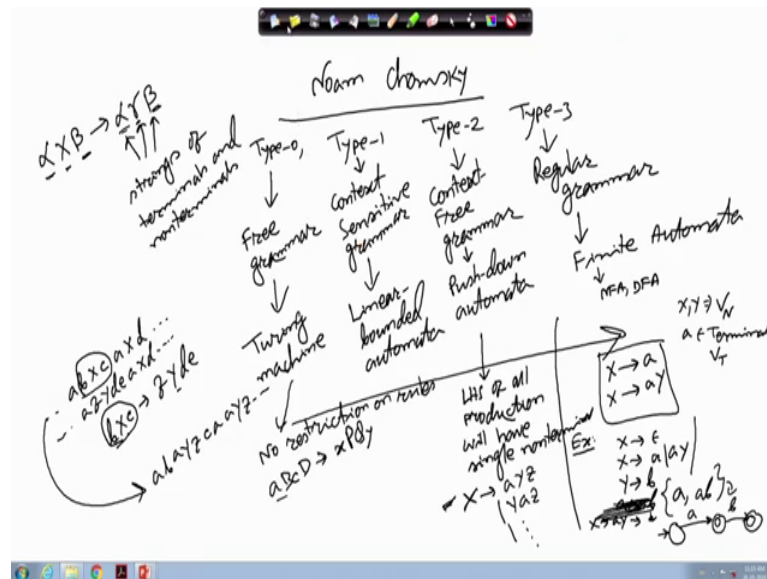


**Compiler Design**  
**Prof. Santanu Chattopadhyay**  
**Department of E and EC Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture – 17**  
**Parser (Contd.)**

There is a language hierarchy that is known as Chomsky's hierarchy.

(Refer Slide Time: 00:20)



So, it is defined by Noam Chomsky. So, according to Chomsky there are four types of languages they are known as type 0, type 1, type 2 and type 3. So, out of this type 0 is the most flexible class of languages because the corresponding grammar that we have is known as free grammar. So, this is called free grammar and the successive classes or successive types.

So, they are more and more restrictive in nature. So, type 3 is the most restrictive version which is known as the regular languages and the corresponding grammar is called regular grammar then we have got type 2. So, type 2 is known as context free grammar, context free languages and the corresponding grammar is called context free grammar and this type 1 is more or (Refer Time: 01:46) more less the constant compared to type 2. So, they are known as context sensitive grammar or context sensitive language.

So, from the name we can understand that this free grammar it can accommodate all other types of languages, but as we are going from type 0 towards type 3 the languages are becoming more and more restrictive in nature. So, for each of these class of languages so we can have different types of accepters available from the automata theory other or the computation theory and this for the regular grammar we have got the finite state automata. So, other the finite state machine or finite automata. So, this is the class of this particular tool can accept type 3 languages. So, we have examples like this NFA, DFA etcetera.

So, all the regular expressions that we have seen in our discussion during lexical analysis they belong to this type 3 language and we can have the corresponding regular expression. So, we can draw the responding NFA and DFA and also we can write down the grammar in some forms. So, I will welcome to that slightly later, then in the context free category. So, here the corresponding acceptor is known as push down automata. So, this is basically apart from the finite machine.

So, you will need a stack for designing and acceptor for the type 2 languages. So, they are known as push down automata. Similarly in context sensitive category we have got linear bounded automata, linear bounded automata which can be used for designing acceptor for this type 1 language and this type 0 or free grammar. So, here we have got Turing machine ok. So, as the acceptor machine that can detect whether a string belongs to the particular language or not so that can be done that can be constructed using Turing machine for free grammar.

Now, out of all this machines Turing machine is the most generic one and naturally we can so that is going to be most powerful; however, constructing Turing machine as you know the Turing machine is a hypothetical machine. So, we cannot construct it because practically because the tape size is going to be infinite. So, this is a theoretical machine only, but it is the most powerful computation platform that we can think about. And as we go from this towards these finite automata they become more and more restrictive in nature and the corresponding acceptor design becomes easier to design.

Now, let us look into the corresponding grammar like will start with the regular grammar and then slowly go to the other category. So, regular grammar we can have grammar rules of the form  $X \rightarrow a$  or  $X \rightarrow aY$  where  $X \neq Y$ . So, these are non

terminal symbols. So, these are they belong to the class  $V_N$  that is the class set of non terminals and  $a$  is a terminal symbol so  $a$  is a terminal symbol belonging to the set  $V_T$  ok. So, the all grammar rules will be of this form  $X$  producing  $a$  or  $X$  producing  $aY$ . If you can take an example like suppose I have got an example like this say  $X$  producing  $\epsilon$   $X$  producing  $a$  or  $aY$  and  $Y$  producing  $b$ . You see that here all the production rules on the left hand side we have got a single non terminal, on the right hand side we have got a single terminal symbol or a single terminal followed by a non terminal. So, either these two combination.

So, if a grammar is like this then will tell that the grammar is a regular grammar. So, we can quickly try to understand what is this grammar. So, you can I see that this grammar it will accept all those languages that can have at most one  $b$  and the  $b$  if the  $b$  is there. So, it will appear only at the end like I can have strings likes say  $a$ ,  $a$ ,  $a$ ,  $b$ . So, this can be derived starting like this  $x$  producing  $a$   $y$  then. So, sorry it is not  $a$ ,  $a$ ,  $a$ ,  $b$ . So, I can have the strings are  $X$  producing  $a$  or  $aY$  and  $Y$  may be replaced by  $b$ . So, I can have a single  $a$  or the string  $a$ ,  $b$ . So, only these two.

So, these two is the languages, these two are the belonging to the language now. So, this is an example of context free sorry this is it is an example of regular grammar and their type 3 grammar and the corresponding language is a type 3 language. Next will look into context free grammar. So, in a context free grammar; so all production left hand side it will have a single non terminal. So, here the idea is that left hand side of all production will have single non terminal, single non terminal symbol.

So, typical examples are like this, I can have say  $X$  producing  $aYZ$  like this where  $X$ ,  $Y$ ,  $Z$ . So, these are non terminals and  $a$  is a terminal. So, on the right hand side there is no restriction. So, another rule maybe say  $Y$   $a$   $Z$ . So, like that I can have a number of rules so, on the right hand side it do not have any restriction, but on the left hand side there is only a single non terminal symbol. So, if the grammar can be specified by means of production rules like this, then the grammar will be called a type 2 grammar or context free grammar and the corresponding language will be context free language.

And as I have told in the last class that most of the programming language construct they belong to this type 2 category. So, there most of the grammar, that will be using for this different different programming languages, so they follow this context free category. So,

that is why these has become very popular for the compiler design codes and everywhere will find that will be talking about this type of languages; then comes this context sensitive grammar. So, as the name suggests the context sensitive. So, every rule has got a context. So, context it is like this if I have a.

So, maybe I have a rule like  $\alpha X \beta$  producing  $\alpha \gamma \beta$  where this  $\gamma$  is a  $\alpha \beta \gamma$ . So, they are all strings of terminals and non terminals strings of terminals and non terminals. So, you see that here the thing is that this  $X$  has been replaced has been we can be replaced by  $\gamma$  only if it is preceded by  $\alpha$  and followed by  $\beta$ .

So, that is why it is there is a context. So, the context is that  $x$  is preceded by  $\alpha$  and followed by  $\beta$ , then only we can apply this rule to change the  $x$  to  $\gamma$  and  $\alpha \beta$  they remain unchanged. So, they just hold the context that is why it is called a context sensitive grammar. So, if you are having a string like say  $a b X c a$  then say  $X d$  like that and suppose I have a rule which says that  $b X c$  can be replaced by say  $f Y d E$  where  $X$  and  $Y$  they are non terminal and the  $b c d, a b c d E F$  so they are all terminal symbols. Now you see that this rule  $b X c$  matches with this part ok. So, I can replace this part by this right hand side.

So, I can from this I can derive the string  $a f Y d e$ , but this  $x$  cannot be modified because the context does not match. So, here the context is  $a d$ , but here it is  $b c$ . So, context does not match so I cannot do this replacement so, it remains as it is had it been a context free grammar. So, like this and it been a context free grammar I would have replaced both axes by these rules. So, I could have written like  $a b$  and then  $X$  replaced by  $Y Z$ ,  $a Y Z$  then  $c a$  then again this  $X$  replaced by  $a Y Z$ . So, I could have done like this, but in a context sensitive grammar since there is a context. So, that this left hand side has got a context. So, you cannot replace it freely, only when the context matches you can do the replacement.

So, it is even better in the sense that you can more precisely frame the grammar rules ok. So, that is why this is more powerful than type 2 languages and type 0 does not have any restriction. So, there is no condition regarding this context like while writing grammar for context sensitive language.

So, you cannot change from this alpha and beta over the rule; so, you cannot. So, this alpha and beta they remain unchanged from the left hand side to the right hand side of the rule, but if in case of type 0. So, that restriction also does not exist. So, there is no restriction on the grammar; no restriction on grammar.

So, you can have rules like  $a B c D$  producing  $x P Q y$ . So, you can have some rule like this. So, it is totally independent of this context. So, I am not maintaining the context of this left hand side or to the right hand side they totally modified.

So, we can have total freedom in the writing down the corresponding rules for the grammar. So, this is the type 0. So, type 0 is the most flexible one as a result none of the automata they can be used for accepting this type 0 languages. For type 0 you have to reward to Turing machine for designing the acceptor and as I have said that we will be mostly concentrating on this type 2. Because type 2 is the context free grammar and most of the programming language constructs they will be based on type 2.

So, we have already seen type 3 which is regular grammar and we have seen that a regular expression. So, you can always construct a finite state machine for that for example, for this language you can very easily make a DFA. So, if this is the start state on a it goes to these state this is the final state and on b also it goes to another final state

So, that is a. So, you can draw on DFA which will be doing this operation ok. So, this type 3 can be there for the regular grammar and that is for regular expression.

(Refer Slide Time: 14:32)


# Grammar

- A 4-tuple  $G = \langle V_N, V_T, P, S \rangle$  of a language  $L(G)$ 
  - $V_N$  is a set of nonterminal symbols used to write the grammar
  - $V_T$  is the set of terminals (set of words in the language  $L(G)$ )
  - $P$  is a set of production rules
  - $S$  is a special symbol in  $V_N$ , called the start symbol of the grammar
- Strings in language  $L(G)$  are those derived from  $S$  by applying the production rules from  $P$
- Examples:


$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid id \end{array}$$


$$\begin{aligned} E &\rightarrow E + T \rightarrow E + T * F \rightarrow E + T * (E) \rightarrow E + T * (T) \rightarrow E + T * (E + T * F) \\ E &\rightarrow E + T * (T * id) \rightarrow E + T * (F * id) \\ &\rightarrow E + T * (id * id) \\ &\rightarrow E + T * (id * id * id) \\ &\rightarrow E + T * (id * id * id * id) \\ &\rightarrow E + T * (id * id * id * id * id) \\ &\rightarrow E + T * (id * id * id * id * id * id) \end{aligned}$$

$2 + 3 * (4 * 5)$



FREE ONLINE EDUCATION  
**swayam**





So, with this we will be going back to our discussion on grammar. So, grammar as I have said that it say 4 tuple  $V_N, V_T, P$  and  $S$   $V_N$  is the set of non terminal symbols  $V_T$  is the set of terminal symbols  $P$  is a set of production rules. And we have seen that depending upon the structure of  $P$ , we can have different types of different class of grammars different types of grammars and  $S$  is a special symbol in the set of non terminal  $V_N$  which is called the start symbol of the grammar.

So, so, and the strings of the language they will be represented by  $L(G)$  then they are the strings that can be derived from  $S$  by applying the production rules or from  $P$  ok. So, any grammar for which we have got for which we have written the production rules, many times what will be happen is that we will not be explicitly mentioning the start symbol in that case the first, the very first non terminal symbol that appears in the grammar specification.

So, that is the start symbol for example, for this particular grammar  $E$  is the start symbol because  $E$  is the first non terminal among the set of rules that we have written. For the second grammar also  $E$  is the start symbol, now starting with  $E$  you can derive strings like this. So, you can apply the first rule  $E$  producing  $E$  plus  $T$ , then you can replace this  $T$  by  $T \text{ star } F$ , then you may decide that this  $F$  I will be replacing by within bracket  $E$ . So,  $E$  plus  $T$  into within bracket  $E$  and then this within bracket  $E$ .

That can be modified to say T by applying this rule E producing T from this you can have E plus T star T producing F ok. So, say do not if say T star F then from this thing you can derive like E producing E plus oh deriving is already there. So, E plus T into T star id from this T can be again be replaced by say F giving us E producing T star F star id, giving E producing T star id star id, giving E producing E producing part is not necessary. T this T T can be replaced by F F star oh sorry this was E plus. So, this E plus part is there F, F star F into id star id then that can give us E plus.

This F can be made to given id then this E can be made to give a T T plus id star id star id. So, this T can be replaced by F so you can get it like this then this F can be replaced by id. So, id plus id star id star id. So, I write these an identifier. So, maybe I have got an expression like 2 plus 3 into within bracket 4 into 5 ok.

So, this shows the derivation starting with the start symbol of the grammar. So, finally, we could derive this string. So, this string we could derive this string using the grammar rules. So, this string belongs to the language accepted by the grammar. So, this appears to be very cumbersome; however, the whole process is automatic. So, once we can write down a piece of code that can do this transformation by suitably selecting the rules from the grammar, then the whole process can be automated very easily.

So, this compiler, this parts are designing task or the two the knowledge that will gather during this parts are design a discussion. So, that will help us to design this type of automated tools.

(Refer Slide Time: 19:40)

## Error handling

- Common programming errors
  - Lexical errors
  - Syntactic errors
  - Semantic errors
  - Lexical errors
- Error handler goals
  - Report the presence of errors clearly and accurately
  - Recover from each error quickly enough to detect subsequent errors
  - Add minimal overhead to the processing of correct programs

So, coming to the error handling; now one also it is very common that the input specification that is given the input file that is given for parsing it has got some errors. So, particularly they are written by for programming languages. So, users are writing programs and programmers often do mistakes ok.

So, those mistakes can lead to different types of errors, first one is the lexical error. So, lexical error are those errors which are ah some which do not constitute any valid word of the language. So, for example, if I have got say if I have got say a one symbol that that is a 2 F. So, 2 F also this is not valid because the it does not fall into any of the regular definition for the language. So, that is a lexical error, then we have got syntactic error. So, syntactic error means there is some grammatical mistake.

Like say if then else statement it says that if some condition, if some condition then some statement else some other statement say S 1 and S 2. Now, if I have not put this then so, if I have written like if c S 1 else S 2. So, as far as the lexical analysis tool is concerned. So, it will identify this individual tokens if else then that tokens that will constitute this condition, tokens that will constitute the statement. So, it will find those tokens, but when it comes to the syntax analysis phase so, it will be taken as a it will be detected as a syntactic error because this does not make a correct statement of the language.

So, it does not follow any grammar rule. So, that is the syntactic error. So, another class of error at the semantic errors. So, it is the program is or the statements are syntactically



correct, but semantically there is some problem this is particularly ah visible for the programming languages where the variables I have got some types and the types are all predefined. So, for example, in the language c so if you have got an assignment like x equal to y plus z then this x y and z they must be predefined variables.

So, in your program you must have define somewhere for example, integer xyz that must have been declared somewhere. So, if this is not there then x y z will not be there in the symbol table and when parser we will try to parse this thing. So, lexical analyzer will written x as id, y as id, z as id. So, that way the, it is an expression id plus id then this is an assignment statement everything is correct syntactically it is absolutely correct. But I cannot derive the type of this variables and whether this plus operator is applicable on y and z that is also not known for example, if y and z are say character arrays then y plus z does not have any meaning.

So, these are semantic errors. So, the semantic errors are also serious and they are to be detected, but that that is not a grammatical error because grammatically it is correct, grammatically we have got this identifier plus identifier. Now meaning of those identifiers or the types of those identifiers we will tell us whether the construction is semantically correct or not. We have got semantic errors then they there are so of course, the lexical errors are there. Now how to do error handling?

So, error handling means somehow the compiler. So, it should detect those errors and not only that the detection. So, it should tell it a, the user that here is the error. So, it should pin point the error like what is the error and at which point in the program the error has occurred. So, error handler it will have the goals like it should be able to ah report the presence of errors clearly and accurately. So, that is the first thing that the some error is there.

So, what is the error and at which place. So, that should be told very clearly, it should be able to recover from each error quickly enough to detect subsequent errors. So, as I was telling in some classes earlier that error handling is important because if you are detect one error and if you just on detection of the first error if the compiler quits the compilation job and it will it ask the user to correct it and maybe user we will correct it and then give it for compilation, just to find that after two three lines again there is another error. So, typically compilers work in a fashion in which it produces all the error

messages together. So, that user can correct all those errors and then give it for compilation.

So, but for doing that it is very difficult because particularly for synthetic error since this compiler or the parser is working based on some automata it may go to a state from where it is very difficult to come to a clean state. So, so that is the, that is a very important jobs. So, somehow the compiler must be able to decide that I have to discard the next few tokens and come to a state which is clean and I can start looking into the new tokens from that point. For example, most of the programming languages so they have got the feature, that any statement that you have it ends with a semicolon.

Now, while doing the parsing operation so suppose there was an error at this point then somehow the compiler must keep through all these tokens and it should come to the next semicolon and from this point onwards it is expected that the effect of that error is gone ok. So, from this point onwards the compiler should start doing the parsing face and try to see whether it can detect more errors and not. So, if the recovery is not proper when the compiler we will produce arbitrary error messages that will mislead the user.

So, this is very important that the user, that the compiler can come up to a decent state from which it can produce proper error messages for the subsequent errors as well as. So, it should add minimal over it to the processing of correct programs. So, if the program this error handling part. So, it should not be such that it creates the code generate makes the code generation process for correct programs to take long. So, it should not happen like that. So, the compilation should be first at the same time it should be able to decide on those ah on those errors and it should be able to recover from there. So, this is a very important, this is a very important thing to do.

(Refer Slide Time: 26:54)

**Error-recovery strategies**

- Panic mode recovery
  - Discard input symbol one at a time until one of designated set of synchronization tokens is found
- Phrase level recovery
  - Replacing a prefix of remaining input by some string that allows the parser to continue
- Error productions
  - Augment the grammar with productions that generate the erroneous constructs
- Global correction
  - Choosing minimal sequence of changes to obtain a globally least-cost correction

Handwritten notes on the slide include:  $S \rightarrow \text{if } C \text{ then } S_1 \text{ else } S_2$  and  $\text{if } C \text{ then } S_1 \text{ else } S_2$  with arrows indicating the flow of the code.

So, we will see how this can be done. There are several error recovery strategies followed by these compiler designers one is known as panic mode recovery. So, panic mode recovery is like this suppose we are at same place and there is a crowd and in the crowd there is a there is some sort of news comes that something some something has happened maybe there is a fire or something like that.

When the everybody wants to come out of the building for example. So, is if people who if the people are inside a building all of them try to come out and we say that is a panic mode may be, it is not mandatory that we should come out so early, but this we most of the people will all the people will do like that. So, this is known as panic mode recovery. So, what the compiler does is that it will discard input symbols one at a time until one of the designated set of synchronization tokens is found.

So, what is the synchronization token? So, as I have said that depending upon the programming language you can define when a particular block ends for example, for the C language. So, you know that say every statement ends with a semicolon. Now if discarding up to semicolon is also not sufficient you find that still the compiler the parser could not come to descend state another synchronization token is the closing braze. Like (Refer Time: 28:18) you normally you whatever we have. So, every block of statements so that is enclosed within a braze, pair of braze.

So, if there is some error in this statement and by discarding this statement we could not come out of the error then what the compiler will do is that it will skip the success is statements also till it comes to the brace. So, that it has fifth and entire block. So, that is one possibility. So, some programming language so they will allow you to have individual procedures and the begin end block markers begin procedure, end procedure markers. So, in that case the compiler can skip up to that point. So, depending upon the programming language that we fall which you are designing the compiler. So, we have got different time synchronization tokens by analyzing the program you can understand what are the, we can what are the synchronization token. So, this is known as panic mode recovery..

Then there is a phrase level recovery. So, phrase level recovery it will try to replace a prefix of remaining input by some string that allows the parser to continue. So, some part of it may be there is some there is something like this for example, suppose my grammars, my language says that there is an if then else statement if condition then statement else statement. Now it may so happen that this then token is then this it is then part is missing.

So, what will happen the parser or it will see this if part condition part then it will see the statement part. So, what the parser can do so, it can be it can intelligently determine that here what is missing is a then token. So, it can insert this then token into the input string. So, that way it will replace a prefix of remaining inputs by some string. So, this is the remaining string from this point onwards. So, this is the remaining string. So, it will replaces this part by this it introduces this then at this point. So, the program did not have, the program that you are considering has got something like this if C S 1 else S 2. So, this part is not yet seen so it has got S 1.

So, it will understand that this I have to put a prefix then at this point then the whole thing will become meaningful. So, this is the phrase level recovery. So, it tries to introduce some phrases at the beginning of this of this input part and then it can continue. So, this way we can have different error recovery policies ok. So, there are error production and global correction that we will see in the next class.