Lecture – 35
Syntax-Based Testing

Hello again, welcome to week 8 we begin the first lecture of week 8 in this series before we move on and look at syntax based testing which is going to be the topic for some time in this week and next week. I would like to spend some time because we are more than halfway through the course it helps to recap and see what we have done till now, what have we done, what have we achieved, what have we understood.

(Refer Slide Time: 00:22)
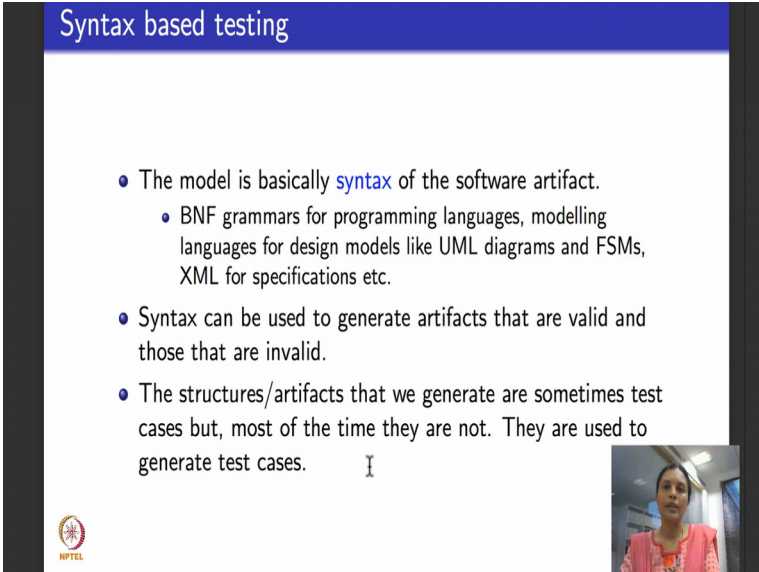
So, after the first week where I introduced you to motivation and initial terminologies of testing we have mainly focused on algorithms and methodologies for test case design. What did we do? We first draw graph based test case design I introduced you to structural coverage criteria, data flow coverage criteria then we took various software artifacts source code, do the control flow graph, data flow graph with the source code, so how to apply the coverage criteria on them. Then we looked at call graphs, so how to apply the coverage criteria on them. Then we looked at CFG's and finite state machine models for requirements and so how to apply coverage criteria on them. There is a nice tool graph coverage web app that is available as a part of the course page.

I have also spent one lecture introducing you to the traditional classical terminologies that people use for source code testing and also for design element testing as a part of these weeks. We also recap some algorithms on graphs depth first search, breadth first search, algorithms to enumerate prime paths and so on. After that exhaustive module on graph base coverage we went on to coverage based on logical predicates there again we. So, our exclusive module recapping logic predicates clauses what they are all, so coverage criteria based on predicates and clauses these assumption active inactive clause coverage criteria and so on. Then we saw how to apply to source code preconditions guards that common finite state machines and so on.

Finally, last week we took black box testing we took input domain input space partitioning. So, how to design test cases by partitioning the inputs in various ways and how to apply to testing a typical software artifacts like code. What are we going to do on from now on in the course we are going to do what is called testing in the course.

(Refer Slide Time: 02:25)

Typically almost every software artifact the deal we deal with be source code design or specification has an underline syntax which tells you what are the building blocks of that software. Like for example, if I take C program I know that you to be able to write an if statement I have to do such a such thing, to be able to write a while statement I have to do such a such thing, I always have to call the main function and so on right.

So, it tells me what we can do or how to write programs which are valid programs syntactically, which are invalid program syntactically. Typically syntax for programming languages are given as some form of grammars we will look at them in detail. You also have syntax for modelling languages like UML diagrams, finite state machines that come as a part of UML diagrams and then many software these days take inputs as XML right, XML is extensible markup language which defines an input format for several different kinds of entities that also has a very well defined syntax.

Every software in artifact some kind of syntax and what we are going to see is can I exploit a work with the syntax of the software to be able to test the software. You can use the syntax to generate artifacts that are valid that are invalid and we will see how to work with valid and invalid syntactical software entities to be able to test them or write test cases for them.

(Refer Slide Time: 03:53)

So, what will be the outline of this? So, we will this whole of this week and well into next week which is week nine we will only be dealing with syntax testing. So, we will as you always look at coverage criteria based on two different entities context free grammars and regular expressions I will introduce both of them to you briefly in this lecture and also discuss coverage criteria. Then in the next lecture I will teach you a term called mutation testing which basically mutates changes the syntax of a program then we will look at test cases coverage for mutation testing, how to do mutation testing for

source code and also for design elements there are specific mutation operators available when you integrate methods for object oriented call coverage we look at all of them. And finally, we will see mutation operators for a markup language like XML which is very popular which basically tells you how to manipulate the inputs to a program.

So, the focus of this lecture will be the first one here. We will tell you what grammars are what regular expressions are, how they come as syntax of programming languages, how to define coverage criteria on them and test them.

(Refer Slide Time: 05:04)

So, typically if you understand a bit of how compilers work. Compilers check if a program is syntactically fine, they compile the program and generate object code. So, every programming language that is meant to be compiled or interpreted comes with an underline syntax. How is the syntax of a typical programming language defined, they here is how it is defined it is typically defined in three different levels. To start with there are what a count words which constitute the lexical level and they tell you how to define words from various characters, how the various characters form tokens.

Generally words has specified using what are called regular expressions which are expressions that constitute regular languages and can be accepted by finite state automata. You should, if you have done a course on automata theory you would know what regular expressions are at a next level after words in programming languages come what are called phrases. Phrases are usually defined using a context free grammar that is

given in the particular normal form called Backus-Naur form. In fact, it is not an arbitrary context free grammar, it is what is called a deterministic context free grammar we will tell you what a CFG is in this lecture. And then at this level the grammar level phrases determine how to confirm phrases. After these we have what is called context information or context sensitive information which cannot be explained, expressed when we do phrases but deals with variables objects the type what do they refer to and so on and context is generally specified using context sensitive grammars.

So, three levels generic levels and which syntax many programming languages are defined, words given as regular expressions, phrases given as a context free grammar in a particular normal form called Backus-Naur form followed by context which is specified using context sensitive grammars. When we deal with mutation testing we are going to mainly deal with words and phrases because that is what you can manipulate syntactically really well and get variants of a software program a variants of an input. So, when we deal with words and phrases because their expressed using regular expressions in context free languages it helps to recap what regular expressions and context free languages are, that is what I am going to do for the rest of this lecture for you.

(Refer Slide Time: 07:21)

So, we will begin with regular expressions I will also tell you what context free languages are. If you want to know more details feel free to pick up any pick up any good book on automata theory or theory of computation you will get to know more about

regular expressions a context free grammars. Feel free also to ping me in fact, there are NPTEL courses on automata theory that are available you can also write to me for any doubts about these.

This is a one of module like we did graph algorithms that you would need to understand syntax based testing a little more thoroughly. So, syntax of a regular expression over an alphabet A, alphabet A is an alphabet that defines the building block of the programming language or very much like the alphabet when natural language that we look at. Once you have an alphabet A the regular expressions are defined using the syntax, you say the empty expression given by phi like this is a regular expression every letter there small a belongs to the alphabets at capital A, single letter is a regular expression on its own. Inductively if I have regular expressions r an another regular expression also denoted by r, please remember this r and this r need not be the same regular expression then there plus r plus r is another regular expression. Similarly, given two different regular expressions r into r they could be same they could be different r dot r or r concatenated with r is another regular expression.

Given a regular expression r, r star is another regular expression. So, what is syntax it says the empty set is a regular expression every single letter from the alphabet is a regular expression addition or plus or union of two regular expressions is another regular expression. Concatenation of two regular expressions is another regular expression, star of a regular expression is another regular expression this is defines a syntax. What do these operators mean each of these regular expression is defined what is called a language over the alphabet A. The set of all words over the alphabet A is denoted by A star and each regular expression r defines a language L of r which is a subset of A star. The language associated with empty regular expression phi as you would expect is the empty set there is nothing it is just an empty set language associated with the regular expression which is just a letter from the alphabet is this set containing the single ten word which is just that letter from the alphabet the same letter from the alphabet.

If I have languages associated with regular expressions r and r prime inductively called L of r and L of r prime then the language associated with the expression r plus r prime is given by the union of the two languages corresponding to r and r prime. The language associated with the regular expression r into r prime is given by the concatenation of the language associated with r which is L of r and the language associated with r prime

which is L of r prime. The language associated with the regular expression r star denoted by L of r star is you take the language associated with r and do the star operation. I hope you are all familiar with how to take union of two languages, it is a normal set theoretic union. What is the dot of two languages? Dot is a juxtaposing or concatenation. Suppose I have a word w from the first language, another word w prime from the second language L of r prime then w dot w prime is you take w and append that w prime and the end of w you can concatenateor juxtapose w and w. L of r into L of r prime is take every word from L of r and concatenate with every other word from L of r prime and this whole language is the concatenation of the two languages.
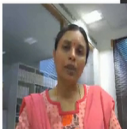
For star, star is 1 0 or more concatenations of the same language, right.

(Refer Slide Time: 11:06)

So, now let us look at some examples to understand how regular expression mean the first one I have answered it for you. What is it say? Here is a regular expression there are three letters in the alphabet ab and c and what I have done here - I have used a star plus b star concatenated with c. If you go back and look at the syntax I have used all the three operations I have used plus I have used dot and I have used star. Plus means take union, dot means concatenate or juxtapose star is the (Refer Time: 17:37) star. So, what is this regular expression mean? It means take any number of occurrences of a because a star is 0 or more occurrences of a and take any number of occurrences of b it to be single a, two a's, three a, say it could be single b, two b, three b's and so on. Take the union of these

two sets which is the plus and concatenate it with the single letter c. So, it means it is strings of only a's or strings of only b's that end with a c that are followed by c. So, the language defined by this regular expression is this language is this string of only a's or only b's that are followed by a c.

Let us look at another example here is another regular expression is says a plus b the whole star in the middle you have abb strictly speaking you should write it as a dot b dot b I have simplified it because I do not have to all the time write the dot notation it is a standard practice followed by another a plus b. So, what it says this that a plus b means it is can be either a or b star means it can be any word that involves a or b any combination of a or b here again at the end its any combination of a or b, but in between they must be a pattern for sure what is that pattern that battle is abb. So, this is the set of all words that contain abb as a sub word in that this is the regular expression.

Now can you tell me what this regular expression we will stand for? I will read it out for you it says a any number of any word over a and b in between there is ab and then this is one occurrence of a or b followed by one more occurrence of a or b. Which means the last letter could be an a or ab, the second last letter could be an a or ab, the third last letter from the right is only ab and then they could be any word over a and b in the front. So, the language for this is, is the set of all words where the third last letter in the word is a b.

So, now, again here is another expression this does not involve plus for a change it only involves dot and star, I will read it out for you. It is b star ab star a the whole star followed by b star. Spend of few seconds thinking about what the language that it could define this b star means that it could end with ab e and this, what is this mean? This means that if there is a word that is not epsilon then there is one a in it for sure and one more a in it and the rest could be or b a need not be there. So, the language that is defined by that what would it be, think about it and if you do not know ask me in the forum and I will tell you.

(Refer Slide Time: 14:24)
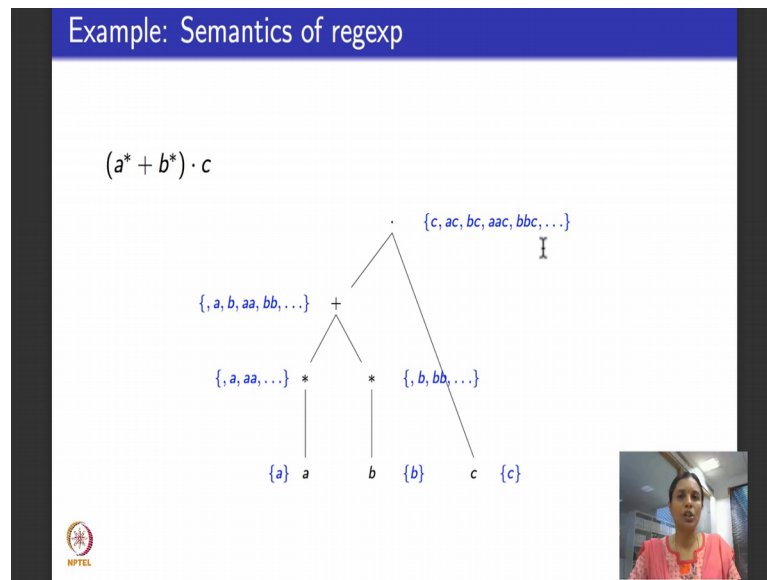
Now, how these are how regular expressions look like. Remember why are we looking at regular expressions because we want to use them in the context of a programming language syntax we are in level one here. So, programming language syntax there is something called a lexical analyzer which parses a regular expression and generates some entities out of them. So, how does that parsing happened? I will illustrate it using an example.

So, suppose I had this regular expression which was the first in the list that we saw here a star plus b star dot c all words ending with the c, if a parse this this is how I get; how do I read this. Read this is a binary tree the leaves of the tree or all letters from the alphabet the internal nodes could have one child or two children if it has one child and it is a unary operator like star like for example, read this part as b star the star operation apply to b, this is a star the star operation apply to a and then I take a star and b star and add it that is this part in the regular expression and then what do I do, I takes separately I concatenate it with a star and b star this is how the tree corresponding to this regular expression looks like and almost all regular expressions you can write such trees.

How do I define what is the language? With every node in the street I can associate a language and I inductively work my way build my way up in the tree to get the language corresponding to the entire regular expression. Like for example, for these leave nodes by a semantics the language singleton a singleton b singleton c for this is star. So, it is 0

or more occurrences of a, this is b star 0 or more occurrences of b I am sorry there seems to be a small typographical error they should be epsilon here before the comma and another epsilon here before this comma and after that this is plus. So, it is all words that have only occurrences of a only occurrences of b and after this is dot which is all words that end with the c. So, this is how you get the semantics of a regular expression.

So, this we just brief quick introduction to regular expressions, in fact, language accepted by regular expressions are called regular languages. We also have automata models for them finite state automata which we saw earlier in the course exactly correspond to languages that are defined by regular expressions called regular languages this theorem called Kleene's theorem. Theorem which let us you convert a given regular expression into a finite state automata and vice versa, any book on automata theory we will help you to understand that better. So, that was a brief introduction to the first part of the grammar which is this words. Now I will move on in tell you what context free grammars are and how are the defined.

(Refer Slide Time: 17:12)

So, here is an example of a context free grammar. So, what is this say, how do I read this, there is a special symbol called S, S stands for start symbol and then there are these production rules each production rule has a left hand side and a right hand side. The idea is you begin with the start symbol keep applying the production rules till you cannot apply them anymore. What do we mean by applying a production rule? Start with S,

replace the symbol S by using one of the rules that are available with the right hand side. So, for S this is the only production rule available in this grammar which is from S you can get aX. So, I have replaced S with aX using this rule now I have X.

There are three other production rules that are available with X. So, I could replace X with an aX replace this X with the bX or replace X with a b. So, let us say I chose to replace X with the bX that is I have used the third rule in this list. I choose to now replace this X with just this b. So, now, you see there are no more of these capital letter symbols available and I no more production rules to replace anything with this, so I stop. When I stop I say this is the word that is generated by the grammar. So, grammar this context free grammar generates this word abb.

So, in general how do you write the language defined by the context free grammar is the set of all words or terminal strings why are they called terminal strings? Because there is nothing they cannot be replaced for, that the all the terminal strings that are generated by the grammar constitute the language given by the grammar.

So, these letters in capital alphabet their symbols in capital letters they are what are called non terminals the symbols in small letters they have what are called terminals. So, now, this grammar I just gave you one word what would be the language defined by this grammar can you think of it. In fact, if you realize that the I can once I start do X as aX right. So, all letters begin with a, all the words begin with a because this is only rule that I can apply to start with and then I can replace the X with aX bX here I can go on replacing I generate one more copy of X to replace it. At some point I say I am going to stop like what I did here I choose this last rule where I replace the X with a b. So, this grammar defines all words that begin with an a and end with a b and this is the context free language. So, I have written it like a regular expression all words that begin with an a and end with a b.

(Refer Slide Time: 19:54)

Context-Free Grammar

A Context-Free Grammar (CFG) is of the form

$$G = (N, A, S, P)$$

where
- N is a finite set of non-terminal symbols
- A is a finite set of terminal symbols.
- $S \in N$ is the start non-terminal symbol.
- P is a finite subset of $N \times (N \cup A)^*$, called the set of productions or rules. Productions are written as

$$X \rightarrow \alpha.$$

So, in general a context free grammar looks like this, it has four entities, there is a set called capital N which is a finite set of non terminal symbols if you go back here then capital end the set capital N consists of the two non terminals S and X. There is a set a of terminal symbols we call it alphabet or sigma, then there is one designated non terminal in the set N called starts non terminal symbol denoted always by S and then P is a finite set of production rules. What is the format of the production rules? Production rules always look like this on the left hand side there is exactly one non terminal which comes from this set and on the right hand side it could be any string that comes from N union a star that is it could be any string that is a combination of non terminal symbols from N and terminal symbols from a.

So, it is a member or an element of N cross producted with N union a star which I for convenience and readability stake write like this. So, this corresponds to this capital N is a member of that set this alpha is a string from this set is that clear please.

(Refer Slide Time: 21:08)

Derivations, language etc.

- "α derives β in 0 or more steps": $\alpha \Rightarrow_G^* \beta$.
- First define $\alpha \overset{n}{\to} \beta$ inductively:
  - $\alpha \overset{1}{\to} \beta$ iff α is of the form $\alpha_1 X \alpha_2$ and $X \to \gamma$ is a production in P, and $\beta = \alpha_1 \gamma \alpha_2$.
  - $\alpha \overset{n+1}{\to} \beta$ iff there exists γ such that $\alpha \overset{n}{\to} \gamma$ and $\gamma \overset{1}{\to} \beta$.
- Sentential form of G: any $\alpha \in (N \cup A)^*$ such that $S \Rightarrow_G^* \alpha$.
- Language defined by G:

$$L(G) = \{w \in \breve{A}^* \mid S \Rightarrow_G^* w\}.$$

- $L \subseteq A^*$ is called a Context-Free Language (CFL) if there is a CFG G such that $L = L(G)$.

So, what is the language corresponding to a context free grammar, language corresponding to a context free grammars defined like this. You start from the start symbol, keep applying production rules one or more times as long as you have a non terminal symbol to replace and after some N applications of production rule when you cannot apply them any more you would get a string full of terminals and that is where you stop.

So, how do I define that I say alpha derives a string beta in zero or more steps, zero or more we always denoted by using the star in the grammar G, how do I define at I define it by induction on the number of steps let us say N is the number of steps. You say alpha derives beta and N steps if either alpha derives beta and one step over alpha derives beta and N step and then I say how alpha derives beta in N plus 1 steps. How do I say alpha derives beta and one step? If alpha is of this form alpha one X alpha two and there is already a production rule in the grammar available which let us you take this non terminal symbol X and replace it with a string gamma which means what in the string alpha 1 X alpha 2 I take X out and using this production rule replace the place where X was there with grammar.
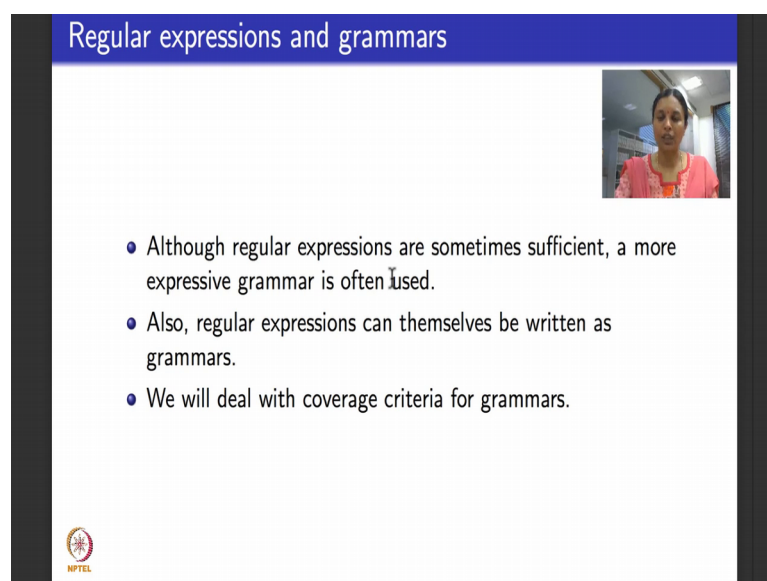
In other words I derive alpha gamma alpha 1 gamma alpha 2 from alpha 1 X alpha 2 by using the production rule X goes to gamma and I keep doing this one step one step inductively, how do I say alpha derives beta and N plus 1 step? If there exist some

intermediate string gamma such that alpha derives gamma and N step and from gamma in one step I can obtain beta, is at clear please.

Now, these intermediate entities that you see alpha gamma and all this is a term in grammar for them they have called sentential forms. Ultimately, so if I go back to this example this is sentential form, this is a sentential form every step in the derivation leads to an entity called sentential form and finally, sentential form end in a string of terminals that belongs to the language generated by the grammar. So, the language generated by the grammar is a set of all words over the alphabet of the grammar such that from the star symbol S I can derive using the rules of the grammar as explained here the word w. Language generated by such grammars context free grammars I call context free languages. You might wonder why the term context free for that if you go back here you will understand, production rules in context free grammars are always up this form, I take a non terminal X I replace it with symbol alpha.

This is irrespective of the context in which X occurs. Like for example, when I use it here I say X occurs here somewhere in between my sentential form without knowing what alpha 1 and alpha 2 is I can directly plug in a rule and replace X with gamma. So, when I do that I am free of the context in which X occurs and that is why it is called context free grammars.

(Refer Slide Time: 24:12)

So, typically regular expressions and grammars are described by using grammar notation when syntax of programming languages are written. So, we really do not write them as two steps, we combined the first two steps and write it as a grammar that defines the combination of both the steps put to together.
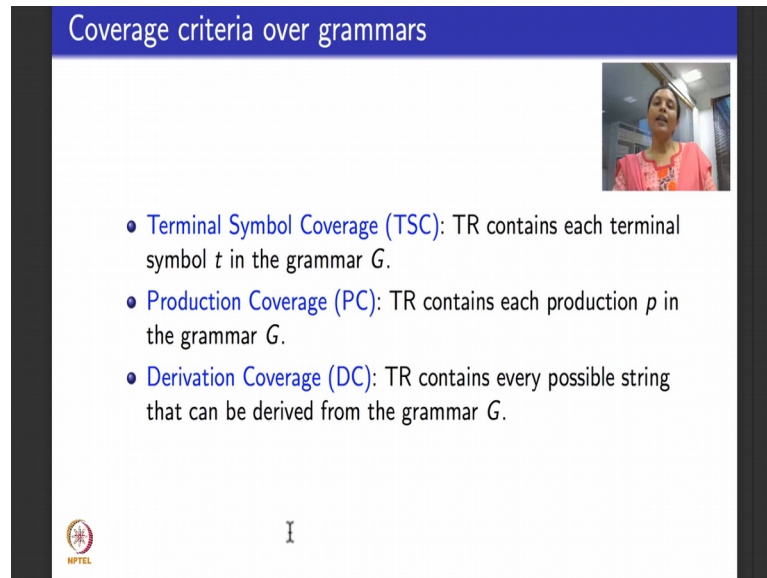
(Refer Slide Time: 24:31)

So, here is an example suppose this is how a grammar will look like here it if is notice it will have notations from regular expressions and notations from grammars.

So, I start with what is called a stream. A stream could involve one or more actions denoted by action star and action could be a non terminal of the form act G or a non terminal of the form act B. Read this is having two different production rules one which says action is act G and action is act B, act G has this format it could be G s n, G is like a terminal string par that is why I have put it within double codes. Act B is B t n, B is another terminal string of put it within double codes, s and t could be a digit of a single unit or three unit it is a digit of length 1 2 3 denoted like this digit to the power of 1 2 3; t is another digit of length 1 2 3, n is has this pattern - how do I read this it is a two digit number denoted by digit to the power of two concatenated with another two digit number concatenated with another two digit number. And what could be a digit? Digit could be anything from 0 1 2 3 4 5 6 7 8 9 right.

So, if I take this grammar how we will I generate strings I start with stream and then I do one of these act of G or act of B based on what I do the string begins with a G or with B

as indicated here it begins with a G or with B. Once I have a G or B I could have two or three digit number like I have 13 here and 17 here and then I have this, two digits followed by a dot followed by two digits followed by a dot followed by two digits which come from this rule 5 n. So, string generated by this grammar out of this form, is that clear.
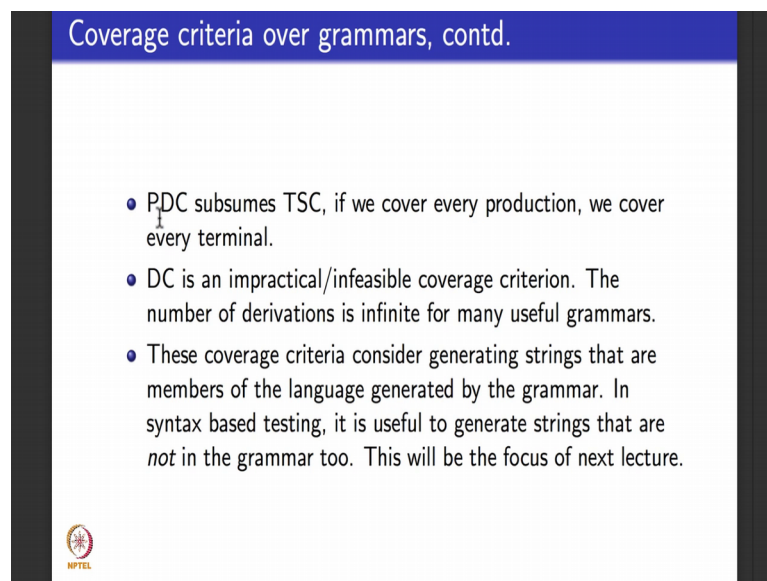
(Refer Slide Time: 26:25)

So, now I again define coverage criteria over grammars. So, what do grammars contain? Grammars contains terminal symbols right non terminal symbols start symbols and production. So, basically the rules say cover each of them, cover every terminal symbol called terminal symbol coverage TSC test requirement contains each terminal symbol t in the grammar G. Cover every production rule which is production coverage TR contains each production p in the grammar G and cover every derive issue DC derivation coverage TR contains every possible string that can be derived from the grammar G.

We have not included a non terminal coverage because if you think about it you will realize it does not make much sense. So, we define three kinds of coverage criteria directly over plain grammars we will see how to apply to grammars of various things like XML, programming languages and all. These three coverage criteria like we did for other things just say cover every terminal, cover every production, cover every derivation.

If you think about it terminal symbol coverage and production coverage make sense because terminal symbol says that every symbol must be generated by the grammar in the alphabet otherwise there is no point in the symbol being there on the alphabet. Production rule coverage also makes sense because if you do not use a production rule then it might as well not be there, but derivation coverage is an infeasible test requirement. Why is that so because there could be infinitely many derivations right as long as there is a rule like this in the example that we solve which let us you take X and give back X you could replace this use this rules several times again and again and again each one is a derivation and result in infinite number of strings.

So, derivation coverage can be sometimes difficult because it says test requirement is derived every possible string typically grammars generate infinite languages and it is not possible to derive every possible string, you will not terminate.

(Refer Slide Time: 28:23)

So, because productions have need to be exhaustive PDC which is production coverage, subsumes TSC if we cover every production we cover every terminal this is what I told you derivation coverage is impractical and infeasible also we do not need them. These coverage criteria considered generating strings that are members of the language. When we do mutation testing we will see how do generate strings that are not members of the language. So, that will be the focus of the next lecture.

Thank you.