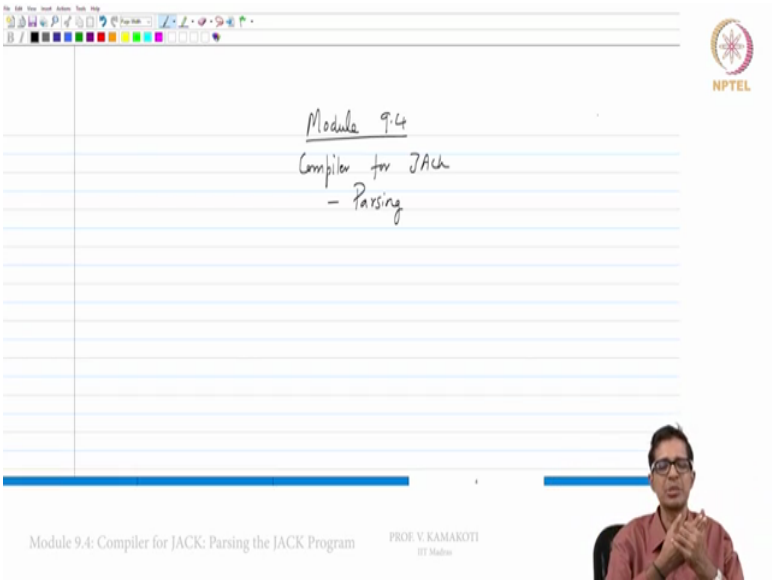
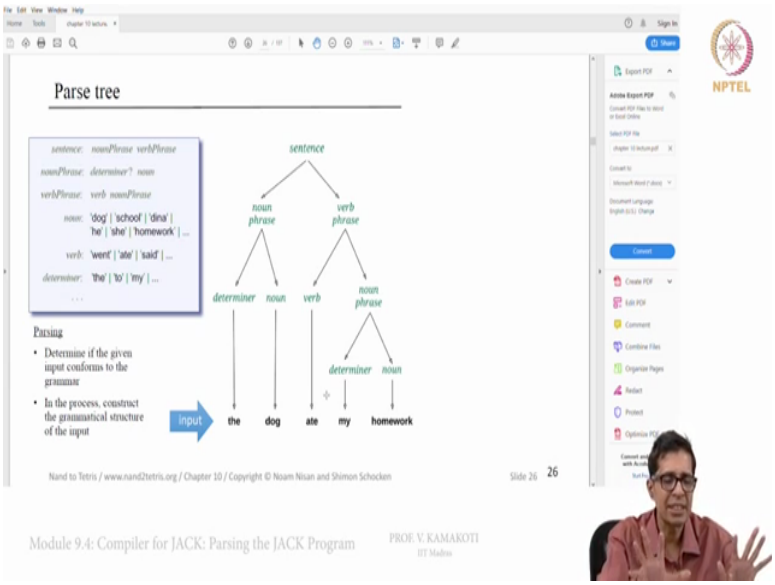
Foundations to Computer Systems Design
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Module 9.4
Compiler of Jack Parsing the Jack Program

(Refer Slide Time: 00:16)



So welcome to module 9 point 4 and in this module we will be talking about the Parsing which is given a grammar and given an input whether that input actually adheres to that grammar and that is what we will be doing as a part of this module.

(Refer Slide Time: 00:35)



So what we did in the previous module, we actually basically contact constructed a tree, a tree is nothing but a hierarchy so you know family tree like that so like this is inherently, right for example the dog ate my homework for example let us take an interesting example, now is this grammatically correct that means using these rules of the grammar can I basically arrive at this sentence, right.

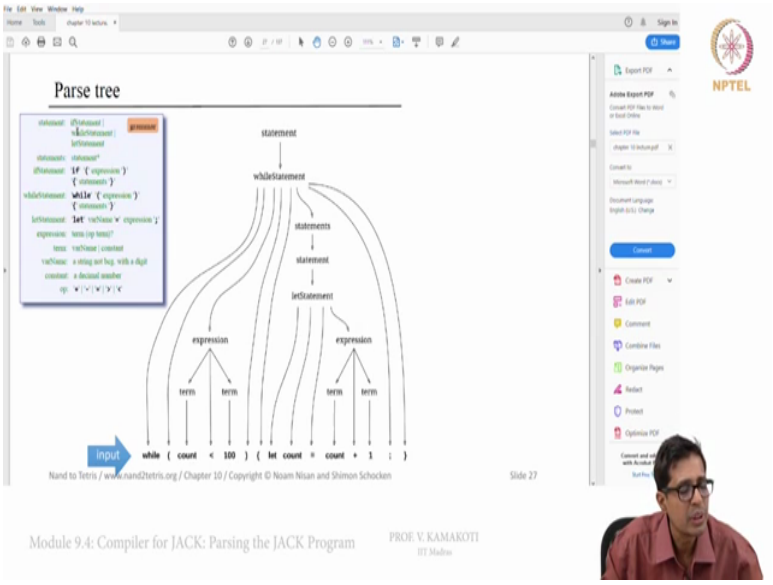
So is this sentence adhering to my grammar how will I prove that yes we start with a sentence here right, the sentence has a noun phrase and a verb phrase noun phrase has a determiner and they know the verb phrase has a verb and a noun phrase, right? So the determiner is the, the noun is dog, the verb is ate the noun phrase again as a determiner and a noun and determiner is my and homework noun as homework.

So from the rules of the grammar I am able to derive that the dog ate my homework is indeed a sentence because the dog ate my homework has a noun phrase followed by a verb phrase the known phrase is the dog why is the noun trace the dog? Because noun phrase can have an optional determiner and a noun and that option determinant is the and dog and hence the dog is a noun phrase when similarly ate my homework is a verb phrase because the verb phrase can have a verb followed by a noun phrase and the verb is in my case associate and the next known phrase can have an optional determiner and known and the determiner in this case is my and homework.

So this my homework is a noun phrase ate my homework is a verb followed by a noun phrase which is a verb phrase and so the dog noun phrase ate my homework followed by ate my homework verb phrase is indeed a sentence and hence this particular sentence adheres to the grammar it agrees with the grammar, so this is grammatically correct as per the rules that I have given for this.

So what I have constructed here is I have parsed this parsed means have looked at every part of this language, I have parsed this entire sentence and I am proving that the sentence adheres to my grammar and so I have essentially constructed what we call as a parse tree, so this is a pass tree.

(Refer Slide Time: 03:09)



Now the same thing that we need to conduct for a construct for everything for example while count is less than 100, so this is the input that I am given while count is less than 100 let count equal to count plus one semicolon, right. Now is this is one statement is this a correct statement yes yeah a statement can have a while statement if statement or while statement or less statement, so I so a statement can be a while statement as per this rule, first rule.

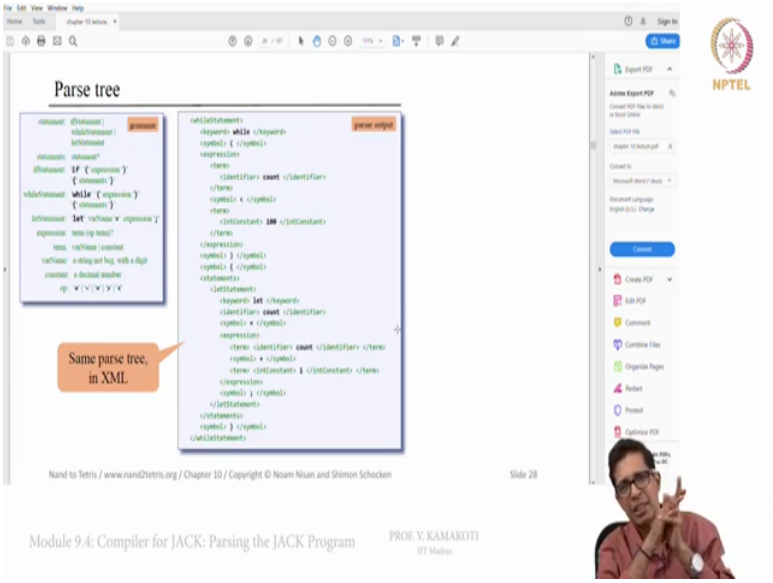
Now that while statement basically should start with a while yes then followed by a parenthesis yes then followed by an expression that expression can have a term and another op and term which is optional I can use it, so term op term the op, so the term can be a variable name so that is count the op can be any of the symbol, so this is less and the term can be again a constant as per this rule again the term can be a constant and that constant can be a decimal number 100, so this is an expression count less than 100 is an expression so there is an expression after that in the whale statement after this expression.

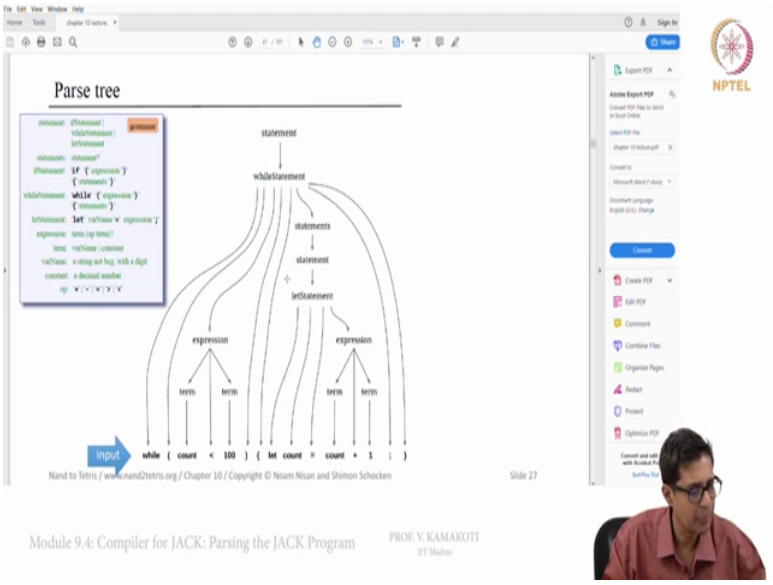
Now you go it to we are going to have a closed parenthesis yes after the closed parenthesis you need to open braces yes that is there then there should be statements at the end of the statements you need to have a closed parenthesis, now let count equal to count plus one semicolon should be a statements, yes. A statements can have one or more statement and so in this case let us say I will have one statement here and that statement can be if statement, while statement or let statement.

So I decide on it is let statement, so statements has one statement see here I am moving the moves here statement statements as one statement, statements as one statement and that statement has let statement, the less statement again should have a let followed by a variable name that is count then equal to then another expression, expression essentially is can have a term op and term and that term is a variable name is count another op can be plus, the next term can be a constant which is one and it should be followed by the left statement should be followed by a semicolon here, so that is the semicolon, ok and then end parenthesis.

So while parentheses count less than 100 close parenthesis, open braces that count equal to count plus 1 semicolon a closed braces adheres to this grammar, so this is correct right. So what we are now doing is what you call as syntax analysis where is there an error in the input in terms of adhering to a grammar that is what we are trying to do and this is what we have achieved here.

(Refer Slide Time: 06:39)





So what we will be doing as a part of this parsing is that this is what we need, so the moment I get this while count less than 100 let count equal to count plus one so this entire statement should whatever we have done here this parse tree, right so should come up here so I will say I start off with while statement then I need to have a keyword while slash keyword then I need to have symbol slash symbol then I need to have an expression.

So let us just take the first level view while statement should have a keyword followed by a symbol, followed by an expression it ends here then again followed by a symbol which is this thing followed by another symbol which is opening of this parenthesis followed by statements, the statements end here then followed by a symbol which is this close parenthesis this and this is the end of while statement.

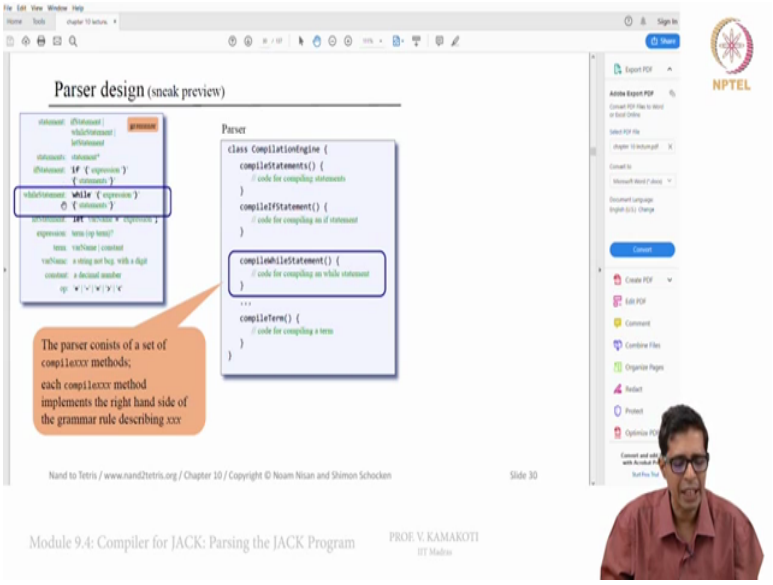
So while parentheses close parentheses, open braces, close braces is there, now intermediately between this two parenthesis I need expression and then between these two braces I need statements, so that we need that we saw here so the expression essentially had a term followed by a symbol followed by a term right, term op term count less than 100 and similarly the statements had a statement which is a let statement for let count etcetera and that let statement basically had a let yeah it is a keyword then an identifier which is a count then a symbol which is equal to then another expression, the expression and term had a term and symbol and a term and so on so forth, right.

So whatever we have generated as a parse tree here we generate a dispatcher XML file this is called extended marking markup language very simple here, just we need to write this down. So from the tokens that we have generated as part of your first part of your module project

then we need to now go ahead and generate this XML file or parser output and that will make the whole thing.

So what we need to now do is from the token how do we generate this XML file that code we have to write, so what happened in the tokenizer we gave safe main dot Jack it gave us main dot tok token, that main dot tok should be taken and it should give us a main dot xml and that is all about this project ten.

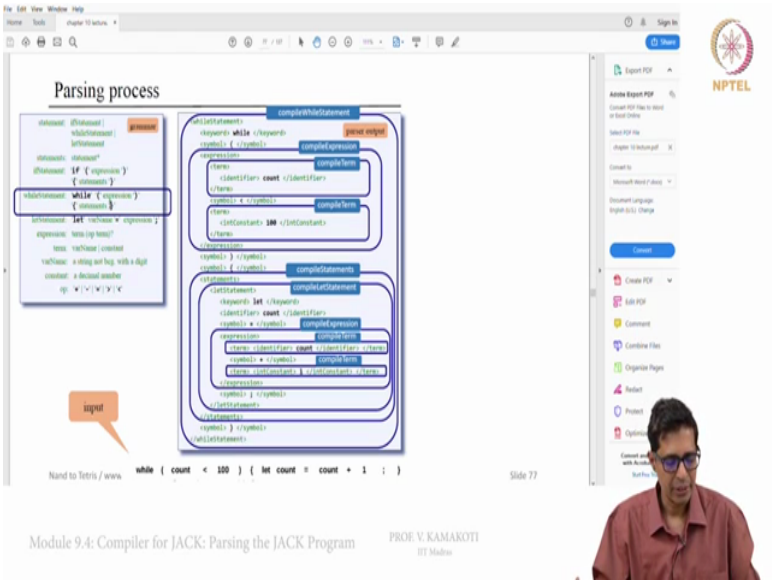
(Refer Slide Time: 09:48)



So how do we go about doing this process, so this is what we have to write so we can write it in I am this is this entire thing is will give you a very sneak purview of this but then so for each of the so we started statement right, so statement basic statements has one or more statement, statement is to start so statements is a start so we have one or more statement and so for each statement so I will have something called compile statements and that compelled statements will essentially have compile if statement, compile while statement, compiles term etcetera for all these things I need a compilation right.

So I have different routines one handling statements, one handling if statement, one handling well statement like that one handling term etcetera.

(Refer Slide Time: 10:43)



So the parsing process is like this you follow the right hand side of the rule and parse the input accordingly if the right-hand side specifies a non-terminal rule x axis called compile x axis and do this and keep doing this recursively, so let us see how this example for example I have while count less than 100 let count equal to count plus 1, so what would be the token for this so I will have a keyword while to start with, so what do you do?

So let us assume that we read while so we know that this is a keyword so immediately we have to now go and call compile while statement right, so immediately I have to go and call compile while statement because while is a keyword and so we go and call compile while statement and what will compile while statement do this is what it will do, so compile well statement will first is say hey I am in the XML file it will say while statement, it will print while statement.

So this is output so while statement then immediately it will go and you have already seen a keyword while so you go and print this keyword while then it has to see it will go and check if the next element is a symbol and that symbol is a open parenthesis right, it will check and if it is true then it will put it here if it is not true then it has to say parenthesis missing when you do a syntax when you do C when you compile the C program you say parenthesis is missing, right.

So who tells particular rule tells this then the next step is that it goes and sees then next it has to see an expression, so immediately it will call compile expression right, so what will

compile so immediately this will put expression here and what will compile expression do compile expression essentially has to look for it term so immediately compile expression will call compile term because immediately does look for a term and that term will term I do not need anything term I should check whether it is going to be an identifier or a integer constant right, term can be an identifier or a integer constant.

So in this case it is an identifier so I just put identifier count identifier and that so how tokenization would have happened, tokenization would have set keyword while keyword, symbol for this parenthesis and count will be as a identifier, so it will see an identifier and it says count right then it comes it (())(13:41) less then so this term should now go and see if there is an op, if there is an op no it does not so this one term is over term var name is over then I go back to compile expression.

Now we will see there is a symbol it goes back that and now it goes and sees yes there is a symbol, so it puts slash symbol there then now it sees the moment it sees a symbol now again it will as term so again it will call compile term and the term in this case we have to check whether it is a constant or this, yes there is an integer constant and then slash term int, ok. So what has happened this I am moving from one token to another token the first token I saw was a keyword while so immediately called compile while statement the moment I call compile well statement that compile while statement will start ensuring that your while is adhering to the syntax of what the while should be, right.

So first there should be a keyword while it is did it then it says that there should be a parenthesis open that also there then there should be an expression, now so the compile while statement will now call compile expression right and now the compile expression immediately will talk call compiled term it will because there is one term surely there should be a term.

Now the term will look now go and compile term will go and check whether it is a variable or a constant so immediately it knows that it is an identifier so it is a variable name, so identifier count it will put their slash identifier and it will go back right. So term is finished now again you will go back to compile expression and then the compile expression will see what next, if it sees a symbol right then it outputs the symbol and after that symbol it should see another term, so again it will call compile term.

Now the compile term will check whether the next is it see a variable or a or a key variable or a constant in this case it sees a constant it finishes and returns after that the compiled expression will see is there anything more nothing so it can be only two so then it returns back, so it returns back to who compile while statement. Now the compile while statement is at this point right now the token now to see it goes back to the compile while statement so the expression is for now the while statement should now see a symbol which is close parenthesis if it does not see that symbol close parenthesis in the token next to token the next token is not a symbol then again it should give an error right but in this case yes he sees a symbol.

Now after that symbol it should now see another symbol which is the opening of the braces opening of opening brace now after that it should see statements right, now immediately it will call compile statements the compelled statements now the immediately the moment I see a compile statements then I will go and check whether compile statement will have zero or many statements, so I will go and check if I have a let or while or if, so I see yes there is a let statement right.

So the moment I go to that statement so I will have compiled let stay, so I see a lit then I make compile let statement, so the moment I see compile let statement now I have variable name let so it should first see what should the let statements sees he should see a keyword let then it should see the next token should be here identifier which is var name it should see an identifier right because var name is an identifier yes so it sees an identifier count then it should see an equal to it is a symbol yes it sees that then it should see an expression, right.

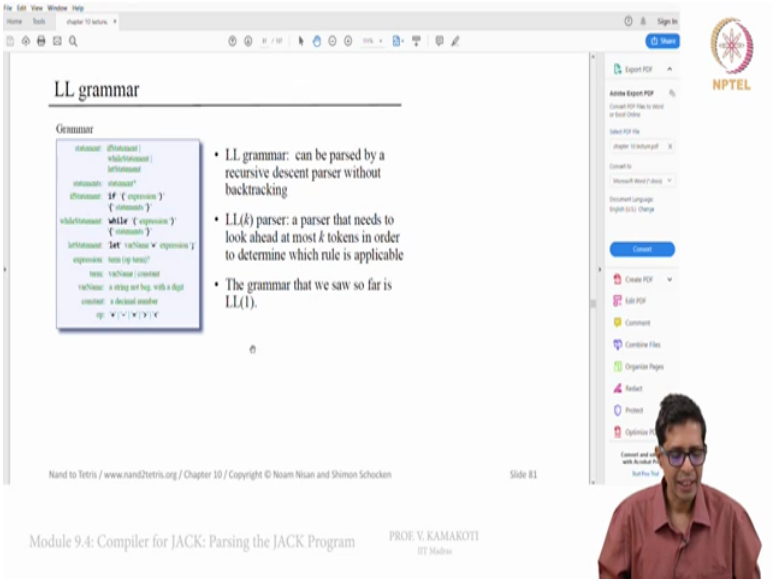
So immediately it will again call compile expression, the compile expression immediately it will call compile term and the term will see (call) inside the term again you will see the identifier compile term will return back again you will see term slash term then you will see a symbol, slash symbol then you see another term which is (inst) constant and then you come back.

So this is how the compiler expression will complete it then you go back after that for the let so you go back to the let us compile let statement right you are compiling this and then you should see a symbol there then this let statement is over then we go back the next statement is over now we go back to compile statements and you do not see you actually do not see any more statements there, so you finish it.

So this is how the while statement now the (ans) once the statements finish then you come back to the while statement again you need to see the last symbol which is closing parentheses right. So this is how the parsing process basically, so you code these routines so I will have something called compile while statement, compile expression, compile term compile statements, compile lets statement etcetera and each of these routines will check if the tokens that are following adheres to certain rules of this grammar, right.

So you have to just the code these four or five routines and then pass the token one by one, line by line so every line had a token right so we have to parse the token one by one, line by line to this thing and every fellow will check whether things are working the whole thing is correct and then it will generate this entire output right. So this this would be the parser output that will be generated right, so this is the entire parsing process this is a very simple thing that we can write, ok right.

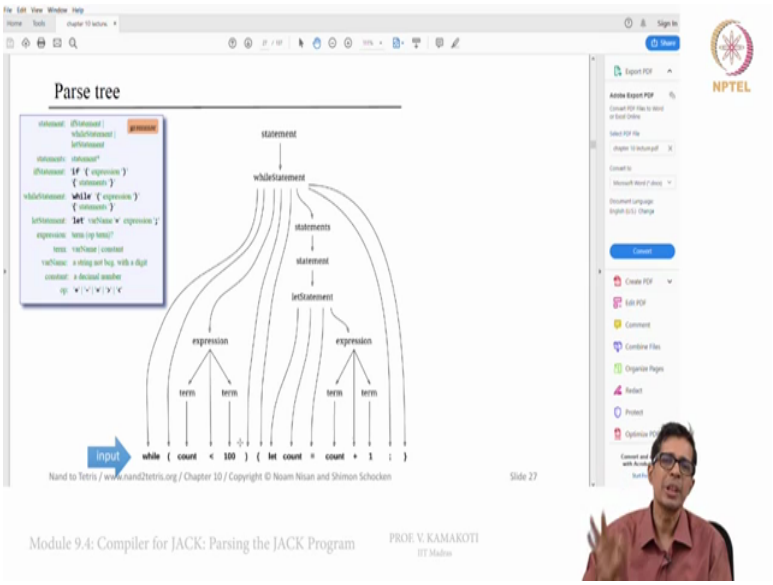
(Refer Slide Time: 20:30)



So a small introduction to finite state or so you will be learning a course called theory of computing at the parse or part of your thing right or in your it is so basically you will be discussing about automata theory etcetera, as a part of your theory of computing course you will be learning this grammar whatever I have taught so far is a one very minor subset of a reasonably large subset of what you will be learning in a theory of computing course, theory of computing basically talks about languages and the grammar computing languages and their associated grammars and that is what we have covered now.

Now what we saw now is called a l l grammar ok which can be passed by a recursive descent parser without backtracking so these are all big words I will basically explain you I will basically explain you what is what is this grammar and everything see what is backtracking and those things right.

(Refer Slide Time: 21:56)



So in this case let us go back to what we saw earlier here when I was let us go back here when we start parsing this we started initial right we started with the let us go back quickly in the entire process of this parsing right let us go there, the entire part of this parsing so we started so while this input came we started with while statement, from the while statement we arrived at while then we have arrived at the symbol then we called the expression and we finished off count less than 100 and then went.

In this entire process we did not revisit our decision right, so by looking at while I we are very clear that this is a while statement this cannot be any other statement it is a while statement, so if at all I want to process this entire statement it should adhere only to the wild rule there is no other rule that will fit into it right, so the first thing the moment I saw while I said I will start using this while statement rules to prove that this statement essentially belonged to this grammar right and I had no doubt about it just looking at while we were able to make the choice.

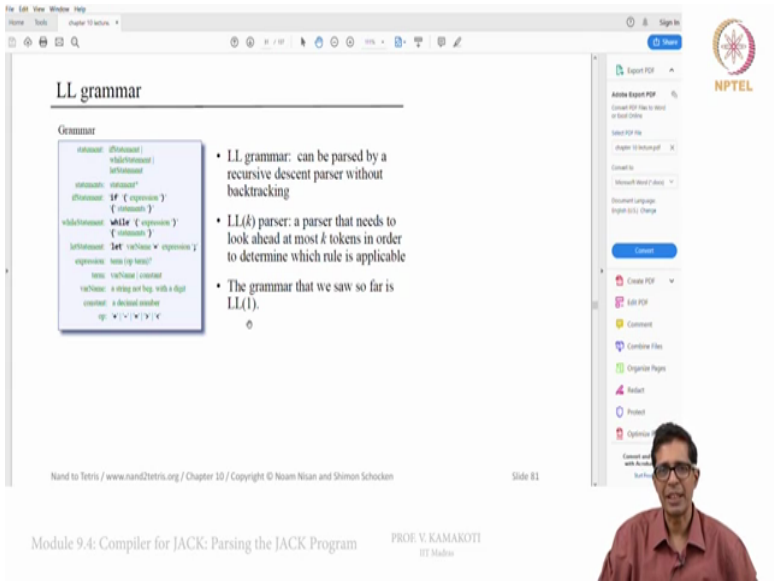
Then after making that while of course there is nothing about the symbol then I had to do there is an expression right, so I come to the expression and here I have a term op term and from the term why can I came to the term I had no doubt whether to use var name or constant

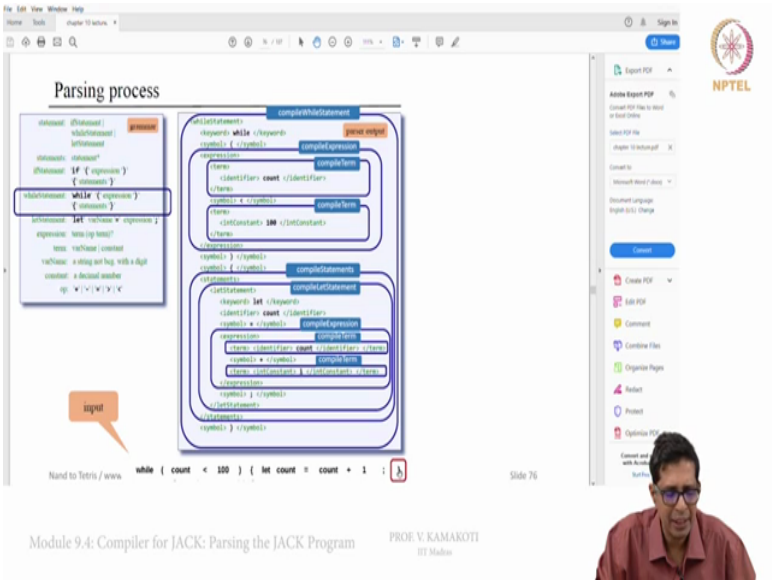
because the moment I looked at count I know that I have to use var name it is not that I used var name and then something went wrong then I went and started using constant.

So there are at least in this whole scene there are at least two cases in the case of statement and term where I have a choice whether when I land up with a statement should I go for if for while or let, if I land up with a term should I go and use a van name or constant right, I have a choice here and similarly when I look at op should I use plus minus equal to less than greater than at any of these instances I never need to I had no doubt of which rule should I use for example when I came to this term I knew that I have to use var name and then count I did not use term constant and then found count and then just by looking at what is the next token I can decide on what rule I should apply.

Similarly when I came to 100 right I saw expression is a term at this time I the moment I saw 100 which is my next token I had no doubt of using this constant to 100 rather than this var name, so when I was looking at count I used to var name while I was looking at 100 I started looking at the constant here right. So at any point in this grammar I never have a reason to actually go back I can just look at the next token and decide which rule should I apply right.

(Refer Slide Time: 26:09)





So that is why so I need not so that is what we mean by this big statements that we have here, we will go to the last bit yeah so this is a grammar can be parsed by a recursive descent parser without backtracking, backtracking means oh I made a wrong choice and go back right and this is called a recursive process because these rules are recursive right, for example statement calls while statement so statements called while statement true statement while statement again calls you statement so the grammar is a recursive grammar right I can call myself in some way in this things.

So these are recursive grammar right so that also we need to, so this parser is a llk parser what means a parser that needs to look at, at most K tokens in order to determine which rule is applicable in this case this is a ll 1 parser I have to look at one token I had to find out which rule I have to apply, so I look at this while and I know I have to do a while, the next is thing I need a symbol I need I have to look at the current token as of now I have to look at the current token to basically find out which I have to look at the current token to actually find out what is the rule I need to apply, actually as of now as we had seen so far I need not actually look at the next token I have to just look at the current token to make the decision, right.

Looking at this particular symbol I know that this is a symbol just one token I have the current token I have to look, so this is basically what we understand by parsing and this is the output that we need to derive, ok. So in the next module we will now show you a little more larger implementation of this parsing and we will show you how it becomes a ll one why we need to look at one more token extras currently we are looking at the current token we have

to look at one more token extra, so why it becomes that we will give you some examples as in the module 9.5, thank you.