string in a derivation.

Then, we have as we have seen we do not expect anything for other. Because we arrive already a terminal string; that means, the derivation ends there. So, there has to be at least one non-terminal to apply any production rule. So, that condition we cannot relax. So, the more general scenario is left side a mixture of terminals and non-terminals, but at

least one non-terminal has to be there and the right side a mixture of terminal and non-terminals as earlier.

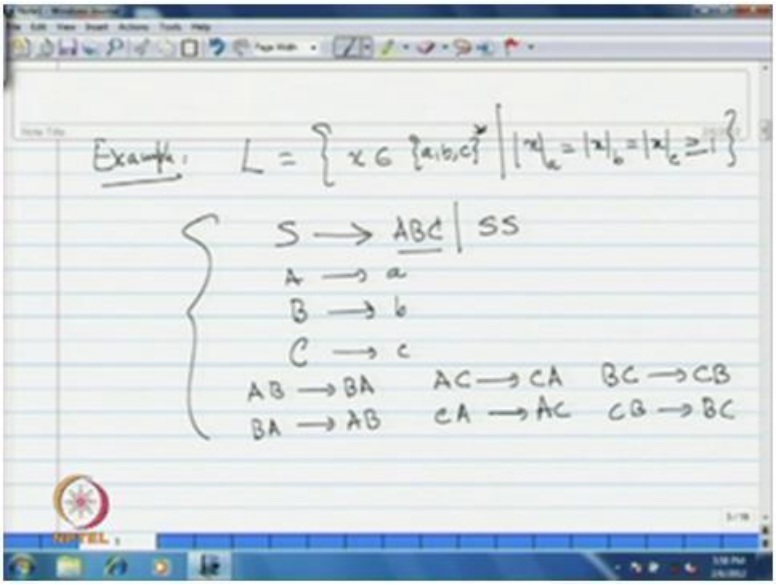
(Refer Slide Time: 04:04)



So, this unrestricted grammar the production rule, which is of the form say alpha goes to beta, where alpha is a mixture of terminals and non-terminals, but at least one non-terminal has to be there. That means, it is an element of this set and as earlier beta is an element of V star, where this V is non-terminals union the terminal symbols. So, this is what we introduce. So, formally a structured grammar structured or unrestricted grammar; we may call it is as simply grammar. Grammar is a quadruple this is a quadruple as earlier N sigma P S, where N finite set because there are all as earlier finite set called non-terminal symbols and sigma is a finite set, this is called terminal symbols and S is the start symbol; S is a non-terminal the start symbol.

Now, this P the production rules it is a finite subset of V star N V star; that means, at least one non-terminal as to be there on the left side cross V star, where V has given here. So, this is how formally we give it as a quadruple N sigma P S, you like in case of context free grammars or regular grammars we give to the same notation. But, here the production rules it is a finite subset of V star N V star; that means, left hand side of each production rule should have at least one non-terminal symbol on the right hand side you will have a mixture of terminals and non-terminals. So, that is how formally an unrestricted grammar can be defined.

(Refer Slide Time: 06:25)



Now, let me look at an example. I will not consider those examples, which we have already established some regular grammars or context free grammar; because you can quickly see as a remark that every regular grammar is you know a context free grammar and every context free grammar is a structured grammar. Because left side you are allowed to use a mixture of terminals and non-terminals with at least one non-terminal; if you fix only one non-terminal on the left hand side; it is just boiling down to a context free grammar. Therefore, I am not going to give those examples, which we have already established some context free grammars.

So, let us look at the example because we have this particular example that we have observed that this language say x in a b c star such that number of a's in x is same as number of b's in x is same as number c's in x. Of course, just I put a restriction that is at least non-empty strings. So, we have observed that this is not context free language. Now, can we have a grammar for this because we have a more relax and you know you have already constructed turing machine also for this. So, we can obviously, as I had mentioned that these grammars are I am going to introduce here they are equivalent to equivalent computational capacity with turing machines.

So, we will naturally expect a grammar for this. So, what kind of thing here? Context free grammar is not possible. So, the left side mixture of terminals and non-terminals sort of thing is required here. So, or more than one non-terminal symbol is required here. So,

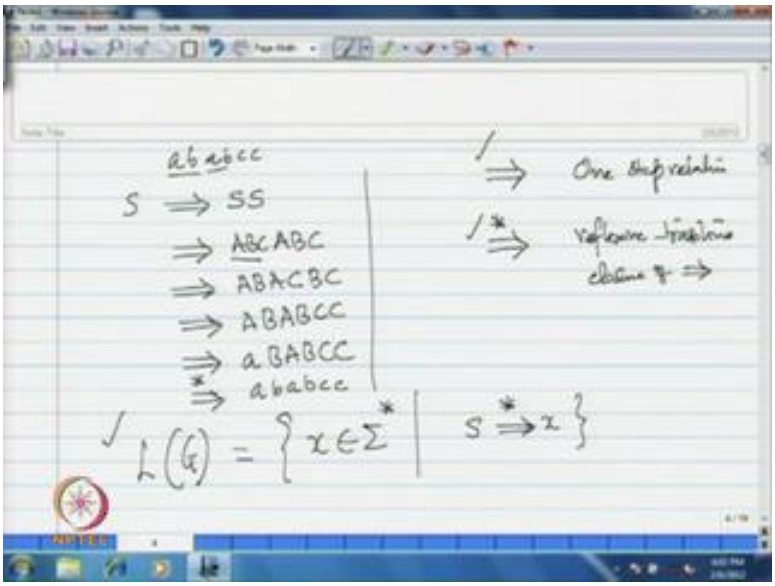
let as look at a grammar for this. You can quickly ascertain and understand that you have to generate a b c's equal number. So, first what I will do I will take the production rule say capital A capital B capital C. So, A B C when I will generate and maybe I will put SS recursively the number of a b c's whatever you want you generate and then. So, for example, by using these two rules you can quickly understand that.

For example, if you want two a's two b's two c's; then you use the production rule SS and two S you have generated and S goes to A B C; if you use two times then you can get this for corresponding to each capital letter of course. I will give the respective small letter say A goes to a; B goes to little b and that C goes to little c for example, I give it like this. Now it is not that A B C's will be the same order because it is it can be mixed for example, all A's come together. So, may be some B's and A's are mixed, but C's may come together. So, whatever is the possibility only restriction I have is number of a's and number of b's and number of c's should be same.

So, what I will do? I will along this non-terminal symbols A B C's they can commute each other. So, for example, A B can be B A whenever you have A B if you want B A you can make if you have B A; similarly I will have a flexibility of making A B. So, this A will commute with B similarly, A can commute with C say A C can be C A; C A can be A C. Similarly, B C can commute with C that is B can commute with C that is B C can be C B or C B can be B C. Suppose, we if I give all these rules you can quickly understand that the logic the way that we have defined you can understand that you know first, what are the number of a's or number of b's number of c's they should be equal. \

So, you wanted to have in a particular string that many times you apply as recursively and get that n number and once you have that required number of capital a's capital b's capital c's, which are equal always. Then, whatever the particular type of string you want what are the permutation that you want your permute this a a's b's c's capital a's capital b's capital c's and then you terminate with the terminal strings and you can quickly generate whatever the string that you want in this language let me give an example.

(Refer Slide Time: 10:47)



For example, if you want a b a b c c you can see that two a's two b's two c's are here. So, the derivation as I had mentioned you may first use SS the production rule and this S you can put A B C and for this also you can put A B C. Now, do not terminate first you arrange using that permuting rules and bring the form whatever that you want. For example, a b can give this first A B there is no problem, but you want a b here. So, this C we will take. So, C can commute with A as well as B. So, I will that rule by commuting this I will get A C then I have B C and you can commute this C with B also. So, I will get A B A B C C. Now, you can terminate this a little capital A can go to little a.

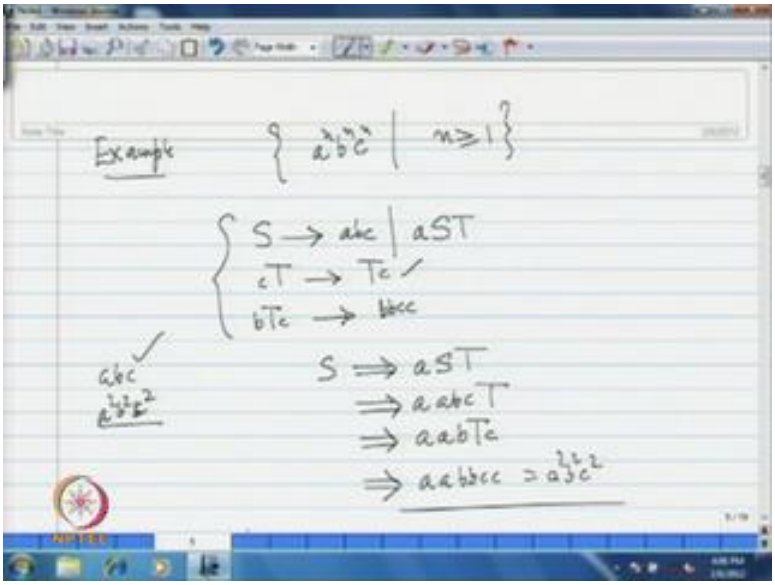
So, that is the situation here and then this capital B. So, you finitely many times I simply write this a you can bring this b then a then b then little c then little c. So, you know how many steps you required I am putting just star here finitely many steps. So, you have a derivation like this. So, quickly we can ascertain that the strings in this language can be generated because as I had mentioned that at least one a or one b one c has to be there because number of a's equal to number of b's equal to c greater than or equal to 1; one a, one b, one c has to be there, but whatever is the order and the logic that I have defined. You can recursively give this and you can permute to the positions, where ever that you want a's b's c's.

So, these are the permuting rules and thus any string in this language you can observe that any string in this language can be generated. Of course, I have not defined formally

the notion of this one step relation and the reflexive transitive closer of that. So, this can be you know as earlier in case of context free grammars. So, I am using this for one step relation; one step relation and its reflexive transitive closer is by star of this reflexive transitive closer of this one step relation. So, this one step relation again what is that given a string if you can apply rule once and to get another string say from alpha to beta; if you can get in by applying one production rule once then we use this symbol between the strings as earlier.

So, that is what using this two we have defined we have shown that S derives this particular string a b a b c c. So, as earlier for a grammar G language generated by G is set of all x in sigma star such that S derives x infinitely many steps, where s is a start symbol of the grammar. So, all these definitions this one step relation definition or reflexive transitive closer of that and the language generated by grammar as earlier; do, we are not going into details of this. But we will just see like what are the what are the languages that you can generate using structured grammars or unrestricted grammars this kind of grammars.

(Refer Slide Time: 14:35)



So, if you recall let me look at this example this is set of all x; I can simply write like this a power n b power n c power n such that n greater than or equal to 1 for example. Once again, we have observed that this particular language is not context free. We have used pumping lemma for context free languages and we have realized that this language

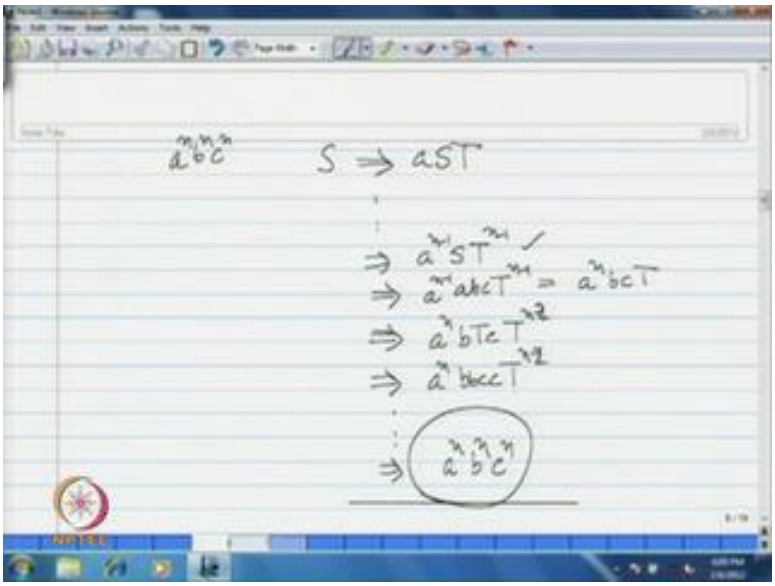
cannot be generated by a this is not a context free language. Now, again the question is can we have a structured grammar for this. So, for this particular language we have defined we have designed the turing machine also you have you have that.

Now, one wants that we do not have that flexibility we have certain restrictions may be let me define like this. I want to generate because n greater than or equal to one some a b c has to be generated and then say let me give recursion and S T say T may be like this and here the idea is as follows. This, T counts how many this this commutes with this T commutes with this little c and whenever you have this b T c; we will make this t essentially generates the respective number of b c's say b b c c. So, if I give an example then you will realize that it generates a power n b power n c power n; you can see that if you want just a b c to be generated it is not a big deal you just get in one step using the first rule there is no problem; because S goes to a b c is the rule.

If you want so, this can be generated if you want a square b square c square say 2 a's 2 b's 2 c's in that order if you want. You see I use this rule a S T and then this S with this rule a b c this T is there. So, T takes care of you know generating the respective number of b's and c's as for that we have used. So, this T can commute with c and come inside. So, what I get like this a a b T c. Now, b T c can give another because each T will generate a pair of b c's. So, a a b b c c. So, what string that we wanted in this language is. So, this production rules that the way that I have defined look at that because this S can produce as many a's as you want and T is the check corresponding to each S you are having a T.

So, this T will generate the respective number of b's and c's in the appropriate position. So, for that purpose this T has to be you know moved in between that b's and c's. Because if T is not moved then what will happen this b's and c's when you are generating you know you required to generate them in this particular pattern after b's then c's should come and after once, you have seen then they should not be any b. So, that is the reason that we have to allow this rule that T commutes with little c and you will be able to generate then the respective in the respective pattern.

(Refer Slide Time: 18:36)



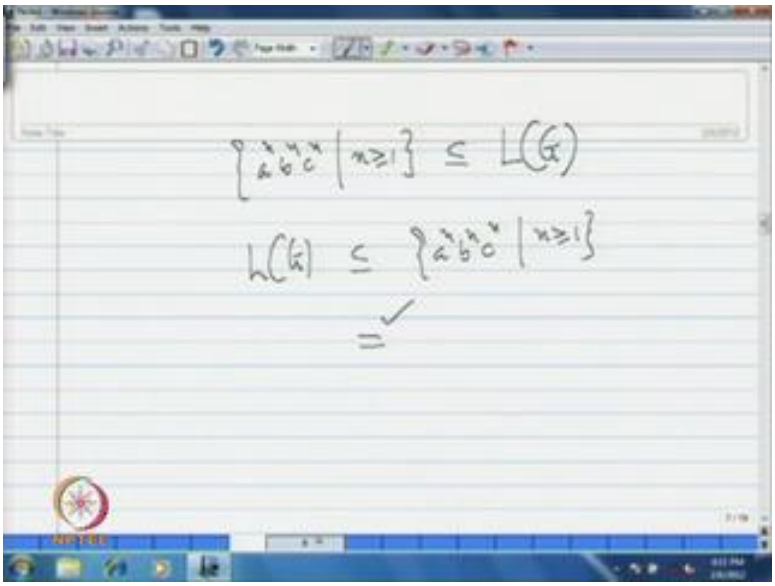
So, just for the sake of you know familiarity just if you want a typically a power n b power n c power n; then what you have to do is you use this rule for n number of times to generate a power n; then what will happen here S T when I am using. So, here every time I get a power n S then this T power n that is what we get. Now, you see so, n minus 1 times we use and then this you terminate with a b c a power n minus 1 a b c c T n that is how we have to do. So, what do you have this is this is a power n b c T. Now, you bring T inside one T. So, a power n b T c T power n minus 1.

This is how you have then this you can generate now 2 b's say b b c c; then T power n minus 1 and so on. You keep bringing this T's inside to generate the respective number; now you have 2 b's here a b c T power n minus 1. So, this will be n minus 2. So, two b's two c's already there; the respective number of T's you bring inside and after finitely many steps you get a power n b power n c power n. So, this is a typical derivation in this particular grammar. Now, what I have observed you take any string, which is the form a power n b power n c power n because any way a 2 b 2 c 2; I have already observed assume n greater than n greater than two, in which case you just use this kind of rules a S T for n minus 1 times.

So, that you get a power n minus 1 S T power n minus 1 after getting this a power n minus 1. You have then S you will terminate with a b c then you have n a's then you have one b c here and it n minus 1 T 's you have. So, those n minus 1 T's you bring

inside to produce the respective number of b c's again side by side so; that means, this T has to commute with this c's what are the that. So, for you have generated then, T will come between b's and c's there you terminate that T by generating one more a pair that b c so that is how we will generate a n b n c n.

(Refer Slide Time: 21:17)

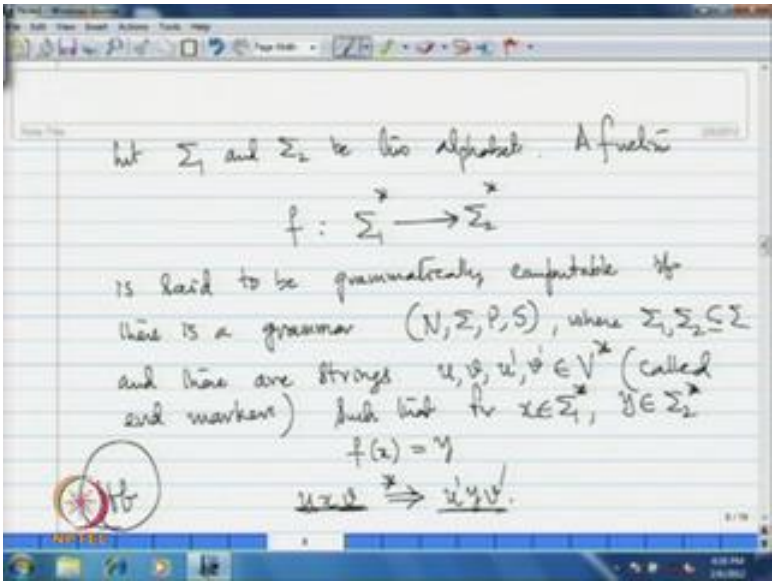


So, by this what do you understand this language a power n b power n c power n such that n greater than or equal to 1 is a subset of the language generated by G. Now, we have to look for the converse; that means, whatever that is generated by this grammar the string is of the form a power n b power n c power n, for which we have to look at the characteristic properties of each rule. Here, I have two terminal symbols S and T unless I use this rule a b c the S cannot be avoided that is what is the property. So, if you start with start symbol S unless I use the rule a S goes to a b c we will always have the non-terminal symbol s in because there only possibility that I can use a S T.

So, I will be producing this many number of t's. Now to avoid this t that means to terminate this capital T; I have to use this production rule. I can use this production rule if I have this T in between b and c. So, I have to use this commuting rule if I have produced certain b's certain c's and so on. So, you use there are only very few rules one two three four rules only. So, observe that applicability of the four rules and then you argue that this the language generated by the grammar is a subset of this at, a power n b power b c power n such that n greater than or equal to 1.

Thus, we conclude that this language generated by this particular grammar is this language you have this equality. So, this is one phase of grammars, but as I had mentioned that these are having the power of turing machines. Turing machines are used as language generators or as computable computing devices. Because we have talked about turing computable functions between you know strings a turing computable function is the one to compute that particular function. You should require a turing machine; now, an equivalent or a parallel mechanism or parallel concept of computability in case of grammars that we introduce.

(Refer Slide Time: 23:39)



So, that is let me give you this formally the definition as in case of turing machines. Let sigma 1 and sigma 2 be two alphabets. A function f from sigma 1 star to sigma 2 star is said to be grammatically computable as earlier we have talked about turing computability; now we are calling grammatical computable; if there is a grammar say N sigma P S, where the sigma 1 sigma 2 or subsets of the sigma and there are strings let me call u v u dash v dash in V star we call them as end markers. We call them as end markers such that such that for x in sigma 1 star y in sigma 2 star if y is a image of x under f if and only if u x v from u x v in finitely many steps in G you should be able to get u dash y v dash.

Look at the similarity of the turing computability with this grammatically computable function. Now, just recall a function is said to be turing computable if there is a turing

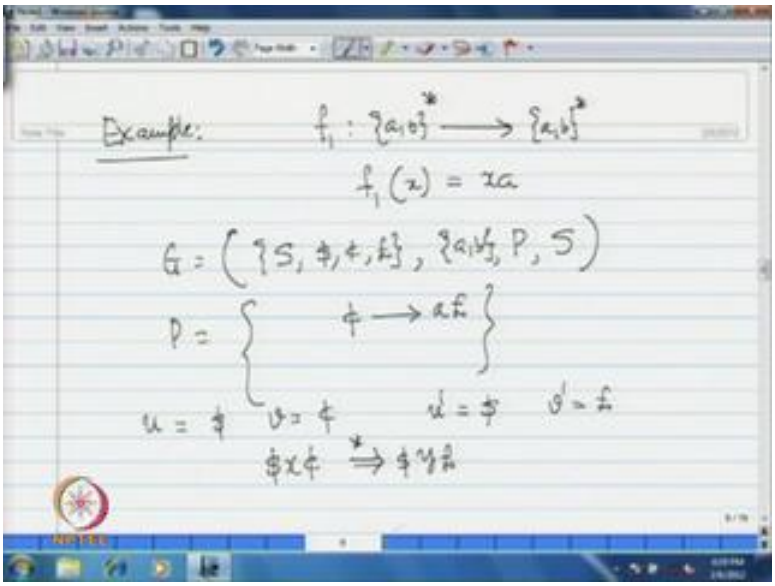
machine turing machine such that if y is image of x under f then by giving x as input in the initial state; you should be able to get y as output and conversely and conversely I mean this if and only if. That means, by giving x as input whenever you are getting y as output in a turing machine what is that condition that y has to be image under f of x. So, the similar the same thing here in place of the turing machine we are talking about through a grammar, but here there is nothing like you know giving input and output therefore, we are talking with respect to certain end markers.

If you want to say a particular function is grammatically computable with respect to a particular grammar further particular grammar you have to first declare these are the end makers. So, these are the starting end markers in this particular contest we are choosing u v and the ending end markers we are choosing u dash v dash. It is similar to you know initializing a turing machine and you know you are stopping a turing machine halting a turing machine by giving the output. In the similar fraction, here we had to introduce the end markers because otherwise in case of grammars we are generating grammar through start symbol.

Here, start symbol will not a play any rule, but any way the grammar that start symbol is a component. So, we are any way considering that is fourth component as well in the grammar, but here the end markers are important when you want to say a particular function is grammatically computable. We have to fix the end marker for that particular grammar for to compute that particular function. Now, look at this definitions take two alphabets sigma 1 and sigma 2 a function from sigma 1 star sigma 2 star is said to be grammatically computable; if there is a grammar n sigma p s where sigma 1 sigma 2 subsets of sigma and there are strings.

These are the end markers u v u dash v dash for the purpose of you knows the starting and ending we are using. So, u u v u dash v dash is elements of v star such that whenever y is image of x under f then you should be able to have this kind of derivation and between this end markers whenever you have this terminals strings x and y with the property that x is in sigma 1 star and y is in sigma 2 star then y has to be image of x under f. So, this condition is very strict this if and only if condition. So, using this condition will be establishing that turing computable functions are also grammatically computable and vice versa. But let me demonstrate first how to construct such grammars to certain examples then we will see accordingly.

(Refer Slide Time: 28:54)

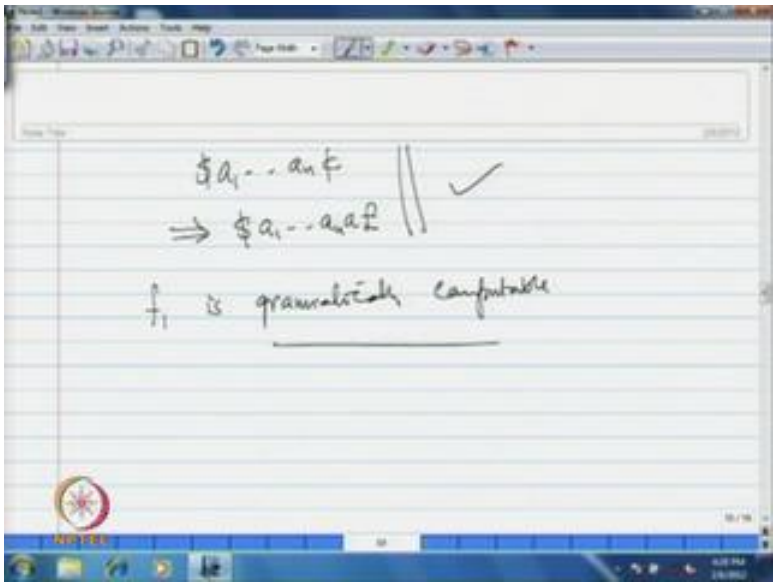


So, let me consider a simple example it is a very easy example to understand this. This grammar generating language is you know very well, but this is little bit different. So, let me first consider say sigma to be say a b. So, a b star into a b star a simple function this f 1 I am defining if you take any x I simply concatenate a to x. So, you can clearly see this is the function it is grammatically computable; what is the grammar I consider let me consider the grammar G to be say any way start symbols since we require.

Let me take that itself as a start symbol and a special symbol some dollar some cent is this known symbol let me use and the terminals a b the production rules I give you and S is that. The production rules what do I give as follows; if I have a with cent I will simply give or I do not require anything because I have to produce a I will just give this as a production rule just only one production rule. So, what is the idea here I declare the end markers u to be this dollar and v to be this cent; I give and u dash same dollar, but v dash I declare it as is forms.

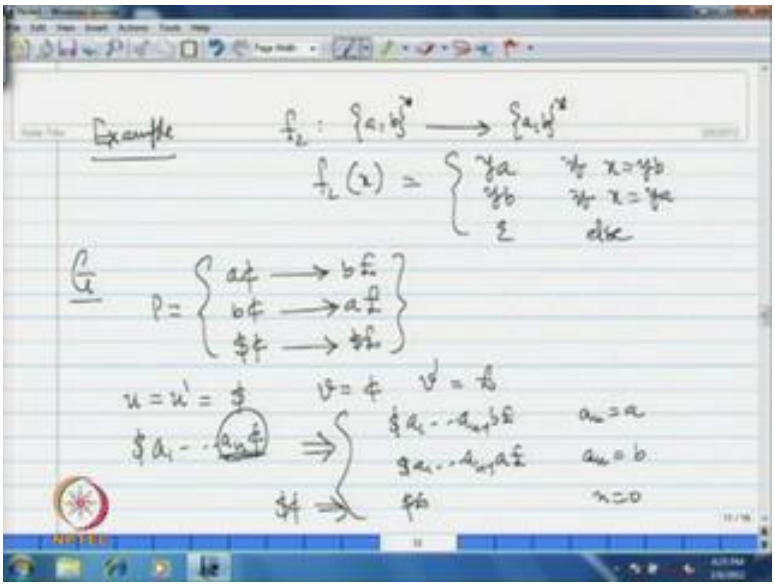
So, the idea here while giving this particular production rule whatever that string x that I consider with this end markers dollar and cent and after finitely many steps whatever that why I wanted to have that is this end marker I am taking the same; but this end marker with this form. I do not get this pound unless I apply production rule. I apply this production rule whenever I have this cent I simply concatenate.

(Refer Slide Time: 31:20)



You can quickly see that using this rules any sting that you consider a 1 a 2 a n this dollar and this cent you apply that production rule only once. Then, you can get the same a 1 a 2 a n you have what are that strings then you concatenate and terminate. So, only one step derivation that you have in this grammar; because only one production rules wherever cent is there; I will simply concatenate and produce pound. Since, this is the end marker for right side and this is for the left side the same thing that we have consider. So, here within one step I get the image of x under this function f 1 and hence so, everything that you can compute in one step only. So, you can quickly see that f 1 is grammatically computable function. So, this f 1 is grammatically computable.

(Refer Slide Time: 32:18)



Let just let me explain this; this conditionally because unconditionally we are putting there. So, f 2 is f 2 is same a b star to a b star defined by f 2 of any x is equal to let me write y a if x is of the form say y b and it is y b if it is of the form y a epsilon else. So, look at the definition of V function if the string x is terminating with b if the ending symbol is b; then I have to remove that b and I have to place by a and the vice versa. Whenever I have to place it by b and if it is empty string it is empty only. So, for a non-empty strings at least a or b you should have. So, you can quickly tell me that what is the production rules that you require? Same earlier if you have with cent I just give b with pound and b with cent if you have and say a pound you give.

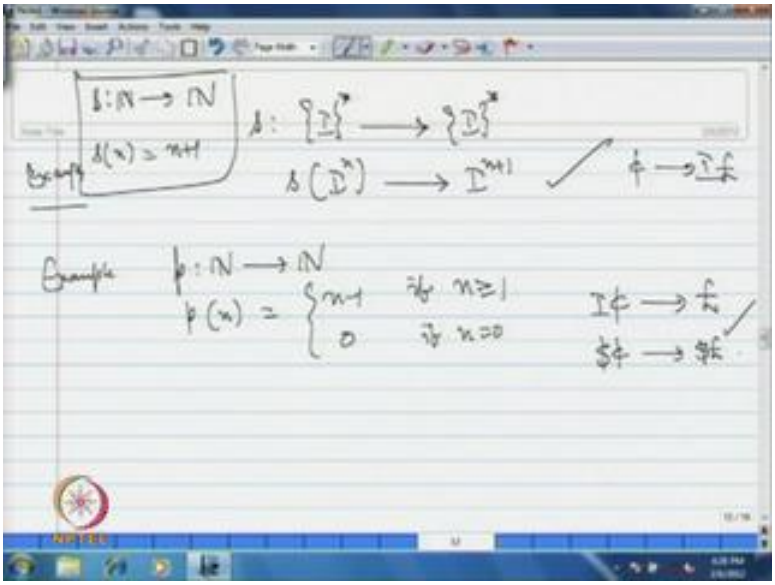
This is important because dollar with cent; that means, if the input is empty string then you have you require this rules. So, these three rules let me consider now the respective grammar G you know what the non-terminal symbols etcetera are. So, this is the p for this grammar. Now, the idea here is with the u u dash I am taking the same dollar, but v to be sent and v dash is the pound that is what we are consider. So, if you take any string a 1 a 2 a n with the start symbols this dollar and cent. Now, what happens this a n based on this a n if a n is a then it is of the form y I require b. So, this a n cent that is a cent will become b.

So, there are the possibilities this dollar is same a 1 a 2 a n minus 1 a and that if a n is equal to a then s cent. We can make it as b pound and if that a n is b this will become in

one step a 1 a 2 a n minus 1 by applying this second rule you will have a pound and if n equal to zero then you have this dollar and cent side by side and hence in one step you can produce this. Thus with the end markers the beginning ends markers u v and the ending n markers; that means, by the time of terminating this derivation. Of course, here also every derivation is you know in one step that you are getting.

Let me consider little better example of course, we have talked about the definition of grammatically computable functions from strings to strings as in case of turing machines; we can talk about this grammatical computable function over numbers also when I am talking about numbers as earlier we will talk with respect to unary representation.

(Refer Slide Time: 36:19)

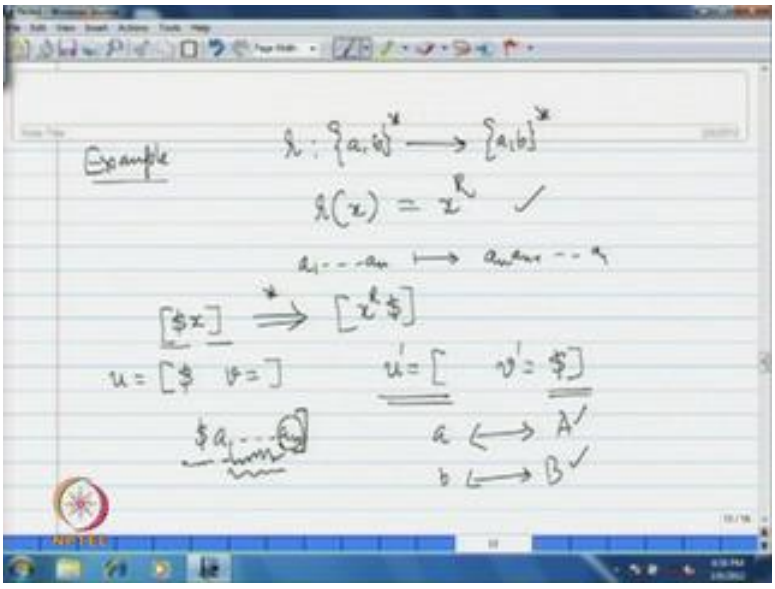


So, let me consider a simple function that is successor function from I star the unary representation the natural numbers are considered I star; that means, essentially I am considering s from natural numbers to natural numbers defined by s of n equal to n plus 1 the successor function. So, here that s is taking I power n I power n plus 1 as we have constructed earlier we can see this is grammatically computable, this particular example. Now, similarly if you want to talk about predecessor function from natural numbers to natural numbers defined by p of n is equal to n minus 1; if n greater than or equal to 1 zero if n equal to zero.

So, for that we have to be little bit careful that you know what are the number of I's you are given if there is an cent the way that you are defining I cent you can make it as say for example, pound. So, that one I is killed the last one if there is no I then the scenario is that you have dollar and cent side by side and hence. So, essentially you have to check the condition because earlier you can simply define that produce one more I at the end by changing the symbol you will get it; that means, let me give you because dollar and cent consistently if I am using I can give the production rule like this; whenever I have cent I will simply make I pound that is all.

So, what are the number of I's existing n number of I's one more I is appended and you declare that pound to be the v dash, but in this case of predecessor function we have to cross check; that means, as earlier if you have this I cent that you can make it pound. So, one I is killed and otherwise if there are no I's then; that means, this dollar and cent will be side by side then you make this is dollar pound. So, if you consider these two rules you can quickly see that predecessor function over natural numbers; you know using unary representation with number of I's you can see that it is they are grammatically computable.

(Refer Slide Time: 38:47)



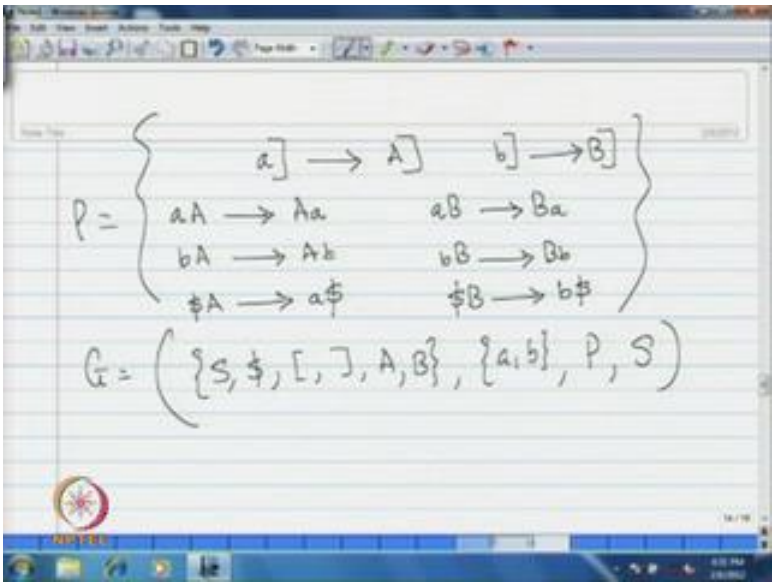
Let me consider another example, which let me stick to a b for the time being say f from no let me write r a b star to a b star defined by r of x is reversal of x. You know reversal of x if whenever you are considering a 1 a 2 a n the string the image is a n a n minus 1

and so on a 1 in reversing the string. So, this is little bit tricky; we have to be careful with the end markers and how we have to pursue this job look at. Now, let me fix may be end markers like this some bracket this dollar x this bracket; this is with this pair end marker u and this is with v after finitely many steps; I would like to produce this x power R this dollar this.

So, here u to be this square bracket dollar and v to be this square bracket and u dash is this square bracket and v dash to be this dollar and square bracket with this end markers I will pursue the job. So, here what I will do whenever I am considering this a 1 a 2 a n this dollar the idea is like this. The square bracket the terminal symbol whenever it is touching this particular terminal symbol corresponding to that I will create one non-terminal symbol some non-terminal symbol and I will allow this non-terminal symbol to commute with each of this terminal symbols here and pass through this dollar; whenever it is passing through the dollar I will make that as a the respective terminal symbol.

For example, if it is I will make it as a capital A let me not to use this because you may think this is the production rule. So, I will convert this as capital A and if it is small b I will convert this as capital B that is how I will choose. So, whenever this is touching this right side bracket the respective capital letters that I am considering. This capital letters can commute with any of the small letters here. So, that it will go here and after crossing this dollar I will leave them back; that means, if the capital A is crossing this dollar I will make it as a small letter if the capital B is crossing this. Then, I will make it as a small b; thus you will pick you will understand that x power R can be generated through this mechanism.

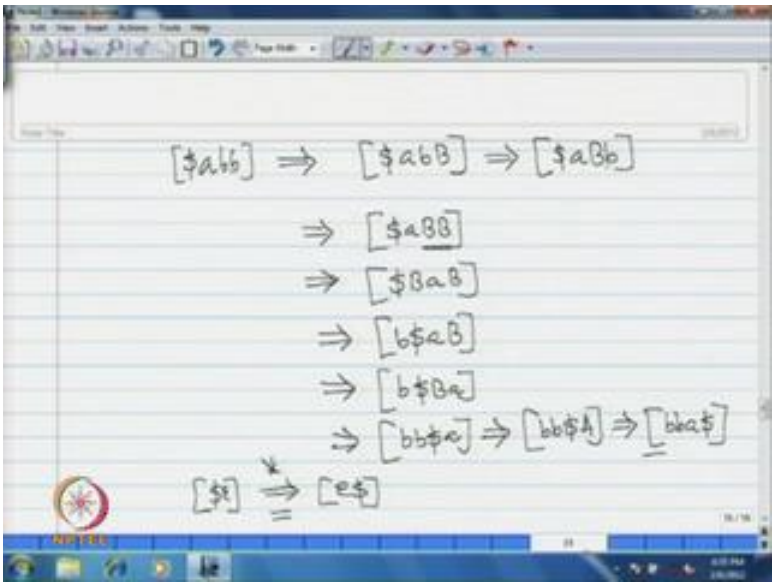
(Refer Slide Time: 41:48)



Let me give the rules. So, the idea is this if I have little a with this bracket the idea is to create the capital A in that place; similarly if you have little b touching to the right side bracket; I will give the rule that it will be make capital B. This capital A and capital B should be able to go through this little a's and b's so that means, this little a capital A the capital A little a; that means, a can pass through this and similarly it can pass through b also. The same property with b's a capital B is this and if you have little b capital B then B b here. Now, this capital A whenever it is touching dollar what do we do, we convert this back to its original form that is a dollar; similarly this dollar capital B is touching this dollar once it pass to that place; we will leave it back to little b dollar.

Now, you look at through an example. So, these are the production rule I am considering these are how many there are four here four here total eight production rules. This is what the production rule set is; you know the grammar here of course, for start symbol purpose you can take dollar as a start symbol does not matter. So, dollar I have used and this square brackets what else capital A and capital B. These are the non-terminal symbols under consideration and terminal symbols little a little b only I have and P has given there and this is S. So, this is the quadruple under consideration this grammar computes this function making a string reversal making a string reversal.

(Refer Slide Time: 43:48)



How it pursues? Let us look at an example suppose I am considering a b b. What is the form that I am considering dollar this bracket and this bracket. Now using that rule two little b with the bracket can be converted to capital B bracket. So, I have a b this is capital B. Now, this capital B can pass through this little a's and little b's. So, what will happen this dollar now you have two possibilities of applying production rule? You can pass this b through a make little b and systematically you can go for the second symbol or if you want to make this little b again capital B, because this small b is touching the right side bracket.

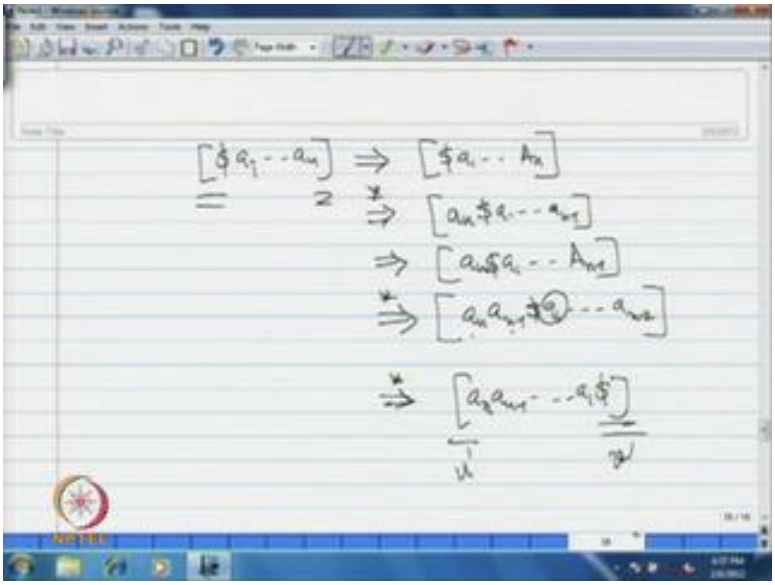
So, let me for fun consider this way look at with these symbols cannot commute among themselves this capital letters b and b cannot commute them. So, the first symbol can commute with small letters only a; you cannot convert unless you pass these symbols. So, let me pass now dollar this capital B a B. So, what I want to point out here whatever the order that the capital letters are generated in that order only they will pass through dollar. The order cannot be changed because this capital can commute with only small letters and this capital letters will become small letters after passing through dollar.

So, the order whatever the way that they are converted to capital letters will not be changed that is what one has to observe here. Now, this capital B is touching this dollar. So, you just pass through that and now becoming the small letter this is how you are you have. Now, this little a unless it became capital letter it cannot pass through dollar and

so, unless you get that particular position you are unable to do that. So, that order is taken care here now pass b through this little a. So, you have this and now this capital B passes through this becomes small letter this and a b b; this dollar will become now capital letter.

So, this is now b b a dollar and you see the end markers u dash v dash you have achieved. This is u dash as declared u dash is left side bracket and v dash now is dollar with this bracket and for empty string you do not have to apply any rule quickly. We can see that if it is empty string the input is of this form here epsilon and with zero number of steps; here let me` write star zero number of steps. In fact, we can see that because these two strings are essentially same.

(Refer Slide Time: 47:01)



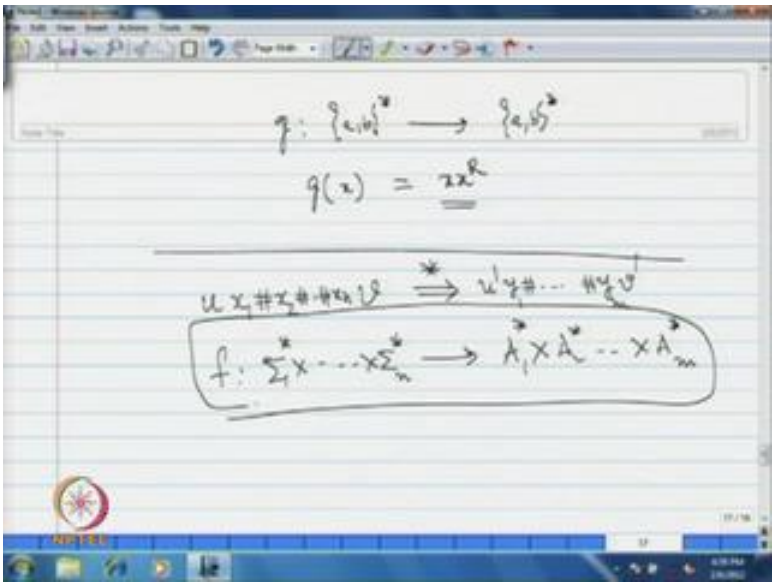
Now, in a general case let us see what is happening in a general case if you consider a 1 a 2 a n you consider a 1 a 2 a n. This becomes capital A n a 1 this is capital A n it may be capital A or capital B I am writing a n corresponding to this. This a n can commute and pass through this after finitely many steps this a n comes here becomes small a n dollar this a 1 a n minus 1. Similarly, this a n dollar a 1 say capital A n minus 1 this may be capital A or capital B again this passes through all these letters finitely many steps and now pass through this dollar become a n minus 1; this dollar a 1 and so on a n minus 2.

Now, you notice that a n has first cross dollar its position is maintained and a n minus 1 is the second symbol to pass through this dollar; so a n a n minus 1 and so on. So, this

symbol last at the last a 1 will be passed through this dollar and will be at the end and dollar will be next to this. So, you will get after finitely many steps using this same process a n a n minus 1 and so on a 1 dollar by the time a 1 passes this dollar will be next to this. So, you will get the final end markers.

This v dash and this is u dash and whatever is the string, which is formed this way and if this dollar is going to next to this you can realize that this will be the reversal of the given string. So, that we can understand that the reversal can be forming reversal is a grammatically computable function; so, all these things that we have done in case of turing machines because given x as input x power r.

(Refer Slide Time: 49:04)



How to create or you know given x may be let me do that kind of thing if I give g from say a b star to a b star; g of x is equal to x x power R. So, this this kind of all such functions we have observed through turing machines that we can create this kind of strings if x is there to create x power r next to that. This kind of things or you know over numbers computing over numbers all those things that we have done through turing machines the same thing; whatever that you have done that computable whatever the computable function that you have observed through turing machine you can observe through grammars also.

Now, for the sake of that generalization, because you have handled with two strings three strings as input and so on n number of strings as input; only thing is here I have to have

certain notation the end markers say for example, u v that you are fixing if you want to give n number of strings as input you may separate with some special symbol like this as in case of turing machine x 1 x 2 and so on x n. We will separating through some special symbols and whatever the output that you wanted after finitely many steps u dash v dash in between suppose; if you want to generate m number of strings say y 1 say special symbol some blank symbol that I am using assume this is not the part of the input alphabet.

If you can do this, then we say some sigma 1 cross sigma 1 star cross sigma 2 and so on sigma n star to for example, sigma let me you say some other sets A star A 1 star cross A 2 star and so on alphabet some alphabet A m star. So, this kind of strings the computable functions in a general case of this form can also be discussed. For example, now you consider two numbers giving as input unary representation and then calculating some of those two numbers product of two numbers or division. So, whatever is the computable function that we are handled through turing machines the similar things that you can handle here.

May be, we will discuss some of the examples through grammatically computable function showing grammatically computable function is you know is much easier than creating a very cumbersome turing machine. So, we will discuss few more examples and then see; so meanwhile whatever the simple turing computable functions that we have discussed. They are all can be taken as exercised to show them as grammatically computable function; that means, the respective grammars that you have create and show that these are grammatically computable functions.