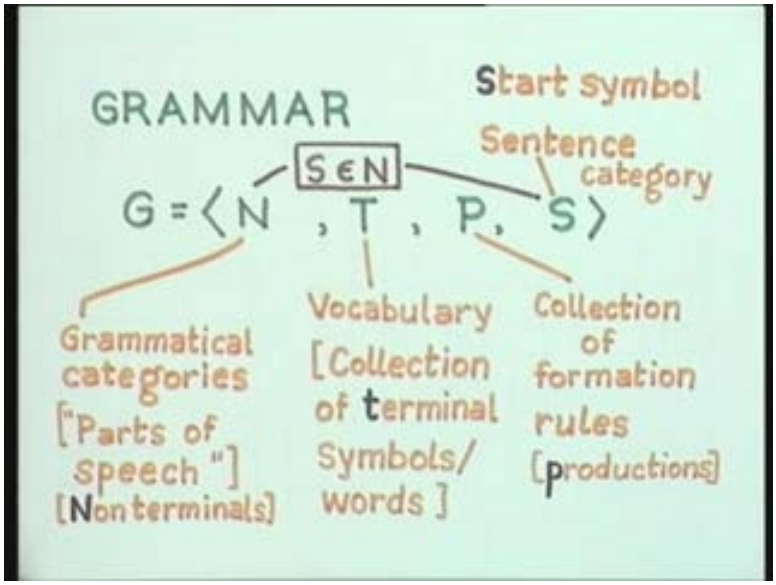
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian institute of Technology
Delhi
Lecture no 3
Lecture Title: Grammars

Welcome to lecture 3. We started with grammar in the last lecture and we will continue in some detail.

[Refer Slide Time: 00:57]

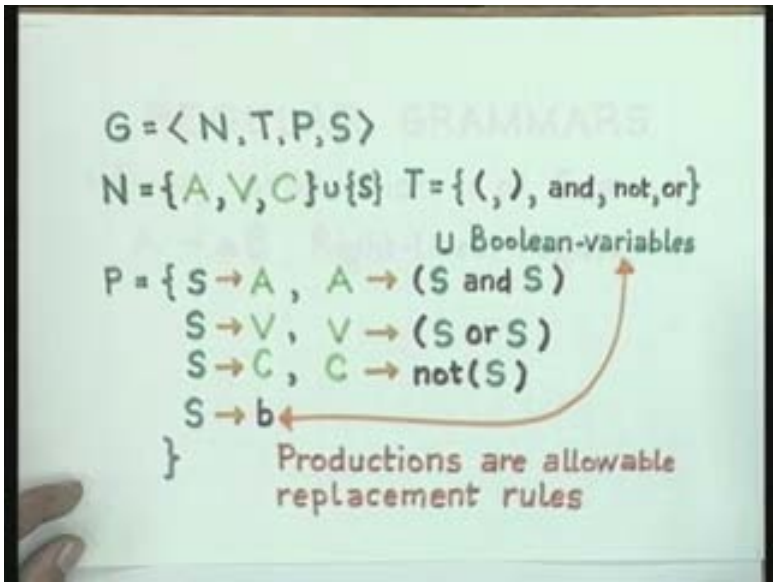


Let me just briefly summarize what we did in the last lecture. As a grammar is a four tuple consisting of a finite set of non terminal symbols or grammatical categories, a finite set of terminal symbols which usually constitutes the vocabulary of a programming language is a finite collection of formation rules or productions which are really rules of replacement and the start symbol really signifies the grammatical category called a sentence of the language.

We went through one simple grammar for example, the generation of Boolean expression. We have various syntactic categories. We have a set of non terminals and the grammatical categories consist of essentially expressions. The V stands for expressions, the C stands for complement expressions and there is a start symbol.

The terminal set consists of open and close parenthesis and the connectives and ‘Not’. While we are dealing with grammar, I will use the color black for terminal symbols and since the grammatical categories are a level of abstraction higher I will use light green for some and dark green for some others.

[Refer Slide Time: 02:57]

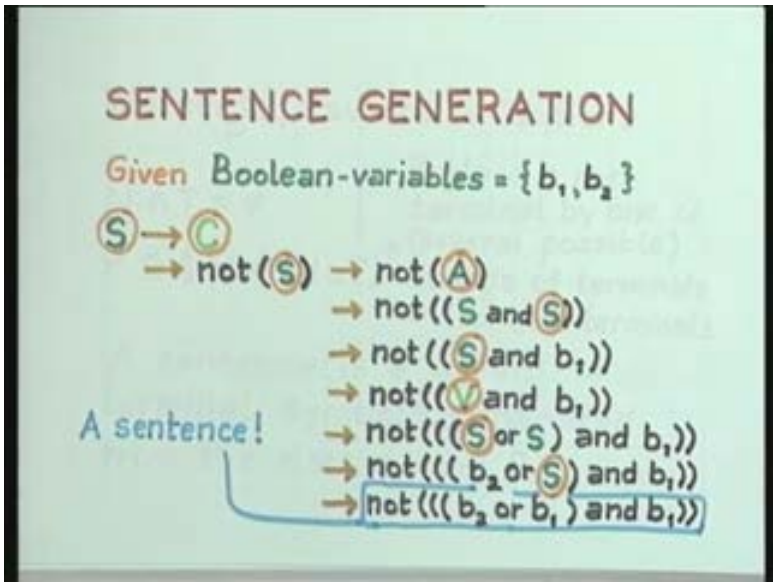


This was the grammar we had. We also saw how a sentence which is a string of terminal symbols can be generated from this grammar by applying the production rules. In each of these cases, I have circled in orange the non terminal symbol which I am replacing and I have several choices for replacing S.

If you choose different choices then you will get a large number of other sentences. You will generate a large number of other sentences since there is absolutely no restriction on how long you can keep getting S. In the case of this grammar you can actually generate an infinite set of sentences.

As you can see, a grammar is a finitary representation of an infinite set. A large part of computer science, mathematics and logic really has to do with how to represent infinitary object in a finite manner and this is one such example.

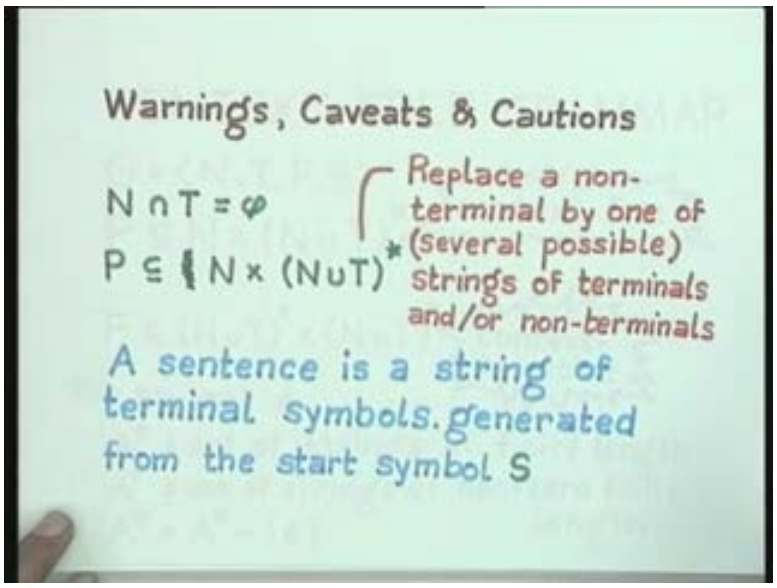
[Refer Slide Time: 04:20]



Some of the warnings and cautions that you must keep in mind are that this set of non terminals and the set of terminal symbols should be disjoint. The production set is really a binary relation from non terminal symbols to strings of non terminal and terminal symbols.

The replacement rule allows you to choose any non terminal symbol and replace it by a string consisting of terminal and non terminal symbols. There is a star here which is to denote that you can replace this set which is really the set of all possible finite strings that can be generated from this set (NuT).

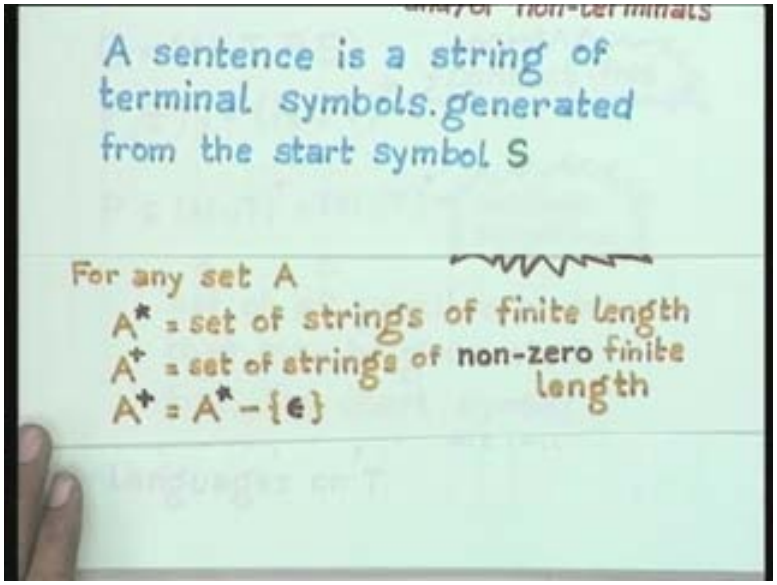
[Refer Slide Time: 05:28]



I will explain some of this in general for any set A, a '*' is the set of all the strings of finite length, in particular the finite length could be a length of 0. A 0 length string is really nothing so it is called the empty string and we usually use the Greek letter epsilon to denote it.

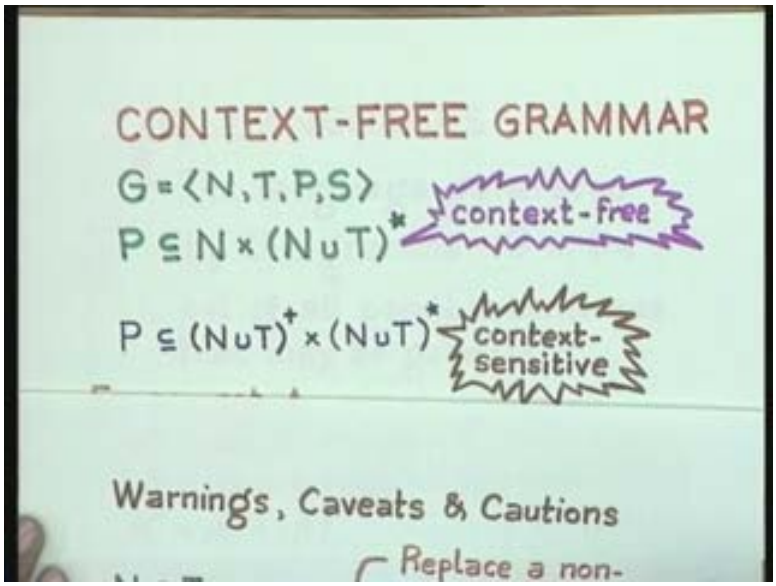
A + is the set of all non empty strings generated from this set of all non empty strings. There is only one string of 0 length that is the empty string. So it is the set of all non empty strings and A + = A * with epsilon removed from it since there is only one 0 length string. So, the particular kind of grammar that we have been considering as an example is what is known as a context-free grammar.

[Refer Slide Time: 06:37]



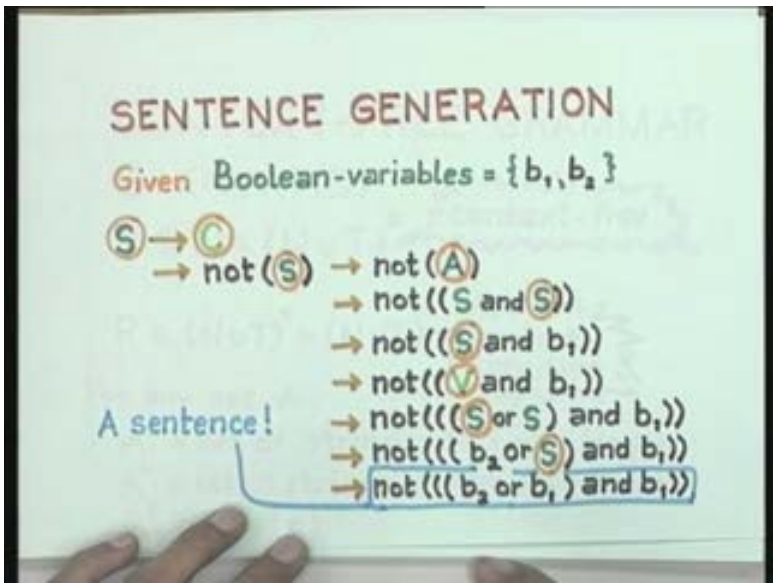
When the production rules are such that on the left hand side of the arrow mark you have a single non terminal symbol and on the right side you have some string of terminals and non terminals that is called a context-free grammar. It is called context-free as opposed to a context-sensitive grammar. A context-sensitive grammar has production rules in which given a certain string of terminals and non terminal symbols you are allowed to replace the non terminal symbols by some other string of non terminals and terminals.

[Refer Slide Time: 07:59]



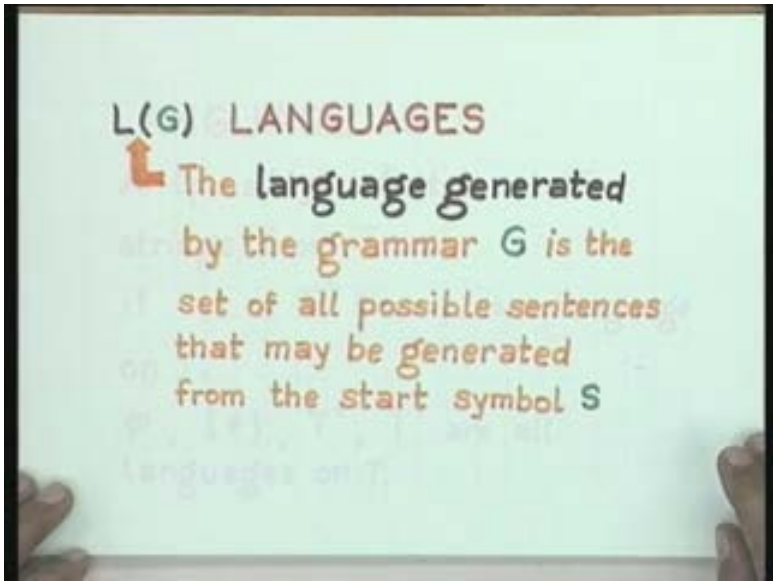
Let us take an example in context-free grammar. Let us take an arbitrary string in this example and we are replacing this S. Let us assume we are choosing this arbitrary string so this S appears in a context and the rest of this is the context. Similarly this S here appears in this context and we are calling this grammar context-free because we are allowing this replacement of S by a string of non terminals and terminals regardless of what context that S appears in. This is the context in which this S appears and we have a uniform rule, uniform in the sense that regardless of what contexts the non terminal appears in, you are allowed to do a replacement.

[Refer Slide Time: 09:13]



In a context-sensitive grammar you could for example specify that a certain non terminal can be replaced by a string only if it appears in a particular kind of context. In particular you might define that context to be an empty context. In general a context-sensitive grammar is really more general than a context-free grammar. You might look upon every context-free grammar production as specifying a context which consists of the empty string. We will go into that a little later but let us first consider some simpler grammars also.

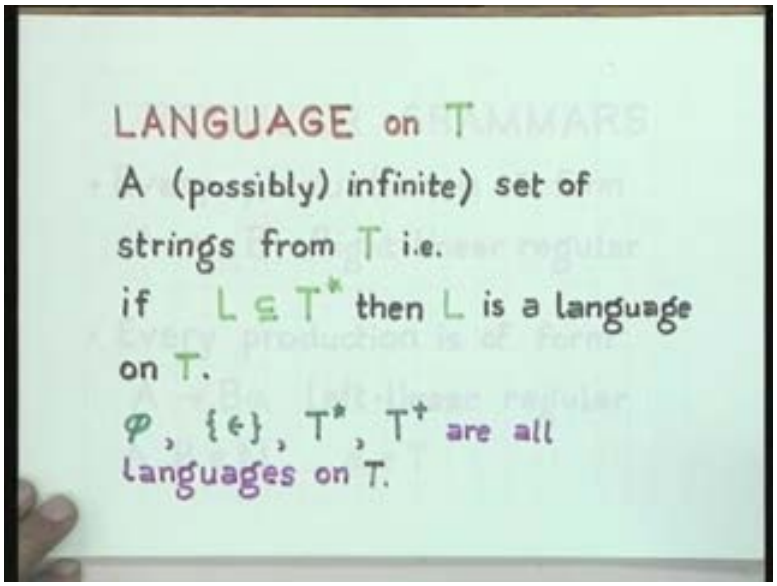
[Refer Slide Time: 10:23]



Let us look at languages. The language generated by any grammar is a set of all possible sentences that may be generated from the start symbol. For example, you can take the start symbol located in some context and generate a string that may not be in the language at all unless that context itself could have been generated from the start symbol.

This is what we would call a language and in general we would call it a language on the set of terminal symbols and a language on a set of terminal symbols is a possibly infinite set of strings from the terminal set. We are saying that any subset of T * is a language and for example here are some trivial languages that you can define on any set T. You have the empty set for strings which is the empty language. You have this language consisting of a single element, the empty string. You have T * which itself is a language and T + itself is a language and in between you have a whole lot of other languages. So you might regard this T + and T * as one extreme and the empty language and the language containing the empty string as the other extreme and you can have lots of subsets in between.

[Refer Slide Time: 12:47]

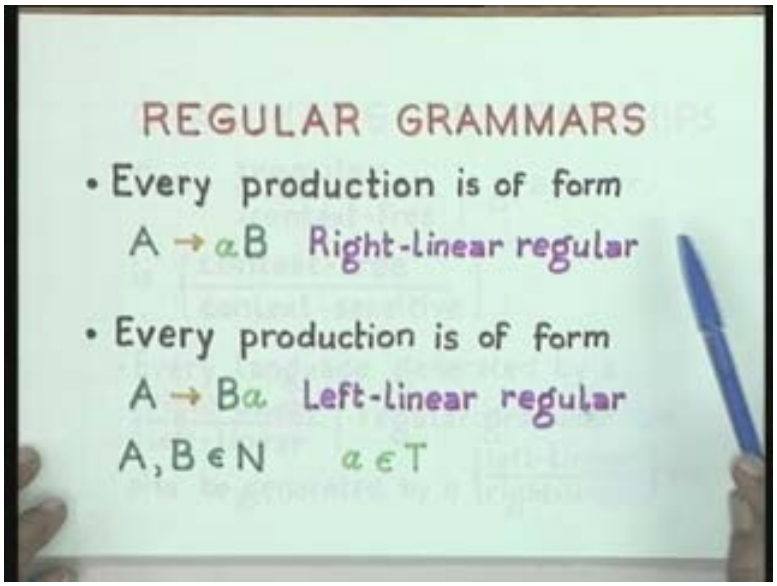


The problem is that when you have a programming language you have an infinite set of possible programs and the problem is of defining exactly what grammar can generate. We will say that just as we have defined grammar we will also define languages in a similar fashion.

First, let us go into some particular kinds of grammars called regular grammars. In a regular grammar supposing you take a grammar in which every production is of a form where this capital A denotes a non terminal symbol this capital B denotes a non terminal symbol and this small ‘a’ denotes a terminal symbol which in fact should have been made black and if every production is of this form then we call this a right-linear regular grammar. The first point to realize is that a right-linear regular grammar is also a context-free grammar.

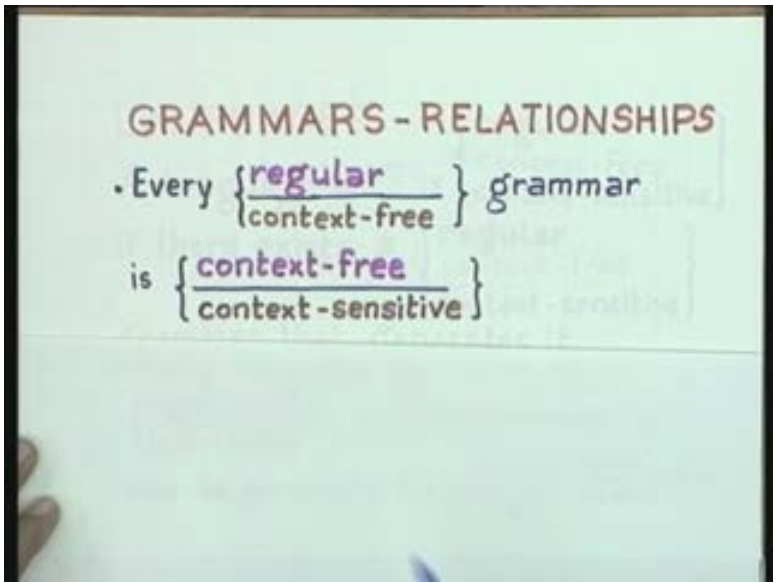
There is really no difference. A context-free grammar allows productions which are not for example a right-linear regular grammar. This means that on the right hand side you should have just one non terminal symbol and one terminal symbol appearing in this order. The terminal symbol is followed by the non terminal symbol.

[Refer Slide Time: 14:30]



In a context-free grammar we do not have that restriction for example we had S. In a context-free grammar if you take any of these rules they are not at all regular, they are not right linear or regular and of course you would also allow just a terminal to be generated.

[Refer Slide Time: 17:58]

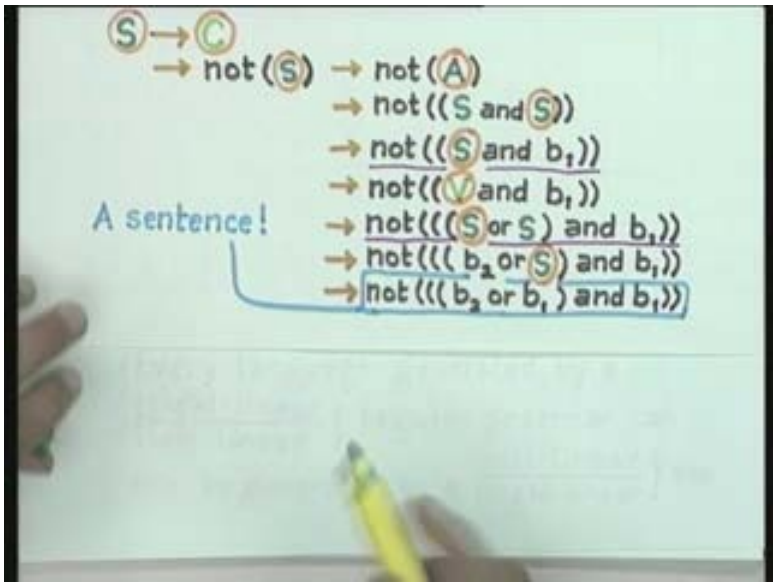


After all you have to generate strings of the language from the terminal set if you always had only non terminal symbols appearing on the right hand side. If that is the case then you will never be able to generate a full sentence of the language. A right linear regular grammar is one in which all the productions are of this form.

Similarly, you might define a left linear regular grammar as one in which all the productions are of this form and of course it has this terminal generation rule too. So, every production that is either of this form or of the other form is a left linear regular grammar. Such a grammar is called a left linear grammar and if you have designed some hardware using finite state machines it turns out that you can actually represent finite state machines that is machines without output. In fact, you can take the state transition diagram of the machine and refer to each state as a non terminal symbol and refer to the input symbol. The input into that state has a terminal symbol so a finite state machine automatically defines a right linear grammar.

Most finite state machines have a start symbol which is the start state. We are talking about a powerful language. Let us summarize what we have looked at. The general properties of grammars are that firstly every regular grammar whether right linear or left linear is also context-free. Every context-free grammar is also context-sensitive. In particular, all the productions of the context-free grammar can be considered in the context of empty strings on both sides of the non terminal symbol.

[Refer Slide Time: 18:42]



In general you can take any of these strings as having a ‘Not’ symbol, an empty string symbol, then an open bracket, an empty string and then A, an empty string and a close bracket. You can look upon every context-free production as appearing in a context which contains an empty string and the empty string implicitly appears everywhere between terminal symbols and non terminal symbols. It is for that reason that every context-free grammar is also context-sensitive. Our interest in grammars is ultimately in generating languages. Supposing you take any language that $T = \{\in\} \cup T$ can be generated by a right linear grammar then it is also possible to define a left linear grammar which will generate the same language. Similarly, if you were to take any language generated by a left linear regular grammar then the same language can also be generated by a right linear one.

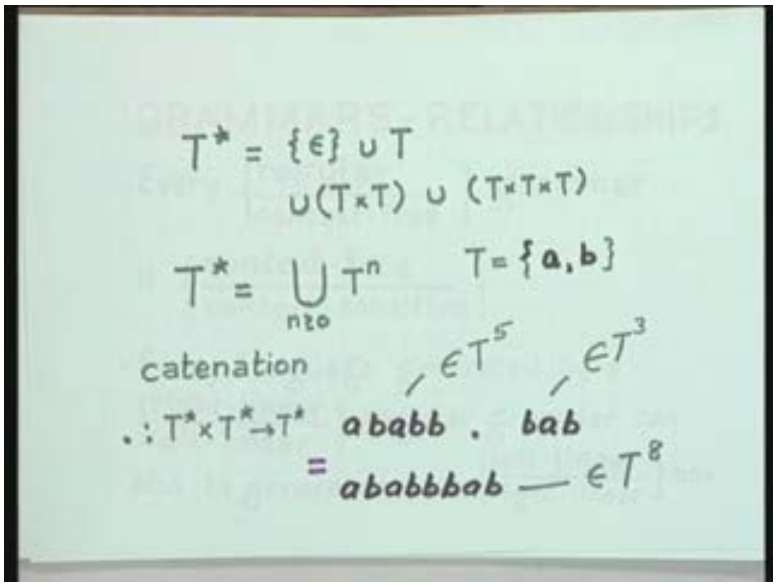
Let us look at the set T *. The set T * for any set of terminal symbols is just the set of all strings obtained by this terminal symbol, T. In particular I can look upon T* as;

$$T^* = \{\in\} u T$$
$$u(T \times T) u (T \times T \times T)$$

$$T^* = U T^n$$
$$n \geq 0$$

I can define a binary operation called catenation. The effect of catenation is to take two strings and put them together. For simplicity let us assume that the set T consists of just two symbols. Let us just call those two symbols a and b. So, I can take a string in T*. Let us take a string in the set, ababb and let me take another string let us say, bab. The operation of catenation which I will denote by just a DOT is to produce the string, ababbbab. The operation of catenation just juxtaposes the two strings. It is a binary operation on strings which just puts the two strings together and gives you a new string. For example; this string is of length 5 so this belongs to the set T 5, T raised to five. This string is of length 3 which belongs to the set T cube and this string has a length 8 and it belongs to the set, T8, T raised to 8 and of course all these sets are subsets of T *and so catenation is really an operation from T * x T * ->T *.
It has a functionality which is to take two strings of finite length and juxtapose it.

[Refer Slide Time: 25:30]

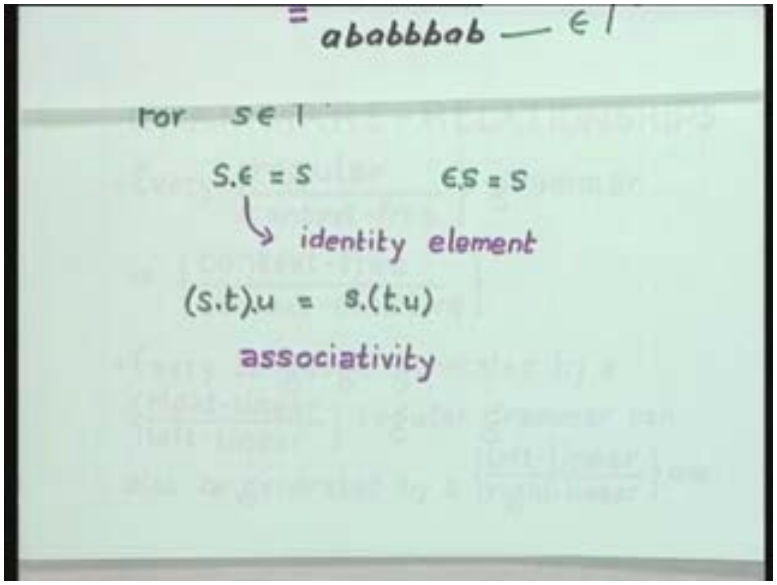


When you take a string and juxtapose an empty string to it you get back the same string. If you take the empty string and you juxtapose some other string to it you get back the other string. The empty string satisfies these conditions that for SET * for any S belonging to T*, SE = S and ES = S. Very often catenation is a juxtaposition operation where we just get rid of the Dot symbol but you could have a Dot in between.

So one obvious property now is that this epsilon, E is in fact the identity element for catenation. It is like a zero for addition.

Secondly, catenation is associative in the sense that if I take three strings s, t and u; I can catenate them in any order. The set T * under catenation and with the empty string is really a monoid because this operation is associative. Therefore, catenation is associative and it has an identity element however it is not commutative so it is not an abelian monoid, it is just a monoid. Now what do we mean by context-sensitiveness? Let us take a production of an arbitrary context-sensitive grammar. Let us keep this here so that we might in fact need it for some reason.

[Refer Slide Time: 28:48]

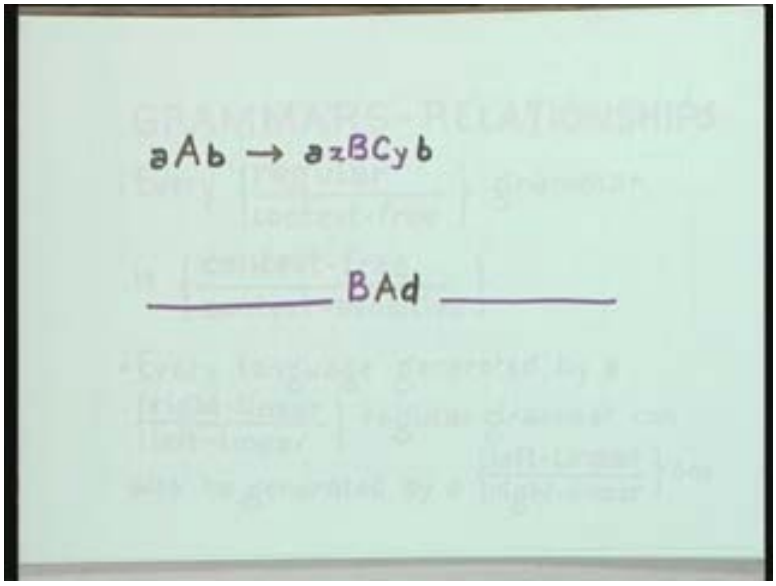


So if you were to take a production of a context-sensitive grammar, what it specifies is that I have a non terminal symbol A and if it appears in some context, a Ab then I can replace this non terminal symbol by some other string which means that the context still is going to be preserved. $aAb \rightarrow axBCyb$

But ‘a’ is going to be replaced so it is a conditional rewrite rule. I can replace a by B, C and may be xyz; x and y padded with some x and y. What this production says is that I can replace a by the string xBCy only if on both sides of the A, I have small ‘a’ and small ‘b’ appearing.

In that sense this production $aAb \rightarrow axBCyb$ is context-sensitive. In the generation process there is some large arbitrary string and it turns out that there are some other symbols which actually are around ‘A’. ‘A’ appears in a context which does not contain small ‘a’ and small ‘b’ on either side. Then A appears in a context in which this rule cannot be applied. So, a context specifies a certain minimal shell within which that element should appear. In the case of our context-free grammar you are implicitly specifying that the context in which it should appear is epsilon on both sides.

[Refer Slide Time: 32:10]



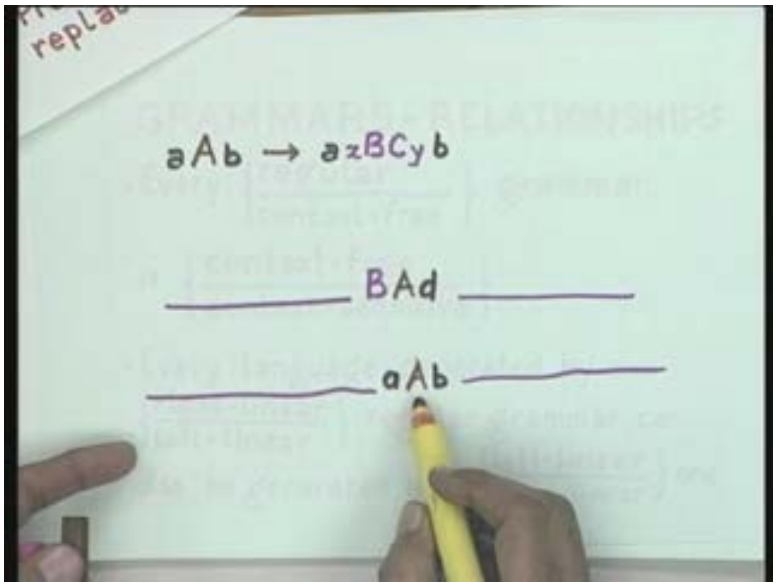
In the case of our context-free grammar we are implicitly specifying that if S appears in an empty context which means that you do not care what appears on either side of S then I can replace S by A. So, a context really specifies the smallest kernel around that symbol which we should satisfy and regardless of what the string is in the generation process, if it is of the form ‘a’A b and regardless of what else occurs in the string, this A is a candidate for replacement by this rule. Context-sensitive grammars are actually more general in the sense that this padding of ‘a’ and ‘b’ is also perhaps not necessary. You might get rid of small ‘a’ and small ‘b’ too. Let us not worry about it because that is the intuitive meaning of context-sensitiveness. You are specifying some minimal padding around that non terminal symbol which will enable a rule to be applied and in the case of a context-free grammar the minimal padding is nothing.

All those rules can be thought of as rules in the context in which on both sides of the padding then the minimal padding that you require is the empty string which means that it does not matter what the rest of the string is in the generation process. Regardless of what the rest of the string is, you can apply the production. As far as programming languages are concerned there are practical reasons why we do not take their context sensitivity into account. It is much simpler to deal with the programming language as generated by a context-free grammar and deal with context-sensitive aspects later on in the process of compilation. A typical context-sensitive feature even in languages like PASCAL is that no variable can appear in a statement unless there is a declaration of that variable. If you have a context-free grammar for PASCAL, it will fail to check on these context-sensitive issues.

Undeclared variables can appear in your program if you just go by a context-freeness but there are no efficient algorithms to recognize or parse context-sensitive languages represented as context-sensitive grammars. We have efficient algorithms to recognize and

parse context-free grammars. If you take context-sensitive grammars then you are not likely to get a linear algorithm. There are no linear algorithms available for parsing context-sensitive grammars so many people usually specify it as a context-free grammar and later as part of the semantics, specify its context-sensitive aspects also. Many people in fact consider context-sensitive to be synonymous with the semantics of the language but that is not quite true.

[Refer Slide Time: 37:10]

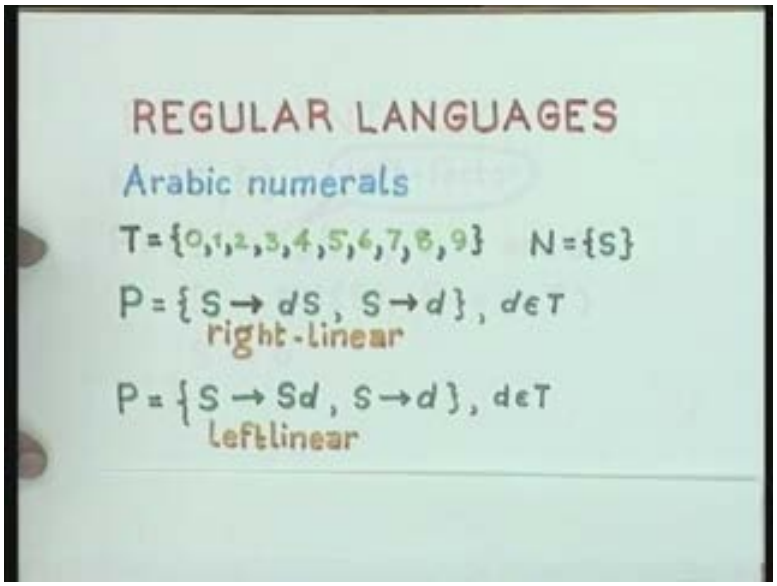


Every regular grammar is context-free; every context-free grammar is context-sensitive and every language generated by a right linear regular grammar can also be generated by a left linear regular one. Every language generated by a left linear one can also be generated by a right linear one. Supposing you have a grammar which is regular but not necessarily right linear or left linear, then some of the rules might be right linear and some of the rules may be left linear. Such grammars can also be always converted into either purely right linear ones or purely left linear ones. In fact that conversion helps us to design machines for recognizing languages of these grammars.

When we are looking at languages we would say that the language is regular if there exists a regular grammar which generates it. Similarly a language is context-free if there exists a context-free grammar which generates it and a language is context-sensitive if there exists a context-sensitive grammar which generates it.

It is possible that for some language you have generated a context-free grammar. It is context-free but the language could still be regular. Similarly, it is possible that you have written context-sensitive grammar for a language which actually could be context-free in the sense that you might be able to come out with a grammar which is purely context-free. Let us just look at a few small examples of regular languages.

[Refer Slide Time: 39:55]



Let us consider our Arabic numerals. We must remember it is one matter to design a grammar and ask what the language generated by the grammar is. It is a different matter to take an existing language and try to define a grammar for it. The numerals were known long before any grammar was defined for them. Actually a large number of numerals comes from the notion of a grammar in natural language and Sanskrit grammar for example was always a neat and rigorous art form. That is one of the reasons perhaps that such a neat notation for numbers was evolved. If you take the Arabic numerals, the terminal set is the set of all possible digits that you have. I am considering representation in decimal which is $0\ldots9 T = \{0,1,2,3,4,5,6,7,8,9\} N = \{S\}$. I require just one non terminal symbol S and I have just these following productions.
$P = \{S \to dS, S \to d\}, d \in T$.

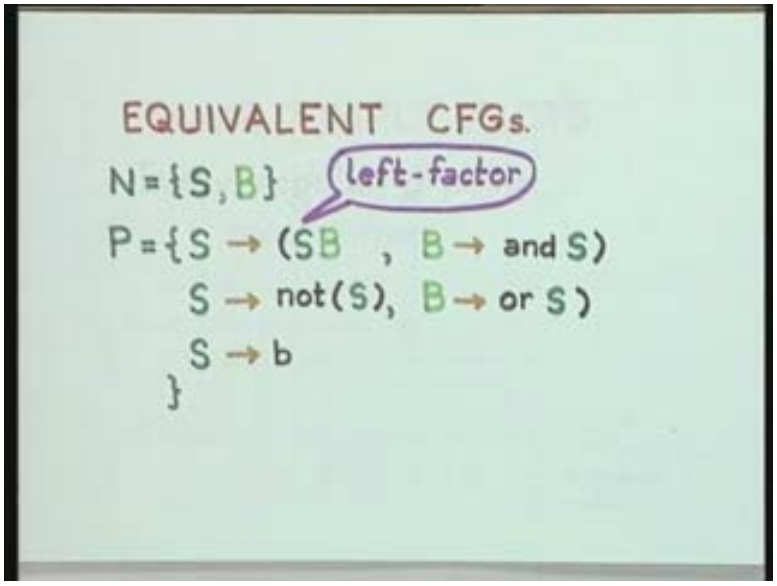
It is a nice and simple grammar that is a right linear grammar and the corresponding left linear grammar which is equivalent to it is $P = \{S \to Sd, S \to d\}, d \in T$. Firstly, it is not at all fully clear what the terminal set is in the sense that the Romans never considered numbers beyond a few tens of thousands.

The Romans had a pattern in the sense that they had symbols and they assumed that they had symbols also for (10,000), (50,000), (100,000), (500,000) but if you continue that pattern then you would require an infinite set of terminal symbols. Supposing you did have an infinite set of terminal symbols then your condition for being a grammar already is violated but supposing you could have an infinite set of terminal symbols then the numerals were written in a very context-sensitive way. For example; an x cannot precede a c, an x cannot precede 500.

It is very context-sensitive in the sense that the roman numerals are not as simple as this. A regular language is implicitly a very simple object. It is easy to see what language it

generates. Very often it is easy to construct a grammar for it and these two grammars are really equivalent. Remember that our ultimate aim is to represent languages somehow in a finitary fashion. The criterion for saying that two grammars are equivalent is that they should have the same terminal set. They need not have the same non terminal set, they need not have the same production set but the language they generate should be the same. The language that we have previewed for the context-free language that we have previously given here is an equivalent context-free grammar.

[Refer Slide Time: 45:24]



For example; I have just factored out the open parenthesis and the S and introduced a new non terminal symbol called B for which I have two rules:

$N = \{S, B\}$

$P = \{S \rightarrow (SB, B \rightarrow andS)$

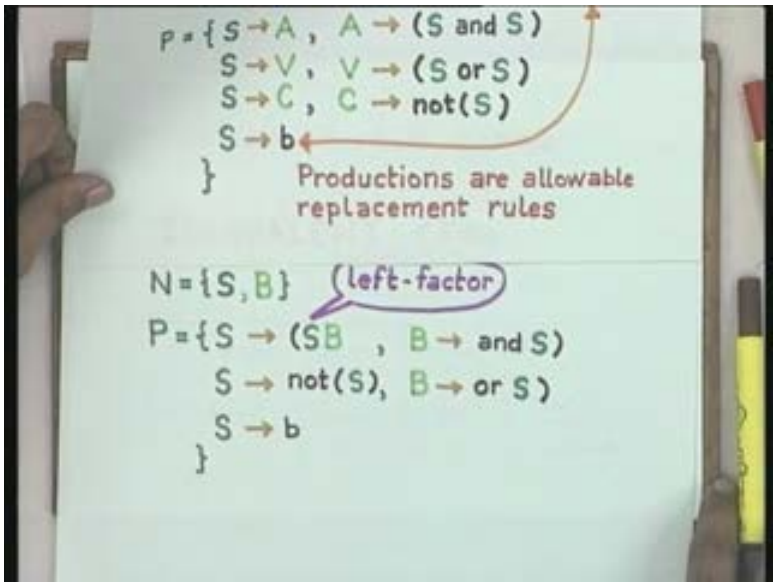
$S \rightarrow not(S), B \rightarrow orS)$

$S \rightarrow b$

}

I have gotten rid of the non terminal symbols ‘a’ and ‘b’ and I had gotten rid of c because it was directly written but I could have had c by choice. So let us quickly go through that grammar.

[Refer Slide Time: 46:04]



Here was the original grammar:

$P = \{S \to A, A \to (SandS)$

$S \to V, V \to (SorS)$

$S \to C, C \to Not(S)$

$S \to b$

$\}$

Here is the equivalent grammar:

$N = \{S, B\}$

$P = \{S \to (SB, B \to andS)$

$S \to not(S), B \to orS)$

$S \to b$

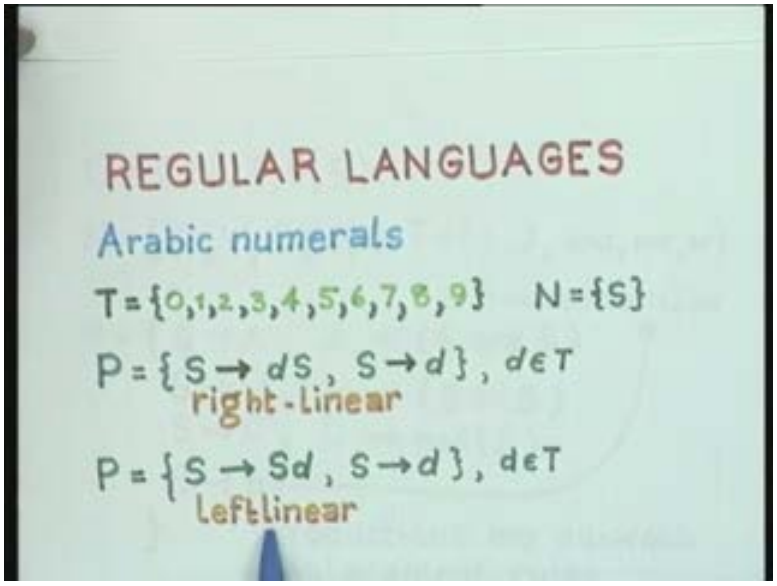
$\}$

We have taken the fact that there is a common occurrence of left parenthesis followed by S. We have factored that out. An equivalent way of writing this grammar is to use a new symbol S. Let us say S -> D and let D produce this string. The elimination of C was just to make a grammar smaller to reduce the number of non terminals. It is important to reduce the number of non terminals because your parsing of the language really depends on how many non terminal symbols there are.

For the same language you might have a variety of grammars. It is a matter of decision making to choose the right kind of grammar which generates that language and the criteria for choosing a grammar are that firstly, the grammar should not be so complicated that it is impossible to parse the language and preferably it should be a context-free grammar. The number of non terminals must be kept low and another

important constraint is that it should facilitate an easy explanation of the semantics of the language.

[Refer Slide Time: 48:27]



In fact the left linear and the right linear numerals have that difference and that is they are both equivalent in terms of actual generation but the fact is that it is easier to specify semantics for the right linear one rather than the left linear one and certain parsing algorithms actually will choose the right linear over the left linear because there is an inherent constraint in those parsing algorithms. The inherent constraint has to do with recursive calls. The recursive calls in the left linear case $P = \{S \to Sd, S \to d\}, d \in T$ could lead to an infinite recursion whereas in the right linear case, $P = \{S \to dS, S \to d\}, d \in T$ they would lead to a finite recursion based on a look ahead.