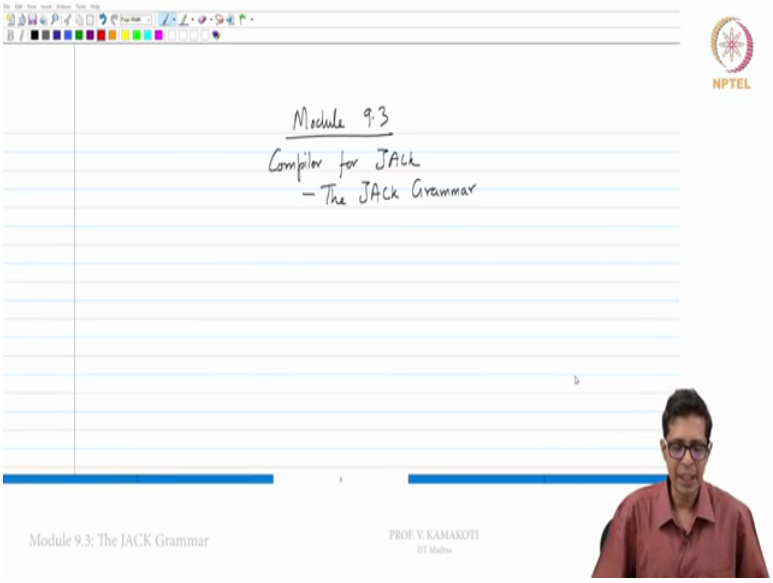
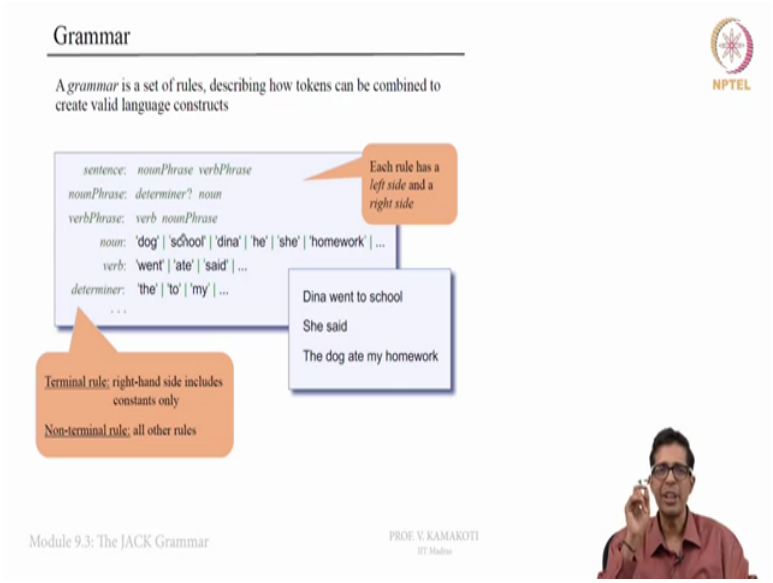
Foundations to Computer Systems Design
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology Madras
Module 9.3
The JACK Grammar

(Refer Slide Time: 0:16)



Welcome to module 9.3 and we will now start with the next part of the front end of the compiler what we call as the parsing. Now before understanding what is parsing we need to understand what is the grammar of the language so like English as a language has grammar the JACK also has a grammar. So how do we capture the grammar of the programming language? We will have a very short introduction but we will try to make it very comprehensive because you need to do a project based on the understanding of this.

(Refer Slide Time: 0:58)



Okay, so now we are going to talk about the parser that you see here and this is the grammar. So let us now start with understanding grammar of a English language for example. Now how does how is what is so as a language we have a collection of sentences, so how is the sentence organized in English? A sentence basically has a noun phrase and a verb phrase, so that is a rule and we need to follow that rule so if you want to talk sensibly in English we need to follow a rule where every sentence that we talk has a noun phrase and a verb phrase.

So sentence is comprises noun phrase and followed by a verb phrase. So you now have something on the left side which is a sentence and that sentence essentially produces two parts namely a noun phrase and verb phrase, noun phrase actually has something called a determiner can have a determiner and a noun, so determiner question means it can you will have a determiner or you need not have a determiner then you can have a noun. So and the verb phrase will have a verb for sure and followed by a noun phrase.

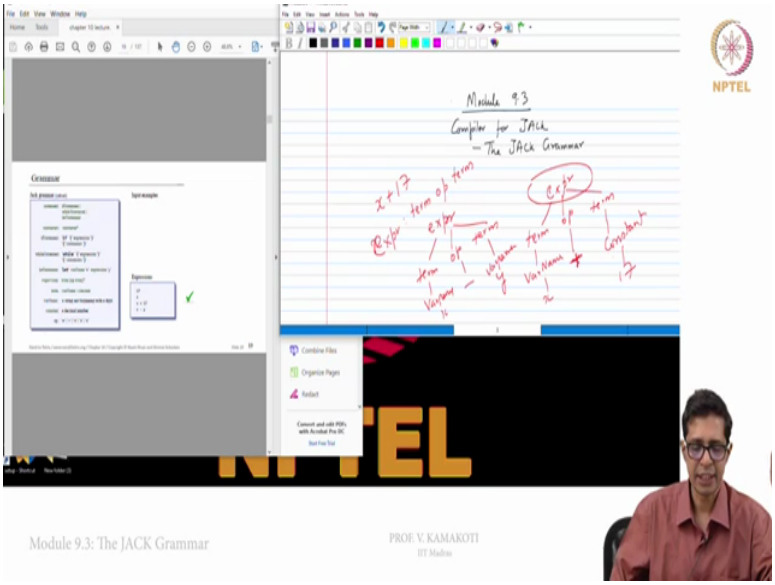
So for example so what are so now the sentence noun phrase and verb phrase comprises noun phrase, verb phrase determiner noun, verb, noun phrase. Now the noun can be like dog, school, dina, he, she, homework. A verb can be went, ate, said, a determiner can be the, to, my. So now sentence gave rise to two entities namely noun phrase and verb phrase, verb phrase again gave entity to two again gave produced two more things like verb and noun phrase, verb ended with went, ate, said, etc. So your, went, ate, said whatever you see here or dog, school does not produce anything more so these are all called terminals and the other things are called non-terminals.

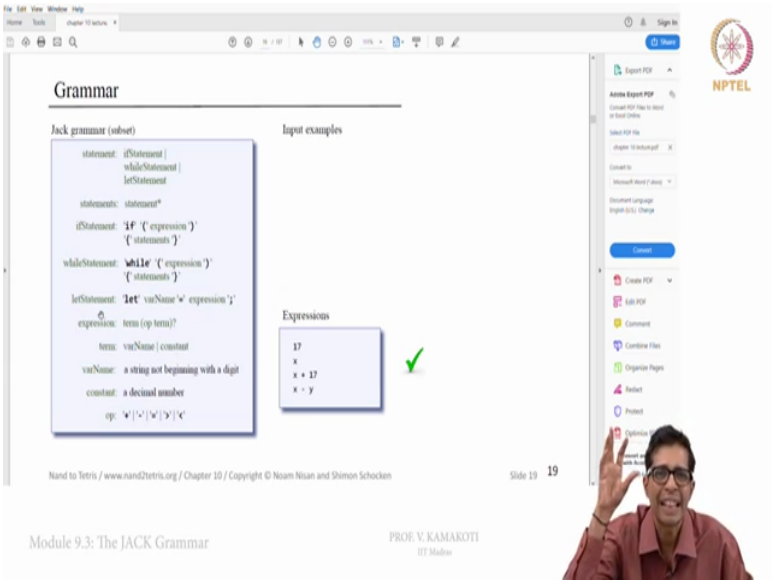
So now you are seeing on the top rules that you see one for sentence, one for noun phrase, one for verb phrase, noun, verb and determiner. These six rules one for each of this can be classified as either a terminal rule or a non-terminal rule. A terminal rule is one in which whatever you see on the right hand side is a terminal, it does not yield to something else or it does not produce something else.

So the terminal rules are noun, verb and determiner because noun leads to dog, or school, or dina, the pipe stands for pipe symbol that you see here basically where I am moving the mouse is basically stands for or so it stops there. So the noun, verb and determiner are basically terminal rules while sentence, noun phrase and verb phrase are non-terminal rules. So a grammar comprises can be represented as a collection of rules and each of this rule will have a right hand side and left hand side and the right hand side, left hand side has one non terminal namely sentence noun phrase, verb phrase like that, right hand side can have either terminals or non-terminal and can be a collection of these terminals.

And the right hand side can be the right hand side can also have terminal or non-terminal that are optional for example what you see here noun phrase has a determiner which is optional. So I can say these school are just say school, the becomes and the determiner and the determiner may or may not exist for a noun phrase. So this is how a grammar is basically represented for computational purposes because we have to now do some computation on this grammar it is not just understanding of this grammar by a human understanding there is a computer understanding of this grammar that is necessary, so we need to put it in more formal term and this is the formal way of representing a grammar.

(Refer Slide Time: 5:58)





Now the grammar for JACK basically has this 4 plus 4, 8 plus 2, 10 rules, right 10 rules and what are those rules? So as we see a program is a collection of statements is a collection of statements. Now every statement so let us start with simple things I could have expressions, your grammar can basically have expressions. For example, 17 is an expression, x is an expression, x plus 17 is an expression, x minus y is an expression.

So now with this as a context let us go and see, so an expression is a term followed by another term if you just look at this particular part an expression is a term followed by an optional op term. So for example and a term can be a var name or a constant and a constant is a decimal number. Now the question asked is 17 an expression? Yes, 17 is an expression

because expression leads to a term and this is optional, so expression just leads to a term, I will not use this, term leads to a constant, constant leads to a decimal number which is 17.

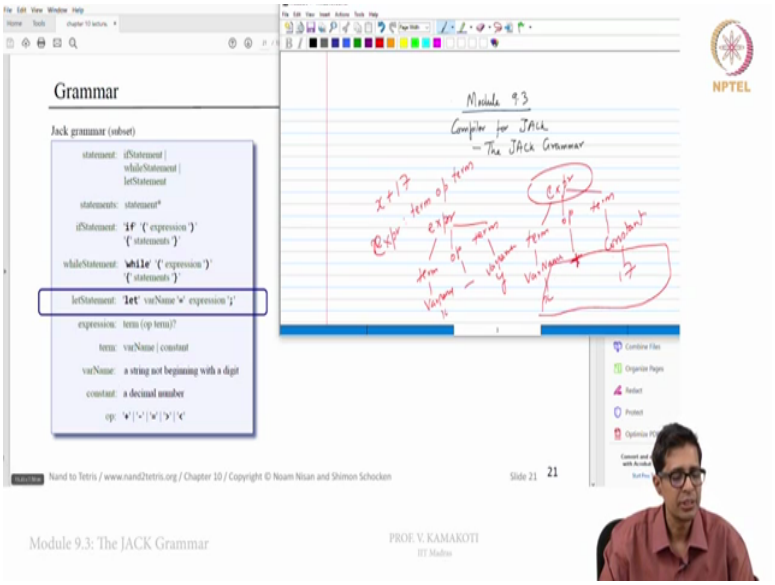
So from expression I can go to expression produces a term, term produces a constant, constant produces a decimal number. So 17 is an expression, 17 is a constant which can be a term, which can be an expression, x is it an expression? Again expression leads to term, term leads to a variable name and a variable name is a string not beginning with the digit and x is one such. So expression leads to term, term leads to variable name, variable name leads to x, so x is a constant, x is a expression.

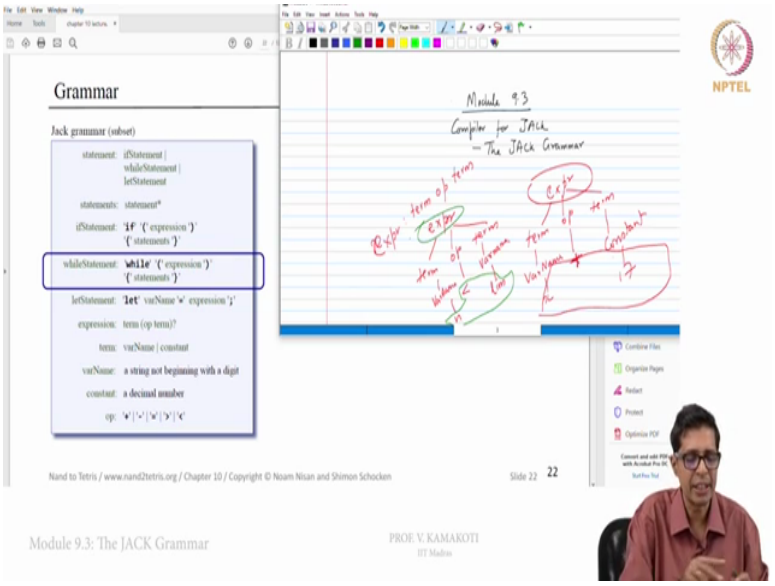
Is x plus 17 an expression? Yes, again x is an expression, so we can just do this as follows just look at this, if I go and look at x plus 17, yes x is an so an expression is basically a term with an operand followed by a term, if you see here this operand followed by a term is optional, but I can put like this. Now so let us expression can be term followed by an operand followed by a term, a term can be a variable name, variable name can be x, op can be plus, operand can be plus you see the last one here and then your term can be again a variable name and this can be say (17) sorry in this case it can be constant and 17 so x plus 17 is an expression.

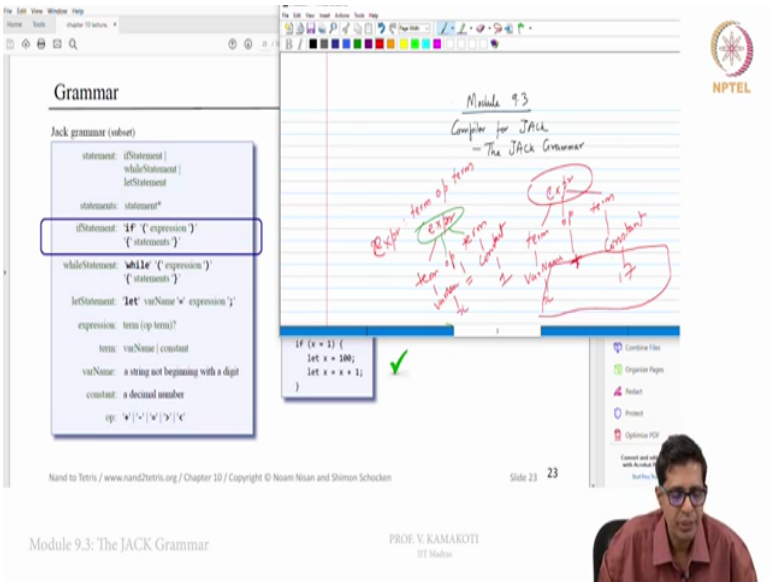
Similarly, x minus y can be an expression, expression can be a term, operand and a term, a term can be a variable name which is x your operand can be minus this is a variable name and this is y. So essentially from the word expression as you see here from the word expression as you see here I am able to use these rules to finally arrive at x minus y. So that means this x minus y is an expression, right.

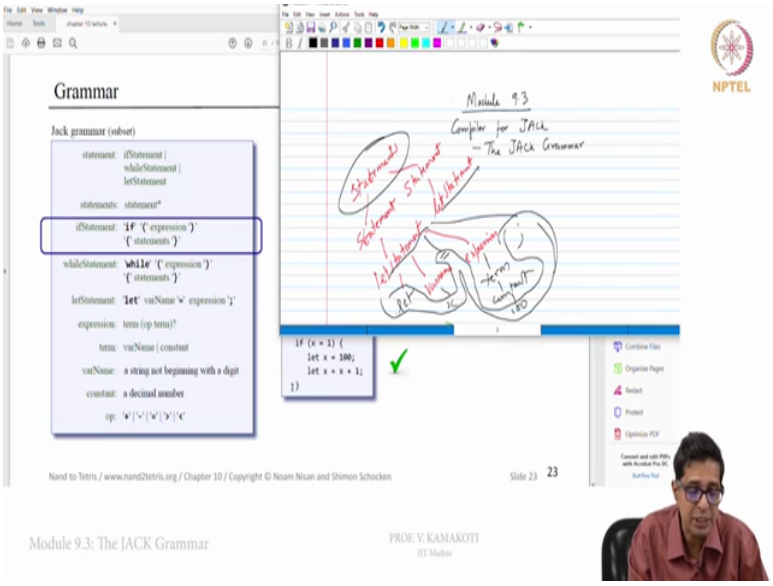
So this is how the grammar is basically captured, so when I see 17, is it an expression? Yes, why? It follows this grammatical principles, I apply rules to prove that 17 is an expression, I apply rule to prove that x is an expression, I applied rules to prove that x plus 17 is an expression, x minus y is an expression. So this is how a grammar is captured.

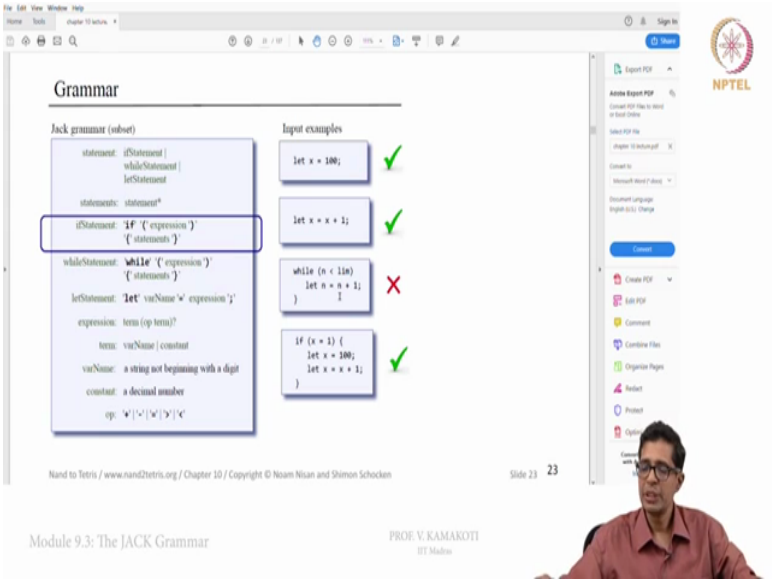
(Refer Slide Time: 11:00)











Now let us go and look at the next example, let x is equal to 100, what is this? This is a let statement, why it is a let statement? Because a let statement has a let command, yes this also has a let command followed by a variable name and a variable name is a string that does not being with a digit that is so x is 1, then it will have a equal to, yes it is equal to, then it should be followed by an expression and you already know that the constant is in expression because expression can lead to a single term or operand many term but that is optional because you have a question mark here, so expression can lead to a term, term can lead to a constant, constant can lead to a decimal number, so 100 is an expression.

So this let x is equal to 100 exactly matches or the rule of let statement and hence let x equal to 100 is a let statement which is grammatically correct. Now let us go and look at the next one let x is equal to x plus 1, there is also a let statement because initially I have let, then I

have a variable name which leads to a string that does not belong to x that does not have a digit to start with x, then there is an equal to, then there is an expression that expression leads to a term op term in this case I will use 1 op term here because this is optional but I can use 1 surely and the term can have a variable name, op can have a plus and the other term can have a constant, so x is equal to x plus 1, x plus 1 is an expression and it matches here.

We have already seen x plus 17 was an expression as you see, so x plus 17 was an expression, similarly we just see that x is as you see here x plus 17 was an expression, so x plus under this also an expression, x plus 1 is also an expression. Now let us see while n is less than lim, let n equal to n plus 1 semicolon this is grammatically wrong because while while agrees, this agrees, this symbol symbol agrees, n is a identifier n is a variable name and variable name is an expression already we know, then so n less than lim is it an expression? Yes, it is an expression only n less than lim is an expression because so let us again see this expression is expression term so term op term, term can be a variable name and op can be less than as you see here and this term also can be variable name so this variable name can be n, this will be lim, so n less than lim as you see here can be derived from expression so n less than lim is an expression so there is no problem here, then you should see a braces, you do not see a braces so there is an error.

So this particular statement while n less than lim without that braces let n equal to n plus 1 does not adhere to our grammar so this is grammatically (correct) incorrect and this is why we get a syntax error so when you compile a C program ad you missed a parentheses or brace there then it gives you a syntax error, so this is a syntax error this is a syntax essentially means grammar, syntax the word essentially means grammar, so there is a grammatical error rate.

Now let us see this if x is equal to 1, let x is equal to 100, let x is equal to x plus 1, this is a very interesting thing. Now let us see if it is a statement then if it coincides then the open parentheses x equal to 1 is an expression because expression is term op term, x is equal to 1 is an expression, correct? Again so expression is term op term and this is variable name, op is equal to (this is another variable name) sorry this is another constant, this variable name is x and this is 1, so from expression I can derive this x is equal to 1.

So x is equal to 1 is an expression, then you should see as you see here a closed parentheses and you see a closed parentheses, then you see an open braces, you see a braces, then you see statements, so what are statements? Statements can be statements can be statement star, what

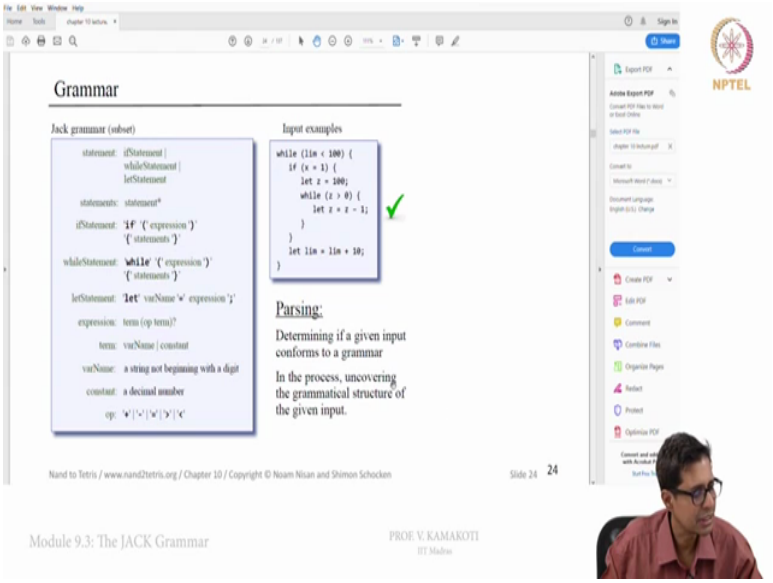
do you mean by statement star? It can be 0 or more statements, if I put a question that means it is optional, it can be or it may not be, but if I say statement star then essentially means 0 or many statements 0 or many statement this is a plural statements is 0 or many statement.

Now let us go and see how this works so I have statements this can be 0 or many statement, so let us say this is two statement since I have a star here I am moving the mouse there I can put two statement and each of this statement as you see above there is a rule for statement can be either if statement, while statement or less statement, so in case each of them is a let statement and from a let statement should have let followed by variable name equal to then an expression followed by semicolon similarly for that let statement.

So this let this will become x, this expression can become a term, term can be a constant and a constant can be 100. So let x is equal to 100 semicolon is basically derived here this is let 100 semicolon it is basically derived. So similarly for the other let statement I can derive let x is equal to x plus 1 semicolon, after so this statements complete statements comprise two statement each statement is a let statement and then that you derive so let x is equal to 100 is one let statement, x is equal to x plus 1 is another let statement. After you finish this you need to have another parentheses here, braces closed braces here which you have so essentially this agrees.

So this is how the grammar is captured and something goes error in this grammar like what you see in the third instance here where I missed a parentheses it is immediately caught as a syntax error. So this is how from basic English language where we have a grammar a computer language like JACK also has an equivalent grammar so and the grammar is basically expressed as a set of rules, a rule basically has a left hand side which derives the things on the right hand side, the right hand side has either terminals or non-terminals, if the right hand side has only terminals then that rule is called a terminal rule you do not progress there is nothing more to proceed after that, if the right hand side has non terminals then atleast one non terminal then it is called a non-terminal rule because beyond that we can keep proceeding and this is a subset of your JACK grammar which basically covers a majority of the things.

(Refer Slide Time: 20:38)



So what is the next step we have to go and find out whether your tokens that you have created basically agrees with your grammar you have given lot of tokens, now we have to agree with your grammar. For example while lim is less than 100 braces, if x is equal to 1, this, this, this, this, this, this does this agree with your grammar? Can I derive this from this this set of rules is the question? Now that is what we term as parsing.

So what we have done in this particular module is basically we have talked about the grammar, how JACK grammar evolves from English grammar and how we can go and test whether a particular input program adheres to the grammar of that language? We have given some examples and we are taking it. So now to what we will do in the next module is what we call parsing, what is important about parsing is that the parsing will actually tell you how to write a program which will basically implement what we have described as a part of this module.

So what is the automation that we can do, so that given a grammar and given an input we need to go and find out whether this input actually matches the grammar if it does not match the grammar then there is a syntax error that we need to report. So we will again meet in the next module.