Compiler Design
Prof. Santanu Chattopadhyay
Department of E & EC Engineering
Indian Institute of Technology, Kharagpur

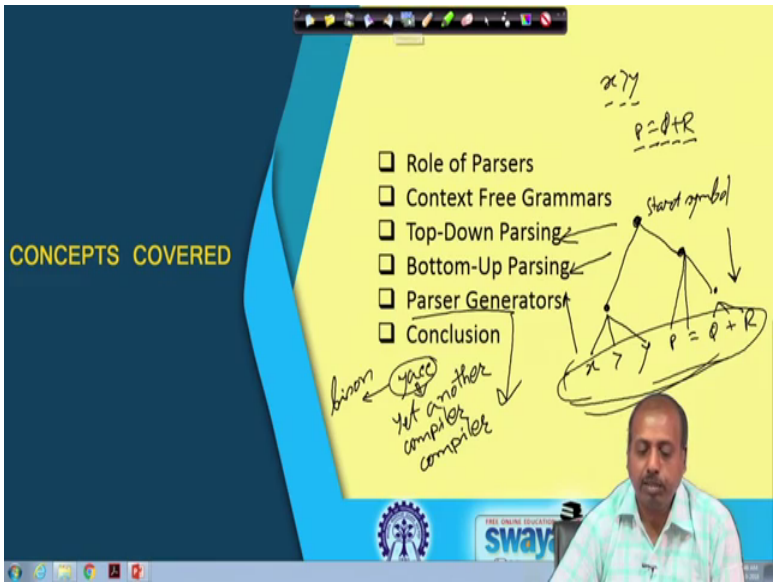
Lecture – 16
Parser

The next phase in the compiler generation process or the Compiler Design process is designing a tool for syntax analysis purpose. So, we have seen so far the lexical analyzer. So, that can return the tokens that are appearing in the language so, valid words of the language. Now, these words are to be connected or there are they need to be sequenced in a proper fashion for some meaningful constructs of the language. So, whatever be the language starting with English to some programming language.

So, there is some sequence in which this token has to appear. So, lexical analyzer so, it cannot determine that sequence. So, it can just give you the tokens that are coming or the words that are coming in the input source file. And, the next job is to try to is to see whether these words are appearing in a proper sequence or not. So, if they are appearing in a proper sequence then they are following the grammar rules of the language. And, accordingly the entire program is accepted as a valid program, entire input file is accepted as a valid input for the language that we are considering.

So, this phase so, we will be discussing on this syntax analyzer design or the task of syntax analysis and it is very complex. In the sense that depending upon the programming language constructs so, which are quite varied in nature. So, it is often very difficult to come up with some syntax analysis tool. And, depending upon the constructs that we have sometimes we can have a very simple type of syntax analyzer whereas, if the language is quite complex, the grammar is quite complex in that case the construction also becomes difficult and many times we need to do a trade off. Like some so, we have to use the knowledge or the intuition of the compiler designer to resolve between different issues that are appearing in the reduction process.

(Refer Slide Time: 02:23)



So, the our discussion will for flow in this way, first we will discuss about role of parsers. So, parser is the technical term for this syntax analyzer. So, it parses the input file into the grammar of the language. So, if it is successful; if it is successful in the sense that if the grammar is the if the input file is grammatically correct.

So, it follows a certain pattern by which this entire program can be derived starting with the start symbol of the grammar. So, that is the role of the parser. So, parser will try to give a proof that this particular input file follows the grammar of the language. And if it is successful in proving it so, it will give it will return a parse tree which is constructed by being in terms of the terminals and non-terminal symbols that are used for specifying the grammar. And, then after that based on that parse tree the compiler designer can try to do so, take some actions.

So, that it can generate code or it can do some other transformation to the input pro input file, so on. So, that is the basic rule of parser. So, as I think now you can follow that it is it has to check all the grammar constructs of the language. So, as a result it is going to be quite involved, then there are different types of grammars. So, we will be discussing mostly on the context free grammars because, most of the programming language construct so, they belong to this category of; this category of grammars ok.

So, most of the programming language they follow their constructs can be specified by means of context free grammars. So, we will be mostly discussing on context free

grammar. After we have a after introducing these grammars and all we will be looking into is 2 class of parsing technique: the 1st class is known as top down parsing and the 2nd class is known as bottom up parsing. So, it is like this that suppose I need to; suppose I need to develop some, suppose I am given an input line say x greater than y then say if as a P equal to say Q plus R something like this is given.

And there are some rules in the language, where there if there are some rules in the grammar that are used for specifying the language. Now, one possibility is that you start with this basic symbols like P equality Q plus R, x greater than y etcetera. So, you start with those symbols x greater than y, then P then equality symbol Q plus symbol R and all that. So, you start with that and then try to combine them in some fashion to such that in a tree type of structure; maybe you try to combine say these 3 together into some symbol. Similarly, maybe you try to combine so, these 3 together into some symbol. Then you try to combine say these 3 into some symbol then finally, you try to combine these 2 into some symbol where this final one is the start symbol of the grammar.

So, this is the start symbol of the grammar. So, what is happening is that ultimately we are producing a tree and the way I have described it; I have started with the at the lowest level the input file itself and from there I am trying to reduce that input file to the start symbol of the grammar. So, this type of approach so, they are known as bottom up parsing because, I am starting with the symbols or the words of the language and then trying to combine them together towards the start symbol of the grammar. Just the reverse approach that is given a program, we can start with the start symbol and then try to see like what may be the possible set of rules.

So, that ultimately it boils down to this particular program. So, it is actually looking in this direction. So, previously we were going in this direction and now we are going in the opposite direction, starting with the start symbol we are trying to go down and reach the final program. So, that particular approach will be known as top down approach. So, in general top down approach they are simpler than bottom up approach. However, there are restrictions also like we will see that for certain languages; so, it may be difficult to construct top down parser because, of the nature of the grammar whereas, it may be possible to construct the bottom up parsers for them.

And in general so, we will see that the bottom of parsers often they make a superset of this top down parsers. So, for whatever any language if you can construct a top down parser you can also construct a bottom up parser, but the reverse is not true. So, this way we will be looking into both top down and bottom up parsing strategies because, bottom up parsing is costly ok. It is so, constructing the parser is difficult. So, it is costly and it takes more time for constructing the parser. So, many simple grammars; so will be we can very easily construct the top down parsing technique.

And then we can they have a parser for that. So, we will look into some examples from both the categories, then we will be looking into some parser generators. So, as we go through the chapter then we will see that the there are algorithms and techniques so, which are fine. So, that they are all automated that is so, once you understand that theory. So, it is possible for us to write down the corresponding program which will act as the parser, but due to the sheer volume of the programming language grammars that we have in today.

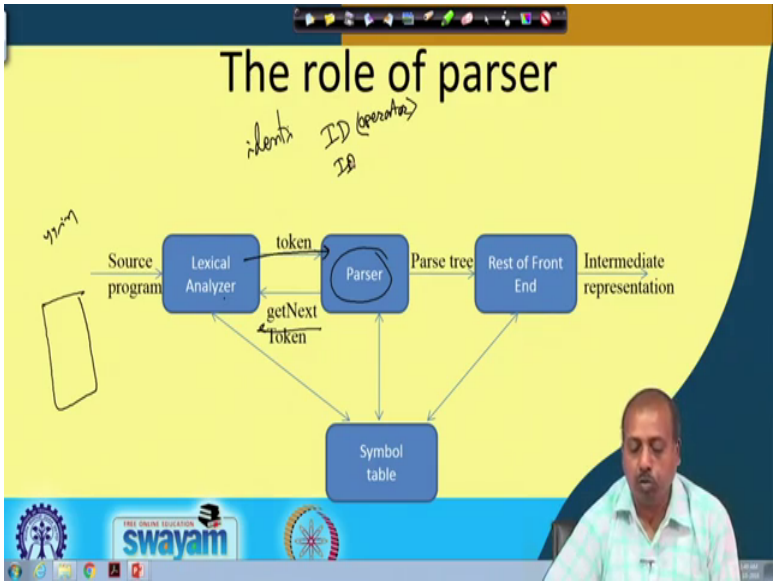
So, it is very difficult to construct those parsers by hand. So, we will be looking into some tool which will be used for this which will be acting as this parser generator. Just like for lexical analysis tool so, we have a lexical analysis job so, we had the 2 legs. So, here also for parser generator so, we have got tools called yacc. Then so, the yacc is yet another compiler generator, compiler compiler Yet Another Compiler Compiler.

So, what does it mean? Why compiler is coming twice, is because there because of the fact that it is a compiler for a compiler. So, it has you specify a grammar and for that it will generate a compiler. So, it is based that is why it is a compiler compiler. So, it compiles a compiler already makes a compiler. So, that is why it is like this is yet another compiler compiler and the abbreviation is yacc. And, just because the abbreviation is yacc some other versions of the tool that has come up one way one of them is known as bison and there are many others I believe.

So, the bison does not have any full form. So, this is basically taking yacc as an animal. So, it is taking bison as another tool, but whatever it is so, basic principle remains same. So, they are all parser generators and they can be used for automatically generating parser from the specification. So, our job is to see that the specification is correct and we

understand how this parser generated so, work actually. So, once we do that so, we will be able to write our own parsers. So, throughout the chapter we will see this thing.

(Refer Slide Time: 10:15)



So, let us start with the role of a parser. The role of a parser is like this. So, the parser actually is sitting somewhere here. So, this is talking to the lexical analyzer tool for tokens. So, it whenever it needs a token to for proceed so, it will ask the lexical analyzer get next token. So, to provide the next token to it and as we understand as we have seen in the lexical analysis tool so, lexical analysis tool so, it actually scans the source file. So, if this is the source file so, it has got the pointer y y in and that pointer from that pointer it will try to see what is the maximally matched token for this for the next inputs part.

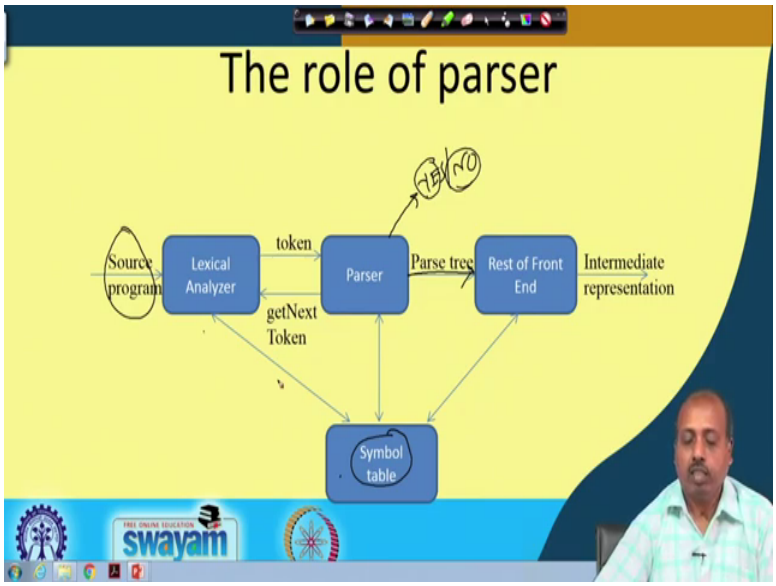
So, accordingly it will find out the token and return it to the parser. So, parser getting this token will try to see like what may be the corresponding action. So, it may try to follow a certain grammar rule for deriving the statement or deriving the line of text like that or it may be that it finds that the token that it has got is not meaningful in this at this point of time. So, that in that case so, that is an error. So, that is a grammatical error. So, typical examples may be that suppose it has seen an identifier ok, suppose it has seen an identifier as a token.

So, ID is the token returned by lexical analyzer. Now, if I consider a language that consists of only arithmetic expressions ok. Then after ID so, it is a expected that I will get an operator here ok. So, it is expected that I will get an operator, but instead if the so,

after getting the token ID from the lexical analyzer the parser asks for the next token. And, the next token if it happens to be again ID; that means, then the parser will understand that there is some grammatical problem or grammatical mistake in the input. So, it can flag that particular error that at this point there is a problem, there is an error because it was expecting an operator and it is getting an identifier. However, the lexical analyzer tool so, it could not understand this particular problem.

So, lexical analyzer tool so, it just scanned the input and whatever is the maximally matched token at that point of time so, it has returned to the parser. So, this lexical analyzer and parser they are working hand in hand. So, they are the whenever parser is requiring token so, it is asking the lexical analyzer tool and then it is proceeding. Then it is trying to see like which grammar rule may be applied for deriving the input and then if it if it is successful then the process will go on. And, if it is if it can do it completely for the entire program then it will generate a parse tree. So, outcome of this parser is basically two things: one is one is an YES NO answer.

(Refer Slide Time: 13:19)



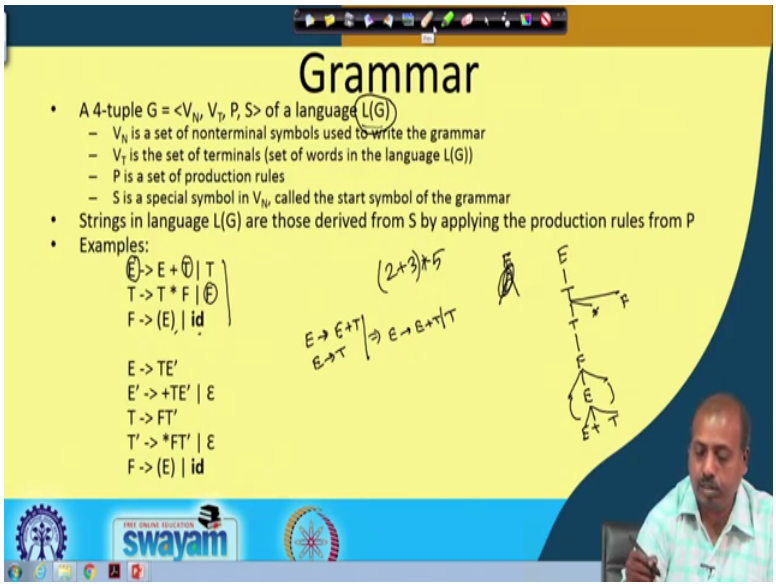
One is an YES NO answer that is whether the source program that is given is following the grammar or not. So, if the source program is following the grammar then the parser will answer YES, if it is not following the answer grammar so, it will answer NO. So, if it answers YES in that case so, we can make it to generate another output which is known as parse tree. So, this will show how the grammar rules have can be applied to

have the input source program derived from the start symbol of the grammar. And then the later stages of the compiler so, they can take help of this parse tree and can generate appropriate output ok. So, it may be some code, it may be some something else whatever it is so, some action. So, whatever it is so, it can do that part and also if the answer is NO, in that case the parser can tell us at which point it found the problem. As I was telling so, it was expecting an operator while it got ID. So, it can flash that message that I was expecting an operator so, there is a mismatch. So, it can flash it can identify that situation. So, it is basically the job of the compiler designer to see that to ensure that those erroneous conditions are checked and those erroneous conditions are appropriately inform to the user ok.

Then the user will correct the source program and again give it for compilation so, it will go like this. On the other hand this parser also takes help of this symbol table. So, as I said as you said that whenever this lexical analyzer finds a new identifier. So, it installs this identifier into the symbol table and returns the index of the identifier to the parser as an attribute in the attribute yy lvl or something like that. So, this parser can again use this symbol table to get further attributes for the symbol.

So, from y from the lexical analyzer so, it has only got the index of the symbol table where the particular identifier is defined, from the symbol table it has got more information like type of the identifier or that that is the type of operators that can be applied on it and all. So, that way the parser can check all those things. So, we have got this particular role of the parser. So, it is central to the compiler overall operation of the compiler. So, it is the main part of the compiler that we have. So, we will see how this parser can be designed.

(Refer Slide Time: 16:11)



To start with we define a grammar ok. A grammar is a 4-tuple G consisting of 4 say 4 parts V N V T P and S. So, the for a language L so, if a grammar is G then the language for it is known as is a denoted by L G. So, L G it denotes the language corresponding to the grammar G and that language has got a and this grammar G definition. So, it has got 4 parse in it V N V T P and S where, V N is a set of non-terminal symbols used to write the grammar. So, V N is that so for writing the grammar so we need some special symbols; so, they will construct the non-terminal symbols whereas, V T is the set of terminal symbols which is a set of words in the language.

So, since there this the symbols that the symbols that are appearing in V T so, they are appearing in the final language. So, they are called terminal symbols whereas, the symbols which are belonging to non-terminal set V N. So, they are not appearing in the final program or the final input so or the final language. So, that is why they are called a non-terminal. So, we should be the final derivation that we have so, there should not be any non-terminal left. So, everything has to be replaced by terminal and then these terminals can be; terminals can be combined in some fashion to using these grammar rules to get the overall parse tree. Then apart from this V N and V T the set of non-terminal and terminal symbols, we have got a set of production rules for the grammar.

So, production rules actually tells like how can we proceed with in the non-terminals, how can how can we replace non-terminals by set of terminals or non-terminals etcetera.

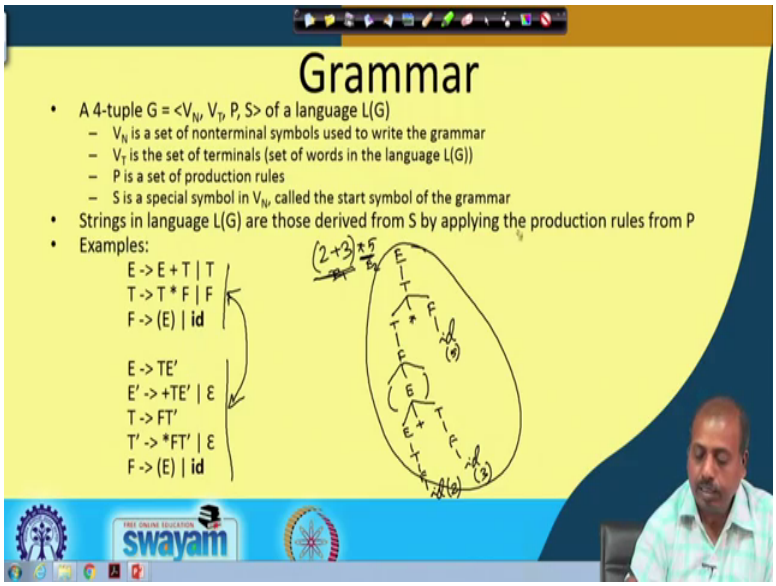
So, we will see some example and S is a special symbol in the set V N set of non-terminal symbols. It is called the start symbol of the grammar ok. Now so, we will take an example. So, this is a grammar this is a grammar; so, in this grammar so, you see that some symbols are termi non-terminal symbols, like symbols like E then T F ok.

So, they are terminal symbols non-terminal symbols whereas, symbols like plus star open parentheses close parentheses id. So, these are called these are the terminal symbols because so, this particular grammar is for arithmetic expressions that has got multiplication addition as the operator and the parentheses is also there. So, you can have an expression like say 2 plus 3 into 5. So, this is supposed to be a valid string of this language. So, how can I have this grammar? How can I have this string?

So, starting with say E so, I can use the first rule E E producing so, this this part. So, we read it as E producing E plus T or T. So, this is actually combination of 2 rules E producing E plus T and another rule E producing T. So, there are 2 rules so, for the sake of brevity so, we just write them together and write it as E producing E plus T or T. So, by combining these 2 so, we will write it as E producing E plus T or T. But for, but you should keep in mind that the these actually mean that there are 2 rules: one rule is E producing E plus T another rule is E producing T, since their left hand sides are common for the rules.

So, we are writing them together. So, let us go back to the point like say for say E I can use this rule E producing E I can use the way I can have a derivation like this E producing T. And, then T producing T star F and then this T producing F producing E and this E produce F producing within bracket E and then this E again giving the remaining part. So, this is actually giving E plus T ok, let me draw it a fresh because it is not.

(Refer Slide Time: 21:21)



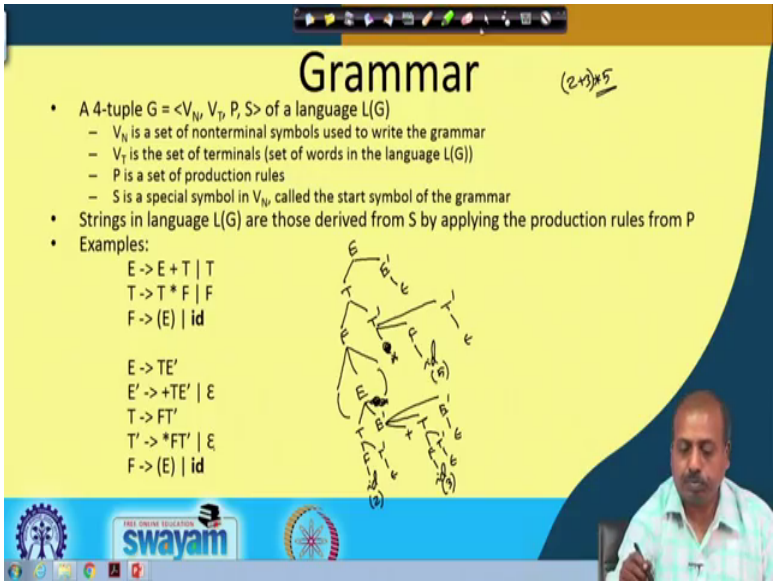
So, the expression that I have is 2 plus 3 star 5 ok. So, I start if I start with E then E producing T first of first of all I have to derive this part. So, this multiplication of 2 expressions so, the this is expression 1 this is expression 2. So, I have to do it a multiplication of 2 expressions. So, that is what we are trying to do. So, T producing E producing T and this T producing T star F ok. Then this from this F I will have id which is 5. Now, for the left T from this left T I have to derive this part, in this 2 within bracket 2 plus 3.

So, from this T we write F and from F we write open bracket E closing bracket and then from this E we have got E plus T and from this E we get T to F to id; which is 2 in our case and then this T gives F gives id which is 3 in our case. So, this is called the parse tree right. So, this will be more clear as we proceed through the remaining part of this course; so, this will be more or less it will be this is what we are exactly going to do in our future lectures. So now, you see that this particular grammar is used for deriving expressions, that involves addition and multiplication.

Now, this is another grammar ok. So, apparently it seems that they are very different from each other, but this is also same as this one. So, we can we will see later that these 2 grammars are equivalent. So, it is not mandatory that for a particular language there can be only a single grammar. So, different people can come up with different grammars and

if you use this particular grammar then the way we should do the derivation is like this that so, we have got 2 plus 3 star 5.

(Refer Slide Time: 23:49)



So, these you have to start with T E and then E we have got only one rule T E dash. And, from this T I have to derive this within bracket 2 2 plus 3. So, for doing that so, we replace this T by F T dash and then this T dash can give rise to epsilon using this rule T dash giving epsilon and from this F we do it like within bracket E and from this E I have to get that 2 plus 3. So, from this T E is from this I again do T and E dash and from this T from this T I will be going to F T dash and this T dash will give epsilon, this F will give id which is 2.

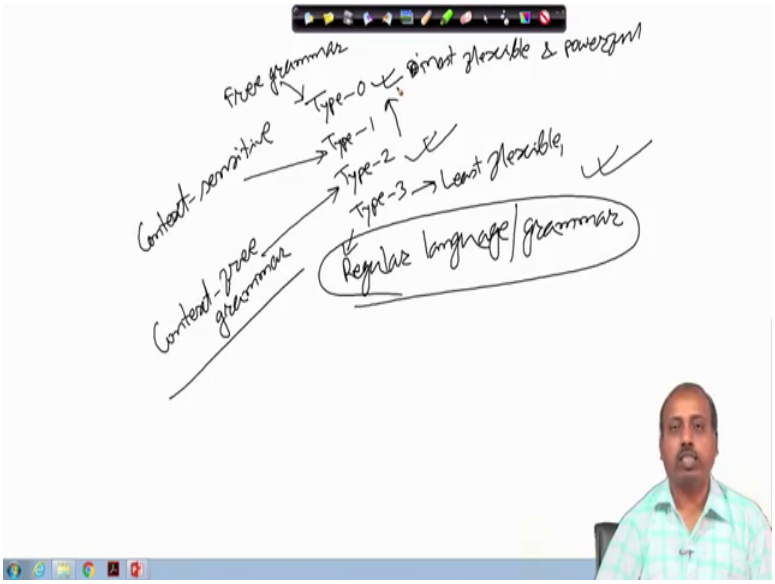
Now, from this E dash I will do plus T E dash to plus T E dash and the now this E dash part can be made to epsilon. And, then this T will be giving me F T dash and then this T dash will give me epsilon. Then this F can give id and which is equal to 3 in our case ok. Now, for this E dash part from this E dash I have to derive this star that star 5 ok. So, for that so, I have to so, for this sorry not from here actually from this T dash I have to do that, that star F T dash that star will come from here.

So, this is star F T dash and this; this T dash will give me epsilon and this F will give me id which is 5 and this E dash will give me epsilon. So, this way I can have another parse tree for the same expression, but using a different language. So, both the parse trees are correct only because, the grammar is different. So, there the trees are appearing to be

different and it appears to be very difficult to draw the parse tree. So, I was just trying to do it intuitively starting with the; starting with the grammar rules and trying to derive the final string.

So, it is very much possible that we get mislead and we go to some other derivation which does not lead to the final string. So, this our discussion on this parser generator will actually avoid all these things. So, it will guide us so, that we can select the proper production rules at different points ok. And, we can operate, we can derive the final parse tree which is which will be correct ok. So, next we will be looking into some error handling mechanisms, but before that so, there can be different types of grammars that I would like to introduce; one is that one type of depending upon their capacity.

(Refer Slide Time: 27:41)



So, there are different types. So, type-0 type-1 type-2 and type-3; out of them type-0 is the most flexible. So, this is the most flexible or most powerful, most flexible and powerful and type-3 is the least flexible least flexible and it is naturally power the power is also less. So, power in the sense that thus the type of languages for which it can construct a grammar ok. So, the if you try to construct a type-3 grammar then you will find that for many languages you cannot do that ok.

And type-1 and type 2 they are some intermediary types ok. So, this type-3 languages so, they are also known as regular language. And, the corresponding grammars are known as regular grammar and incidentally the regular expressions that we have seen in our lexical

analysis discussion so, they fall into this particular category, they fall into this class of regular grammar. Then this type-2 so, this is known as context free grammar; context free grammar so, most of the programming language constructs they will belong to this context free category ok.

So, most of the constructs that you have in normal programming languages, that we are familiar with; so they belong to the context free grammar. So, most of the time our discussion will be centered around context free grammars only. Then type-1 it is known as context sensitive grammar. So, they are known as context sensitive grammar because, we will take some examples later. So, where it will show that it can be more powerful than context free grammar and type-0 grammar so, they are known as free grammar ok.

So, accordingly so, if you try to design a recognizer then designing recognizer for type-3 is the simplest job that we have already done in terms of finite automata NFA and DFA. Type-2 is more complex where you will need a stack apart from the finite automata. So, that there it is known as pushdown automata so, that we will be able to do that. Then if the constructing the other acceptors for type-1 and type-0 they will be more and more difficult ok. So, most of the time we will be restricted to context 3 type type-2 grammars only, that is context free grammars because programming language constructs they belong to this particular category.