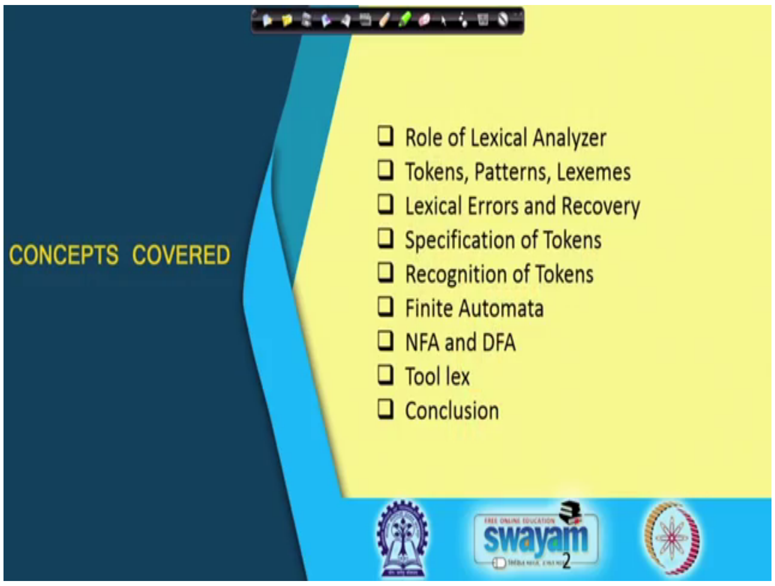
Compiler Design
Prof. Santanu Chattopadhyay
Department of E & EC Engineering
Indian Institute of Technology, Kharagpur

Lecture – 07
Lexical Analysis

So, today we will start with the first phase of a compiler which is known as the lexical analysis phase. So, the task of the lexical analysis phase is to identify the valid words that we have in the input program for the language that you are considering. Now the major challenge that we have here is that, there may be different types of formatting and that formatting is often user specific. Like, one user maybe putting 2 blanks after every word, another user maybe putting one tab after every word.

So, like that the formatting is up to the user. And very often for a to a programmer we suggest that you should do a good amount of indentation so that the code is easily legible and all. So, this type of artistic things that are introduced in the program so, they are of no meaning when it comes to the machine code. So, they are to be handled by the first phase of the compiler, so which is no which is the lexical analysis phase. So, here we will look into that lexical analysis phase in detail.

(Refer Slide Time: 01:21)


CONCEPTS COVERED

- ❑ Role of Lexical Analyzer
- ❑ Tokens, Patterns, Lexemes
- ❑ Lexical Errors and Recovery
- ❑ Specification of Tokens
- ❑ Recognition of Tokens
- ❑ Finite Automata
- ❑ NFA and DFA
- ❑ Tool lex
- ❑ Conclusion

The topics that we are going to cover a like this; first we will discuss about the role of this lexical analyzer. Then in the context of lexical analysis we will introduced some definitions tokens patterns and lexemes. So, they will be used throughout our codes to determine to define the data items that are exchanged between different phases of the compiler.

So, these tokens are identified and generated by the lexical analysis phase. Then there are some lexical errors that may occur like, maybe some word which is least spelt. Like say somebody while writing say i f if writes f i fi. So, were lexical analysis tool it can identify that type of mistakes and it can it can correct those errors, or at least suggest that possibly the user has user wanted to write f a i   f, but has written f i.

So, that way the user can be given some feedback. And not only identifying the error sometimes we need to recover from the error also. Like for example, getting the character f, so the lexical analysis tool will wait to see the next character. So, next character it finds i, then it understands ok f i is some error then it needs to convert it into i f, and then introduce those characters all to the input stream by replacing the original characters f i by that.

So, this way there maybe some error recovery scheme into that the compiler designer can think about and accordingly introduced into the compiler. Then how were these tokens specified? So, that is one important thing. And there can be different rules in the programming language that will tell us like how can we define different tokens. For example, the identifiers in a programming language it has got a particular format. If you considered say the URL of different websites so, it has got a particular format. So, that format or if I consider that as that u entire URL as a single word, that word structure is definitely different from the variable declaration stay that we have in some programming language.

So, that way we should be able to specify the desired tokens very clearly and that should be very unique. And second thing is that after the specification phase has been done, we have to have some recognizer for that. So, specification phase. So, it will define a set of rules by which the tokens will be defined, and in the recognition phase actually the lexical analysis tool it will scan the input program to see what are the tokens that are appearing in the, what are the words that are appearing in the program, and accordingly return corresponding tokens to the person.

Now, for the recognition part, so we will see the tool finite automata it is. So, the as I said that entire course hinges heavily on the automata theory. So, we will be using finite automata for to act as a tool that can identify the tokens that are there in the input program and to generate the words that are there in the input program and generate the corresponding tokens. And in there finite automata category so, we will look into 2 variants of for the finite automata that are used by the lexical analysis phase.

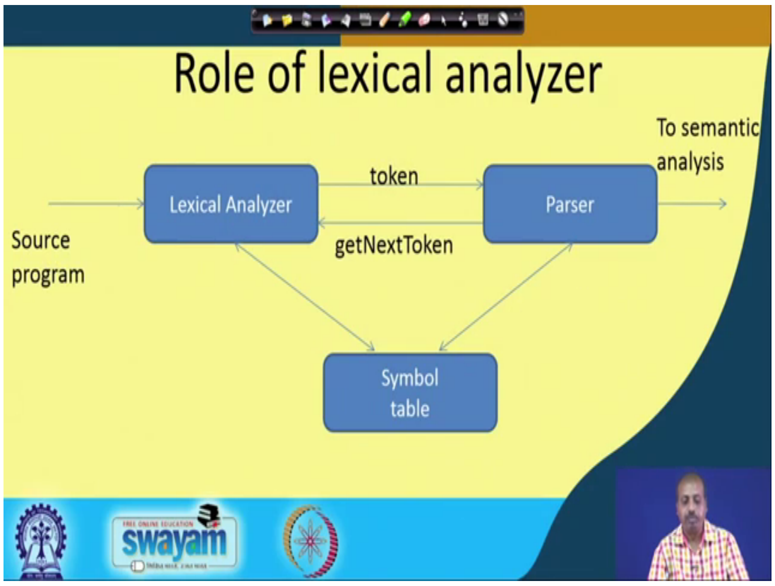
One is known as Non-deterministic Finite Automata or NFA, another is known as Deterministic Finite Automata or DFA. Both of them are equally powerful; however, specification strategy for NFA and DFA is slightly different. In general, we can find that NFA's are, NFA's will have less number of states compared to DFA, but to visualize a DFA maybe it is easier for us. But if we can visualize the NFA then we can get more compact representation for the token.

So, initially it seems that the DFA is the better one, but after some practice you will find that it is easier to construct NFA then to construct a DFA. And DFA often may often have large number of stakes compared to NFA. So, complexity of DFA maybe more, though their capabilities are same NFA and DFA they are perfectly equivalent to each other. Then we will look into one tool that was originally introduced with the Unix operating system known as lex.

So, which given a specification, so it can generate a lexical analyzer program for some language. So, foreign language, so you can define it is the word the rules defining the words of the language and then feed it to the lex tool. And the lex tool will come up with a program lex dot y y dot c, which is a c program which can be used by the compiler designer to automatically generate the lexical analysis part. So, you do not need to write explicit code for the lexical analysis tool. Because the language maybe very complex the input programming language maybe very complex.

So, writing the corresponding lexical analyzer from scratch may be difficult. So, this tool actually helps us, and it has become so popular that nobody now nobody thinks in a different way apart from writing the lex specification and generating the lexical analyzer automatically. Finally, we will conclude the discussion.

(Refer Slide Time: 07:17)


Role of lexical analyzer

So, start with the role of lexical analyzer. So, as you see this tool takes the input source program of some language and it will scan through this program and accordingly it will generate tokens. And as I said previously that this lexical analyzer and the parts are they work hand in hand. And they are they, so the demarcation line between the two is very thin ok.

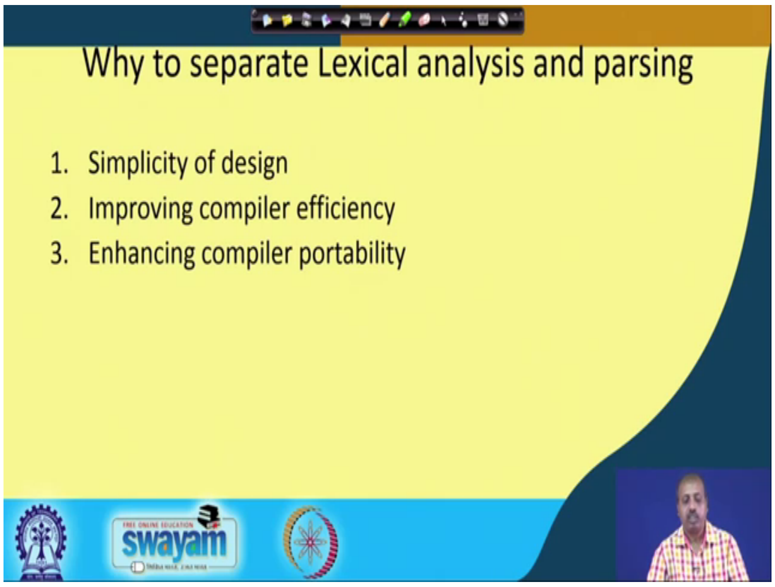
So, the way it operates is that this parts are whenever it requires some further token, it will request the lexical analyzer tool to give it the token, and give it the give the next token. So, intern the lexical analysis tool, it will scan the input file wherever it was the input program wherever it was scanning previously from that point onwards. And accordingly it will identify the next valid word that appears in the input program of the language, and return it, returns the corresponding token to the person. So, what is token? So, we will come very shortly to the definition. For the time being you can just take it as if this is some quantity which corresponds to the valid words of the programming language and each valid word has got a different token identifier.

Now, these parts are intern. So, it gets the tokens and accordingly it understands the it checks whether the program is syntactically correct or not; the entire program. So, this lexical analyzer tool is giving it words, and the parts are is actually checking for sentences. So, if the sentence sentences are correct, then it will give it to the semantic analysis tool which will do the type checking, which will do say your other associated actions meaning of the program it will try to identify, and then it may go to old generation phase.

So, both lexical analyzer and parser, they utilize this symbol table data structure. So, as I say previously, that symbol table stores all the identifiers that we have in the program. And so, this lexical analyzer whenever it finds a new identifier, it informs the symbol it puts it into the symbol table, telling that this is the next symbol that we have, and it is put into the symbol table.

And the parser whenever it needs the reference to any symbol, so it will refer to the symbol table. So, who will populate the symbol table? So, that is often the a choice of the compiler designer, sometimes it is done by a some compiler designer will put the job on to the lexical analysis phase, some will put on the parser phase depending on the ease of programming and type of language. So, different approaches are taken by different complier designers.

(Refer Slide Time: 10:11)



Why to separate Lexical analysis and parsing

1. Simplicity of design
2. Improving compiler efficiency
3. Enhancing compiler portability

Now, as I said that this lexical analysis phase and the parsing phase they are working hand in hand. So, why should we separate them into 2 different phases, why not design the entire thing together? So, there are 3 basic reason for doing it, first of all simplicity of design. So, actually this whole compiler can be designed as a monolithic software, monolithic piece of program, where starting from the lexical analysis till code generation everything or even the code optimization.

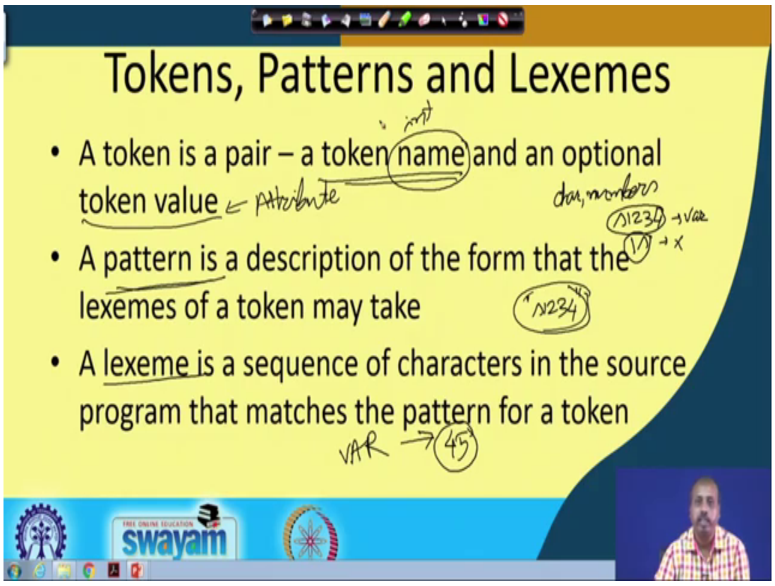
So, everything can be done together, but that will make the whole program very complex. So, just to avoid it just to avoid that complexity so we have got simplicity of design so that this lexical analysis phase if it is separated, then some portion or some action, so they are separated into a different phase.

So, designing that phase become easier for the compiler designer. It improves comp compiler efficiency, because now this later phases of the compiler they need not work with the input stream or the input character strings that I have in the input program. So, they can be working with some tokens, and we will see very shortly the tokens are each token is given some integer identifier. So, we can say that the later part of the program of the compiler that the compiler that is designed, so they will be working with integers. And as you know if we have a program that does number clenching that is often much simpler to design compared to program that uses character strings and some manipulation around them.

So, that way this compiler efficiency improvement so, this is better if we have got this lexical analysis on a separate phase. It also enhances compiler portability. So, from one system to another systems if we want to port the compiler, then take the compiler to some other system then the formatting the input format may be slightly different on to the target system. So, they are so those things can be taken care of by the lexical analysis phase itself. So, the later phases of the compiler they can be kept as it is.

Only the front end part so that needs some changes and it can be done. May be from one for the same for the same language if we go from 1 platform to another platform, may be the syntax will the way in which words are defined may be slightly different. And those things can be taken care of by the lexical analysis phase itself.

(Refer Slide Time: 12:55)



Tokens, Patterns and Lexemes

- A token is a pair – a token name and an optional token value
- A pattern is a description of the form that the lexemes of a token may take
- A lexeme is a sequence of characters in the source program that matches the pattern for a token

Coming to the definition of tokens, patterns and lexemes; a token is a pair, a token name and an optional value. So, every token is given in name and it has got some value. The value is integer, and most often this token name and that we have, so it is given as it is also, given as integer and this token value part. So, it may be consisting of it may be consisting of some attribute of the token. A pattern in turn, so if pattern is a description of the form of a, form that the lexemes of a token may take. Whereas, a lexeme is a sequence of characters in the source program that matches the pattern of a token.

So, this lexeme is, so lexeme is the input string that is there that is read from the input file. And we have got some patterns that are valid for the program. For example, if I say that my identifier in my programming language is like this. Any sequence of character and any sequence of character and numbers, any sequence of characters and numbers, but the first symbol there must be a character. So, if I have got something like s 1 2 3 4. So, this is a variable in that as per my definition.

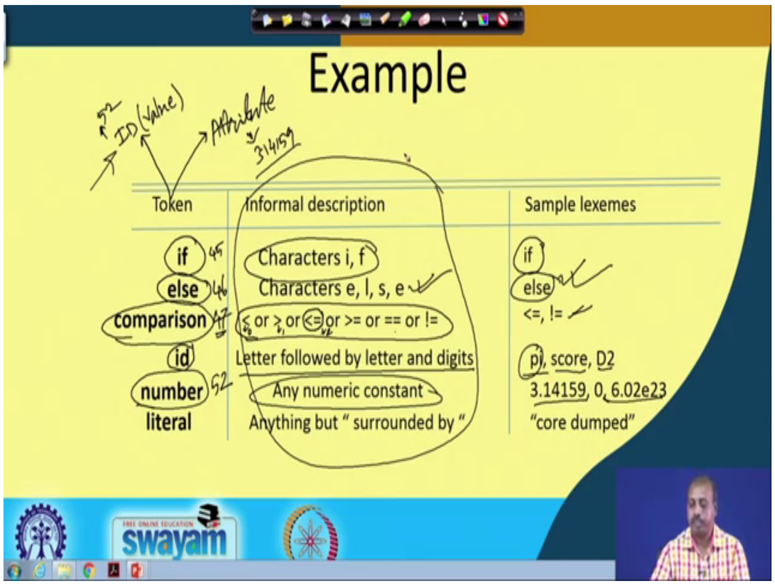
Then say, but say 1 s, so this is not a variable, this is not a variable. So, in my input string in the input program if I get s 1 2 3 4 so, this is the lexeme that you are getting. So, this is the lexeme, a sequence of character in the source program. And the pattern is I am telling I will say how do I specify this pattern, but pattern is actually telling a rule by which I can define what are valid a lexemes. Like here I am telling that any sequence of characters such that the characters and numbers such that first symbol is a character is we will define a variable.

So, any so as per my as per my definitions so, this is a valid definition valid variable, but this is not a valid variable. So, this is the pattern, so how do we write a pattern that will see a shortly, but somehow it will capture this rule. And now after getting this lexeme, matching with this particular pattern, which pattern is called the variable. So, after this lexeme s 1 2 3 4 matches with the pattern variable, so what is returned is the token.

So, this token name, so this is often taken as integer and this there can be an optional token value; so, which is often called attributes also. So, each token has got a value and a or a name and a value or attribute. So, in the compiler designer may give that variable the corresponding number may be say 45.

So, on getting this s 1 2 3 4 on the input string; since, it is matching the pattern of character number sequence for variable, so the compiler the lexical analyzer will return this token name as 45, and as a value it can return the symbol table location for the variable s 1 2 3 4. So, this way depending upon the token that we have we can have a value that is in our name and attribute or token value. So, that way it can that that pair can be there, so it can be token name and token attribute, and name is often it is an integer. But that is not mandatory some compiler designer may use token names as some other format also may not be integer. But in general, it is taken as integer. And the next tool that I am talking about, so that also takes it as integer.

(Refer Slide Time: 17:23)



Example

Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but " surrounded by "	"core dumped"

So, coming to the some coming to some example like say this say this particular pattern, so characters i and f. So, this is a pattern. So, this is the lexeme, so that is appearing in my input string. And according to the token that is returned is if. So, though it is written as a character string i f, but you can understand that these what I want to mean a something different than this one.

Similarly, when it is say characters e l s e, the formal informal description of the lexeme is e l s e these 4 characters coming together. The sample lexeme is else e l s e that. So, these thing appears on the input file. So, this matches with this particular rule that is a character sequence e l s e. So, this matches with the, so this input sequence matches with this particular pattern, and then it will return this as the token.

And as I said that the may be, so all these tokens they are given some numbers. So, this may be say 45, this may be 46. So, like that randomly some values, some integer values unique integer values may be assigned. Similarly, so say all the symbols less than greater than less or equal greater or equal equality and not equality. So, these are this defines another rule ok. So, this defines another rule or another pattern, and some sample lexemes that appears in the input program maybe this less or equal, or not equal to, so these are these are appearing on the input file as sample lexeme.

And now the token that is returned is comparison. So, this comparison token may be given the number say 47 and the lexical analysis tool. So, when the parts are asked for the token, if it gets on the input file less or equal the lexical analysis tool will return the value 47 to the parser. But that is not sufficient because I need to tell which comparison operator I am talking about.

So, accordingly it can have some attribute part the token can have some attribute part, and there somehow we tell that I am talking about this symbol. Maybe each of them is again given a code. So, this is 0, this is 1, this is 2, so like that each of them may be given a code. And that code may be returned as the attribute of the token comparison. Similarly, so for say for identifier, the rule maybe later followed by letter or digits. So, this is the rule, so this is the pattern that I am talking about. So, this lexemes that appears in the source program may be pi score D 2. So, these are some character strings appearing on the input program.

So, since the for lexical analysis tool, when it finds p i, pi. So, it find that there is no rule that defines lexeme the corresponding pattern pi, but this particular rule matches ok. So, this sell it because matches because it starts with the letter and it is it continues with letters or numbers letters or digits, so this part matches. So, it will return the token id, but it has to do something more, because this pi most probably pi is a variable. So, it will you put it into the symbol table. And as an attribute part, so it may return the symbol table offset to be used by the parser.

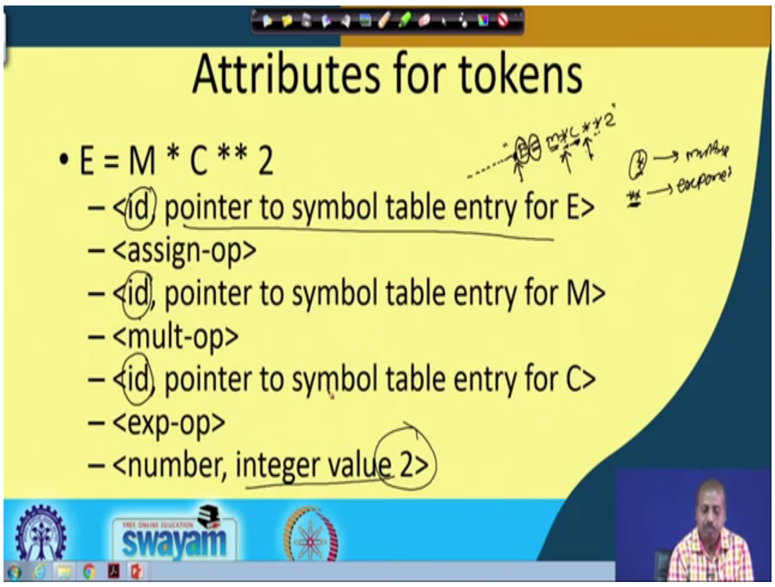
Some numbers, so this is a 3.14, so 14159, so this appears in the input program or say something like this. So, the rule that I have is any numeric constant. So, this is the pattern ok, and accordingly the token returned is number. So, here also token is number, but it the token does not talk about the value that is actually worked in there. So, this token has got 2 parts as I said. It has got a identity the token id and the attribute ok. So, token id or token value and then attribute part. So, this id is suppose this here the corresponding token number that is given is say 52.

So, this id value is 52 and in the attribute part we return the actual value. So, maybe 3.14159 so that parser if necessary, so it will it will it will check the grammar rule whether for checking the grammar rule it will look into the token id, but for doing some code generation and all maybe it will need this attribute value; so that also it can do.

So, this is how this parser and this lexical analyzer will work together. So, literal is anything that surrounded by double core. Anything, but is surrounded by this double code symbols like core dumped. So, this is sample lexeme, so this is a literal. So, this way this lexical analysis tool so, it will be looking into the rules that we have. So, we have not yet seen how to capture this informal description part by means of pattern rules. So, that we are going to see very shortly. Once that is done, so this for the language I can define a set of rules that can define, that can guide valid lexemes that are that can be there.

Now, the lexical analysis tool will scan the input file and it will identify lexemes that are occurring in the program, and accordingly it will return the corresponding token to the parsing phase. So, this is how this whole thing will work.

(Refer Slide Time: 23:21)



Attributes for tokens

- E = M * C ** 2
  - <id, pointer to symbol table entry for E>
  - <assign-op>
  - <id, pointer to symbol table entry for M>
  - <mult-op>
  - <id, pointer to symbol table entry for C>
  - <exp-op>
  - <number, integer value 2>

Like say when I talking about attributes of tokens. Suppose I have got this with this particular the string appears in my program E equal to MC square. So, E it appears in my program M into C then 2 star, then 2 this whole thing appears in the input program somewhere.

So, lexical analysis tool, so it will start scanning from. So, the when the parser as asked it to return next token at that point the lexical analysis tool might have seen up to this much in the program. So, it has seen up to this much. So, it scans the input and it finds the symbol E and after that there is equality. So, if it scans through the valid lexeme rules where valid pattern rules that are there in the program, that are there in the language and it finds that E it can be an identifier. So, it will return the token id and as the value part or attribute part. So, it will return pointer to the symbol table entry e.

So, the variable e must have been defined previously in the program. So, accordingly when that point it if the variable e was seen, so it was put into the symbol table now sometime later when this line e equal to MC square comes, then the lexical analysis tool gets the symbol e, and it identify e and it can search through the symbol table find out what is the offset of e in the symbol table and it can return that pointer to symbol table for e to the parser. So, this is the token value and this is the token attribute.

Now, it finds, so next time the parser asks the lexical analyzer to return the next token. So, at that time it will find this equality as the next token. And equality is the token the corresponding token name is assigned of. So, this it returns the token assigned of. So, we do not need any other attribute for this particular token. So, no attribute is mentioned. Next when it is asked, so it will find this M. So, it will return id and pointer to symbol table entry for M. Then it finds next token is star, so it will return the multiplication operator, then it will be finding the next time it will find this c, it will return the token as id and pointer to symbol table entry for C. Then it will find 2 stars, 1 star, 2 star

Now, it can, so what, so here there is a confusion, because here on getting one star it was it has returned mult op as the token, but here getting the first getting the first star itself, it could have returned returned mult op to the parser. But it does not do that, it actually what the lexical analysis tool does is that it finds the maximum match. So, there is a rule in my language which says that the if you get a single star that is a multiplication, but if you get 2 stars that is an exponentiation. So, this rule is there in my language.

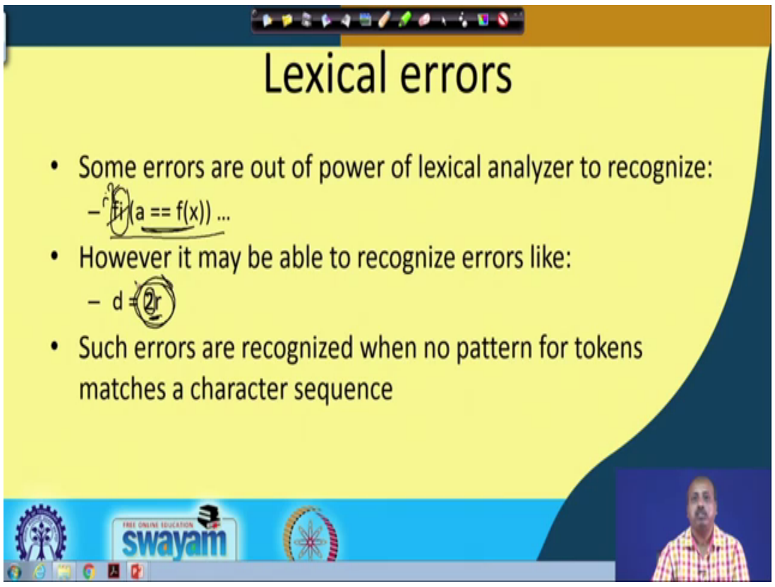
So, the lexical analysis tool, so it will try to find maximum match at this point and it will find that the maximum match is this on,e not this one. Whereas, at other places the maximum match was this single star. So, at this point when I was looking for maximum match, so this was a single star. Because star C does not have any meaning for the programming language; so that is considered here.

So, naturally this here it is a maximum match, so star star it matches. So, it will return the exponential operator. And then this 2 is a number, now the as I said that it will have an attribute, so attribute value is integer value 2.

Now, you look into these 3 places, so here also it is returning id. So, here also it is returning id here also it is returning id. At all the 3 places it is returning the same token, ok. If this token id is given the number say 45. So, at all the 3 places it is returning the value 45 to the parser. But as an attribute it is differentiating between them. In the first place it is returning pointer to E, second place pointer to M and third place pointer to C. So, all these information are required like as far as the grammar checking is concerned, so these pointers are not necessary. But if you are trying to check what are the types of this variables, and if you are trying to generate code for that, then you need to have the corresponding symbol table pointers, ok.

So, for basic syntax checking it is not needed. But for later stages of this code generation and all, so these pointers will be useful. So, that is why this lexical analysis tool it not only identifies the words and returns are corresponding tokens, it also identify some attributes that may be meaningful for that particular word and returns them to the parser. Similarly, when it is considering this number so, it is returning the value of the number as 2. So, this is done here. So, this way all these tokens, so they have got some attributes and these attributes are returned to the parsing phase.

(Refer Slide Time: 29:05)



Lexical errors

- Some errors are out of power of lexical analyzer to recognize:
  - fi(a == f(x)) …
- However it may be able to recognize errors like:
  - d = 2r
- Such errors are recognized when no pattern for tokens matches a character sequence

There can be lexical errors, so lexical errors as the situation is such that errors are out of power of lexical analyzer to recognize. So, this like say this one. Say some errors the tool cannot identify, like say this one in some c language program we have written something like this.

Now, so we it cannot identify the situation because this fi; so ideally it seems that fi should be if, ok. But it should not do it automatically, because it may so happen that user has defined a function whose name is fi. And this a equal to equal to fx is an expression. So, value of this expression is parsed as a parameter to the function fi, and that that will be evaluating the function fi.

So, this has got some other possible meaning also, but it is, but actually if this is made if then that is the most probably that is the correct one. But the other one the existing one that fi and within bracket a equal to equal to fx, so that is also may be correct.

So, we have got confusion. So, the lexical analysis tool cannot handle this type of situation this type of errors; however, it can identify this as an error. Because in my programming language for the identifiers it is said that it should start with a symbol, it should start with a character and then it should continue with characters or digits.

Now, this, so no symbol can start with a 2. So, this is not going to be a symbol. So, it, but at the same time it cannot take 2 as a number, because if it is a number when after that between 2 and r there should be an operator. So, a number must be it must be separated by at least one blank space. So, that way the lexical analysis tool can identify an error at this point. And it can tell the user that there is an error at this point.

So, such errors are recognized when no pattern for lexemes matches if character sequence. So, if you do not find any match, then we can say that there are some there is some problem with this particular sequence of characters that are occurring, so that is an error. Another possibility is that there may be some foreign characters in my input string, and then the lexical analysis tool since it knows the alphabet, so it can inform that those foreign characters are there; so it is not in your language, so accordingly it can generate some error.