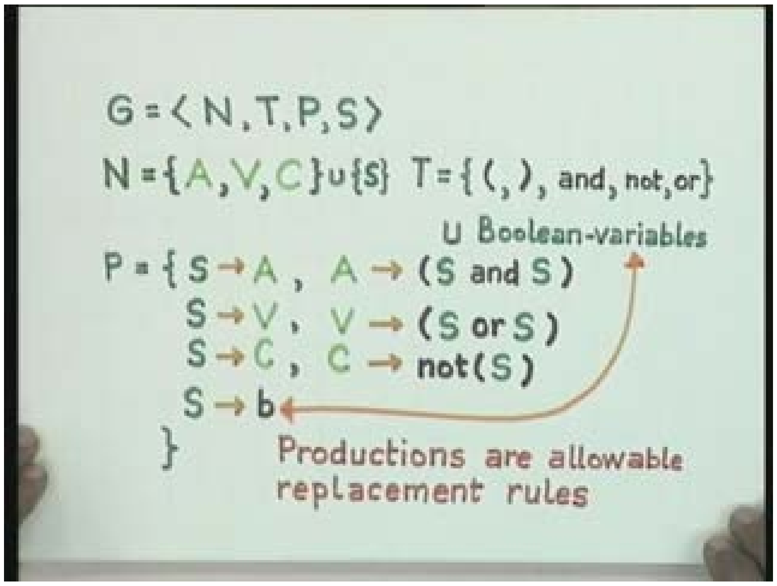
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi
Lecture no 4
Lecture Title: Ambiguity

Welcome to lecture 4. In this lecture we will talk about ambiguity but before that we will go through a definition of a simple programming language and then look at how ambiguity comes about naturally. Let us get back to our context-free grammar which you have seen several times. Let us look at the sentence that we generated using the grammar.

[Refer Slide Time: 00:55]



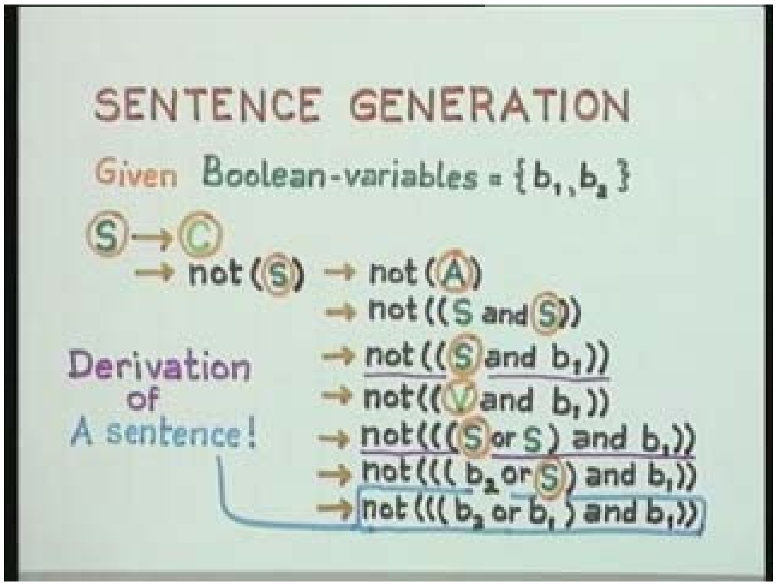
$G = \langle N, T, P, S \rangle$

$N = \{A, V, C\} \cup \{S\}$ $T = \{ (,), \text{and}, \text{not}, \text{or} \}$ $\cup$ Boolean-variables

$P = \{ S \rightarrow A,\ A \rightarrow (S \text{ and } S)$
$S \rightarrow V,\ V \rightarrow (S \text{ or } S)$
$S \rightarrow C,\ C \rightarrow \text{not}(S)$
$S \rightarrow b$
$\}$

Productions are allowable replacement rules

[Refer Slide Time: 01:15]



SENTENCE GENERATION

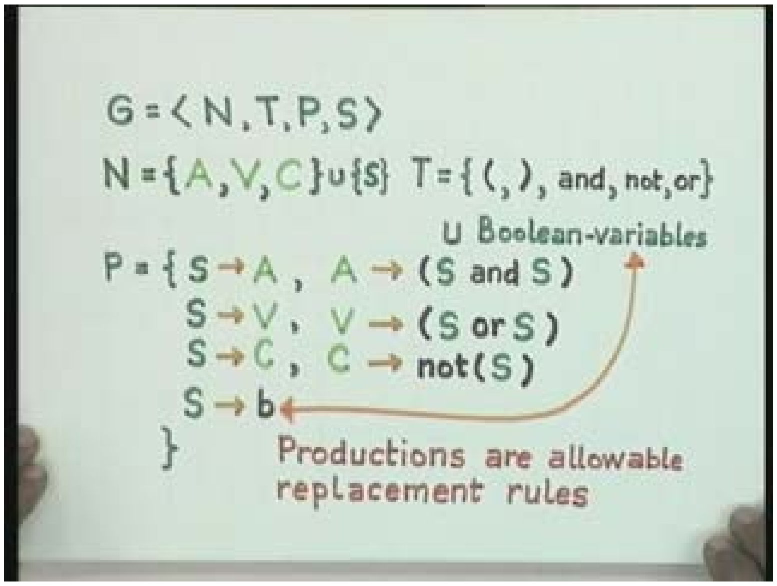
Given Boolean-variables = $\{b_1, b_2\}$

$S \rightarrow C$
$\rightarrow$ not(S) $\rightarrow$ not(A)
$\rightarrow$ not((S and S))
$\rightarrow$ not((S and b_1))
$\rightarrow$ not((V and b_1))
$\rightarrow$ not(((S or S) and b_1))
$\rightarrow$ not(((b_2 or S) and b_1))
$\rightarrow$ $\boxed{\text{not(((}b_2 \text{ or } b_1\text{) and } b_1\text{))}}$

Derivation of A sentence!

Let us take a look at our sentence generation carefully: If the ultimate aim is to generate this sentence of this grammar$\rightarrow$not (((b_2 or b_1)) and b_1)) then let us look at how we have fired the productions.

[Refer Slide Time: 01:45]



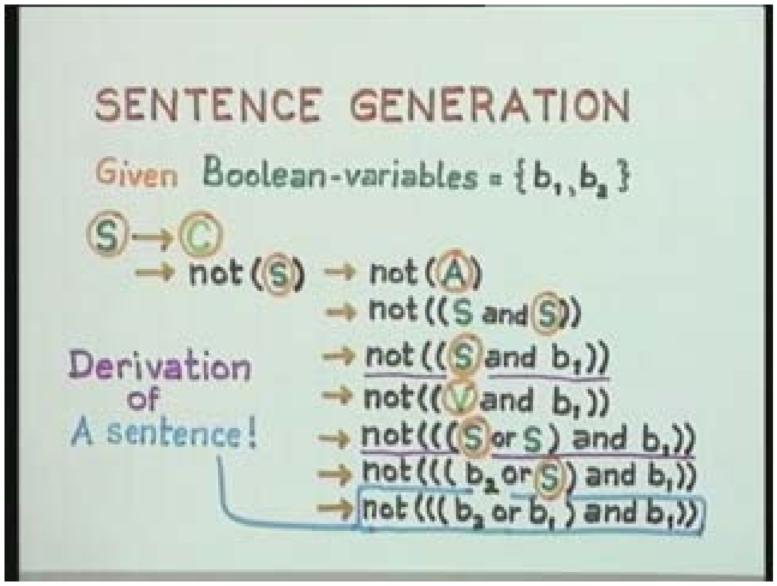
G = < N, T, P, S >

N = {A, V, C} ∪ {S} T = { (,), and, not, or} ∪ Boolean-variables

P = { S → A , A → (S and S)
S → V , V → (S or S)
S → C , C → not(S)
S → b
}

Productions are allowable replacement rules

[Refer Slide Time: 02:00]



SENTENCE GENERATION

Given Boolean-variables = { b_1, b_2 }

$S \rightarrow C$

$\rightarrow$ not (S) $\rightarrow$ not (A)

$\rightarrow$ not ((S and S))

$\rightarrow$ not ((S and b_1))

$\rightarrow$ not ((V and b_1))

$\rightarrow$ not (((S or S) and b_1))

$\rightarrow$ not (((b_2 or S) and b_1))

$\rightarrow$ $\boxed{\text{not (((} b_2 \text{ or } b_1 \text{) and } b_1\text{))}}$

Derivation of A sentence!

Initially we chose one out of four possibilities. The four possibilities for us were S replaced by A, S replaced by V, S replaced by C and S replaced by an identifier. In fact if you are to generate the sentence $\rightarrow$ not (((b_2 or b_1)) then there is really no other possibility you should take. Any other possibility will not give you $\rightarrow$not (((b_2 or b_1)).
So out of the four possible choices, we have to choose $S\rightarrow C$. Then there was only one possibility for C and we have chosen$\rightarrow$not (S). Here again unless we chose the possibility A, we cannot generate the ultimate sentence$\rightarrow$not (((b_2 or b_1)) and for A there is only one possibility namely $\rightarrow$not ((S and S)).

Now we actually have two possibilities. We have chosen to fire $\rightarrow$not ((S and S)) with S goes to b_1 which is essential in the sense that if you had to produce the b1 in the end you would have been forced to apply S goes to b_1.
However, there was another possibility, which is that we could have ignored the right hand side ‘S’ in$\rightarrow$ not ((S and S)) and instead fired the left hand side ‘S’ in$\rightarrow$ not ((S and S)).

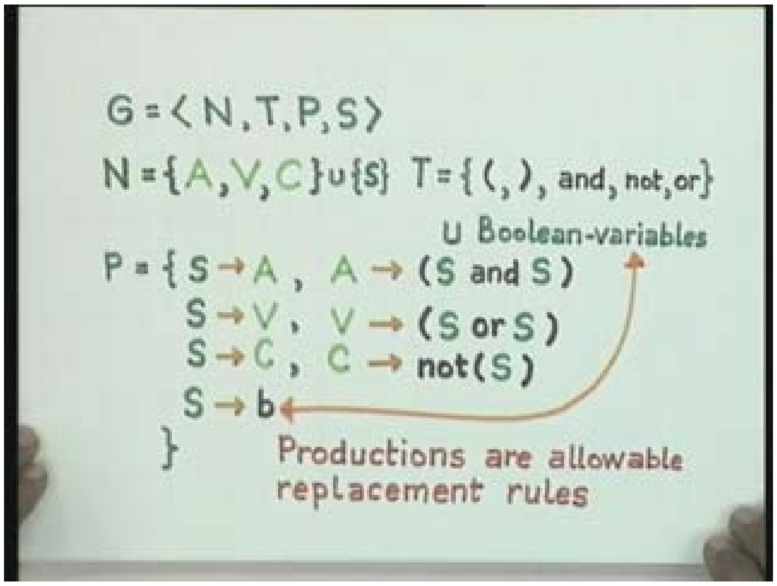
If we had chosen ‘S’ on the left hand side then we could have got this step $\rightarrow$not ((V and b_1)) first. These two steps, $\rightarrow$not ((S and b_1)) and $\rightarrow$not ((V and b_1)) would have been commute. If we would have chosen to apply a production on S always keeping in mind that $\rightarrow$not (((b_2 or b_1)) and b_1)) is the ultimate sentence that we have to generate, we would have got Not (V and S) and then again we have the possibility of either firing this V in $\rightarrow$not ((V and b_1)) or ‘S’ on the right hand side in $\rightarrow$not ((S and S)). We could have chosen any one. You could have either replaced S by this b1in $\rightarrow$not ((V and b_1)) or we could have gone ahead with this V and replaced it by ‘S or S’. Supposing you had chosen V you would have ‘S or S’ and the S on the right in not $\rightarrow$ ((S and S)) would still be there in $\rightarrow$not (((S or S) and b_1)). When you have three possibilities of replacement you could choose any of them.

Out of the various non terminals in the intermediate sentence generation you could have chosen anyone and applied an appropriate production. There would be many such derivations of the same sentence depending upon the order in which you have chosen to apply productions in a sentence. What you choose really does not matter.

The various derivations that you have, just give you various orders. In the example that we have taken; we have not chosen any particular order. We could have chosen to always fire a production of the left most non terminal symbol in an intermediate string.

Here in $\rightarrow$not ((S and S)) we have violated that; we have not chosen the left most. You might consider the derivation as one of several possible derivations of the sentence. Since we are talking about a context-free grammar and the replacement of non terminals by their right hand sides in the production rule, we are talking of something that is independent of context. If there are several non terminals it really does not matter which non terminal is chosen first for replacement, provided you choose the right one which will ultimately give you the sentence. If you were to justify each of these steps in a derivation by which production you have applied, (you could number these productions as 1, 2, 3, 4, 5, 6, 7….) and you could write a justification which just tells you which production number you applied.

[Refer Slide Time: 06:56]



$G = \langle N, T, P, S \rangle$

$N = \{A, V, C\} \cup \{S\}$ $T = \{ (,), \text{and}, \text{not}, \text{or} \}$

$\cup$ Boolean-variables

$P = \{ S \rightarrow A,\ A \rightarrow (S \text{ and } S)$

$S \rightarrow V,\ V \rightarrow (S \text{ or } S)$

$S \rightarrow C,\ C \rightarrow \text{not}(S)$

$S \rightarrow b$

$\}$

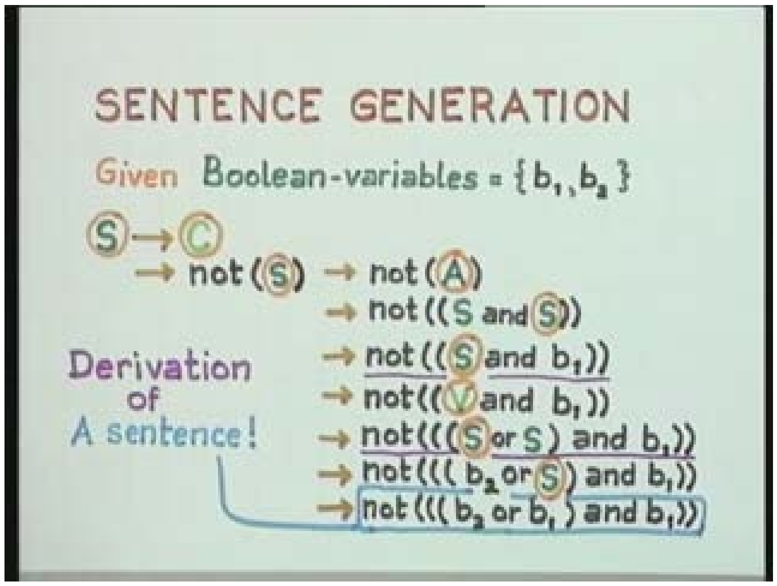
Productions are allowable replacement rules

Then you could essentially permute the order of applications of these productions. For example; in $\rightarrow$not (S) $\rightarrow$not (A) you do not have much choice but in $\rightarrow$ not ((S and S)) you have two possible choices either an application of production 4 which is S$\rightarrow$b or an application of production 2 which is S$\rightarrow$V.

You could have just permuted the order in which you apply these productions but your intermediate strings will not be the same. But you could have permuted the order of application of the productions and you could have derived the same sentence.

So what it means is that the application of productions for the derivation of sentences in a context-free grammar need not be totally ordered. It is because of the context-freeness of the grammar there is independence at an intermediate stage between the various non terminals that you can choose to replace.

[Refer Slide Time: 08:32]



SENTENCE GENERATION

Given Boolean-variables = $\{b_1, b_2\}$

$S \rightarrow C$

$\rightarrow$ not (S) $\rightarrow$ not (A)

$\rightarrow$ not ((S and S))

$\rightarrow$ not ((S and b_1))

$\rightarrow$ not ((V and b_1))

$\rightarrow$ not (((S or S) and b_1))

$\rightarrow$ not (((b_2 or S) and b_1))

$\rightarrow$ $\boxed{\text{not (((}b_2\text{ or }b_1\text{) and }b_1\text{))}}$

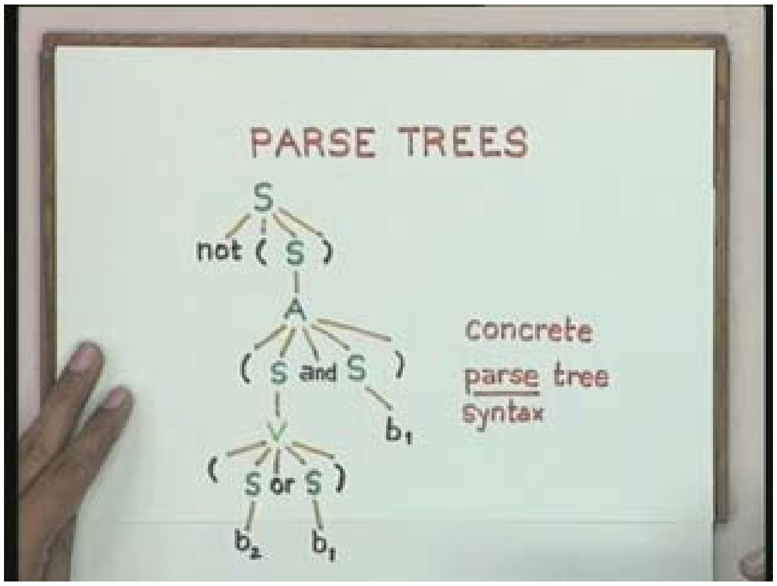
Derivation of A sentence!

So, there is nothing sacrosanct about our derivation. We could have got another derivation by just applying the same productions in a different order. However there is a certain amount of order. For example, you could never have applied any productions before applying the production $S \rightarrow C$. You would never have any of these possibilities before applying this production C goes to ‘not’ (S) if you are interested in generating this sentence$\rightarrow$not (((b_2 or b_1)) and b_1)). Given the sentence you would have to apply $C \rightarrow$not (S) before any of the others. But in $\rightarrow$ not ((S and S)) you have a choice whether you want to replace the S on the right side or the S on the left side. The actual production that you have to apply is still place-specific. If you have to generate b2 or b1, you have no alternative but to replace the S on the left hand side eventually but it does not matter whether you replace the ‘S’ on the left hand side before you replace the ‘S’ on the right hand side or after it. In any derivation, this independence between two or several other possible replacements that might be there, gives you a partial ordering on the application of the productions.

This partial ordering is really what is sacrosanct. It specifies that there are certain productions which have to be applied in a certain order but other productions need not be applied in the same order and they could be applied independent of each other.

If you were to look at the various derivations possible for this sentence $\rightarrow$not (((b_2 or b_1)) and b_1)) we could take all those derivations and collapse them to give us the partial order. We could collapse them to actually look for independence and dependence. In fact, we can draw a tree of exact dependencies. That tree is known as a parse tree. There is a tree of the same sentence that we have generated and this tree tells us what the dependencies in the applications of various productions are.

[Refer Slide Time: 11:20]



The root of a parse tree is always the start symbol. In the generation of this sentence, the first production that was applied was S $\rightarrow$C.

Then the next production was C$\rightarrow$Not S. I am looking upon the whole of 'Not' as a single symbol and open parenthesis as a single symbol. Remember our convention that black denotes terminal symbols. The eventual strings that you generate will all be in black. The other colors denote certain abstractions.

The first production was S $\rightarrow$C, C goes to Not(S) and then there was an application of S which yielded A. Then you have 'A' yields (S and S) and at this point we are at the same position we were in the original sentence$\rightarrow$ not ((S and S)). The fact that you can choose the S on the left hand side first or the S on the right hand side first does not really matter. Eventually in whatever order you apply the productions for them, if you are interested in generating the sentence that you have generated, the right hand side S should go to a b_1 and the left hand side S should be expanded into a V. This V should be expanded into (S or S) and it does not matter in which order you perform these two productions. If you are interested in generating the sentence$\rightarrow$not (((b_2 or b_1)) and b_1)), the first S should produce a b_2 and the second S should produce a b_1. That is also the reason why I have used a brown color for the productions.

We obtain a tree which we can call the parse tree and the leaves of this tree are all terminal symbols. Notice that all the leaves of this tree are black. If you read the leaves from left to right you get the sentence that you generated. You have 'Not (((b 2 or b1) and b1))' which is what this sentence is. We are most interested in the parse tree rather than in the actual derivation.

For any sentence we have a corresponding parse tree. For generating any sentence we do not necessarily have a unique derivation. In this grammar that we have defined, for every sentence there is actually a unique parse tree. Parse trees are very important from the point of view of compiling language implementation and from the point of view of specifying semantics. Only a parse tree is really sacrosanct and the fact that there are many ways of traversing this parse tree gives you several possible derivations.

Let us just summarize that briefly. We can look upon a parse tree as presenting a partial order in the firing of productions and we can look upon a derivation of a sentence as just one of many possible traversals of the parse tree of the sentence. For each sentence we would like to have a unique parse tree not necessarily a unique derivation. You can look upon a derivation as just a linearization of a partial order.

A topological sort just takes a partial order and linearizes it. In topological sorting, you sort it to provide a linear order, a total order of all the elements such that the dependency specified by the partial order is maintained. But where dependencies do not exist, you might place them in any order you like.

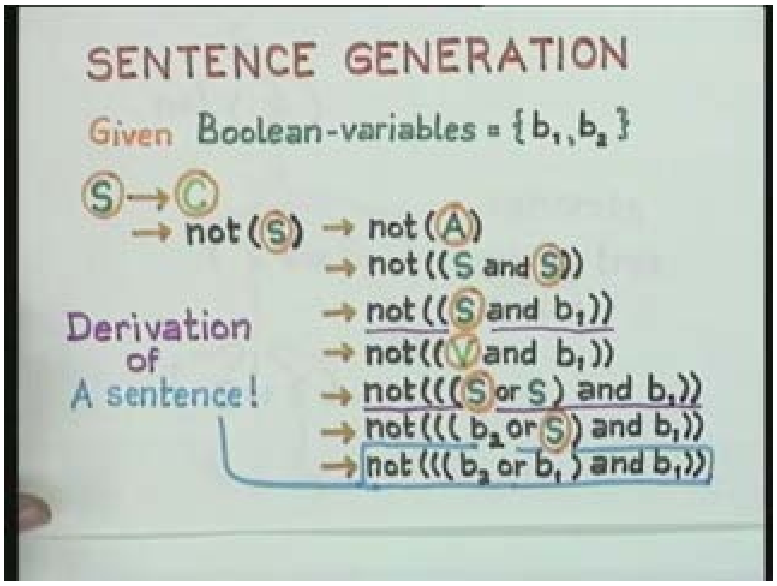
A linearization of a tree always gives you a total order. For partial orders it is also true that a partial order is completely defined by the set of all possible linearizations of the partial order. Essentially a parsed tree is also completely specified since it is a partial order by the set of all possible derivations or traverses you can make of the past.
This is a fundamental property in the theory of partial orders which is that you can look upon every partial order as actually a set consisting of all total orders of that set and as being completely defined by the set of all total orders.

The parse tree that we have presented is somewhat syntactical. It is syntactical in the sense that from the point of view of compiling or language implementation, the nature of each of these terminal symbols does not really matter. As far as it is concerned it is some stream of symbols that come and it just looks upon them as a stream of symbols. But any kind of language also has an implicit type of terminal symbols which is clearly distinguished. For example; we make a clear distinction always between an identifier and an operator. We might call this a concrete parse tree where we make no distinctions at all between identifiers and operators. If they are terminal symbols they are leaves of the parse tree.

What we are more often interested in is what might be called an abstract parse tree where in the same sentence you actually make a distinction between what are the operators and what are the operands.

If you look at the sentence that we have previously generated, it is clear that our intention was to define a language of Boolean expressions where the operations are ‘Not’ and the operands are these Boolean variables like b1, b2 etc.

[Refer Slide Time: 21:33]



SENTENCE GENERATION

Given Boolean-variables = $\{b_1, b_2\}$

$S \rightarrow C$

$\rightarrow$ not(S) $\rightarrow$ not(A)

$\rightarrow$ not((S and S))

$\rightarrow$ not((S and b_1))

$\rightarrow$ not((V and b_1))

$\rightarrow$ not(((S or S) and b_1))

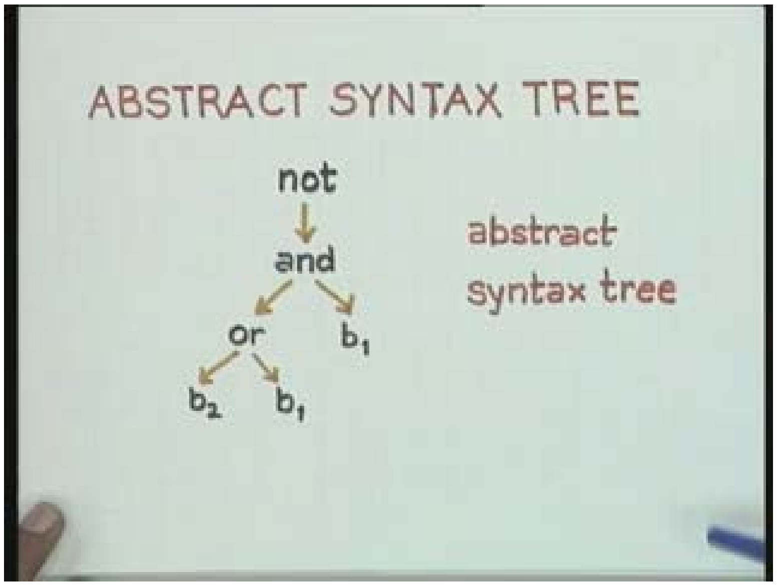
$\rightarrow$ not(((b_2 or S) and b_1))

$\rightarrow$ $\boxed{\text{not(((} b_2 \text{ or } b_1 \text{) and } b_1\text{))}}$

Derivation of A sentence!

This clear distinction between the operands and operators is what leads us to an abstract syntax tree. This syntax tree actually elevates and replaces non terminals. We have designed the language in such a way that we could easily elevate the operators to the intermediate nodes of this tree. We can talk of a root operator which is ‘Not’. We can talk of intermediate operators ‘and’ and ‘or’.

[Refer Slide Time: 22:46]



ABSTRACT SYNTAX TREE

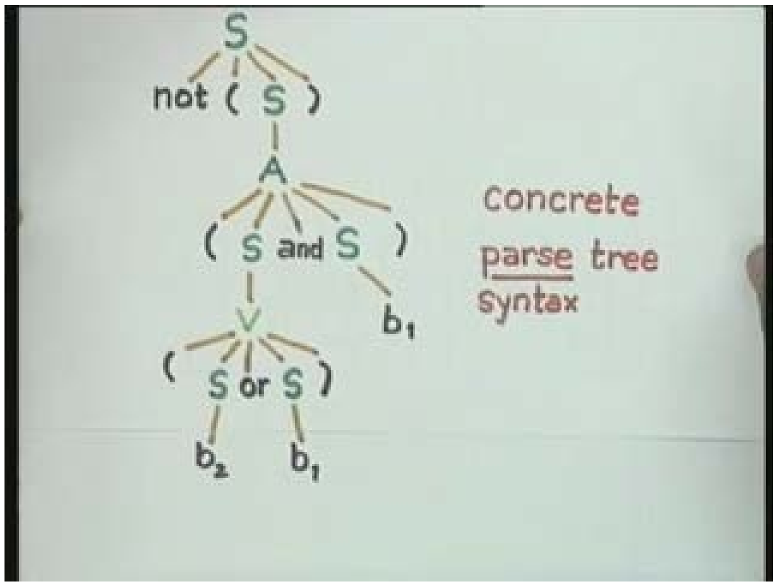
$$\begin{tikzcd}
 & \text{not} \arrow[d] & \\
 & \text{and} \arrow[dl] \arrow[dr] & \\
\text{or} \arrow[d] \arrow[dr] & & b_1 \\
b_2 & b_1 &
\end{tikzcd}$$

abstract syntax tree

In an abstract syntax tree the operators are always the intermediate nodes and the leaves are all the identifiers of the operands. This is not absolutely full proof but when we talk about the distinction between operands and operators we are making a distinction between the various kinds of terminal symbols that are there in the parse tree and we are elevating some of them. If you look at the ultimate programming language that we are interested in, we are really not interested in the non terminal symbols.

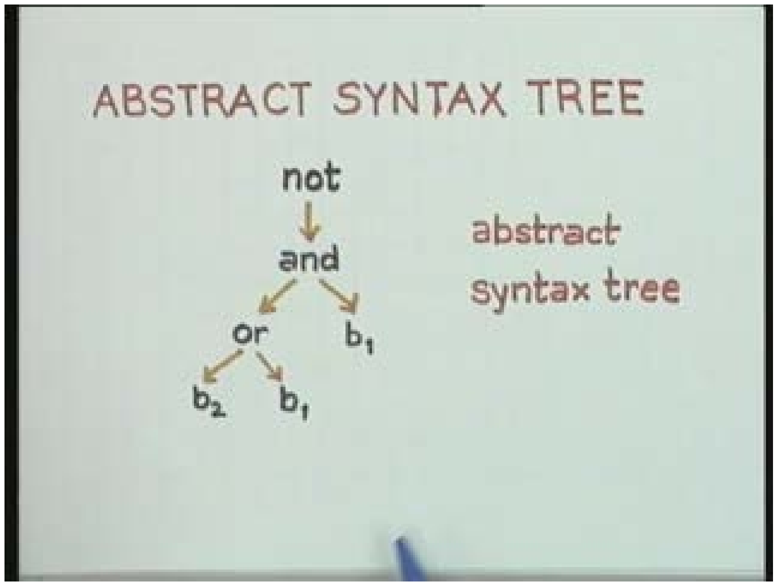
Those terminal symbols in any programming language have some meaning. There are some distinctions between what an identifier is and what an operator is. If we want to bring about this distinction then we are not interested in the concrete syntax tree but in the order in which we should apply the operators on the operands.

[Refer Slide Time: 23:52]



In fact, if you look at this sentence the reason why we included parenthesis in all our mathematical language is to specify an order of application of operators on operands. This is especially true for infix operators. It is not absolutely essential to have parenthesis but you can see that otherwise we would have to have a uniform post fix notion or a uniform prefix notion if you have to avoid operators. In fact, every language construct can be regarded as an operator. Essentially if you are interested in giving meanings to languages, you are not interested in the concrete parse tree.

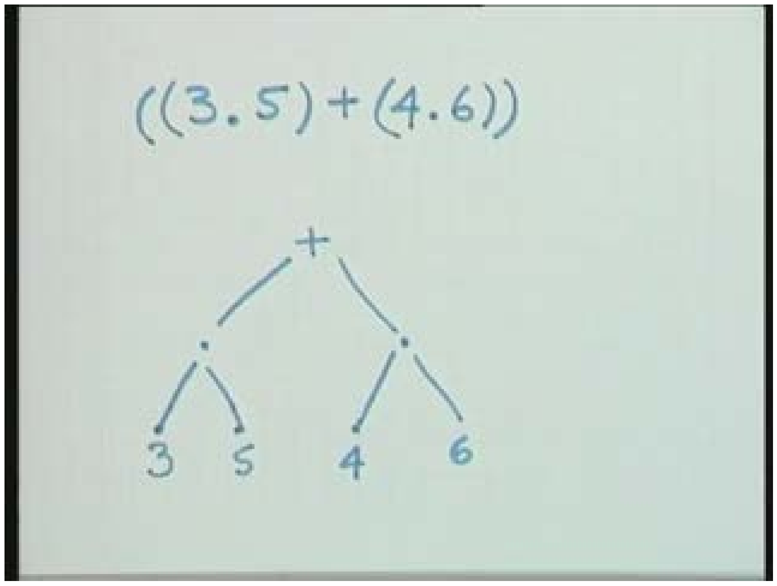
[Refer Slide Time: 24:54]



ABSTRACT SYNTAX TREE

We are interested in the abstract syntax tree. In an abstract syntax tree there are no brackets. Given an arbitrary expression there are various ways of evaluating that expression. You can choose to evaluate one operand rather than another unless there is an explicit dependency that you cannot evaluate one operand before the other.

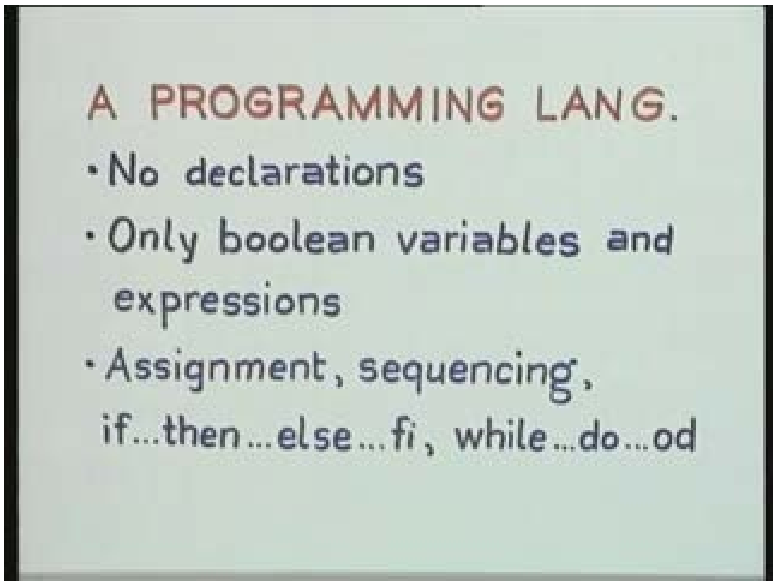
[Refer Slide Time: 25:42]



Take an arithmetic expression like ((3.5) + (4.6)). In our normal arithmetic there is an implicit order. You first apply multiplication before doing addition. It does not matter in which order you do the multiplications and then you do the addition.

All our evaluation mechanisms actually refer more to an abstract syntax tree than to the concrete parse tree of the expression. We have to keep this in mind. We will often use these abstract syntax trees once we cross syntax. Let us then define a small programming language.

[Refer Slide Time: 27:10]



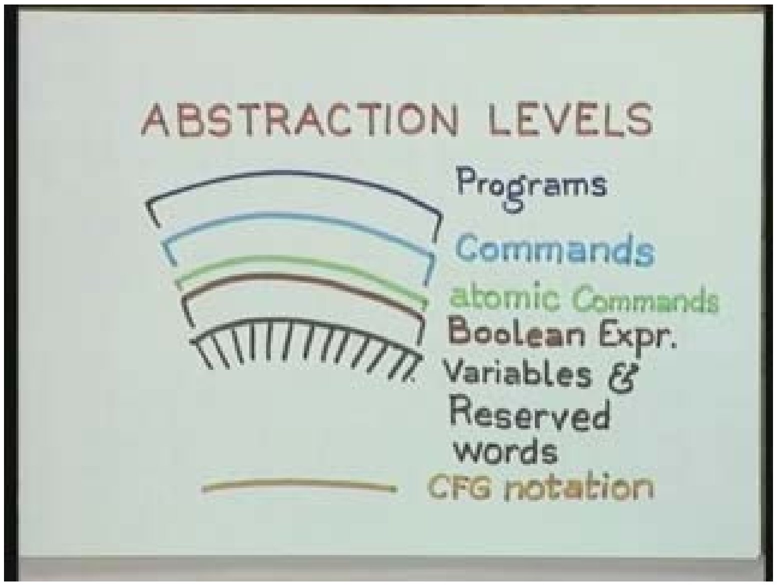
A PROGRAMMING LANG.

- No declarations
- Only boolean variables and expressions
- Assignment, sequencing, if...then...else...fi, while...do...od

This programming language contains no declarations. It contains only Boolean variables and expressions. It contains the following commands; assignment, sequencing, a conditional and a simple looping command. I would like to specify the syntax of this programming language so that it clearly gives me all the possible syntactically valid programs in the language and I can then generate from the grammar all possible syntactically valid programs of the language. We are talking about the kinds of programs that would be syntactically valid and the kinds of programs that would not be syntactically valid. But now there is a question of formalizing it and giving production rules for the syntax in such a way that if you had to build a compiler or a translator for this language, you should be able to do it without any problems.

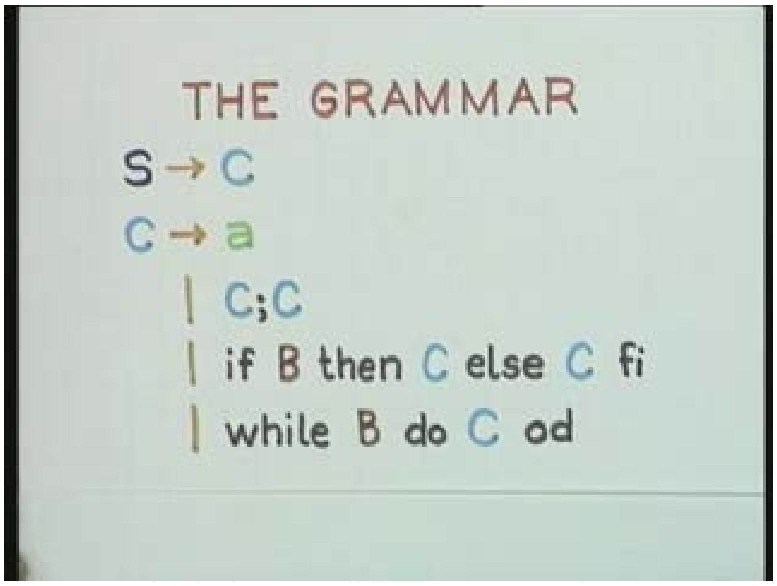
In fact with our specifications we can start writing programs. But for the purposes of translation and compilation you require to specify in some more detail. Let me just summarize my coding of various constructs.

[Refer Slide Time: 29:23]



This brown is part of the context-free grammar notation, which is the notation for productions. This brown looks very much like the bark of a tree which is why branches of parse trees are coded in this color. The actual terminal symbols are of course in black. Since we have two different kinds of entities, commands and Boolean expressions, I have used this dark brown for Boolean expressions and I have used light green for some atomic commands. The assignment is an atomic command. For all other compound commands I use this light blue and the actual program or a sentence of the language is in dark brown. You will be informed about changes in colour coding but this is how I have defined the grammar. We will define the grammar in a top down fashion.

[Refer Slide Time: 30:50]



THE GRAMMAR

```
S → C
C → a
  | C;C
  | if B then C else C fi
  | while B do C od
```

Let us look at this grammar. I have the start symbol S. I am only interested in the sentences of the language and the sentences of this language are all programs. Unlike a language like Pascal there is no program heading and there are no declarations. What is a program in this language where $S \rightarrow C$?

-This rule $S \rightarrow C$ specifies that a program is any command. Any command is a full program of this language. What is a command of this language? - A command could be an atomic command or it could be a sequence of two commands. It could be a conditional command or a looping command. I have used brown bars to indicate that $C \rightarrow A$ is four productions:

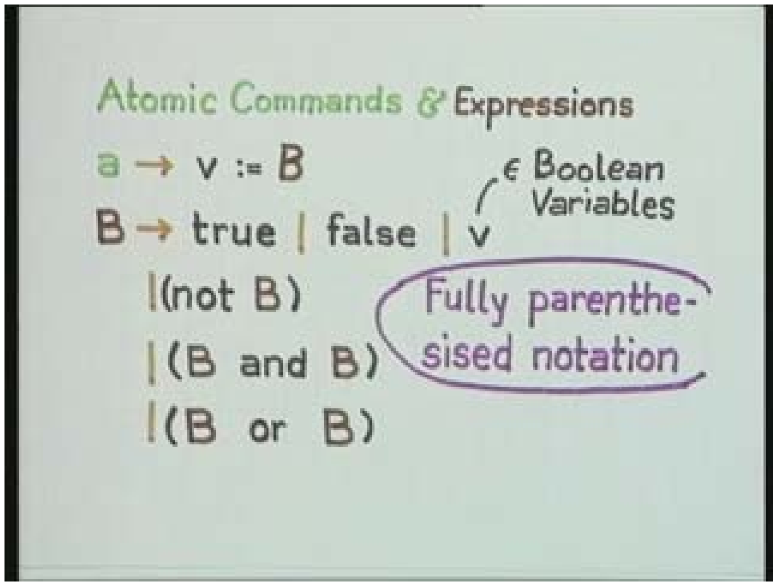
$C \rightarrow a$
$C \rightarrow C ; C$
$C \rightarrow$ if B then C else C fi
$C \rightarrow$ while B do C od

This bar actually specifies the various alternatives that you have for each non terminal symbol. At the command level or what in most of these imperative languages are called statement level these are the various productions. We can look upon the definition of this programming language in terms of several levels and this 'S' gives you the program level and how the program level goes into the command level.

I have so far not specified what this atomic command, 'a' is. At this point this command level essentially tells you how to form compound commands from simpler commands. It just says that any simple command is a command and the sequencing of two simple or compound commands. That is why it is in blue; this semi colon is the reserved word of the language so it is in black. The sequence of the two commands is a command in itself and if B is a Boolean expression then this conditional compound command might contain some compound commands inside it while B command contains a Boolean expression and possibly a compound command.

It means that as far as the grammar of commands is concerned this B and this 'a' are also terminal symbols of this level of the grammar specification.
But in order to get a complete definition we should also look into these atomic commands and the Boolean expressions. Let us look at the atomic command. Since I specified that there is only assignment statement there is only one atomic command.

[Refer Slide Time: 34:50]



Atomic Commands & Expressions

$a \rightarrow v := B$

$B \rightarrow$ true | false | v

|(not B)

|(B and B)

|(B or B)

∈ Boolean Variables
Fully parenthe-sised notation

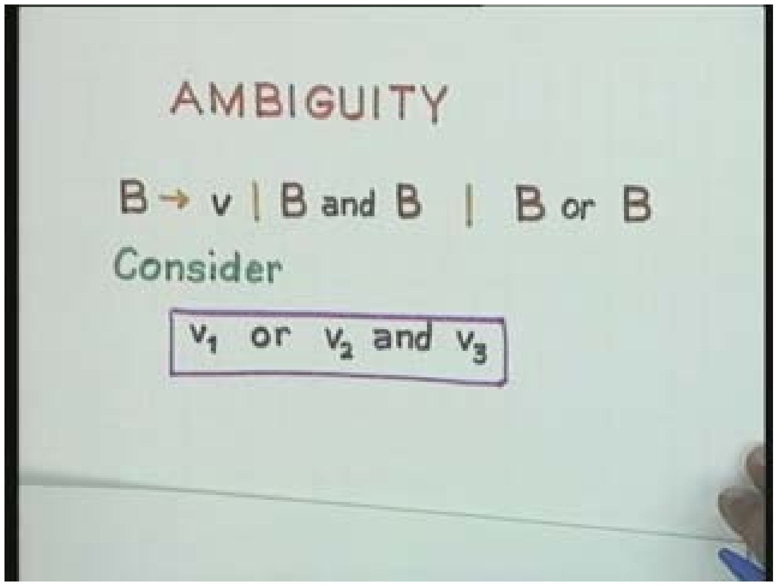
The grammar or atomic commands is just of this form, $a \rightarrow v := B$. Let us assume that v is a Boolean variable and B is a Boolean expression then v assigned B is an atomic command.

The terminal true is a Boolean expression and the terminal false is a Boolean expression. Any Boolean variable, v is a Boolean expression.

If you take the language of Boolean expressions alone as a separate entity these true$|$false are the terminal symbols of the language. The next level specifies how to make compound Boolean expressions from simpler Boolean expressions.
If B is a Boolean expression then 'not B' is also a Boolean expression and given two Boolean expressions (B and B) is a Boolean expression and (B or B) is also a Boolean expression.

I have changed the grammar in the sense that here I have defined a fully parenthesized notation. For all Boolean expressions that are not atomic I have enclosed a new parenthesis. Supposing we get rid of true and false and instead we defined the grammar without parenthesis.

[Refer Slide Time: 37:25]



AMBIGUITY

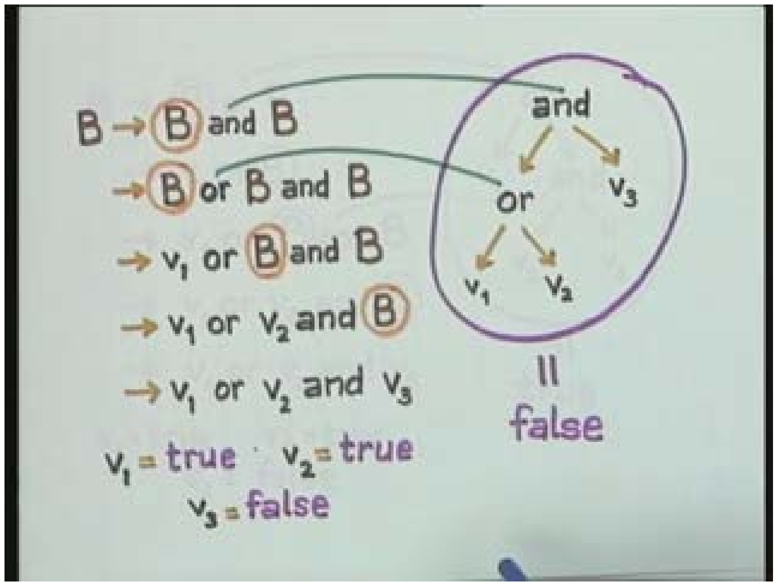
$B \rightarrow v \mid B \text{ and } B \mid B \text{ or } B$

Consider

$\boxed{v_1 \text{ or } v_2 \text{ and } v_3}$

Consider this sentence v_1 or v_2 and v_3. This sentence is generated by this grammar. It is very easy to give a derivation for that. Moreover this sentence can actually be derived in two different ways.

[Refer Slide Time: 38:48]



$B \rightarrow B \text{ and } B$

$\rightarrow B \text{ or } B \text{ and } B$

$\rightarrow v_1 \text{ or } B \text{ and } B$

$\rightarrow v_1 \text{ or } v_2 \text{ and } B$

$\rightarrow v_1 \text{ or } v_2 \text{ and } v_3$

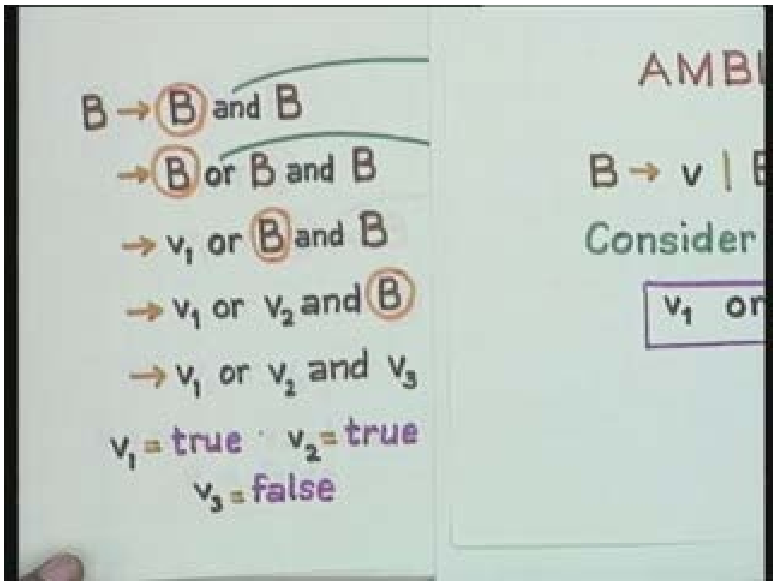
$v_1 = \text{true} \quad v_2 = \text{true}$

$v_3 = \text{false}$

Look at this sentence v_1 or v_2 and v_3 and this is my grammar, B $\rightarrow$v|B and B|and B or B. I could do apply the B or B rule and derive B $\rightarrow$B or B. I could choose to expand it and get (since my first symbol has to be v_1 I do not really have any other alternative) v1 or B. Then I can choose to replace B of v1 by B and B and v_1 or v_2 and B. I can choose to expand B and replace it by v_2. Then I can choose to expand B of 'v_1 or v_2 and B' and get $v_1$or v_2 and v_3.

I could also make a different derivation. For example; I can decide to apply this B and B of B $\rightarrow v|$B and B$|$to B$\rightarrow$B and B and I can choose to expand the second B of $\rightarrow v_1$ or B and B and get v_1 and then I choose to expand the second B and replace it by v_2 and then I choose to expand the third B of v_1 or v_2 and B and replace it by v_3 and I get the same sentence $v_1 =$true $v_2 =$true $v_3 =$false.

[Refer Slide Time: 39:36]



$$\begin{aligned} B &\rightarrow \boxed{B} \text{ and } B \\ &\rightarrow \boxed{B} \text{ or } B \text{ and } B \\ &\rightarrow v_1 \text{ or } \boxed{B} \text{ and } B \\ &\rightarrow v_1 \text{ or } v_2 \text{ and } \boxed{B} \\ &\rightarrow v_1 \text{ or } v_2 \text{ and } v_3 \end{aligned}$$

$v_1 =$ true $v_2 =$ true

$v_3 =$ false

AMBI

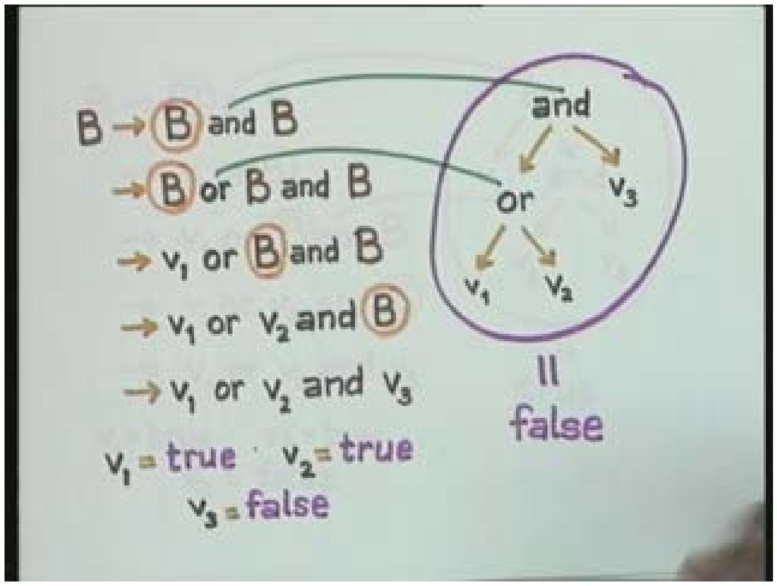
$B \rightarrow v \mid$

Consider

$\boxed{v_1 \text{ or}}$

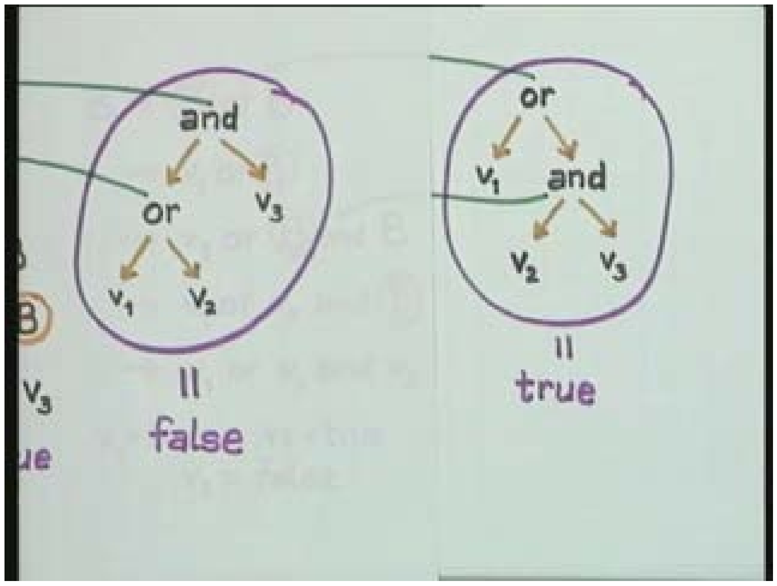
If you were to look at the way in which the productions have been done you will find 'and', which is a root operator. If you were to look at the abstract syntax tree then 'and' is a root operator and below there is an 'or' and this is the abstract syntax tree that you get. I have chosen to give you the abstract syntax tree. (You could for example take the concrete parse tree also of this derivation) If you look at the abstract syntax tree of the other derivation where 'or' is the root operator and 'and' comes inside.

[Refer Slide Time: 40:40]

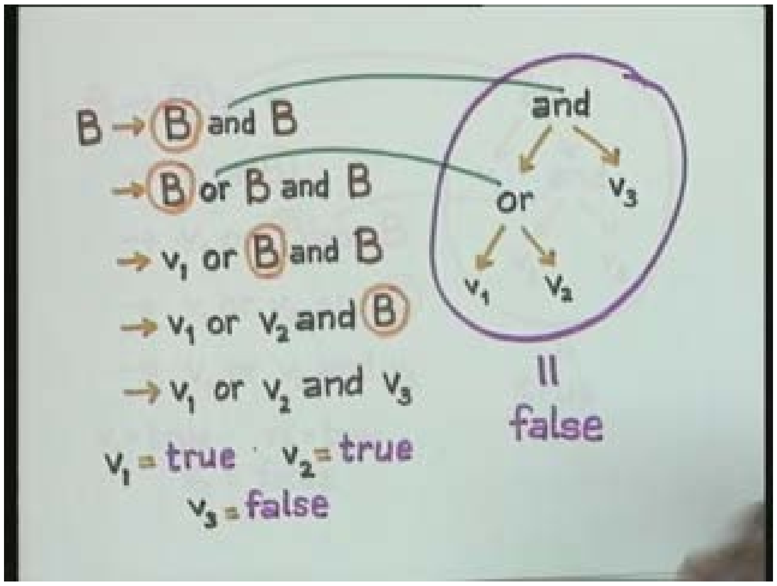


If you look at the two trees you can see that they are both actually different. There are not just two different derivations of the same sentence, there are two different derivations with different syntax trees and these two different syntax trees actually affect how you can specify the meaning of this language. For example if v1 and v2 had the values true and v 3 had false then the evaluation of this syntax tree would give you a value of true and the evaluation of this syntax tree would give you a value of false.

[Refer Slide Time: 41:15]



[Refer Slide Time: 41:16]



$B \to B \text{ and } B$

$\to B \text{ or } B \text{ and } B$

$\to v_1 \text{ or } B \text{ and } B$

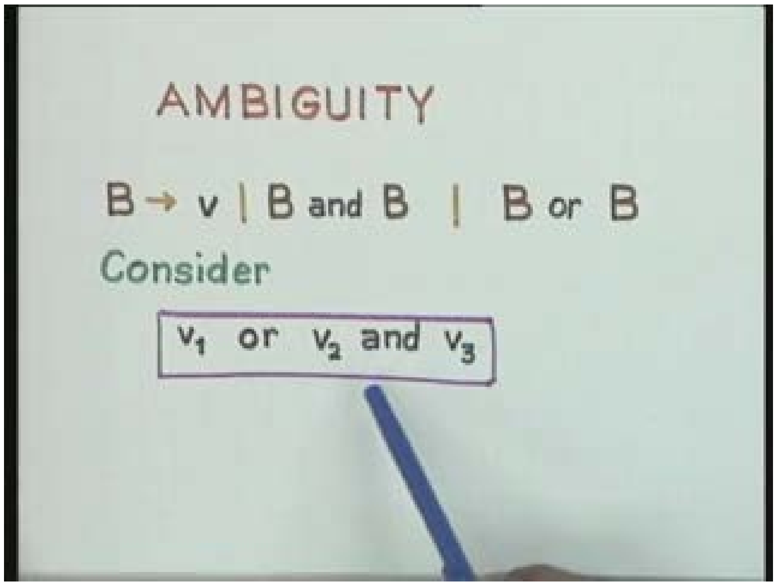
$\to v_1 \text{ or } v_2 \text{ and } B$

$\to v_1 \text{ or } v_2 \text{ and } v_3$

$v_1 = \text{true} \quad v_2 = \text{true}$

$v_3 = \text{false}$

[Refer Slide Time: 41:55]



AMBIGUITY

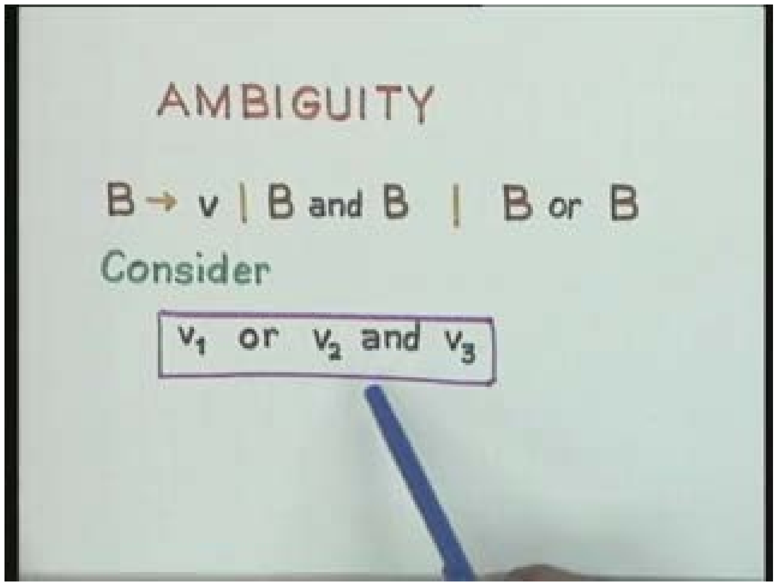
$B \to v \mid B \text{ and } B \mid B \text{ or } B$

Consider

$\boxed{v_1 \text{ or } v_2 \text{ and } v_3}$

If you look upon syntax as ultimately having to specify a meaning then you would want a unique meaning to be specified. In that sense this grammar for example falls short of being an adequate representation of a unique expression language, an expression language with unique meanings. So we call such a grammar ambiguous. It is called syntactically ambiguous but the syntax is really a via media in which you are going to specify semantics. It is only an initial handle for the specification of semantics.

[Refer Slide Time: 41:58]



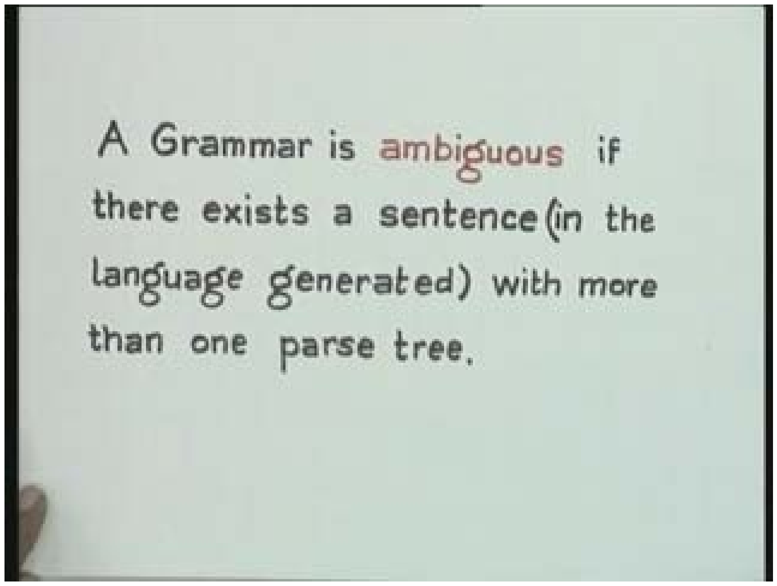
AMBIGUITY

$B \rightarrow v \mid B \text{ and } B \mid B \text{ or } B$

Consider

$\boxed{v_1 \text{ or } v_2 \text{ and } v_3}$

[Refer Slide Time: 42:30]



A Grammar is ambiguous if there exists a sentence (in the language generated) with more than one parse tree.

Let us look at grammar. A grammar is ambiguous if there exists a sentence in the language with more than one parse tree. Ambiguity is an important constraint in the sense that it is not just that there are two different derivations, it is that there are two different parse trees. Those parse trees are important both from the point of view of translation which means running programs. If you are looking into the problem of compiling then you are really looking at the problem of executing programs in order to get meanings. It is necessary for a program to be interpreted in exactly one way. So, the compiler for that program and the user of that programming language both come to an agreement on how that program is to be interpreted. Ambiguity has very serious consequences in specifying the meanings for the execution behavior of programs.

Actually a lot of our programming languages do have ambiguity. The programming language that we have defined is totally unambiguous. Take the following grammar:

$S \rightarrow C$

$C \rightarrow a$

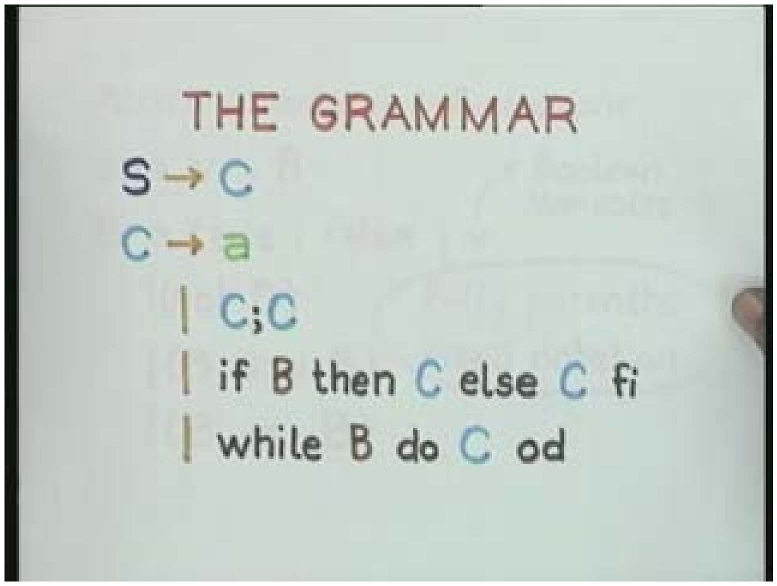
$|C; C$

$|$if B then C else C fi

$|$while B do C od

It specifies a language in which there is absolutely no ambiguity. There is exactly one parse tree for every sentence in the language.

[Refer Slide Time: 44:18]



THE GRAMMAR

S → C

C → a

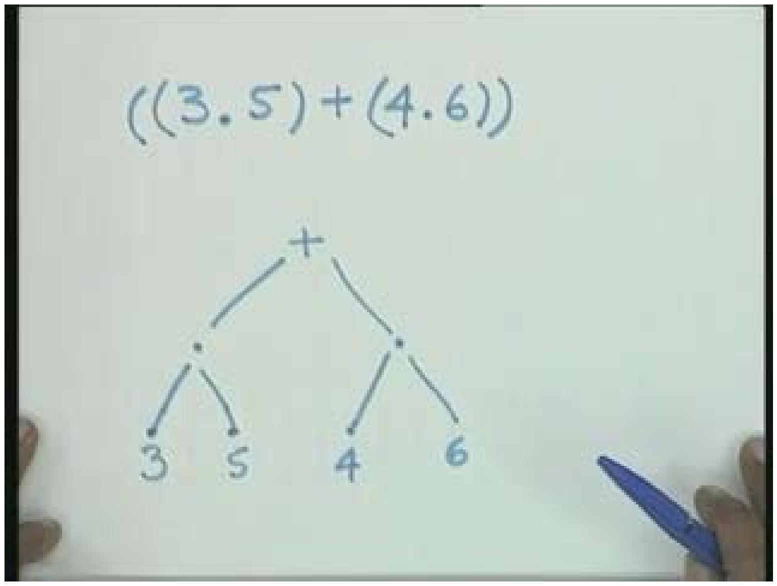
| C;C

| if B then C else C fi

| while B do C od

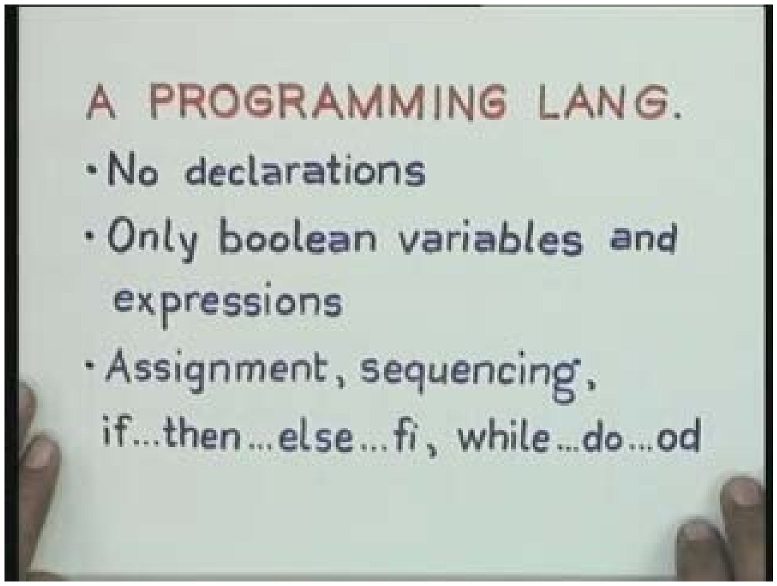
In the expression language there is always ambiguity in most languages but that is because the expression language allows you to use normal mathematical notation. For example, it allows you to specify various details with an implicit order of evaluation. Our normal mathematical notation allows you to specify an expression without any parenthesis. The syntax tree for the expression is going to be the same even if you remove the parenthesis. If you remove the parenthesis there is absolutely no reason except for normal mathematical convention, why you cannot construct a syntax tree.

[Refer Slide Time: 45:25]



But it is just that our mathematical convention says that there is a precedence of operations which ensures that multiplication is done first and addition is done later. If there is syntactic ambiguity then the order of evaluation should be that multiplication should precede addition otherwise the order of evaluation is as specified by the parenthesis. It is the normal mathematical notation which most programming languages implement. At the command level also there are syntactic ambiguities for example in languages like Pascal. One is known as the dangling else problem. If you look at the difference between the conditional that we have defined and Pascal, it is that firstly we do not have both an ‘if then’ and an ‘if then else’ construct. Our conditional construct has a perfect bracketing.

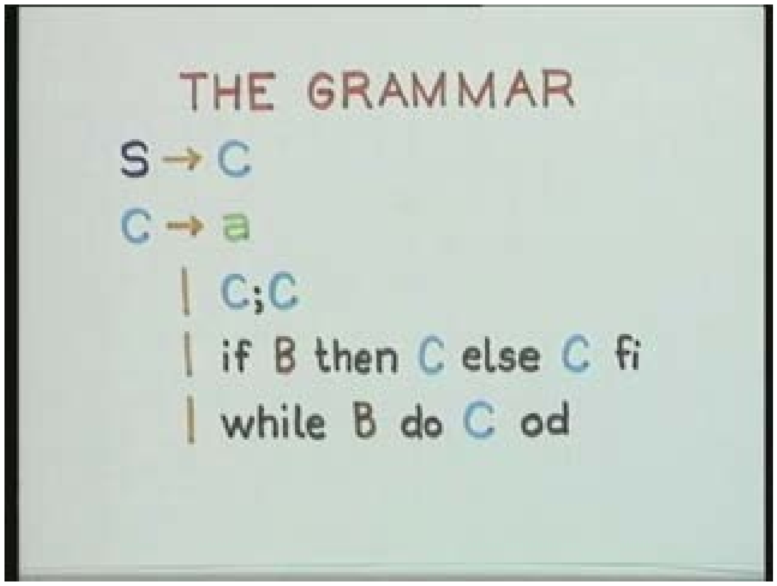
[Refer Slide Time: 48:00]



A PROGRAMMING LANG.

- No declarations
- Only boolean variables and expressions
- Assignment, sequencing, if...then...else...fi, while...do...od

[Refer Slide Time: 48:55]



THE GRAMMAR

$S \rightarrow C$

$C \rightarrow a$

$\quad | \ C;C$

$\quad | \ \text{if } B \text{ then } C \text{ else } C \text{ fi}$

$\quad | \ \text{while } B \text{ do } C \text{ od}$

There is an ‘if’ with a ‘fi’ and a ‘while’ do with an ‘od’. There is no possibility of ambiguity except in the case of one construct.
The sequencing is ambiguous.