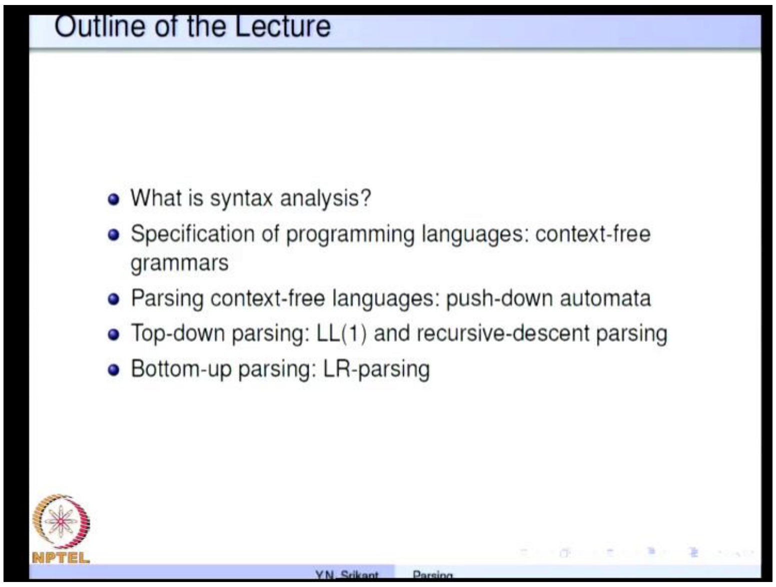
Principle of Complier Design

Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 5
Syntax Analysis: Context-free Grammars, Pushdown Automata and Parsing Part-1

Welcome to the lectures on syntax analysis. So, in this sequence of lectures, we will learn about context-free grammars, pushdown automata, and parsing.

(Refer Slide Time: 00:28)



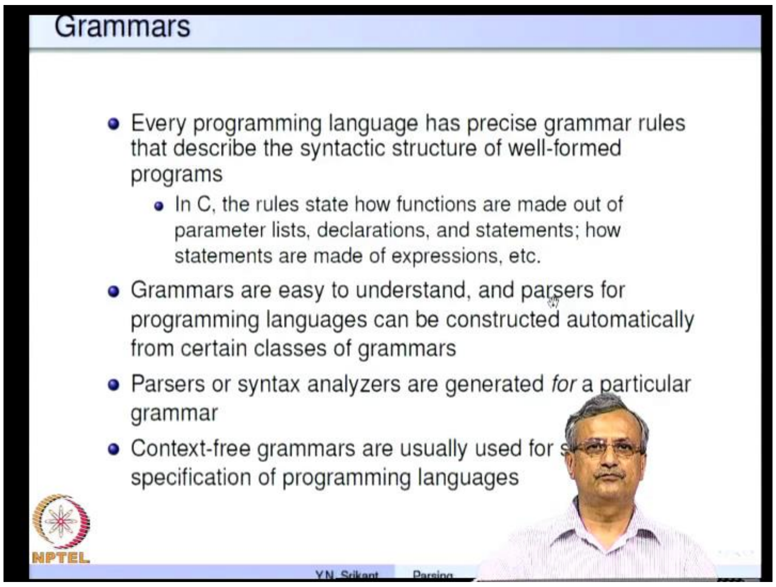
Outline of the Lecture

- What is syntax analysis?
- Specification of programming languages: context-free grammars
- Parsing context-free languages: push-down automata
- Top-down parsing: LL(1) and recursive-descent parsing
- Bottom-up parsing: LR-parsing

Y.N. Srikant Parsing

So, we will understand what exactly is syntax analysis and then study context-free grammars, which are the basis for specification of programming languages. Parsing context-free languages is based on push-down automata just like regular language recognition was based on finite-state automata. We will study two types of parsing; one is the top-down parsing, the other is the bottom-up parsing. So, in top-down parsing, we will study LL(1) and recursive-descent parsing techniques. And in bottom-up parsing, we will study LR-parsing techniques. And there is also a tool called yacc, which is based on LR parsing. We will see examples of how to use it for parser construction.

(Refer Slide Time: 01:19)



Grammars

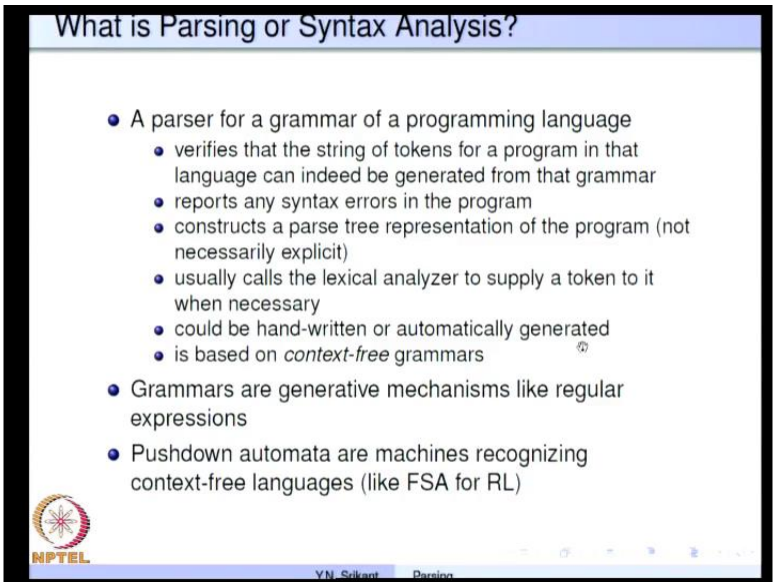
- Every programming language has precise grammar rules that describe the syntactic structure of well-formed programs
  - In C, the rules state how functions are made out of parameter lists, declarations, and statements; how statements are made of expressions, etc.
- Grammars are easy to understand, and parsers for programming languages can be constructed automatically from certain classes of grammars
- Parsers or syntax analyzers are generated *for* a particular grammar
- Context-free grammars are usually used for s specification of programming languages

So, what exactly are grammars? So, every programming language has to be described very precisely. And a grammar is used for describing the syntax of the programming languages. So, for example, if you consider a language such as C or Pascal, a grammar can be written to describe the syntactic structure of well-formed programs – correct programs. When we say correct, we do not mean correctness at run time, but correctness as far as the syntax is concerned. The grammar rules for such a grammar state how functions are made out of parameter lists, declarations and statements; and in turn, they will also say how statements are made up of expressions; and in turn, how expressions are made up of numbers, names, parenthesis, etcetera. Grammars are very easy to understand as we will see. And parsers for programming languages can be automatically constructed from grammar specification of certain types of grammars. Not all grammars can be used for automatic construction of parses, but certain types can be.

And, it is important to note that, parsers or syntax analyzers are generated for a particular grammar. So, in other words, I told you that, there is a tool called yacc, which will be used for generation of parsers automatically. So, we input a particular grammar and then outcomes a parser, which checks sentences based on that particular grammar. For a different grammar, we need to generate a parser all over again. But, if there are couple of grammars available, different grammars available for a particular language; based on the restrictions placed in the tools, one of those grammars may satisfy the restrictions and that can be chosen for parser generation. It really does not matter which grammar shows in as long as the restrictions of the generators are met.

And, context-free grammars as I already said, they are usually used for syntax specification of programming languages. So, context-free grammars are subclass of programming languages. So, I told you during lexical analysis, there are different types of languages and grammars. So, there are regular languages, context-free languages, context-sensitive languages and type-0 languages. So, context-free grammars are used to specify context-free languages. And these are the most useful for programming language purposes.

(Refer Slide Time: 04:19)



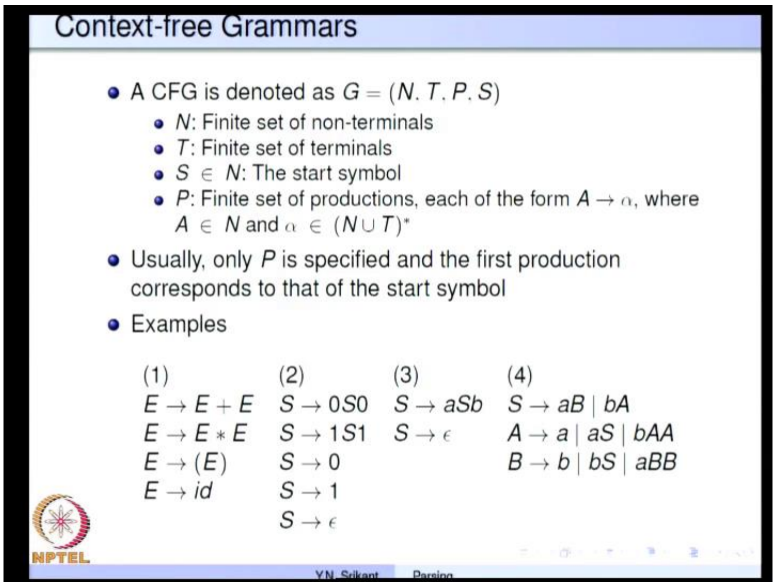
What is Parsing or Syntax Analysis?

- A parser for a grammar of a programming language
  - verifies that the string of tokens for a program in that language can indeed be generated from that grammar
  - reports any syntax errors in the program
  - constructs a parse tree representation of the program (not necessarily explicit)
  - usually calls the lexical analyzer to supply a token to it when necessary
  - could be hand-written or automatically generated
  - is based on *context-free* grammars
- Grammars are generative mechanisms like regular expressions
- Pushdown automata are machines recognizing context-free languages (like FSA for RL)

Y.N. Srikant Parsing

What exactly is parsing or syntax analysis? So, what does the parser do? You have given a programming language; we wrote a grammar for it; and then let us say we also wrote a parser based on it; or, we generated a parser based on this grammar. So, it vary… The parser verifies that, the string of tokens for a program in that particular programming language can indeed be generated from the grammar that, we have provided as a basis for the parser. Tokens are nothing but the entities, which a lexical analyzer carves out of character streams. So, the tokens form a sentence in a particular language and the parser checks whether that sentence is indeed from the language of the parser. It reports any syntax errors as in the case of lexical analyzers. And it constructs a parse tree representation of the program. We will see what parse trees are. But, it is not always necessary to construct a tree explicitly; sometimes it is possible to do without it. It usually calls the lexical analyzer to supply a token whenever it finds that it requires a token to proceed further. Of course, they could be hand-written or automatically generated as well. And our ((Refer Slide Time: 05:54)) are all based on context-free grammars. So, grammars of course, are generative mechanisms; whereas, machines such as finite-state automata or pushdown automata are accepting mechanisms. So, grammars are very similar to regular expressions. So… And pushdown automata are the machines corresponding to context-free languages such as just like FSA or the finite-state automata or the machines corresponding to regular languages.

(Refer Slide Time: 06:30)



Context-free Grammars

- A CFG is denoted as $G = (N, T, P, S)$
  - N: Finite set of non-terminals
  - T: Finite set of terminals
  - $S \in N$: The start symbol
  - P: Finite set of productions, each of the form $A \to \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$
- Usually, only P is specified and the first production corresponds to that of the start symbol
- Examples

(1)	(2)	(3)	(4)
$E \to E + E$	$S \to 0S0$	$S \to aSb$	$S \to aB \mid bA$
$E \to E * E$	$S \to 1S1$	$S \to \epsilon$	$A \to a \mid aS \mid bAA$
$E \to (E)$	$S \to 0$		$B \to b \mid bS \mid aBB$
$E \to id$	$S \to 1$		
	$S \to \epsilon$		

Y.N. Srikant Parsing

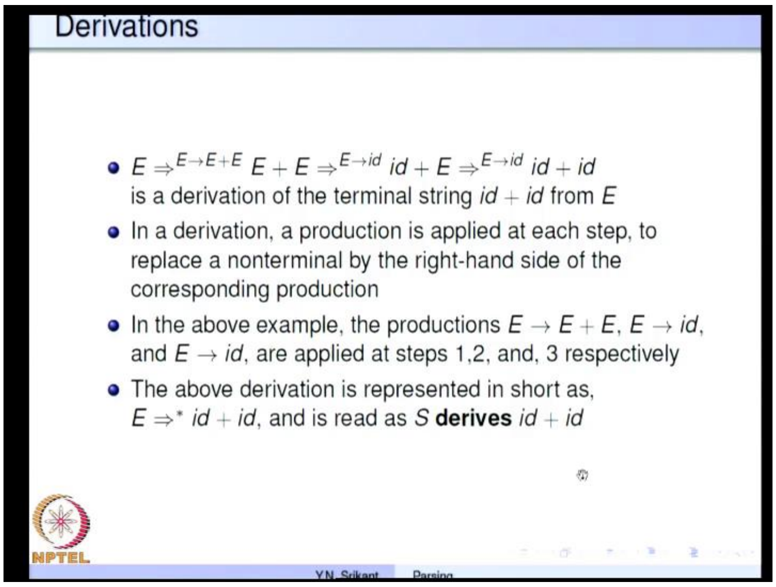
So, let us define a context-free grammar. So, a context-free grammar is denoted as G equal to a quadruple N comma T comma P comma S; where, N is a finite set of what are known as non-terminals or variables; T is a finite set of what are known as terminals. These are the terminals I mentioned in lexical analysis would correspond to the tokens. So, tokens of a lexical analyzer are the terminals of a context-free grammar. S is a non-terminal, which is special; it is a start symbol; and P is a finite set of productions. And for context-free grammars, all the productions are of the form A arrow alpha ((Refer Slide Time: 07:17)) going to alpha; that is how it is read; where, A is a non-terminal and alpha is a string – combination made up of a both non-terminals and terminals. So, N union T star. So, usually, whenever there is no requirement, we do not mention all the N, T, P, S components separately; but, we just provide P. Assume that, the first production ensures the start symbol on its left-hand side. So, let us take some examples.

The first example is E to E plus E; E to E star E; E to parenthesis E, and to id. So, these are the four productions corresponding to the grammar. Here there is exactly one non-terminal. The bold face symbols are all non-terminals and lower case symbols are all terminal symbols in our examples here. So, E is the only bold face symbol; and that corresponds to the single non-terminal. And then terminals are plus star, the left parenthesis, right parenthesis and the id. So, 1, 2, 3, 4, 5 terminal symbols; one non-terminal. These four rules correspond to the productions. These… As I said, the first production shows the start symbol on the left-hand side.

So, E is the start symbol. So, let us later understand what exactly this particular grammar generates. But, for the present, let us move on with other examples and understand other possibilities of specification. Second example also, there is exactly one symbol, which is the non-terminal, that is, the S. The terminal symbols are 0 and 1. Epsilon as usual is the null string, but it is neither a non-terminal or a terminal; it is the empty string. But, to some extent, informally, we can say it is a terminal symbol, because it does not generate any more symbols from it. So, it cannot be a non-terminal.

The third one has two productions: S going to aSB and S going to epsilon. And again here S is the non-terminal; a and b are the terminal strings. For the fourth one, the notation is slightly different. So, the first production starts with S. So, this is the start symbol. The non-terminals are S, A and B. These are the only ones, which are mentioned all over. Little a and little b are the terminal symbols. So, this vertical line are the bar, which is present between aB and bA; indicates that, these are the two productions S going aB and S going to bA. This is a short hand for writing productions with the same left-hand side non-terminal. So, this would be A going to a or A going aS or a going BAA; and this would be B going to b or B going to bS or B going to aBB. So, even here I could have written the four productions as A going to E plus E bar E star E bar parenthesis E parenthesis bar Id. And even this could have been written in a single line separated by bars; the same is true for three as well. It is just a question of making it convenient to read.

(Refer Slide Time: 11:03)



Derivations

- $E \Rightarrow^{E \to E+E} E + E \Rightarrow^{E \to id} id + E \Rightarrow^{E \to id} id + id$ is a derivation of the terminal string $id + id$ from E
- In a derivation, a production is applied at each step, to replace a nonterminal by the right-hand side of the corresponding production
- In the above example, the productions $E \to E + E$, $E \to id$, and $E \to id$, are applied at steps 1,2, and, 3 respectively
- The above derivation is represented in short as, $E \Rightarrow^* id + id$, and is read as S **derives** $id + id$

Y.N. Srikant Parsing

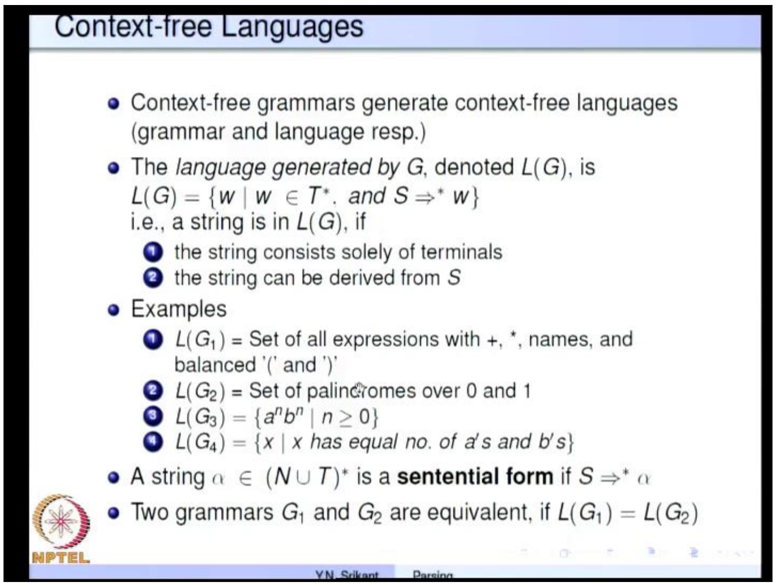
Now, let us move on to a very important concept known as a derivation. This is necessary to understand what exactly is a sentence derived by a grammar. So, here is an example E; then we have an arrow; and there is a production written here E to E plus E; and there are three symbols E plus E written here. Similarly, again E to the arrow E to Id and Id plus E. Finally, another arrow E to Id and Id plus Id. So, this is a derivation. So, as you can see, the first symbol is the start symbol of this particular grammar. So, now, we are looking at this grammar and derivations from this particular grammar. The last three symbols Id plus Id are the terminal symbols. So, if Id plus Id is regarded as a terminal string, then this sequence of steps that we have shown here corresponds to a derivation of the terminal string Id plus Id from the non-terminal or the start symbol E.

So, what is written here is this symbol; the arrow – big arrow corresponds to derivation. So, we read this as E derives E plus E. And the production, which is written here is used to tell the reader that, the derivation uses the production E to E plus E. So, the way this works as the same is true here, the production used is E to Id; and E derives Id plus E. And again E plus E derives Id plus E; and Id plus E derives Id plus Id. So, at each of these steps, there is the production, which is used; and it is noted at the top of this arrow symbol.

The way the derivation works; here is a non-terminal E. And we know that, there are four possibilities for the non-terminal E. There are four productions, which start with E on the left-hand side. So, to derive any particular string, we could choose any of these, which one is appropriate depends on the terminal string that we want to derive. In this case, we want to derive Id plus Id. Therefore, the only production, which has plus in it is E to E plus E. So, let us try out that particular production. So, this symbol E is now replaced by the right-hand side of its production namely, E plus E. That is the reason why we have written E plus E here. So, this E plus E is known as a sentential form. So, in this sentential form, this is an intermediate form before we reach Id plus Id. We can choose to replace this left E or the right E – any one of them and in any particular order. It really does not matter; order does not matter.

So, let us assume that, we are replacing the left E by an appropriate right-hand side of a production. Again, there are four possibilities. And since we are only looking at a particular string known as Id plus Id, we do not want to derive any stars or parenthesis. So, we choose the production E going to Id for that purpose. So, the left E is now replaced by the right-hand side of the production E going to Id. So, this becomes Id. And the rest of the sentential form remains; that is, plus E. In the third step, Id and plus cannot be defined further. There is nothing to do. They are already terminal symbols. And this E can be replaced by another Id. And in that process, we use the production E to Id. So, E is replaced by the right-hand side of the production E to Id; and this becomes Id plus Id. So, this entire process I just now described gives you the string Id plus Id starting from the start symbol E. So, we write this process very concisely as E derives star say 0 or more steps; E derives Id plus Id. So, this is a derivation of the string Id plus Id from E.

(Refer Slide Time: 15:37)



Context-free Languages

- Context-free grammars generate context-free languages (grammar and language resp.)
- The *language generated by* G, denoted $L(G)$, is $L(G) = \{w \mid w \in T^*, \text{ and } S \Rightarrow^* w\}$
  i.e., a string is in $L(G)$, if
  1. the string consists solely of terminals
  2. the string can be derived from S
- Examples
  1. $L(G_1)$ = Set of all expressions with +, *, names, and balanced '(' and ')'
  2. $L(G_2)$ = Set of palindromes over 0 and 1
  3. $L(G_3) = \{a^n b^n \mid n \geq 0\}$
  4. $L(G_4) = \{x \mid x \text{ has equal no. of } a's \text{ and } b's\}$
- A string $\alpha \in (N \cup T)^*$ is a **sentential form** if $S \Rightarrow^* \alpha$
- Two grammars G_1 and G_2 are equivalent, if $L(G_1) = L(G_2)$

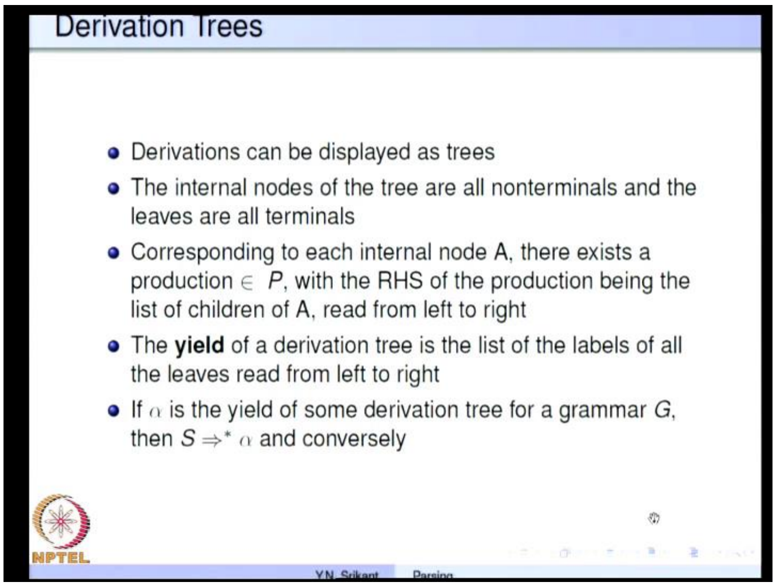
Y.N. Srikant Parsing

So, once we understand how to derive strings from the start symbol, we are also ready to define the language, which is generated by a context-free grammar. And then we will take up more examples of the grammars and the further derivation trees and so on. So, here context-free languages are specified by context-free grammars. And context-free grammars are set to generate context-free languages. So, how do… Once we are given a grammar G, the language generated by that grammar is denoted as L of G. Just as for regular expressions, we wrote L of R, here we write L of G.

So, this is a set of strings w. w is a set of strings. So, it has no non-terminals in it. So, T star; T is a set of terminals; and T star is its closure. So, w is in that closure of T star. So, that means it is a string of terminal symbols. And we just described the derivation process. So, S derives w; S is a start symbol. So, all those strings, which can be derived from the start symbol and belong to the terminal string set star, that is, a closure of the terminal string, is actually in the language L of G. So, this is how we describe, rather define the language generated by a grammar.

So, for the first example, which had these four productions; as you can see, this language corresponds to arithmetic expressions with plus star parenthesis and Id. And then the second language is the set of palindromes of over 0 and 1. The third language is a n b n with n greater than or equal to 0. So, let us look at that. So, here we have 0 S 0 or 1 S 1 or 0 or 1 or epsilon being generated by S. So, as you can see the number of 0's and number of 1’s are balanced on both sides of s. So, once a 0 is on the left side, there is a 0 on the right side as well; that means this can generate palindromes. Here this generates any number of a’s and any number of b’s, which are equal in number on both sides of S. And therefore, easy to see that, it generates a n B n. And the fourth one is not so intuitive. So, it generates strings with equal number of a’s and b’s. So, you can actually check out a couple of reservations and make sure that you understand the operation of this particular grammar. So, a string alpha, which is a combination of both non-terminals and terminals is a sentential form. So, I mentioned this already; so if S derives alpha. So, the difference between a sentential form and a sentence is the sentence has only terminal symbols; whereas, the sentential form has both non-terminals and terminals. And when are the two grammars G 1 and G 2 equivalent? Only if their languages are exactly identical.

(Refer Slide Time: 19:18)

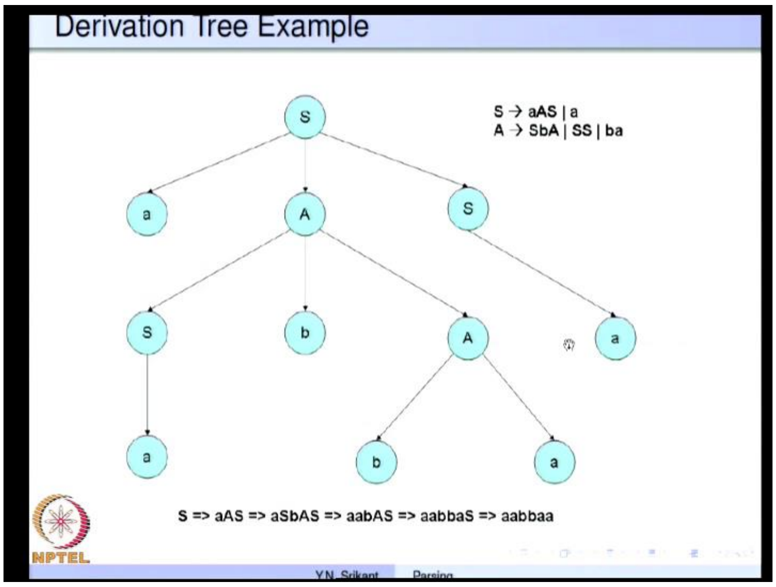


Derivation Trees

- Derivations can be displayed as trees
- The internal nodes of the tree are all nonterminals and the leaves are all terminals
- Corresponding to each internal node A, there exists a production $\in P$, with the RHS of the production being the list of children of A, read from left to right
- The **yield** of a derivation tree is the list of the labels of all the leaves read from left to right
- If α is the yield of some derivation tree for a grammar G, then $S \Rightarrow^* \alpha$ and conversely

So, let us move on and understand the concept of derivation trees before we take up more examples of derivation of strings.

(Refer Slide Time: 19:32)



Derivation Tree Example

S → aAS | a
A → SbA | SS | ba

S => aAS => aSbAS => aabAS => aabbaS => aabbaa

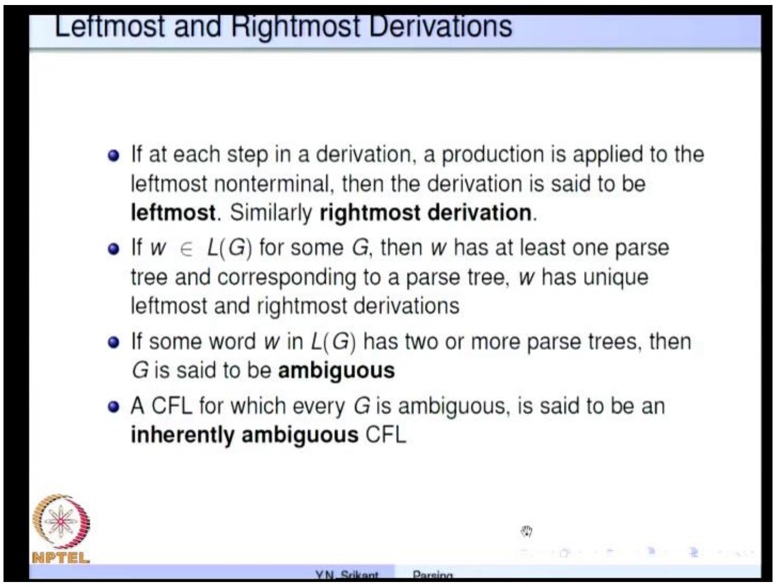
So, let me first show you a picture and with an example, and then go back to the definition. So, here is a simple context-free grammar S going to aAS or a; A going to SbA or SS or ba. And let us consider the string aabb… – aabb and then aa right. So, here is a derivation of the string aabbaa from the start symbol S. So, S… We first use the production S going to aAS. So, we get the sentential form aAS. Now, the first a is expanded by another production. So, a going to S b A. So, we get little a, then S b A, then followed by S. Now, we expand the S that was recently acquired. So, we apply S going to A and we get aab capital A capital S. Now, it is time to expand the capital A. So, capital A goes to you know ba and we get aabbaa and a capital S. Finally, capital S gets the last a. So, this is the derivation of the string from its start symbol.

So, a derivation tree really shows which productions were applied with which points and how the string was derived from the start symbol. So, the first production applied was S going to aAS. So, a small tree is shown here with S as the root; little a, capital A and capital S being the three children. And then I said the capital A is expanded further with the production A going to SbA. So, again the same structure with the three children is present here as well, but the symbols are different. This A gives rise to A. So, the production S going to A was applied here. This A gives rise to b a; and the production A going b a was applied here. Finally, this S gives rise to a and the production S going to a was applied here. So, the structure of the derivation tree is very simple at any internal node. So, this S, this A and this A, and this S are all internal nodes; whereas, these are the leaves – a – this a, this b, this b, this a and this a. These are the leaves. So, all the internal nodes are non-terminals and all the leaves are terminal symbols. And every non-terminal node corresponds to a production, which is applied there; and the right-hand side of production will be represented as the children. The nodes corresponding to the right-hand side will represent the children of this particular node. So, that is true here. And we cannot develop the parse tree further once we reach the terminal symbols. So, derivations can be displayed as trees. So, I already showed that.

And now, there is another important property here. If we have actually expanded in this derivation sequence, we always chose to expand the leftmost non-terminal. But, we could have easily expanded any non-terminal, which in this particular form. So, for example, here we could have expanded capital S, capital A or capital S. And finally, after we exhaust all the non-terminal symbols, the terminal string would possibly be same. Unless we have used a different production to expand a particular non-terminal, to get a particular string, the same productions will have to be applied at various places during the derivation. And when you apply the production is really immaterial. So, the internal nodes of the tree are all non-terminals and the leaves are all terminal symbols.

And, as I already said, corresponding to each internal node A, there exists a production in P, with the right-hand side of the production being the list of the children of A read from left to right. And the yield of a derivation tree is the list of all the labels of the leaves from left to right. And finally, if alpha is the yield of some derivation tree, S derives alpha. And if S derives alpha, then alpha is the yield of a derivation tree for some grammar G. So, here the yield is this a – concatenated with this a, this b, this b, this a and this a. And S derives this aabbaa. And we have displayed a derivation tree here. And for this derivation tree, this is the yield.

(Refer Slide Time: 24:52)


Leftmost and Rightmost Derivations

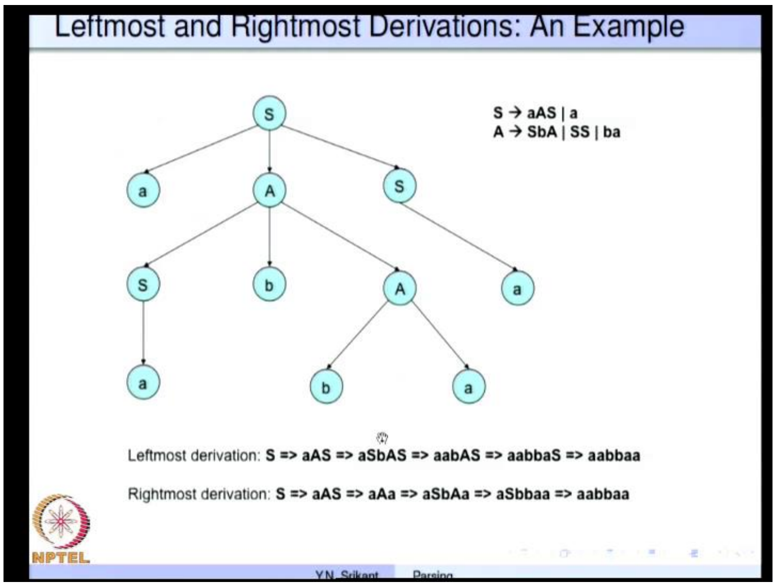
- If at each step in a derivation, a production is applied to the leftmost nonterminal, then the derivation is said to be **leftmost**. Similarly **rightmost derivation**.
- If $w \in L(G)$ for some G, then w has at least one parse tree and corresponding to a parse tree, w has unique leftmost and rightmost derivations
- If some word w in $L(G)$ has two or more parse trees, then G is said to be **ambiguous**
- A CFL for which every G is ambiguous, is said to be an **inherently ambiguous** CFL

Y.N. Srikant Parsing

So, as I said, there are many ways to derive strings; many derivations are possible. So, if we expand the leftmost non-terminal in a derivation at each step, then it is called a leftmost derivation. And if we choose to expand the rightmost non-terminal first at every step, then it corresponds to the rightmost derivation. And if there is a string w, which is generated by the grammar or it belongs to the language of the grammar, then w has at least one parse tree. And corresponding to a parse tree, w has unique leftmost and rightmost derivations. So, this is possible only if the grammar is not ambiguous.

And, if the grammar has ambiguity, then the next bullet already says that a word has two or more parse trees, then the grammar is ambiguous. So, if there is a unique parse tree, then the grammar is unambiguous; and if there is more than one parse tree for the same string, the grammar is ambiguous. And if there is a unique parse tree, then w has unique leftmost and rightmost derivations for that particular parse tree. Of course, if there are many parse trees, for each of these parse trees, there would be a unique leftmost and rightmost derivation as well. So, a context-free language for which every G is ambiguous is what is known as an inherently ambiguous language; but, we are not so particular about this language – inherently ambiguous variety, because it is not of much use to us.

(Refer Slide Time: 26:46)



Leftmost and Rightmost Derivations: An Example

S → aAS | a
A → SbA | SS | ba

$$\begin{tikzcd}[column sep=small] & & S \arrow[dll] \arrow[d] \arrow[drr] & & \\ a & & A \arrow[dl] \arrow[d] \arrow[dr] & & S \arrow[d] \\ & S \arrow[d] & b & A \arrow[dl] \arrow[dr] & a \\ & a & b & & a \end{tikzcd}$$

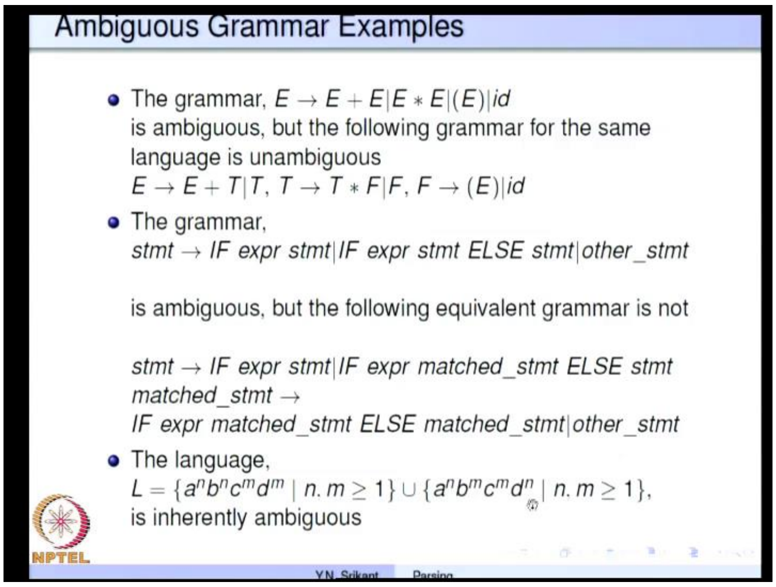
Leftmost derivation: S => aAS => aSbAS => aabAS => aabbaS => aabbaa

Rightmost derivation: S => aAS => aAa => aSbAa => aSbbaa => aabbaa

Y.N. Srikant Parsing

Now, let us understand ambiguity further. First of all, leftmost and rightmost derivations. So, the same grammar and the same derivation tree and the same sentence as well. So, we have seen this derivation already. This is the leftmost derivation. And for the same string, suppose we expand the rightmost non-terminal; so in aAS, we do expand S. So, we get aAa. Here we expand A; there is no option; we get aSbAa. Here we expand the rightmost a. So, we get aSbbaa. And finally, the S is expanded to little a. So, giving us the string aabbaa. So, this is the leftmost and this is the rightmost derivation.

(Refer Slide Time: 27:35)



Ambiguous Grammar Examples

- The grammar, $E \rightarrow E + E | E * E | (E) | id$
  is ambiguous, but the following grammar for the same language is unambiguous
  $E \rightarrow E + T | T$, $T \rightarrow T * F | F$, $F \rightarrow (E) | id$
- The grammar,
  stmt → *IF expr stmt*|*IF expr stmt ELSE stmt*|*other_stmt*

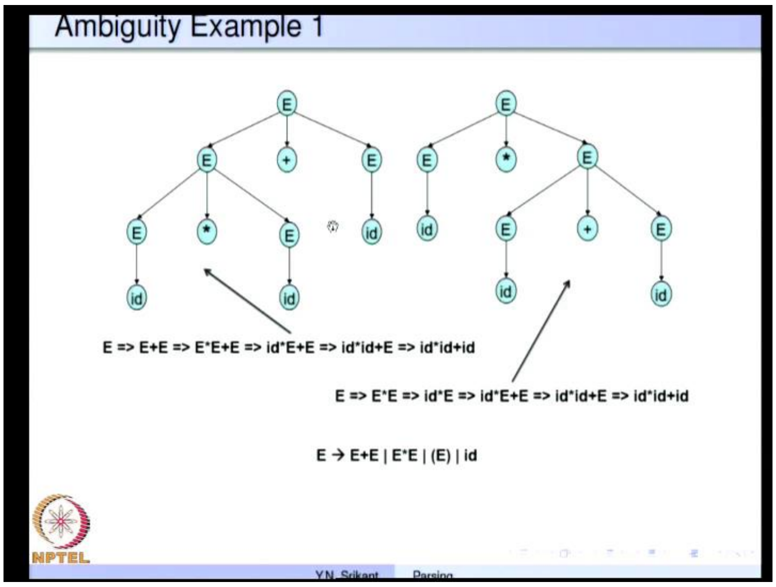
  is ambiguous, but the following equivalent grammar is not

  stmt → *IF expr stmt*|*IF expr matched_stmt ELSE stmt*
  matched_stmt →
  IF expr matched_stmt ELSE matched_stmt|*other_stmt*
- The language,
  $L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$,
  is inherently ambiguous

Y.N. Srikant Parsing

So, let us understand ambiguous grammars, because these are very important to us. If the grammar is ambiguous, then the tools for generating parsers will be in trouble. Further, even our parser techniques will be in trouble. So, we must understand ambiguity appropriately. So, as I already said, ambiguous means there are two parse trees for the same grammar and for the same string. So, for example, the grammar E to E plus E star E parenthesis E parenthesis id – this is ambiguous. But, we can actually design another grammar, which yields the same language, generates the same language – E to E plus T or T; T to T star F or F; F to parenthesis E parenthesis or id. The difference between these two grammars is; for this grammar, plus and star are at the same precedence level; whereas, here star has more precedence than plus. So, let us look at an example to understand this better.

(Refer Slide Time: 28:59)



Ambiguity Example 1

E => E+E => E*E+E => id*E+E => id*id+E => id*id+id

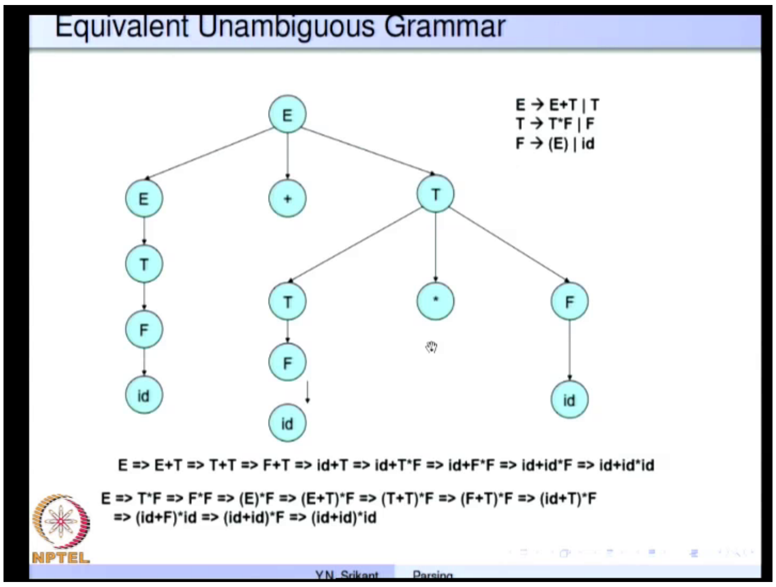
E => E*E => id*E => id*E+E => id*id+E => id*id+id

E → E+E | E*E | (E) | id

So, we have a grammar E to E plus E; E star E; parenthesis E parenthesis id. And let us look at the string id star id plus id. So, you can look at this particular string id star id plus id as the multiplication first and then addition. Or, you can look at it as addition first and then multiplication. And correctly, for each of these interpretations, this is correct according to the… Both interpretations are correct according to the grammar simply because we have not mentioned whether plus has precedence over star or star has precedence over plus. So, let us see what happens in the parse tree representation. So, let us assume that, the structure at the highest level is E plus E. In other words, we first do the star and then we do the plus. So, we have E to E plus E. So, this is a first step in the derivation. Expand the left E. So, it becomes E to E star E plus E. So, E to E plus E. Then this left becomes E to E star E. And these two yield id’s. So, this is one parse tree.

The second parse tree says at the highest level, look at it as E star E. And then the second one is E plus E. So, plus is done first and then the star. So, this corresponds to this particular derivation sequence. So, for each of these parse trees or derivation trees, we can write down the leftmost derivation and the rightmost derivation. That would be unique up to the parse tree. But, for this parse tree, the leftmost and rightmost derivations would be different. You can see that very clearly right here. This is the leftmost derivation for this parse tree; and this is the leftmost derivation for this parse tree. Obviously, these two are very different. It will be the same case with rightmost derivations as well.

(Refer Slide Time: 31:21)



Equivalent Unambiguous Grammar

E → E+T | T
T → T*F | F
F → (E) | id

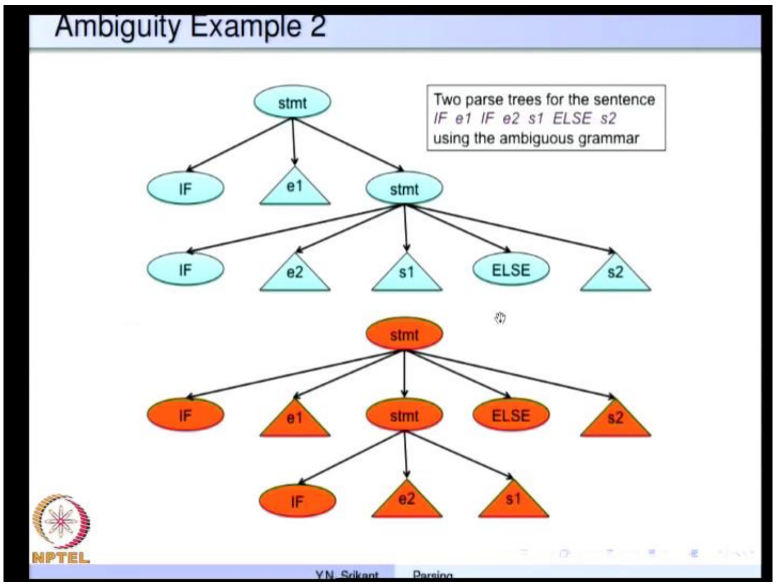
E => E+T => T+T => F+T => id+T => id+T*F => id+F*F => id+id*F => id+id*id

E => T*F => F*F => (E)*F => (E+T)*F => (T+T)*F => (F+T)*F => (id+T)*F
=> (id+F)*id => (id+id)*F => (id+id)*id

So, let us consider the unambiguous grammar equivalent to ambiguous expression grammar that we studied so far. So, this is the unambiguous grammar – E to E plus T or T; T to T star F or F; F to parenthesis E parenthesis or id. And let us take the string id plus id star id. So, let us see what happens in the derivation. So, first of all, E to E plus T is a possibility, because we would look at it as id plus and then id star id. So, id star id gets done first. And then this E gives rise to T, F and id. So, this is the derivation sequence. And this T gives rise to T star F; F becomes id and T becomes F and id. So, now, suppose we try to look at the same string – id plus id star id and try to look at it as star first and then positive – plus. So, that is, let us begin with the production E to T star F – E to T and then it is T star F.

So, I just skipped one step here just to compress the derivation; E to T and then T to T star F; and T star F becomes F star F. But, once you reach this stage, it is imperative that we use the parenthesis E parenthesis as the right-hand side of F in order to generate E, because without E, you cannot get E plus T. But, once we use parenthesis E parenthesis, we have introduced to new symbols: parenthesis left and parenthesis right. So, the string that we finally generate becomes parenthesis id plus id star id, which goes to show that, we have disambiguated the grammar. And whenever you need to do plus between these two and then do a star, we need to include them in parenthesis; otherwise, it would be interpreted as star first and then plus. So, this grammar is indeed unambiguous. And for every string, there is exactly one derivation tree.

(Refer Slide Time: 34:07)



Ambiguity Example 2

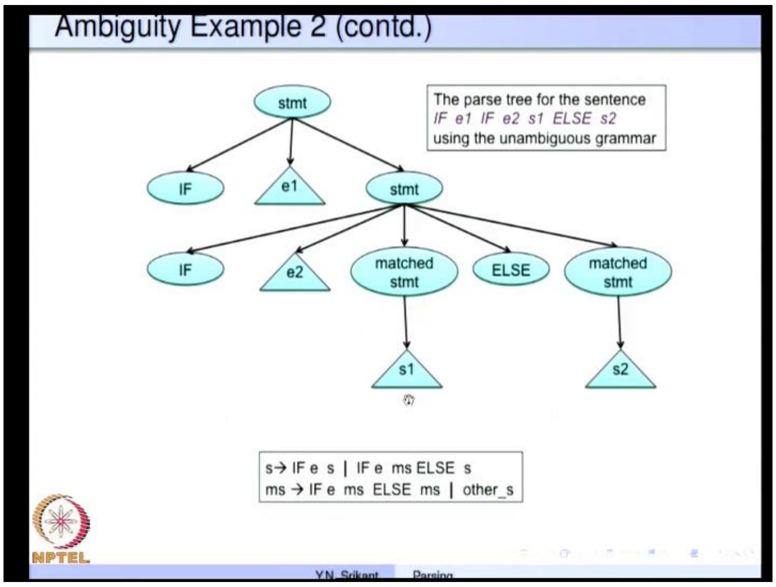
Two parse trees for the sentence
IF e1 IF e2 s1 ELSE s2
using the ambiguous grammar

Another example for ambiguity. Before that, the text form of this is here. So, statement going to IF expression statement or IF expression statement ELSE statement or other statement. So, this is a common piece of grammar generating IF THEN ELSE statement. So, it says IF followed by expression followed by statement; that is the IF THEN statement; otherwise, the other option is IF expression statement, ELSE statement. So, this generates the ELSE part as well; otherwise, any other type of statement. This happens to be ambiguous. But, the second grammar is unambiguous as we will see very soon. Let us finish off this last part. The language is a n b n c m d m with n, m greater than equal to 1 union with a n b m c m d n with n, m greater than equal to 1. So, this language with these two sets in union is inherently ambiguous.

It is very difficult to… It is impossible to write down a grammar, which get us to this L, but is not ambiguous. The problem is some strings are generated by the fall in this category and also in this category. The grammars for this and this are always different. And therefore, it is very difficult to say… It is impossible to rather write down an unambiguous grammar for this particular language. So, let us proceed with our example. So, here is the classic example of the ambiguity in IF THEN ELSE. The statement is IF e IF e2 s1 ELSE s2; very similar to C syntax.

Here this can be seen as either an IF THEN statement with the inner else belonging to the inner statement; that is, IF e2, then s1 ELSE s2 is one unit. And if e1 and the whole thing is another unit. So, we have a statement – IF e statement; and this inner statement is IF e2 s1 ELSE s2. This is one interpretation. We could also say the ELSE belongs to the outer IF. So, we would say IF e1, and then IF e2 s1 is the IF THEN ELSE – IF THEN statement; and the ELSE corresponds to the outer IF. So, the derivation tree for that would be as follows. So, there is a statement IF e1 statement ELSE s2. So, the outer one is the IF THEN ELSE statement and the inner one becomes IF THEN statement. So, both are correct as far as the language is concerned. But, the grammar generates two types of parse trees; and therefore, it is ambiguous.

(Refer Slide Time: 37:27)


Ambiguity Example 2 (contd.)

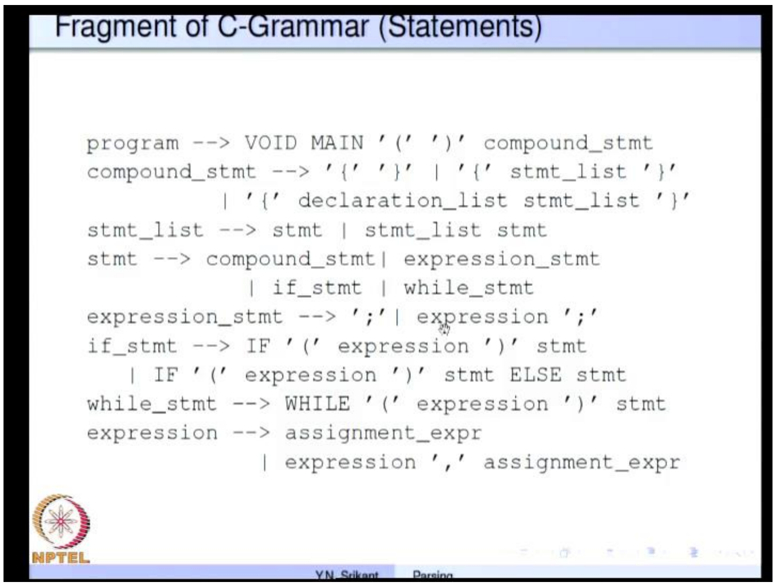
The parse tree for the sentence
IF e1 IF e2 s1 ELSE s2
using the unambiguous grammar

s→ IF e s | IF e ms ELSE s
ms → IF e ms ELSE ms | other_s

So, now, look at the same sentence and let us see how it is handled by the unambiguous version of the same grammar. By the way, let me add that it is impossible to convert ambiguous grammars into unambiguous grammars in any automatic fashion. In certain cases, we will be able to rewrite or write another grammar such as the ones we have shown in which the grammar – second grammar is unambiguous. But, this cannot be automated; this is an undecidable problem. So, here is the unambiguous grammar. So, IF e, then statement, I have shown s as statement and e as expression just to write a little less; or, IF e, then s; otherwise, IF e, then matched statement ELSE statement.

What is a matched statement? It always generates IF THEN ELSE. IF e, then matched statement; ELSE matched statement; otherwise, other statements. So, the beauty of this grammar is the matched statement always generates an IF THEN ELSE matched statement. So, the interpretation that, whether the ELSE belongs to the outer IF or the inner IF is eliminated here. It is always… If you want to generate an ELSE, it must be generated from s here. So, that means the ELSE corresponds to this IF directly. And IF we are not generating any s, ELSE then we can use this IF e then s. But, once we have generated this ms, the ms in turn cannot generate IF THEN statements, because that is the one, which gives rise to ambiguity here. So, it always generates only matched IF THEN ELSE statements. So, in our case, we have a statement here and it generates IF e statement. And this statement in turn generates IF THEN ELSE statement. So, there is no other possibility for this particular rule. It is not possible to say IF e use this, because if we had used this alternative, the outer ELSE would have… But, the inner IF THEN could not have been generated, because ms always generates matched IF THEN ELSE statements. So, this is the only derivation tree that is possible for this particular sentence.

(Refer Slide Time: 40:21)



Fragment of C-Grammar (Statements)

```
program --> VOID MAIN '(' ')' compound_stmt
compound_stmt --> '{' '}' | '{' stmt_list '}'
          | '{' declaration_list stmt_list '}'
stmt_list --> stmt | stmt_list stmt
stmt --> compound_stmt| expression_stmt
           | if_stmt | while_stmt
expression_stmt --> ';'| expression ';'
if_stmt --> IF '(' expression ')' stmt
   | IF '(' expression ')' stmt ELSE stmt
while_stmt --> WHILE '(' expression ')' stmt
expression --> assignment_expr
             | expression ',' assignment_expr
```

Y.N. Srikant Parsing

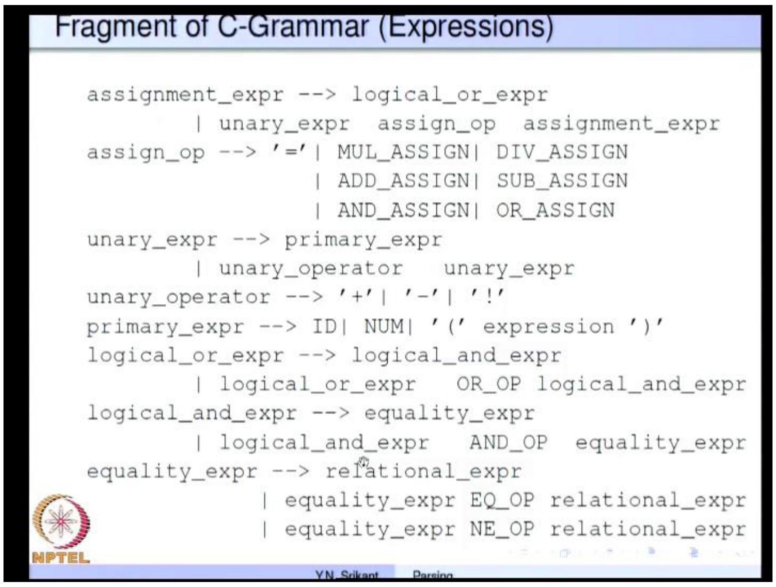
So, let us now take up a fragment of C grammar. So, this is a fairly large example, but still it is a fragment of the entire C grammar; the entire c grammar is much bigger. So, I have chosen very important parts of the language in showing you this grammar. So, a main program is denoted by the non-terminal program. And as we know, this starts with VOID MAIN, then the two parenthesis followed by a body, which can be called as compound statement. A compound statement in general – it could be empty. So, just two flower brackets or it could have flower bracket followed by a list of statements followed by another flower bracket. This is also well-known. Or, we could have flower bracket followed by some local declarations and another set of statements. So, all these three possibilities exist. So, these are the local variables, which hold only within this particular block.

So, what is a statement list? It is either a single statement or a list of statements. So, this mechanism is used to generate as many statements as we want. So, this statement list can go on generating individual statements if we apply it again and again. So, let us say first time if we apply statement list going to statement list followed by statement. So, we have generated one statement. Second time we again apply this to generate a second statement; then we could apply the same production to generate a third and so on. And once we have generated the required number of statements, we can stop with this terminating production statement list going to statement.

And, what is a statement? It is either a compound statement or expression statement or if statement or while statement. There are many other statements in C, but we will confine our attention to just this to show how grammars are written; the others can be added very easily. And we have seen compound statement already. So, let us see what exactly is an expression statement. Expression statement could be just a semicolon, that is, null body or it could be expression followed by semi colon. So, any expression in general in C is a statement as well.

That is the reason why this is expression followed by a semicolon. And in IF statement, expression is expanded later; IF statement – obviously, we have seen the grammar already; IF expression statement or IF expression statement, ELSE statement. This is ambiguous. But, for our purposes, it does not matter, because we already know how to write an unambiguous grammar for this particular syntax. While statement is while expression statement. And now, we come to expression. So, expression has assignment expression or expression comma assignment expression. So, again, this can be used to generate as many assignment expressions as necessary.

(Refer Slide Time: 43:49)



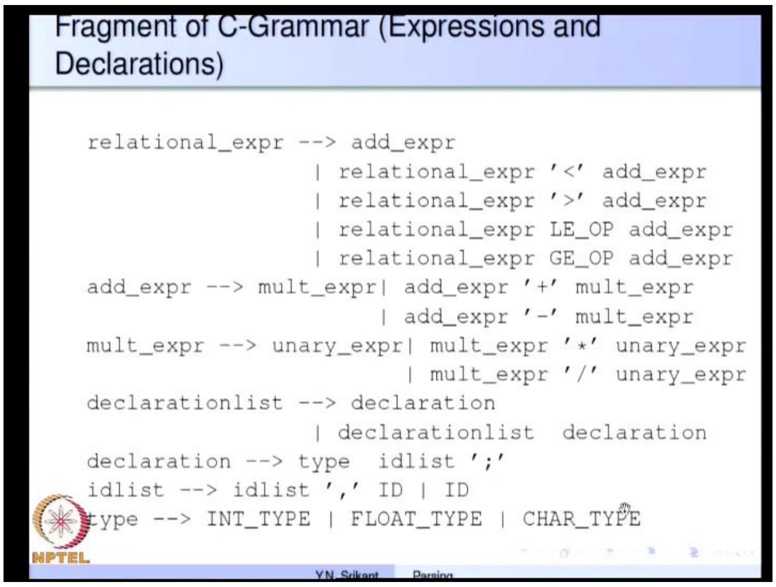
Fragment of C-Grammar (Expressions)

```
assignment_expr --> logical_or_expr
        | unary_expr  assign_op  assignment_expr
assign_op --> '='| MUL_ASSIGN| DIV_ASSIGN
                 | ADD_ASSIGN| SUB_ASSIGN
                 | AND_ASSIGN| OR_ASSIGN
unary_expr --> primary_expr
        | unary_operator   unary_expr
unary_operator --> '+'| '-'| '!'
primary_expr --> ID| NUM| '(' expression ')'
logical_or_expr --> logical_and_expr
        | logical_or_expr   OR_OP logical_and_expr
logical_and_expr --> equality_expr
        | logical_and_expr   AND_OP  equality_expr
equality_expr --> relational_expr
             | equality_expr EQ_OP relational_expr
             | equality_expr NE_OP relational_expr
```

And, assignment expression assures how to break the ambiguity. So, for example, at the highest level or logical or expressions; otherwise, unary expression assign op followed by assignment expression. So, let us say we take this logical or expression. So, the logical or expression then goes to logical and expression; logical and expression goes to equality expression; equality expression goes to relational expressions. Let me go further and then come back. Relational expression goes to add expression; add would be multiply and then star slash plus minus etcetera. So, the… What I wanted to show you is if you write an expression – some expression with A or B etcetera, there is a unique way of parsing it, because this expression grammar happens to be unambiguous. However, in places of or, we could always have used plus and things of that kind. So, that is something that we really cannot avoid so easily; that is a semantic part. So, let us go further. So, assign op is either this assignment or multiply assign star equal to, slash equal to, plus equal to, minus equal to, and equal to, or equal to – these are all assignment operators.

Unary expressions are primary expressions or unary operator followed by unary expression. So, unary operator is plus, minus or not. So, one of these three. So, primary expression is some kind of a terminator; id or num followed by a parenthesis or parenthesis expression. So, logical or expression is used to generate logical or expression or op logical and expression. So, this shows that, or is at the higher level and is at the lower level, and gets higher priority than or. And and expression gives you logical and expression, and op equality expression. So, equality gets higher priority than and op. And equality expression has EQ OP, NEQ OP or relational expression.

(Refer Slide Time: 46:35)



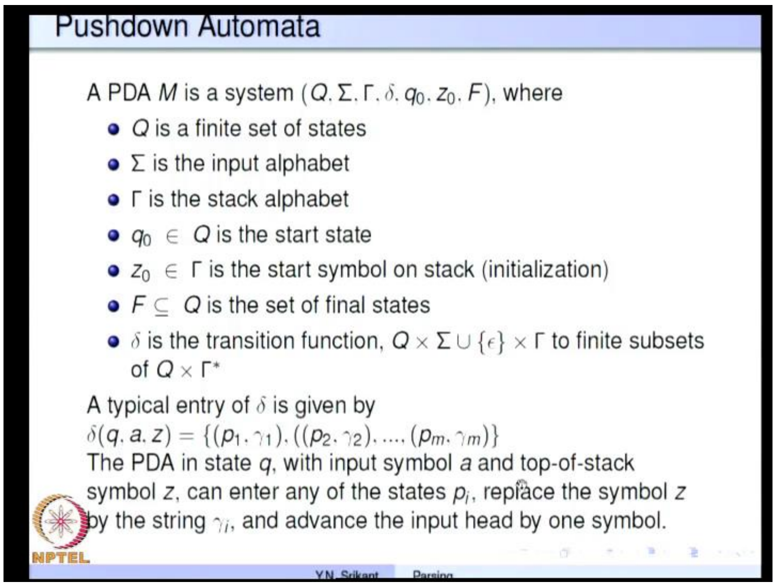
Fragment of C-Grammar (Expressions and Declarations)

```
relational_expr --> add_expr
                 | relational_expr '<' add_expr
                 | relational_expr '>' add_expr
                 | relational_expr LE_OP add_expr
                 | relational_expr GE_OP add_expr
add_expr --> mult_expr| add_expr '+' mult_expr
                      | add_expr '-' mult_expr
mult_expr --> unary_expr| mult_expr '*' unary_expr
                        | mult_expr '/' unary_expr
declarationlist --> declaration
                 | declarationlist  declaration
declaration --> type  idlist ';'
idlist --> idlist ',' ID | ID
type --> INT_TYPE | FLOAT_TYPE | CHAR_TYPE
```

Y.N. Srikant   Parsing

So, these equal to, not equal to operators get lower precedence than relational operators, which are here – less than, greater than, LEQ, GEQ. And here is add expression. So, add expression has plus minus. They get higher priority than these. And multiply expression has star, slash; which get higher priority than plus minus. So, one has to… I just wanted to show that, writing a grammar for fairly large language is a very nontrivial exercise; and one has to pay lot of attention to the precedence of operators and many other details. Then we have declarations, which are generated by declaration list followed by declaration – this part; and this is a terminator part of the declaration list. So, each declaration has type followed by a number of id's. So, id list is used to generate id's. And type can be integer or float or character. Many other possibilities exist, but I just wanted to show you a few samples here.

(Refer Slide Time: 47:41)



Pushdown Automata

A PDA M is a system $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, where

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $q_0 \in Q$ is the start state
- $z_0 \in \Gamma$ is the start symbol on stack (initialization)
- $F \subseteq Q$ is the set of final states
- δ is the transition function, $Q \times \Sigma \cup \{\epsilon\} \times \Gamma$ to finite subsets of $Q \times \Gamma^*$

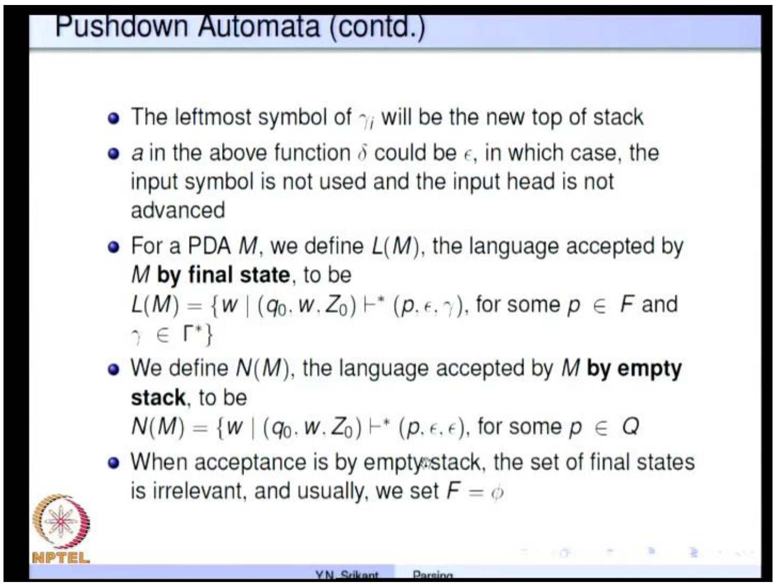
A typical entry of δ is given by
$\delta(q, a, z) = \{(p_1, \gamma_1), ((p_2, \gamma_2), ..., (p_m, \gamma_m)\}$
The PDA in state q, with input symbol a and top-of-stack symbol z, can enter any of the states p_i, replace the symbol z by the string γ_i, and advance the input head by one symbol.

Y.N. Srikant Parsing

So, that is about the grammars; and a large example as well. So, let us now look at some of the machines, which can be used to implement parses. So, this pushdown automata is a machine, which can be automatically derived given a grammar; and it can be used to parse context-free languages, which is preferred by context-free grammars. So, let us understand how it works. This is a stack-base system. It is very similar to a finite-state machine. In a finite-state machine, we had Q, we had sigma; we had delta, we had q naught; and we had F. Here… And the meanings of those are exactly the same. Q is a finite set of states; sigma is the input alphabet; q naught is the start state; F is the set of final states; and delta is the transition function. The meaning of the transition function is different. So, we will come back to that very soon. And here is gamma, which is the stack alphabet.

So, we… As only if there is a stack, the symbols which can be stored on the stack or form the stack alphabet; that is the finite alphabet again. And z naught is the symbol initialized for which the stack is initialized. So, the start symbol on the stack. And what is delta? So, delta – a typical entry is shown here. It has state; delta has the state as the first parameter; the input symbol as the second parameter; the top of stack symbol as the third parameter. So, given this combination, it can go to any number of states and rewrite the stack using gamma 1, gamma 2, etcetera. So, if it goes to the state p 1, it writes gamma 1 on to the stack; if it goes to p 2, it writes gamma 2, etcetera. So, this is generally the non-deterministic version of a pushdown automata. So, a finite state automaton also had this possibility. We just said a finite state machine non-deterministic variety could go into any of the states mentioned after the equal to part of the delta. So, here along with the states, we also say what is the stack rewriting that is to be carried out. The important thing is the stack symbol is replaced by this gamma 1 or gamma 2, etcetera; and the input is advanced by one symbol. So, we are going to see examples of how this works.

(Refer Slide Time: 50:43)



Pushdown Automata (contd.)

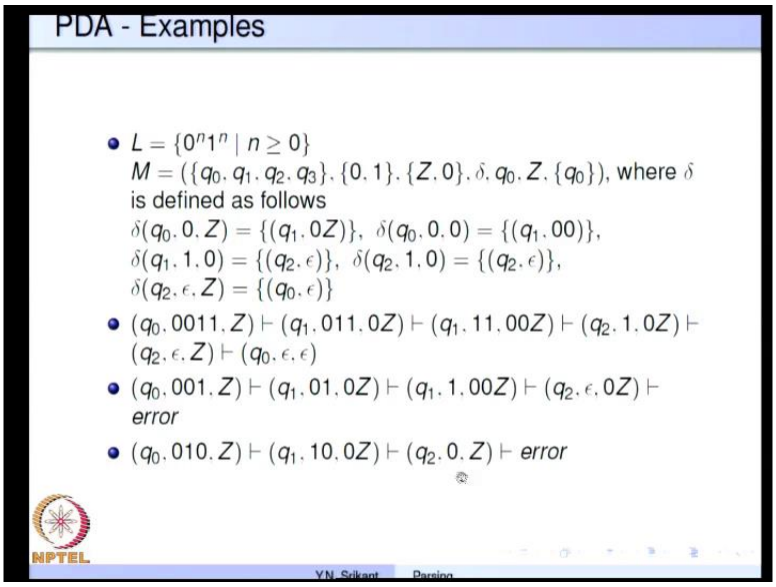
- The leftmost symbol of γ_i will be the new top of stack
- *a* in the above function δ could be ϵ, in which case, the input symbol is not used and the input head is not advanced
- For a PDA M, we define $L(M)$, the language accepted by M **by final state**, to be $L(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \gamma)$, for some $p \in F$ and $\gamma \in \Gamma^*\}$
- We define $N(M)$, the language accepted by M **by empty stack**, to be $N(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$, for some $p \in Q$
- When acceptance is by empty stack, the set of final states is irrelevant, and usually, we set $F = \phi$

Y.N. Srikant Parsing

The leftmost symbol of gamma i, which is actually going to replace the top symbol of the stack will be the new top of stack. So, if abc is gamma i, then a will be the new top of stack symbol. a in the above function delta could be epsilon. So, here this a could be epsilon. That is why the definition contains Q cross sigma union epsilon, and then cross gamma. And it could go to finite subsets of Q cross gamma star. So, a state and a string of stack symbols. So, if the input is not read, then we mention this a as epsilon; in which case the input symbol is not used and the input head is not advanced; otherwise, whenever there is a non-epsilon symbol, the input is read and the input head is advanced.

We define a language, which is accepted by the machine M by final state. So, we say language accepted by M by final state. And there is another variety called language accepted by M by empty stack. This is very straightforward. So, this is the set of all strings w, which are terminal strings really. So, starting from the initial state q naught with the entire input unused, and the stack start symbol Z naught on the stack, a series of moves gives you the state p, the input is emptied, and some strings gamma on the stack. This is not very relevant. But, the important thing is p is a final state and the input is empty. So, if this happens, then the string is in the language. If it is by empty stack, then starting from the same initial configuration q naught, w, Z naught reach the configuration p, epsilon, epsilon; where, p is not necessarily a final state, but the input is empty and the stack is empty. So, in such a case, we actually say this is an automaton, which accept by empty stack. And since the final state is not relevant, we can set F as empty here.

(Refer Slide Time: 53:21)



PDA - Examples

- $L = \{0^n1^n \mid n \geq 0\}$
  $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \{Z, 0\}, \delta, q_0, Z, \{q_0\})$, where δ is defined as follows
  $\delta(q_0, 0, Z) = \{(q_1, 0Z)\}$, $\delta(q_0, 0, 0) = \{(q_1, 00)\}$,
  $\delta(q_1, 1, 0) = \{(q_2, \epsilon)\}$, $\delta(q_2, 1, 0) = \{(q_2, \epsilon)\}$,
  $\delta(q_2, \epsilon, Z) = \{(q_0, \epsilon)\}$
- $(q_0, 0011, Z) \vdash (q_1, 011, 0Z) \vdash (q_1, 11, 00Z) \vdash (q_2, 1, 0Z) \vdash (q_2, \epsilon, Z) \vdash (q_0, \epsilon, \epsilon)$
- $(q_0, 001, Z) \vdash (q_1, 01, 0Z) \vdash (q_1, 1, 00Z) \vdash (q_2, \epsilon, 0Z) \vdash$ *error*
- $(q_0, 010, Z) \vdash (q_1, 10, 0Z) \vdash (q_2, 0, Z) \vdash$ *error*

Y.N. Srikant Parsing

Let us take a simple example. We have L n equal to 0 and 1 n; and n greater than equal to 0. The machine corresponding to it would be… It has four states: q naught, q 1, q 2, q 3; the input is 0, 1. The stack symbols are Z and 0; delta is defined very soon; q naught is the start state; z is the stack symbol; and q naught is also the final state. So, recognizing this language is very easy. You push all the zeros on to the stack. And then once 1 appears in the input, start cancelling or popping the stack 1, 0 for each one. And if we reach a state, where the input is empty and the start of stack symbol has been reached, then we go to a final state. So, q naught, 0, Z; so input symbol is 0 and start symbol is on the stack. So, we go to the state q 1 and push 0 on to the stack. Remember – this 0 is the top new top of stack. And q naught on a 0, and 0 – we go to q 1; push the 0 on to the stack. q 1, 1, 0 – we pop; and go to the state q 2. So, this should have been q 1; this is not q 0; q 1, 0, 0; it is q 1, 0, 0. So, in the state q 1, we go on pushing the zeros.

And, once we reach 1 – the middle part, we start popping; we go to q 2; and what is written on to the stack is epsilon. So, here the 0 was pushed on to the stack. Here also 0 was pushed on to the stack; see the whole 0 is already present here; still present. And when we are in state q 2, we go on popping the stack against the input; cancel the input against the stack symbols. And in state q 2, if we exhaust the input and see the top of stack symbol – start stack symbol Z, then we enter the state q naught with epsilon input; and q naught happens to be… With epsilon as the stack; that means we have accepted the input. So, here is the movement possible – q naught, 0 0 1 1, Z; q 1, 0 1 1, 0 z. So, we have pushed a 0; we have pushed a second 0; we have popped the first one against a single 0; we have popped the second one against the 0; and now, we have reached a state, where we have nothing on stack and nothing on input. Therefore, this is an acceptance by empty stack.

And, here this is an illegal input – 0 0 1. So, we start pushing the 0. And then we push the second 0. We pop the first one against the 0. But, then there is nothing to do. q 2 has no other move defined. So, we end up in an error. Here also this is another illegal input – 0 1 0. So, we push a 0. Then we pop the 1 against a 0; but, we are left with a nonempty input and almost empty stack. So, this is an error situation. So, we will stop here and continue with pushdown automata in the next lecture.

Thank you.