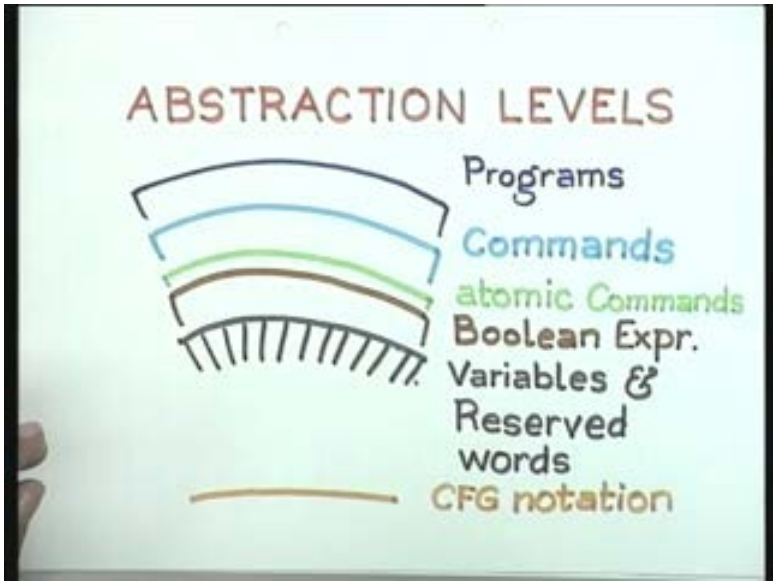
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi
Lecture no 5
Lecture Title: Syntax

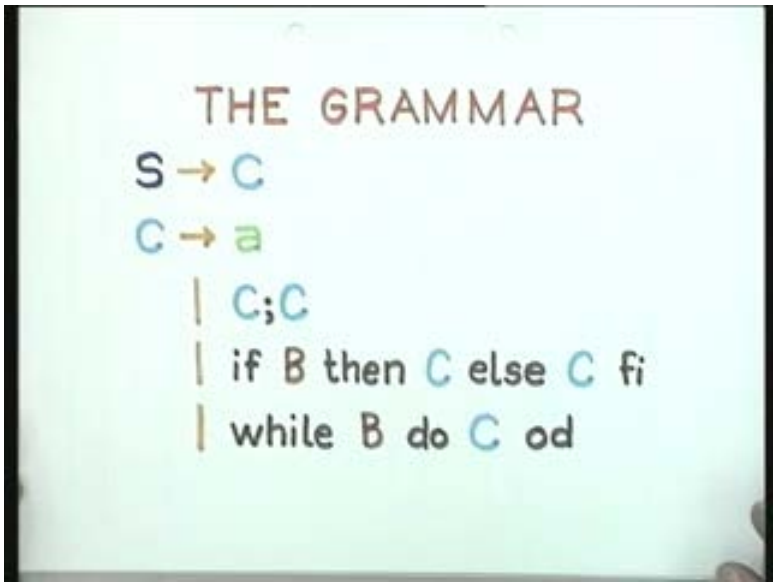
Welcome to lecture 5. We will do a slightly more complicated programming language for which there is also a compiler but before that let us briefly summarize what we did last time and also answer the question. We will follow more or else the same abstraction levels except that I will change brown.

[Refer Slide Time: 00:47]



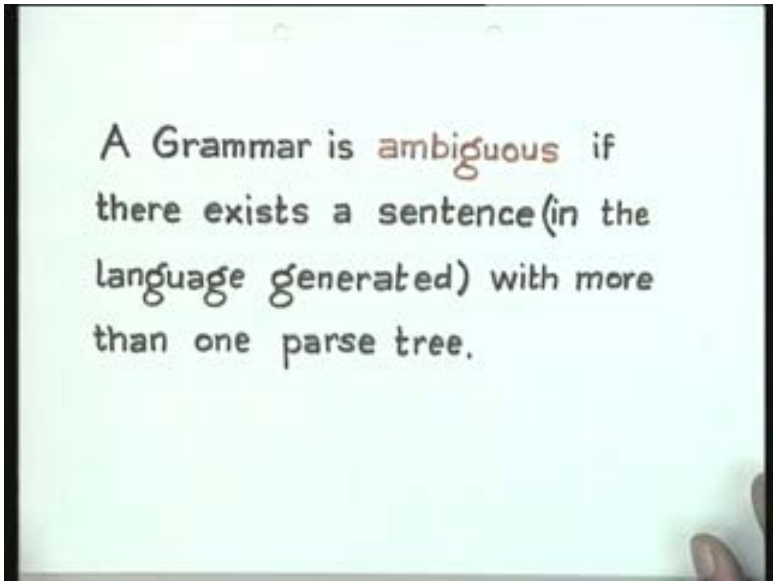
Programs, commands, atomic commands and a context-free grammar notation will remain the same.

[Refer Slide Time: 01:06]



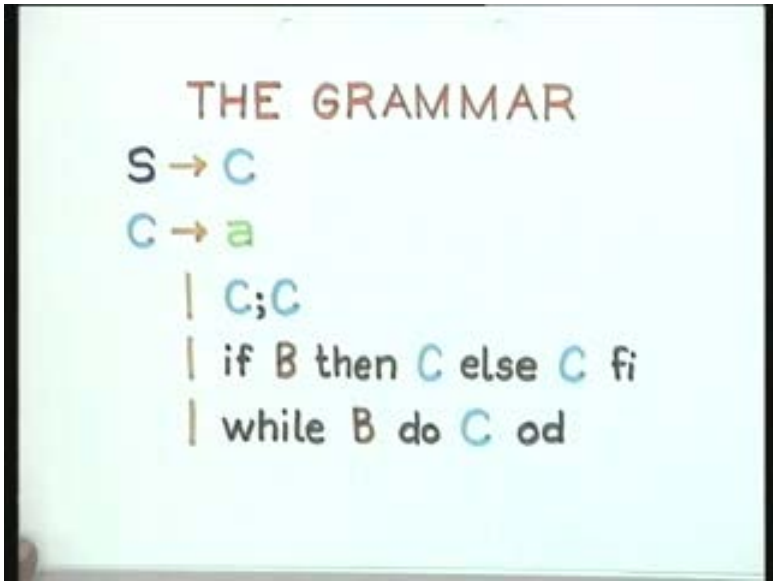
Let us look at the grammar we had last time. Firstly, the grammar of programs was that every program is just a command. A command could either be some atomic command or it could be a sequence of commands; it could be a conditional with an ‘if b then c else c fi’ where all these words in black are reserved words including this which you may not have seen. It acts as a closing bracket for ‘if’ and similarly the ‘od’ here acts as a closing bracket for the ‘do’. We also defined the notion of ambiguity for example; we asked the question whether this given grammar was ambiguous.

[Refer Slide Time: 02:05]



As I said our grammar is ambiguous if there exists a sentence in the language generated with more than one parse tree. It is implicit that if a grammar is ambiguous not only will there be a sentence with more than one parse tree, the same sentence will also have more than one abstract syntax tree because the abstract syntax tree is just obtained from the parse tree by elevating the operators to the root nodes and replacing the non terminal symbols by their appropriate operators. That is for a restricted class of parse trees. Hence the question was how this grammar is ambiguous.

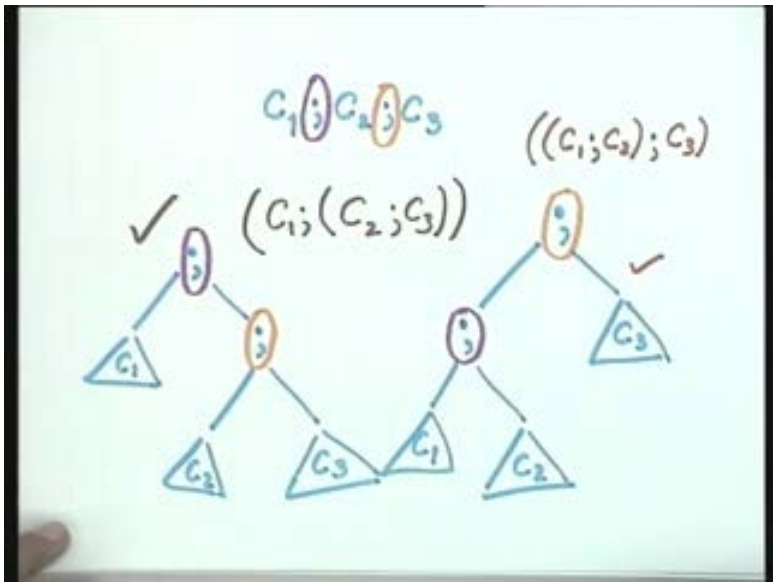
[Refer Slide Time: 02:57]



In the case of these two control structures, the conditional and the loop, I have eliminated the ambiguity by introducing two new reserved words, the closing bracket words ‘fi’ and ‘od’. However, this sequential composition or the sequencing operator which is a binary operator on commands actually gives you ambiguity.

Let us assume you have three commands C_1; C_2; C_3. These three are either atomic or compound commands but these three commands actually give you two different parse trees. These various commands themselves can expand into trees.

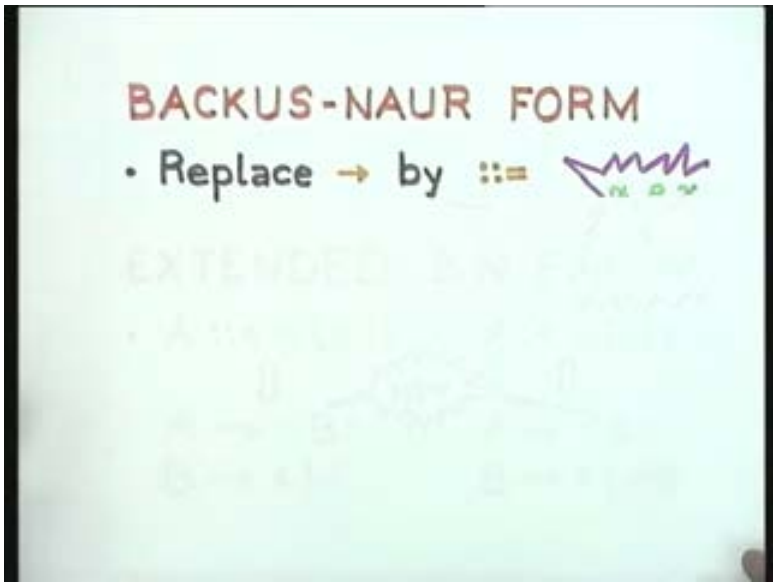
[Refer Slide Time: 07:50]



There is another possible parsing where the first semicolon forms the root of the left sub tree and the second semicolon is the root of the tree. Strictly speaking there is ambiguity in this grammar but the sequencing operation in any programming language is associative. The two trees really correspond to different bracketing. One tree corresponds to the bracketing where you have $((C_1;C_2);C_3)$ and another tree corresponds to the case where you have $(C_1; (C_2; C_3))$ but in general sequencing operation in the semantics is really a function composition operation and the function composition itself is associative and sequencing is also associative.

As far as the implementation of the language is concerned, any implementation can take any decision with regard to the semicolon operation. The fact that it is ambiguous does not matter as far as the runtime behavior or the meaning of programs is concerned. That is a small matter which we have disposed off but in other cases it can change as we saw an example of Boolean expressions where there was ambiguity if you did not introduce parenthesis and it can actually change the value of the Boolean expression depending upon how you parse the Boolean expression.

[Refer Slide Time: 08:10]



Let us see what a language reference manual would look like. Most languages since Algol 60 use a notation called the Backus Naur Form. It is actually a notation created by John Backus and Peter Naur in the definition of Algol 60. The Algol 60 was the first language which used a rigorous syntactic form based on context-free languages and context-free grammars to define the language accurately.

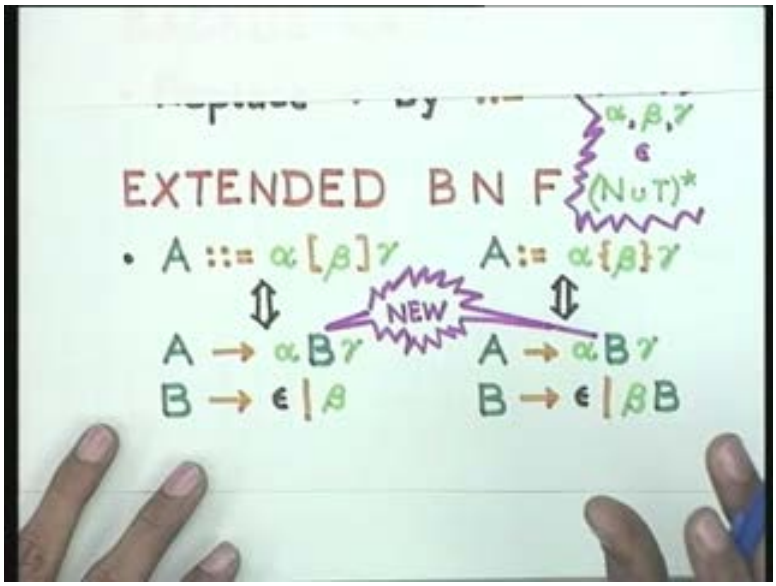
Before that for example Backus was involved in the creation of FORTRAN and the net result was that since there was no clear syntactic definition of the FORTRAN language every FORTRAN compiler written by various people gave different interpretations to the syntax of FORTRAN and what that resulted in was that FORTRAN programs were not compatible across machines.

For example; the way one compiler treated the FORTRAN syntax was different from another compiler and moving programs from one machine or one compiler to another became a huge problem. It became a huge problem in the sense that you required a whole team of programmers to either entirely rewrite that program to suit the new compiler or the new architecture of the machine or it required substantial rewriting and patching up of programs so that they would run correctly on the new machine or on the new compiler.

By that time of course context-free grammars had become quite popular as a form of theoretical study and Backus and Naur defined the Algol 60 language using this notation $\cdot$Replace $\rightarrow$by $:=<\alpha\ \beta\ \Upsilon>$. To ensure readability they did not use single character non terminal symbols but used full words. They wrote statements within angle brackets and since there was no arrow mark, '$\rightarrow$' on the type writer they used double colon equals, '$:=$' which has now become standard.

They wrote all the productions in this form · Replace $\rightarrow$ by $::=<\alpha\ \beta\ \Upsilon>$ in full non terminals being enclosed in angle brackets and the arrow being replaced by '::=' but actually the 'Extended Backus Naur Form' is more convenient. The 'Extended Backus Naur Form' just adds the power of specifying regular expressions within context-free in a convenient fashion. It is merely for convenience. Firstly, regular expressions are also context-free. But there are many constructs which for example allow for options or 0 or more occurrences and in order to specify them by a context-free grammar with the limited notation, new non terminal symbols are introduced to allow for those kinds of iterations. So, the extended Backus Naur Form essentially uses the Backus Naur Form extended to include iterations in choice.

[Refer Slide Time: 12:07]



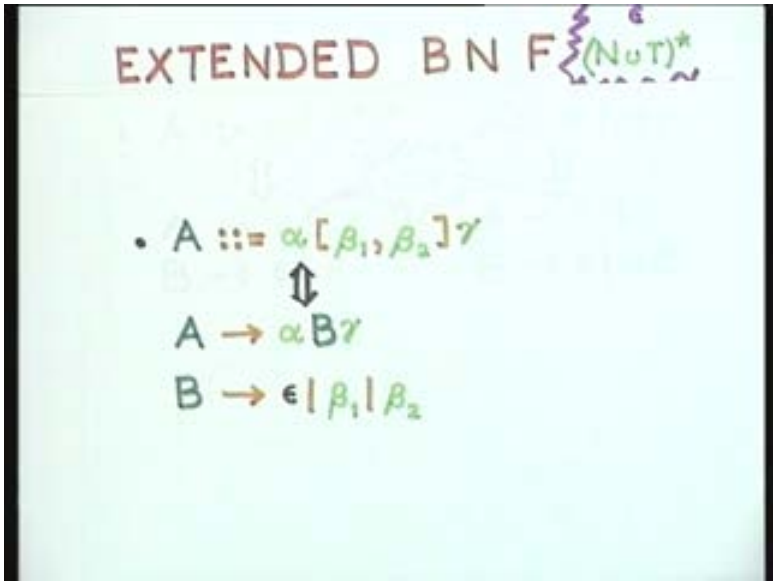
For example; you had a Backus Naur Form production of this form A where Alpha (α), Beta (β) and Gamma (Υ) are strings of terminals or non terminals. If you had a production of this form A ::= α[β]Γ (note that I am using my light brown brackets for the Backus Naur Form notation or for the context-free grammar notation) this would be equivalent to the two productions A -> aBΓ where B is a new non terminal symbol which is not already there in your set of non terminals and B -> E/β.

You will see this extended Backus Form Notation quite often. For example; on any Unix machine if you run the man pages for some command you will find that there are various options and switches given and they are usually enclosed in brackets of one kind or the other.

Normally they use square brackets to represent the various options. A typical example would be that if you had more than one option as in this case; the several options are either separated by bars or by commas and what it means is that this is equivalent to this set of productions with a new non terminal symbol since in the definition of a programming language you do not want to clutter it up with new non terminal symbols

which do not have any particular significance, except that they aide in writing out a grammar more systematically.

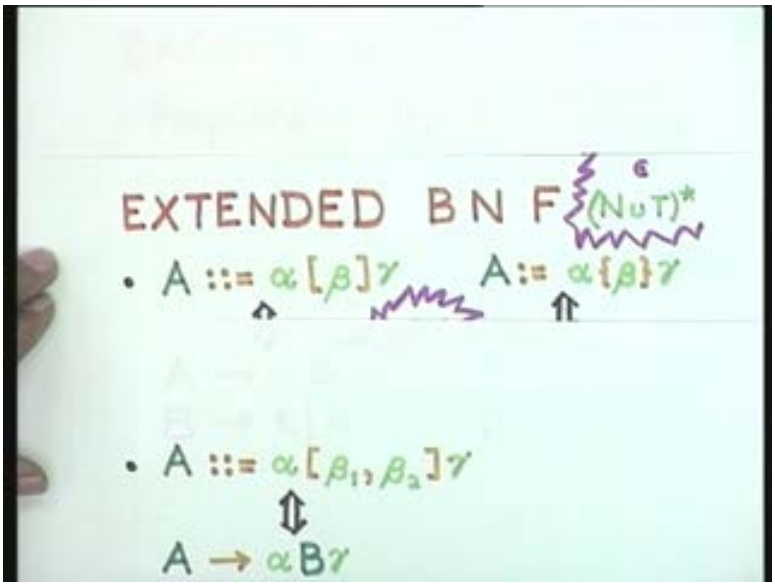
[Refer Slide Time: 13:48]



If they do not have any logical significance you would not like to introduce them. For example; if in a language like Pascal you allowed both the statements, the ‘if then’ statement and the ‘if then else’ statement, then the else clause is an option which ideally could be separated out. But then the else clause actually belongs to the ‘if’ and the ‘then’ as a logical grammatical entity so you would not like to separate it. You would put that else clause in the definition of your language, if your language allowed an else clause and a normal one arm conditional like ‘if then’. You would put the else clause within these square brackets, [ ]. You can therefore reduce the amount of the number of non terminal symbols.

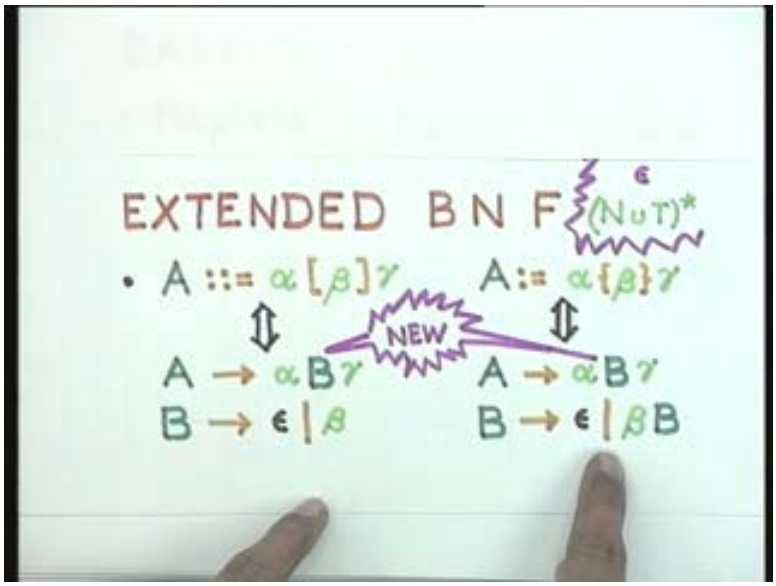
Remember that a real world programming language, which is actually being used is quite a large piece of syntax by itself without complicating it further by adding these extra non terminals that do not have any particular significance. They have significance for the compiler and for accurately specifying the language. They should have significance for example with respect to ambiguity, parsing etc. But otherwise as a logical entity they do not have grammatical significance from the non terminal in which they are actually specified as an option. You can have a 0 or more repetitions of some option.

[Refer Slide Time: 16:13]



If you were to have a production of this form $A ::= \alpha[\beta]\Upsilon$ then this is equivalent to saying that there are two productions of the form A -> α B Υ where B is a new non terminal and β denotes 0 or more repetitions of the string B in your production of the string β. B either goes to $\in$ which denotes a 0 number of repetitions or it goes to β B so that you could have more than 1occurrence of a β. This is the extended BNF notation which is normally used. The extended BNF notation is also very convenient for other practical reasons. For example; if you look at any Pascal manual the syntax diagrams of Pascal are directly equivalent to the extended BNF notation. Therefore you just have to follow the arrow marks in the syntax diagram and they actually give you the productions.

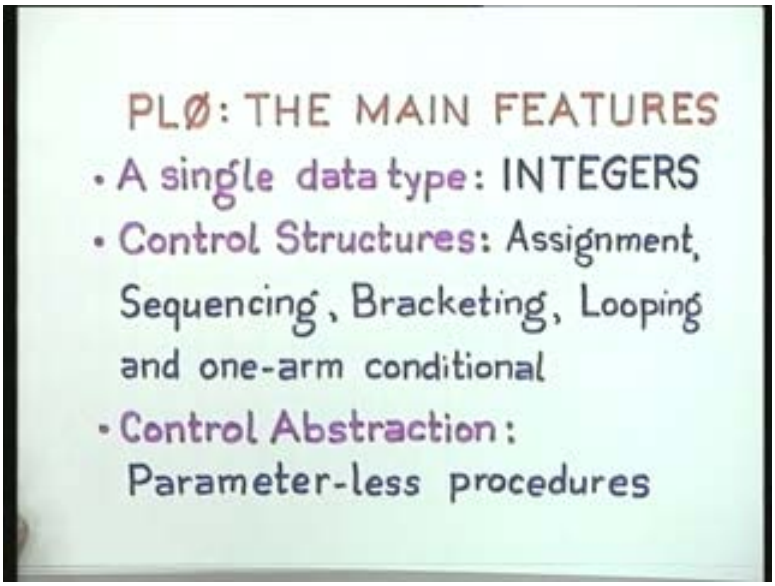
[Refer Slide Time: 17:21]



They correspond more to the extended BNF notation than to the ordinary context-free grammar notation that we have already seen. It is important to know this for reading manuals, for learning a new language and in general to know about the language. Let us look at a language. The last time we looked at a toy language which did not even exist. Let us now look at a language which actually does exist. This is the language PL0, which is very much like Pascal. It was designed for the purposes of teaching programming languages, compilers etc. as a first course by Niklaus Wirth himself; the designer of Pascal.

He has also written the compiler which is available with us. PL0 is even smaller than Pascal and it has a single data type. The main features are that it has a single data type of integers and no other data types. The only control structures are assignment, sequencing, bracketing, looping and a one arm conditional that means it has an ‘if then’ statement.
There is no ‘if then else’ statement. You can program an ‘if then else’ as two one arm conditionals in sequence where you can negate the Boolean. There are no Boolean data types in this; what it means is that you will have to encode your Booleans as integers. You can probably use 0 for false and 1 or any thing greater than 0 for true. But it does contain one important feature and that is the control abstraction mechanism.

[Refer Slide Time: 18:30]

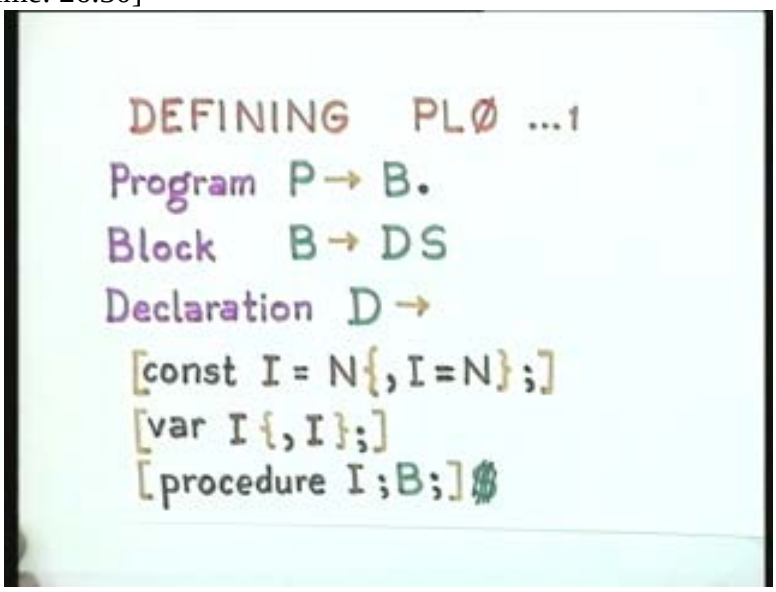


It actually contains parameter-less procedures which allow for a step wise refinement of programs. They in turn allow for complicated programs to be written in a structured fashion and it also allows these parameter-less procedures to be nested. That is you can have a nesting which allows for a step wise refinement in the development of programs.
Let us look at the syntax of PL0. I will keep using the arrow mark instead of the double colon equals, for a production, but I will use the extended BNF notation otherwise.

Let us define it in a top down fashion. My start symbol is P in this case. I will not explicitly specify the terminal symbols and the non terminal symbols. It is obvious that the terminal symbols are those that are colored black and the non terminal symbols are others. The start symbol however is P.

A PL0 program is a block which terminates with a period, $P \rightarrow B$. As in the case of Pascal programs you terminate the program with a period. A block consists of a declaration followed by a statement, $B \rightarrow DS$. For brevity I do not write full names for the non terminals. I have tried to use single letter non terminal symbols whose meaning is quite obvious. This clause, [const I=N {, I=N};] is optional since it is enclosed in light brown brackets. This word ‘const’ is a reserved word. I have put these, I and N in dark brown; they stand for identifier and number.

[Refer Slide Time: 26:30]



The reason I have put them in dark brown is because they are not really non terminals. There could be an infinite set of identifiers possible and definitely an infinite set of numbers. We are considering an ideal machine for which we are specifying the syntax. The actual limits on the numbers and the lengths of the identifiers are just going to be implementation dependent. That is not part of the syntax definition of the language. They are actually terminals but then there are an infinite number of them possible.

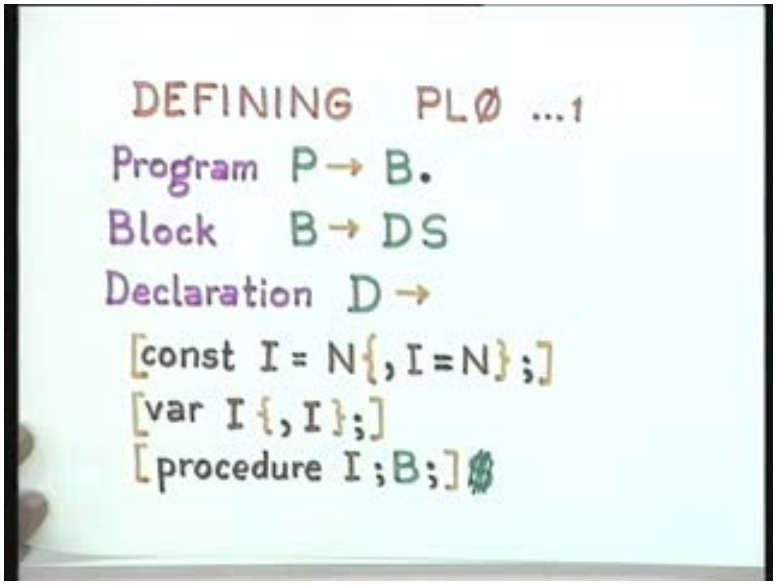
Just take this dark brown, I and N at face value as identifiers that is names and numbers. Since we have only integers those are the only kinds of data type we have and you could have a number of constants specified separated by commas. So the comma is a reserved word of this language and the moment the word ‘const’ occurs, there has to be at least one constant definition. This combination specifies one or more occurrences of the clause I = N. This also specifies how they should be separated. They should be separated by a comma. If there is a ‘const’ reserved word it should be terminated by a semi colon. So, you can define a sequence of constants in a single declaration and terminate the entire declaration by a semicolon. You do not need to have any constants at all and so this entire clause $[\text{const } I = N \{, I = N\};]$ is optional.

Whether you have any constant declarations or not you could have variable declarations but you do not need to, so even that clause is optional. Since there is only one kind of a data type, it is not necessary to explicitly specify what the variables are going to be of. You can just specify the variables separated by commas but the moment you have this reserved word ‘var’ occurring, you have to have at least one identifier which in this case is ‘I’. If you have more, they should be separated by commas and if you have this ‘var’, the entire variable declaration will have to be terminated by a semi colon. This is also optional. You could have procedure declarations and they are parameter-less procedures and ‘procedure’ is a reserved word. There should be an identifier which should be terminated by a semi colon and there should be a block which is again terminated by a semi colon. The entire procedure is also optional. You do not need to have any

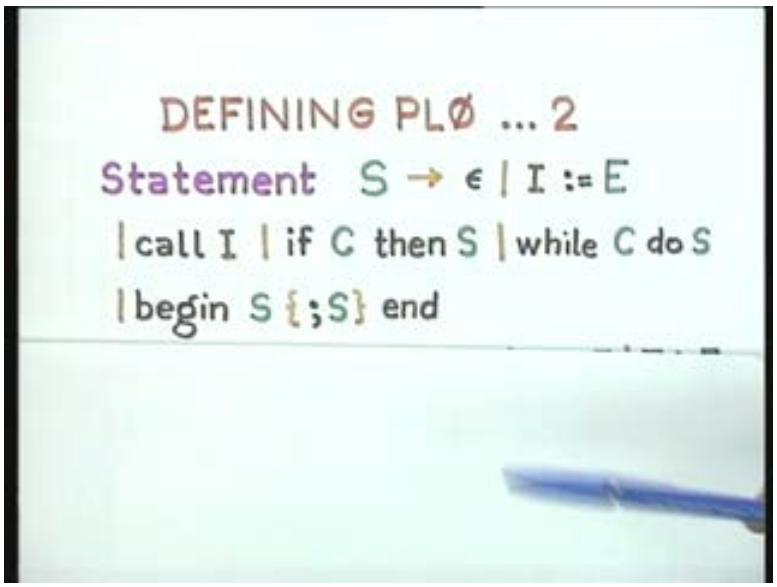
procedures in your program and whether you have one or more of these clauses, [const I =N {, I =N};]
[var I {, I};] [procedure I; B;]they should occur in this order and you have a statement.

Since all the three declarations are optional, you could have an empty declaration too.
The production $D \rightarrow \in$.

[Refer Slide Time: 26:30]



[Refer Slide Time: 29:45]

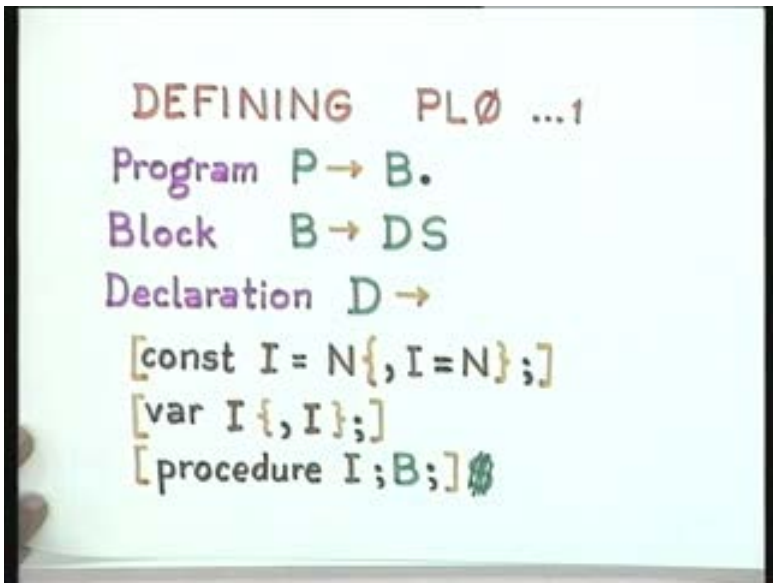


Let us look at the statement which is S. A block just consists of a declaration followed by a statement. Let us look at this definition of statements. A statement itself could be empty; a declaration could be empty, which means an empty string itself is a program. A statement could be empty otherwise you have an assignment. An assignment statement is a statement where an assignment of course is an identifier. Colon equals an expression. Note that there is absolutely no relation between the identifiers declared in the declaration and the identifiers that are used in the statement 'Statement $S \rightarrow \in | I := E$'. The syntax is context-free but the language feature is such that you cannot use a variable without declaring it.

That is a context-sensitive feature which is not specified in the syntax of the language. Then you have an explicit procedure call statement, 'I'. You can 'call' an identifier I and the implicit meaning is that this identifier 'I' should have been declared as a procedure otherwise you cannot use 'call'. 'Call' is a reserved word. You have the one arm conditional, 'if C then S' which is 'if' a condition 'then' a statement and 'if C then S' is a single statement. You have the looping construct 'while C do S', which is just as in Pascal and you have compound statements. You can colas a sequence of statements by bracketing them with a 'begin' and 'end' and call that a single statement, '|begin S {; S} end'.

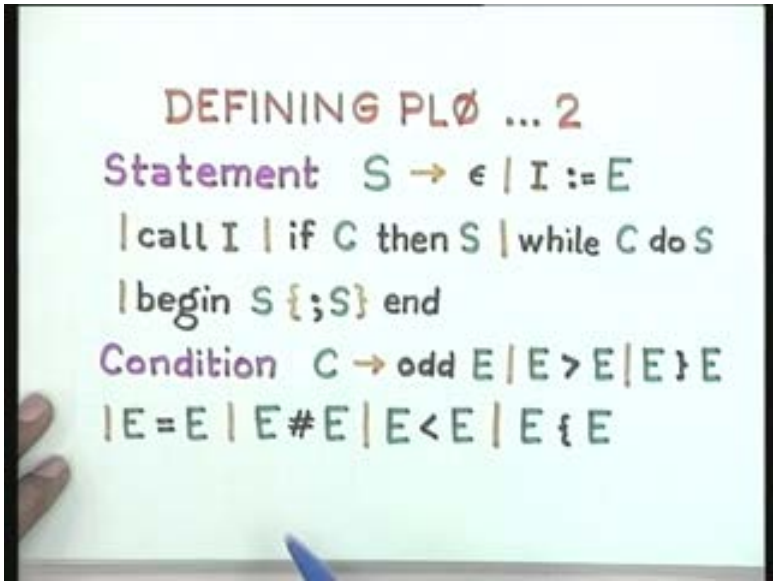
This just says that you can have a 'begin', you can have an S (note that since '$S \rightarrow \in$', the 'S' on the left hand side in '|begin S {; S} end' could also be empty.)
Then you can have 0 or more occurrences of 'S {; S}'. For example; just a pair of brackets 'begin end' itself is a statement. It is a trivial statement which does nothing. It corresponds to a no operation in hardware. So, you could have 0 or more occurrences of this grammatical entity, 'S {; S}' and they are all separated by a semicolon and '|begin S {; S} end' is a statement.

[Refer Slide Time: 29:55]



If you look at the only non terminals that have not really been defined, they are I and N. We will define them in the end but all other non terminals here have been defined but they have also been defined at the expense of introducing new non terminals. For example; expressions and conditions are aspects that we have to define.

[Refer Slide Time: 30:05]



Firstly, there is a unary condition for an expression E (note that since the only data type available is integers, the only expressions available are also only integers). This unary predicate 'odd' applies over all expressions and yields a 'true' or 'false'. The reason for using this 'odd' is partly because it is nice to have some unary predicate. Secondly, the reason for choosing 'odd' is because there is a direct jump on not equals in most hardware so, where ever you have jump, or even in your high level programs, a large number of your programs are of the form 'while some variable is not equal to 0 do something'. In all those cases you could check for oddness of that variable because a variable is also an expression. You could use C $\rightarrow$ odd E as a condition. Otherwise you have various binary relational operators. I have simplified the original language to a certain extent by using single-letter relational symbols. That is why you have odd looking symbols. This is a standard greater than, E $>$E. This is greater than or equal to, E $\geq$ E. This is equals, E $=$E. This is not equals, E $\neq$ E. The original Pascal compiler for example allowed this, $\neq$ as a not equal symbol. This is less than, E $<$E and this is less than or equal to, E $\leq$E.

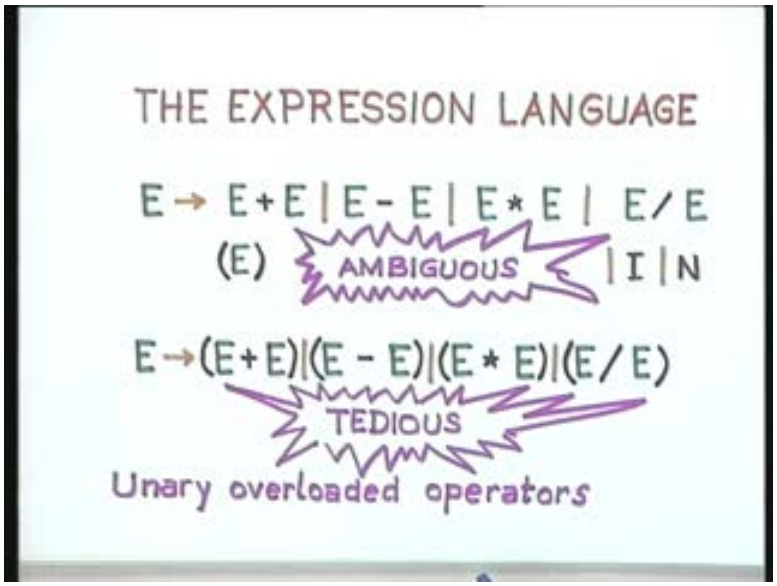
The original PL0 compiler as defined by Wirth actually assumed that the '$\leq$' and '$\geq$' are already available on the key board, but they are not. So, I will have to change that. Conditions really depend upon expressions either through unary predicates or through binary predicates.

Let us look at that expression language. Before I get into the expression language I would like to say that in the case of the expression language you have a compromise between two extremes. One is that all of us think of expressions really in this form,

$E \rightarrow E + E | E - E | E * E | E / E$
(E).

Consider the four operators that you have; addition, subtraction, multiplication and division. We normally think of an expression as being a sum of two expressions, a difference of two expressions, a product of two expressions or a quotient of two expressions and we use brackets (E).

[Refer Slide Time: 37:45]



However, what happens is that this grammar is ambiguous because it does not use parentheses sufficiently. It is possible to generate sequences which you understand to have a certain priority and order of evaluation. If you define the expression language without parentheses the compiler need not process it and therefore, it is ambiguous.
Every variable is an expression and an integer constant is also an expression. Those are the atomic statements of this grammar. The other extreme is that you fully bracket every expression. Whenever you have a binary operator, you put a bracket around the pair of operands. So you can have (E + E), (E – E), (E * E) and (E / E) enclosed in parenthesis.

However, most of us find it tedious to actually write parenthesis over every thing and you will have to key in parenthesis every time. Whenever there is a binary operator, for instance, you would have to key in parenthesis. So if we are looking at abstract syntax in string form we will just assume them to be fully parenthesized. If we are looking at abstract syntax as in tree form, we will just draw the trees corresponding to whatever order of evaluations we want because any fully parenthesized notation can be translated into an appropriate abstract tree which preserves the order of evaluation of the

expressions and vice versa; given any abstract syntax tree, you can transform that into a fully bracketed string of symbols.

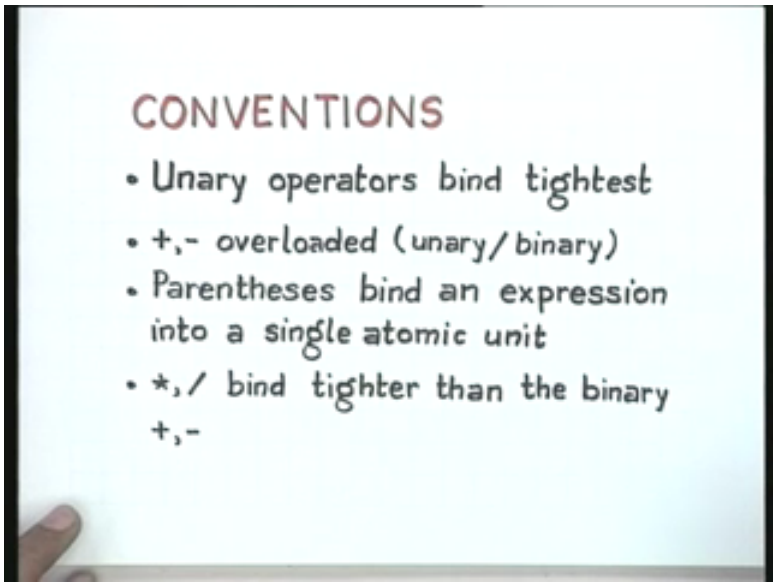
But from the point of view of a compiler because it is ambiguous it is clearly unacceptable. As far as this is concerned, it is tedious for every programmer to write fully parenthesized versions and this makes it inconvenient. So, one has to strike a reasonable compromise and try to define the expression language in such a way that it is not ambiguous, not tedious and it follows all the normal conventions of mathematical notation. In parsing, this '+' and '-' are also overloaded unary operators. For example; you can take negative numbers or you can write positive numbers as plus something so then they become unary operators.

A negative number is just the unary minus of a non negative integer. It is a unary operator there whereas when it occurs in such a form these '+' and '-' are binary operators. We have to take that into account too. There is overloading and we have always used a lot of overloading too in our mathematics. For example in our programming too; addition, subtraction, multiplication and division are used for both real data types and integer data types. In the Pascal language they are also used for set operations. They are overloaded tremendously and one should take care of it.

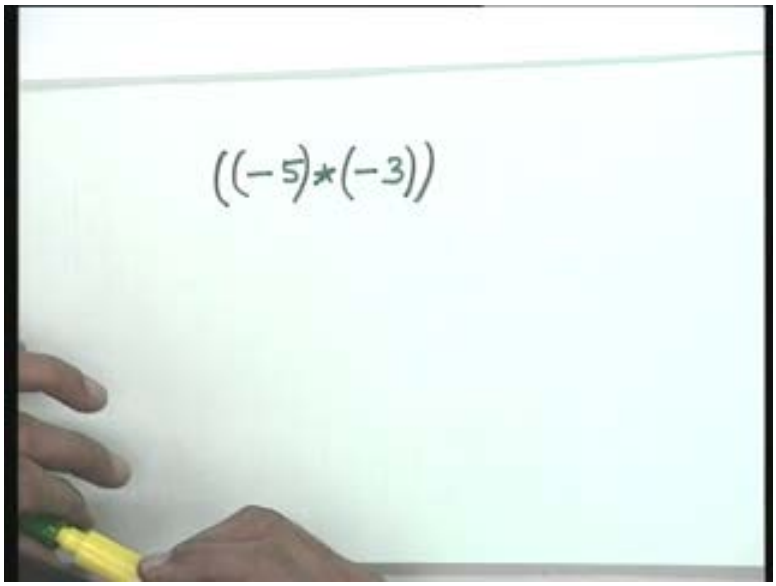
We follow some normal conventions. They are the following. The unary operators usually bind the tightest. This means that a unary operator's influence extends over to the first available symbol and it takes precedence over all other operators except when there are parentheses. If you have a unary operator and a large expression enclosed in parenthesis then it is the negation of that entire expression and not of the first symbol after the left parenthesis.

Plus and minus are of course overloaded and the normal mathematical convention is that multiplication and division bind tighter than the binary operators, '+' and'-'. However, multiplication and division lose precedence over the unary operators, '+' and '-'.

[Refer Slide Time: 37:50]



[Refer Slide Time: 39:00]



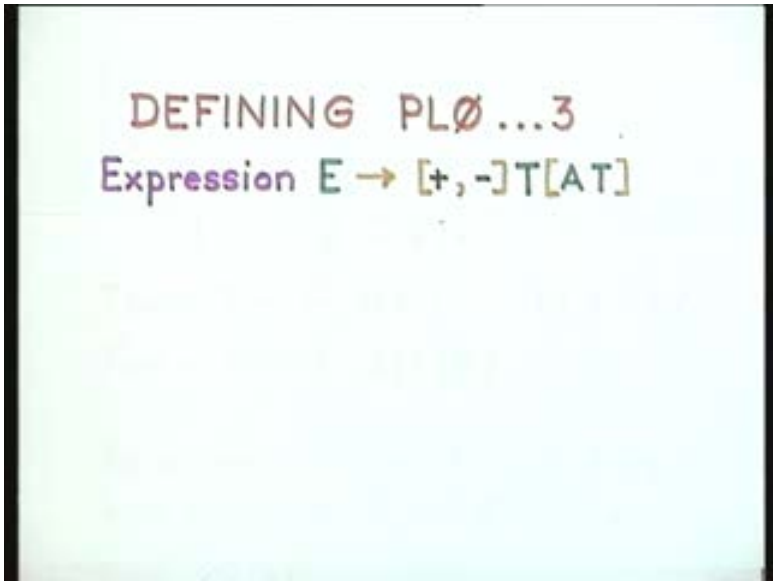
For example; if you have an unparenthesized expression of this form,
((-5)*(-3)) or ((-5)*(3) then the first minus refers to the first 5 on the left and not to the entire expression and the '–' before 3 refers to 3 and this '*' binds -5 and -3 and so the appropriate bracketing is this, ((-5)*(-3)).

We have to take these conventions into account for giving a friendly user interface as far as the expression language is concerned so that people with a normal knowledge of mathematics, mathematical notations and mathematical conventions can write programs and expressions that they have been normally trained to write.

Let us look at the expression language. The provision of this convenience means that you require a fairly large number of non terminals before you can expect to define it unambiguously.

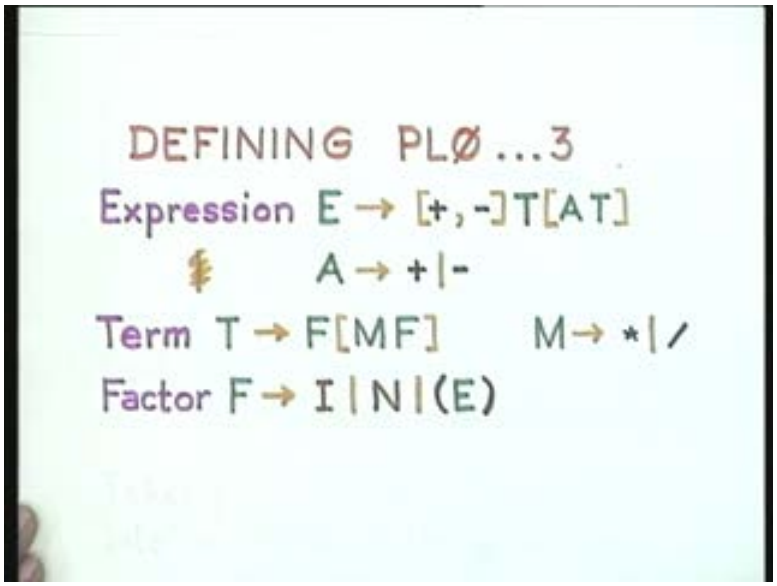
This language of expressions, Expression E→[+, -] T [AT] is available in all books which deal with parsing or compiling etc. You can see at a glance that it will work. An expression is a term which might be preceded by a unary plus or minus or since [+, -] is an optional clause it may not even be preceded by it. It might be an unsigned term followed by optionally an addition operator, A and a term, T.

[Refer Slide Time: 40:27]



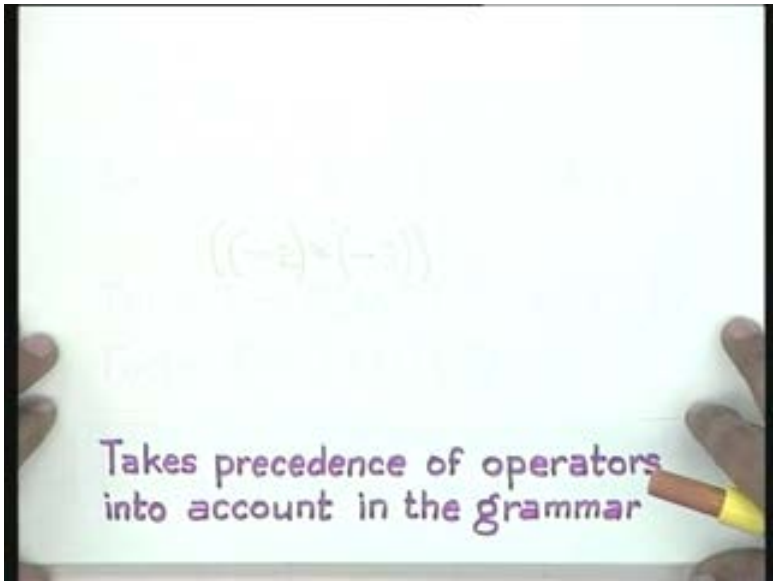
The addition operators are just the binary plus and minus. Let us disregard this. Any expression can be regarded either as a term, a signed or an unsigned term or a signed or an unsigned term followed by an addition operator and another term. A term is either a factor or a product of two factors or the quotient of two factors. This M is a multiplicative operator, M -> *|/ (a star and division). The multiplication and division are multiplicative operators and a term is something of this form, Term T→F [MF}. Either it is a factor, F or it is a product or quotient of two factors. A factor is anything regarded as a single unit. Either it is an identifier, which means that we are normally talking about variables or constants, or it is actually a number specified as part of the expression or it is a whole expression in itself enclosed in parenthesis (E). These three non terminals; E, T and F are mutually recursive because E is defined in terms of T, T is defined in terms of F and F is again defined in terms of E. They are mutually and circularly recursive. They actually take into account the fact that you can look upon an expression as basically the sum of two things, so you do not expand out into a term.

[Refer Slide Time: 42:58]



If you have the outermost operation as an addition to be done, you have some large expression whose root operation is an addition operation. That means that if it is the last operation to be done then your context-free grammar is such that the left operand (supposing is just an identifier) will allow it to go from E to T then from T to F, F to E again or F to I and from 'I' again it will keep circularly revolving. This grammar really takes precedence of operators into account and this syntactical definition is absolutely essential for writing the compiler.

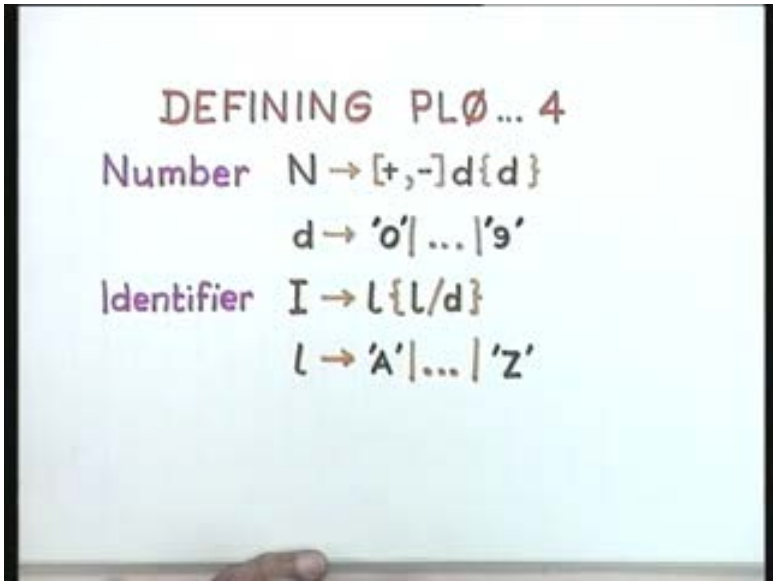
[Refer Slide Time: 44:41]



For pragmatic reasons it is necessary to have this kind of a syntactic definition but semantically we will just look upon an expression as a signed or an unsigned expression or a binary operator just in terms of abstract trees. We will either look at it, if you want to represent them linearly, either as fully parenthesized expressions or as abstract syntax trees.

Let us come to the last part which is the definition of a number. A number is either a signed or an unsigned integer. This [+, -] is an optional clause and the definition is that it should have a digit followed by more digits. A digit is defined as a character 0 to 9. We follow normal Pascal rules. An identifier should start with a letter, L; in the case of the PL0 compiler all the alphabets consist of only upper case letters. It is very trivial to modify it to include lower case letters too. What really distinguishes a number from an identifier is the occurrence of a letter or a plus or minus symbol or a digit which is what really distinguishes it.

[Refer Slide Time: 45:25]

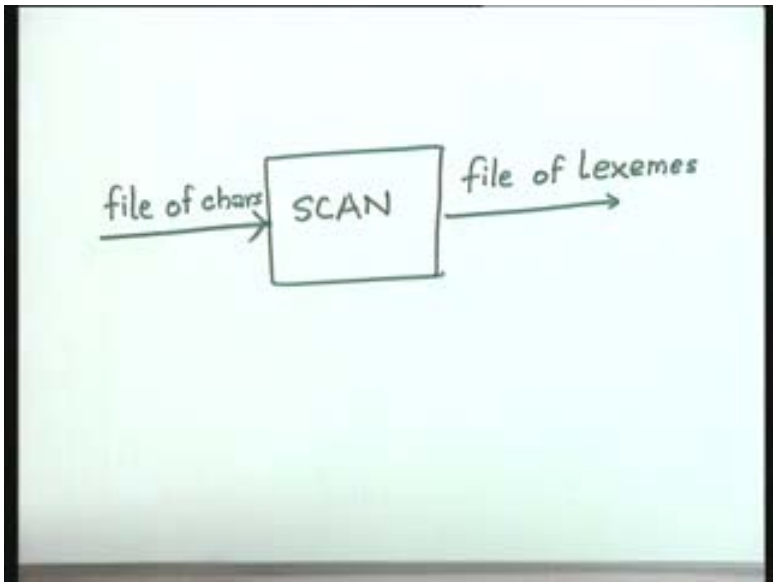


This plus or minus is really an operation. If you have a negative number you are actually going to take the corresponding integer and negate it as an operation. In the case of an identifier that is why it should begin with a letter and it may be followed by one or more letters or digits. The reason for removing N and I productions from the main grammar is that the rules are really not part of parsing. They are part of what is known as a lexical analysis. For example; you can write such rules also for recognizing that the word ‘while’ -w h i l e has been recognized as a single word. The word ‘begin’ has been recognized as a single word. These are actually part of the process of what is known as scanning or lexical analysis.

Typically, a program written in this language is a file of characters and we would like to divide up the program into words or what are known as lexemes which really describe

each entity in the program. Instead of a file of characters we would like to regard it as a file of words. We can regard it as a file of words by recognizing all the reserved words by scanning all the words and deciding whether they are reserved words. In case they are not reserved words you should be able to treat them as identifiers or if they are constants, you should be able to read out the entire constant in this case, a string of digits representing an integer or a plus or minus followed by a string of digits representing an integer.

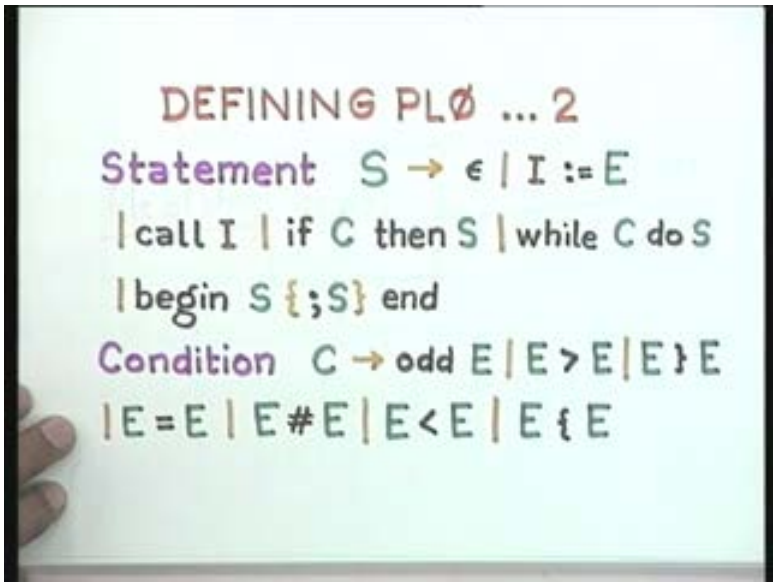
[Refer Slide Time: 50:25]



Then you would treat the whole unit as a single unit. So, a scanner typically takes a file of characters and gives you a file of words. The user program is just a file of characters and you get a file of lexemes. The word file is used in a very general fashion. It does not mean a desk file but any unbounded ordered sequence of objects. The process of scanning converts a file of characters into a file of lexemes and then the process of parsing actually takes over the handling. That is one reason why I have not worried too much about these reserved words.

The process of scanning would have created a single lexeme out of all these. They are all a single unit and after scanning they would lose the status of being a string of characters. They would be some single unit in the form of an element of some structured data type which gives the identification as to what this unit is. It could be an index into a table with other details such as it is a reserved word or it is an identifier or a constant. It is striped in a more complicated language whether it is an integer type, etc. and will be filled on later after the process of parsing is through.

[Refer Slide Time: 50:30]



You will actually create a huge table of the amount of information that you have to extract from the program through all the process of compiling and keep it for use for type checking, run-time type checking and compile-time type checking to detect undeclared variables which are also a good way of detecting spelling mistakes.

All these terminal symbols will actually become single units in some table and the file of lexemes will just be a single unit which gives an index into the table so that that table is resident always in memory for reference during the process of compiling. For example; if you have to check various context sensitive issues such as; if it has been declared before, what its type is if it has been declared before, if it has been assigned the right type and if it is being used in an expression in the right type you require a table of information for each identifier, reserved word or each lexeme to know whether it is being used correctly in the program.

We have covered the syntax of the language and we will define the semantics of the language. We will start the next lecture with the basic notions of semantics. We will then add on new features to this toy language and see how they have to be defined. The syntactic definitions of these new features are not very critical because you have all the basic material. As long as you can parenthesize expressions and the new features that you are introducing without ambiguity or as long as you can define them in some reasonably good syntax, it is not very important how they look. It is more important how the abstract syntax trees look and what meaning you give to the abstract syntax tree.