

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

Lecture – 08
Lexical Analysis (Contd.)

So, if errors are there. So, we can have we can think about some error recovery policy.

(Refer Slide Time: 00:19)

Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token then
then
then
- Delete one character from the remaining input { } ?
- Insert a missing character into the remaining input if, if 2/b 2/b +, +
- Replace a character by another character
- Transpose two adjacent characters

The slide features a yellow background with a blue border. At the bottom, there is a banner for 'swayam' (Free Online Education) with logos of various institutions. Handwritten notes in black ink are present next to each bullet point, providing examples or additional context.

So, the problem with the lexical error is that, the input string input file that we have an input stream. So, there is some problem in the input stream itself. So, if you are thinking about correction. So, we have to correct the input stream and if there is no look back ok. So, lexical analysis tool so it is scanning the input file from one side and going towards other side. So, it has already scanned some portion which needs to be corrected.

So, the lexical analysis tool has to undo some operations that it has already done for correcting the text or doing some recovery. So, one possible recovery mode is the panic mode of recovery. So, panic mode of recovery here it is said that a successive characters are ignored until we reach to a well formed token. So, some tokens are very unique for example, semicolon is a unique token, then say if you are considering its a C programming language then this symbols like open brace and closing brace so they are unique.

Similarly, if we get a closing parenthesis. So, that may be meaning end of an expression. So, like that certain tokens are certain character sequences are unique. So, if you find that there is some error so when the when it was going through this part. So, it can go on, it can go on ignoring all these characters still it gets a closing brace because after the closing brace it is I can assume that from this point a new correct token will start.

So, in a panic mode of recovery. So, it will ignore all these tokens and it will be starting a phrase with the next synchronizing token so or synchronizing symbol. So, this is good because I can the lexical analysis tool. So, it can come out of the error and continue processing of the remaining one. So, another possibility is that delete one character from the remaining input.

So, this is just one more one possibility that we you delete one character with the expectation that the from the next character I will get a valid one, as I was telling that the example that I took previously that its a 2 r. So, after getting 2 so, I was expecting. So, to return this as a number I expect that at this place I should have a blank or some operator say plus or multiplication some operator.

So, what I do? I just ignore the next character. So, maybe after r after this r there is a blank to if I delete this r when I will get this to as a valid token. So, this way I can delete one character from the remaining input and accordingly I can make a valid word out of the, whatever I have scanned or I can insert the missing character into the remaining input.

So, here also the same to same example like it is 2 b then I can insert one blank in between in that case this you will be returned as we have some number and after that we will b returned as an identifier. That will solve the problem for the lexical analysis tool. Now whether this is correct or not that is not the concern because it may or may not be correct, but the lexical analysis tool it got stuck at that this point.

So, it was unable to proceed further. So, if you do this then it will be able to proceed with the remaining input and maybe it will do something, maybe it will flash some error message and the user will correct it.

But if it can give a large number of error reported to the user otherwise it will stop at that point and it will come out from this compilation phase telling that there is a wrong, there

is an invalid character at this point patriotic only this much that is not there. Sometimes we replace a character by another character. So, that is also another recovery policy. So, if you so, there may be alternative words that are available. So, maybe one in some character in some words only one character is wrong.

So, what it does is that it modifies that word completely with a valid word. So, normally for this purpose we have got this text processing software which do this thing. So, they have got some dictionary of words and it whenever it finds that some word is coming which is not matching. So, will try to replace it by some valid word.

So, it will try to replace a character by another character or it can transpose to a two adjacent characters. So, two characters suppose that I have got this if I. So, if I do a transpose for these two characters will change the interchange their position and it will give me i f similarly. So, for while I am trying to write the qr then maybe I have actually written teh n and then if I transpose these two characters then it will become t h e n this type of corrections can be done by the error recovery phase.

So, whether so how many of them will be done and whether it is done always. So, it is up to the compiler designer. So, language design will not talk anything about error recovery. So, and it has got nothing to do with code generation also because if there is a mistake in my program. So, that is the mistake that is an error.

So, it is not generate the code, but the compiler designer I can think about this type of possible errors and flash good number of error messages good amount of error messages so, that the program can be corrected by the user. So, it is up to the user to it is up to the compiler designer to identify possible or visualize possible mistakes that the user may do and accordingly come up with some policy for the recovery.

(Refer Slide Time: 06:37)

Input buffering

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
 - In C language: we need to look after $=$, $<$ or $<=$ to decide what token to return
 - In Fortran: DO 5 I = 1.25
- We need to introduce a two buffer scheme to handle large look-aheads safely

Handwritten examples on the slide:

- $x = y$ (with an arrow pointing to the $=$)
- $x < y$ (with an arrow pointing to the $<$)
- $x <= y$ (with an arrow pointing to the $<=$)
- $x - y$ (with an arrow pointing to the $-$)
- $x + y$ (with an arrow pointing to the $+$)
- $x * y$ (with an arrow pointing to the $*$)
- x / y (with an arrow pointing to the $/$)
- $x \% y$ (with an arrow pointing to the $\%$)

Input buffer example: `E = M * C ** 2 eof`

Footer: swayam

Sometimes we need to do some buffering, some lexical analyzer it needs to look at the some symbols to decide about the token to return.

So, typical example maybe this symbols like in a C programming language we have got the symbol say minus then equality less than. So, we have to look head and then only you can see you can do something written the proper token for example, example if you have got say x minus y.

So, at this point if the input, if your input point are somewhere here you have got a this minus symbol should I return the token minus ok. So, it is not it may or may not be correct because it may be the user as written x minus minus plus y, user has actually written this and you are at this point now. Now getting the single minus you should not written a minus the lexical analysis tool it should look I head see the next minus and then this blank and then it should say that this minus might have. So, this is basically auto decrement mode similarly this equality looking into the looking into one equality. So, whether it is correct or not that is the question, because I may have two equalities so, x equal to y and x equal to equal to y.

So, both are valid in the C programming language and if the user has written like this then if you are returning that it is an equality operator. So, that is wrong. So, it has to you have to say that it is a equal check operated by see you both these equality symbols you should return a single token which is the equality check. So, this way this lexical analysis

tool its, it needs to look ahead similarly is less than we have got many variants like less than is there less than equal to is there then we have got this left shift.

So, all these are there in the programming language. So, depending upon the programming language you may need to look ahead for the next few characters to know what is the valid token and in general the policy followed by the lexical analyzer is that it returns the maximally matched token ok. So, it will written the maximally matched token. So, whichever token matches with the next maximum number of characters so, that token will be returned a very nice example that we have this is the Fortran language. So, Fortran language it has it has got a loop constructs.

(Refer Slide Time: 09:20)

Input buffering

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
 - In C language: we need to look after -, = or < to decide what token to return
 - In Fortran: `DO 5 I = 1.25` (handwritten: `DO 5 I = 1, 25`)
- We need to introduce a two buffer scheme to handle large look-aheads safely

Handwritten notes on the slide:
- `DO 5 I = 1.25` is circled, with arrows pointing to `1.25` and `5`.
- `DO 5 I = 1, 25` is written next to it.
- `DO 5 I = 1.25` is circled, with arrows pointing to `1.25` and `5`.
- `DO 5 I = 1, 25` is written next to it.
- `DO 5 I = 1.25` is circled, with arrows pointing to `1.25` and `5`.
- `DO 5 I = 1, 25` is written next to it.
- `DO 5 I = 1.25` is circled, with arrows pointing to `1.25` and `5`.
- `DO 5 I = 1, 25` is written next to it.
- `DO 5 I = 1.25` is circled, with arrows pointing to `1.25` and `5`.
- `DO 5 I = 1, 25` is written next to it.

Code snippet: `E = M * C ** 2 eof`

So, where the structure is like this that you look. So, DO then a label and then we have got some variable I equal to say 1 comma 25 and then you write the body of the loop and then somewhere at this point. So, you have this label and here you say continue here. So, means that this body of the loop will be repeated, first with the I is equal to 1 then I equal to 2 etcetera up to I equal to 25. So, this is the, this is a valid so this is some example that we are doing.

So, actually the user wanted to write in this case DO 5 I equal to 1 comma 25, but by mistake this 1 comma 25 has been written as 1 dot 25. So, this 1 dot 25 so this is a real number which is allowed in Fortran language and Fortran language it does not does not recognize this blanks in between the symbols, it just ignore this blank. So, this entire DO

5 blank b I so, this whole thing is reduced to a variable DO 5 I. So, it thinks that this DO 5 I is a single variable that is equal to 1.25. So, this is this is the interpretation of this particular statement.

So, until and unless we have seen you so whether this a comma or a full stop or dot. So, you cannot know whether you should written DO as a loop token or this DO 5 I as a identifier as an identifier. So, you see the level of complexity that this lexical analysis tool is going to face. So, it has to look ahead quite some, quite a few some characters to see whether a comma appears after the first number after the equality symbol. If it is not then it is the whole that the entire left hand side will be reduced to a variable name where as if it is a comma in that case it has to written only the do part and the input pointer should be pointing to the next character here; where as in this case if it is 1.25 then the input pointer will come to this point.

So, this type of things are there so, it designing lexical analysis part is not that symbol apparently it seems it is quite simple I will just do a simple string match and where ever I get the match I will written it. So, that is not the case because programming language are often very critical in nature. So, and that is up to the language designer they have done it like that.

So, we have to follow that. So, sometimes you need to introduce a two buffer scheme to handle large look ahead safely so that when one buffer pointer is advancing through one buffer, the other buffer holds the next symbol next input string. So, that way there is no possibility of over flow, sometimes we need to save some amount of buffer information on to some other so other buffer so that these symbols are not lost in the process any way so, they are all implemented in the lexer tool. So, while designing the compiler we do not really feel that we have to do so complex things, but the lexer tool that is there. So, it takes care of that.

(Refer Slide Time: 12:58)

Specification of tokens

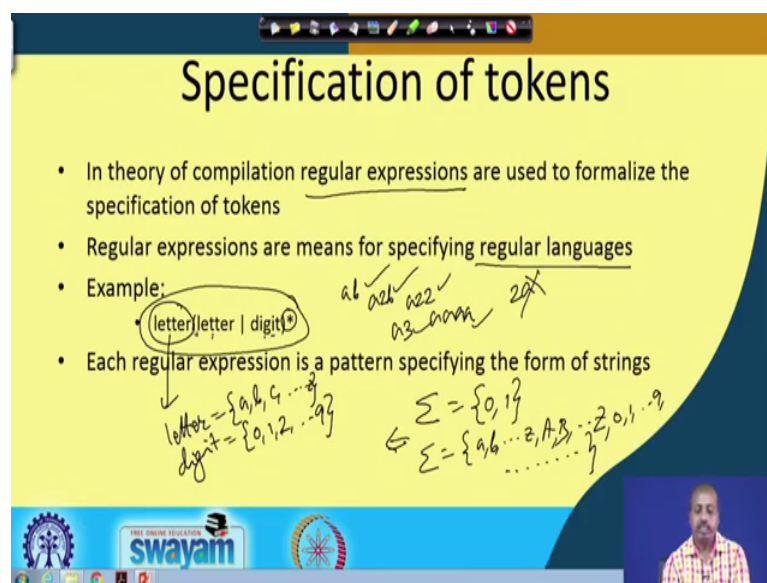
- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:

(letter|letter | digit)

α_1 ✓ α_2 ✓ α_3 ✓ α_4 ✓
- Each regular expression is a pattern specifying the form of strings

$\text{letter} = \{a, b, c, \dots, z\}$
 $\text{digit} = \{0, 1, 2, \dots, 9\}$

$\Sigma = \{0, 1\}$
 $\Sigma = \{a, b, \dots, z, A, B, \dots, Z, 0, 1, \dots, 9\}$



Next we will come a very important part of this lexical analysis phase which is known as specification of tokens, like how do you tell what are the valid tokens in your programming language? So, any language or not only programming language any language. So, as I said any language has got an alphabet which defines the characters or symbols that the language can help.

So, if we are the, if we are talking about say the set of sentence is over binary symbols then my alphabet set is often represented as sigma. So, this has got 0 and 1 unit only two symbol, then if I say that we have got a say the English language text then all the symbols a, b up to z then this capital A, B up to Z then we have got these numbers 0, 1 up to 9.

Then all the symbols that I that I have in the English language so they will come as a alphabet set for the language. So, from this alphabet set we have to define some rules by which I will tell what are the valid strings or valid words for the language. So, here we will be using a particular structure known as regular expression to formalize the specification of tokens ok. So, we will be learning about regular expression for that purpose. So, regular expressions are means for specified regular languages. So, I will come to this regular language parts slightly later, when we go to the parser phase parsing phase and we will talk about the grammars and all for the time being we take it as accepted that the regular expressions can specify regular languages.

So, what is a regular language we learn it later. So, a typical example of regular expression is like this letter then with in bracket letter then vertical bar digit bracket close star. So, where this letter is a set of symbols so, any letter maybe I can say the letter set is, the set letter is all the lowercase letter that we have in the English language then digit. So, this is a set of symbols 0, 1, 2 up to say 9.

So, these are digits. So, it says that this whole thing is a regular expression which must start with a letter and after the letter I can have a letter or a digit and this letter or digit can occur any number of times. So, this star at the end means that with whatever portion on which we have applied this star that can appear any number of time.

So, if I if I like write a b so that is a valid that is a valid word for this particular regular expression then I can have say a 2 b or a 22 or a 3. So, a all a s anything that we can write like this, but anything that star does not start with a letter is not valid, like if I ask whether this two a is a valid word as per this rule or not. So, it will say no. So, this not correct, but all these are correct, all these are correct, but this is not. So, so we have got this particular rule where this. So, this is a regular expression this whole thing is called a regular expression.

So, time back I was telling how are you going to specify the patterns that are allowed for the programming language. So, this is actually the, this is done by means of the regular expression. So, you have to define the set of regular expressions for your language that will define all words that you have in your language.

So, each regular expression is a pattern that specifies the form of string for the language. So, the type of strings which will correspond to this regular expression so that will be identified by this regular expression. So, every token now you can understand that, each token has got a corresponding regular expression. So, for example, I have previously I talked about that token if ok.

(Refer Slide Time: 17:34)

The slide is titled "Specification of tokens" and contains the following content:

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
 - `letter(letter | digit)*`
- Each regular expression is a pattern specifying the form of strings

Handwritten notes on the slide include:

- A box with the text "if comparison" and an arrow pointing to the list.
- The text "if comparison" written vertically.
- A list of comparison operators: `>`, `<`, `>=`, `<=`.

The slide also features a Swayam logo and a small video feed of a presenter in the bottom right corner.

So, these that is the corresponding regular expression is simply if this is the corresponding regular expression similarly if I have got say comparison operator. So, the comparison, this particular token.

So, the corresponding the regular expression is equality or greater than or less than or greater equal or less equal. So, this way you can specify the token. So, the corresponding regular expression. So, each token will have got a corresponding regular expression and the task of the lexical analysis tool is to see which of these regular expressions maximally match the next part of the input. So, if this is your entire, if this is the entire input file that you have may be at present the lexical analysis tool is at this particular line and in this particular character. Now, if you ask when the parser asks it for the next word or next token what it does it scans from here and sees which token matches maximally ok.

So, we it finds the maximal match. So, may be the maximal match up to this much entire thing maximally matches with some token then this entire thing will be. So, that is checked by the regular expression. So, this entire part matches the corresponding token will be returned. So, it is the task of the lexical analysis tool to see which token matches or which regular expression matches maximally from the given set of regular expressions and return the corresponding token to the parser. So, this is how this lexical analysis tool is going to what.

(Refer Slide Time: 19:28)

Regular Expressions

- ϵ is a regular expression denoting the language $L(\epsilon) = \{\epsilon\}$, containing only the empty string
- If a is a symbol in Σ then a is a regular expression, $L(a) = \{a\}$
- If r and s are two regular expressions with languages $L(r)$ and $L(s)$, then
 - $r|s$ is a regular expression denoting the language $L(r) \cup L(s)$, containing all strings of $L(r)$ and $L(s)$
 - rs is a regular expression denoting the language $L(r)L(s)$, created by concatenating the strings of $L(s)$ to $L(r)$
 - r^* is a regular expression denoting $(L(r))^*$, the set containing zero or more occurrences of the strings of $L(r)$
 - (r) is a regular expression corresponding to the language $L(r)$

$L(r|s) = L(r) \cup L(s)$

So, we start with the definition of regular expression. So, epsilon so it is a regular expression denoting the language L of epsilon equal to epsilon containing only the empty string. So, throughout our discussion. So, this epsilon will be these will be calling as this a special symbol. So, will be call this as null ok. So, wherever you find this epsilon you take it as null no is a wide sort of thing you can say.

So, this is special regular expression so if your language. So, ai if your language contains only the null string that is very regimentally language that has got only one string which is a null string. So, that is empty string. So, that is represented by the regular expression epsilon, then if you have got the symbol a small in the alphabet set Σ then a itself is a regular expression corresponding to the language that has got the string which contains only a single a .

So, if my Σ in the alphabet set I may have all this symbols say a b and c fine then L of a L of a means it is talking about the language that has got the alphabet a only it does not have the other alphabets b and c . So, naturally so L of a is only the, it has got only a only a single valid string in the language which is a single a similarly L of b . So, each symbol that you have in the alphabet set constitutes a language consisting of a single occurrence of that symbol ok. So, L of b is a single b L of c is a single c .

So, it defines those language that contains the single b or single c in it now comes combination of two or more regular expressions. So, if r and s are two regular expression

with corresponding languages L of r and L of s then r vertical bar s is a regular expression denoting the language $L_r \cup L_s$. So, when I say L_r so this is L_r is nothing, but set of strings, set of strings in the language in the language with r as regular expression. So, that is L_r . So, this is a set it defines the all the strings that are possible in the language identified by the regular expression r .

So, when I say r vertical bar s or we read it as r or s . So, if L_r and L_s are two languages then this r vertical bar s it will correspond to the language $L_r \cup L_s$. So, L of r or s is nothing, but L of r union L of s . So, any string which belongs to the language L_r also belongs to the language L_r or s . Similarly any string which belongs to the, which belongs to the language L_s also belongs to the language L_r or s . Next we look in to the next rule it says that if $r s$ is a regular expression so $r s$, r is a regular expression and s is regular expression or they are occurring side by side.

(Refer Slide Time: 23:21)

Regular Expressions

- ϵ is a regular expression denoting the language $L(\epsilon) = \{\epsilon\}$, containing only the empty string
- If a is a symbol in Σ then a is a regular expression, $L(a) = \{a\}$
- If r and s are two regular expressions with languages $L(r)$ and $L(s)$, then
 - $r|s$ is a regular expression denoting the language $L(r) \cup L(s)$, containing all strings of $L(r)$ and $L(s)$
 - rs is a regular expression denoting the language $L(r)L(s)$, created by concatenating the strings of $L(s)$ to $L(r)$
 - r^* is a regular expression denoting $(L(r))^*$, the set containing zero or more occurrences of the strings of $L(r)$
 - (r) is a regular expression corresponding to the language $L(r)$

$r = ab$ $p = bc$ ab $abbc$

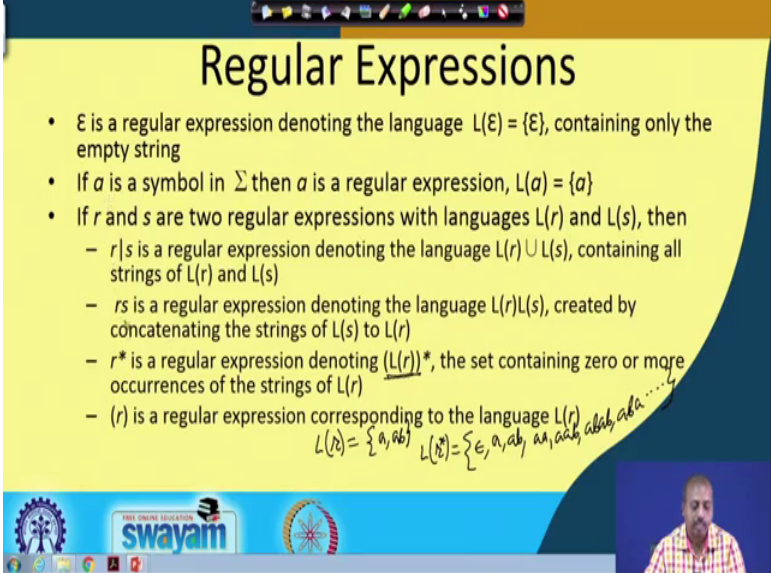
So, for example, a is a regular expression and b is a regular expression or I can say $a b$ is a regular expression and $b c$ is another regular expression. So, that way may be my r equal to $a b$ and s equal to $B c$ in that case $r s$ is this one $a b B c$. So, if $r s$ is a regular expression then that will denote the language L_r followed by L_s .

So, any language any string that is created by taking one string from L of r and another string from L of s and concatenating the second string to the first string. So, you have got say one language L of r suppose it has got the language, it has got the strings a , $a b$ and a

c and L of s has got the string x and y then L of r s. So, this will have all these, all these strings like a x, a y, a b x, a b y, a c x and a c y that is you take the first part from the language L r and second part from the language L s and concatenate the two strings together. So, what you get is the set of strings that are allowed in this language.

So, if you have constructed individual, individual languages for the regular expression r and s. So, you can take concatenation of those strings of different languages and tell them that it is the regular expression for; it is the language for the regular expression r s. So, this is there then we have this special regular expression r star. So, r star means this star symbol it means that it is 0 or more occurrences of the regular expression r.




(Refer Slide Time: 25:33)



Regular Expressions

- ϵ is a regular expression denoting the language $L(\epsilon) = \{\epsilon\}$, containing only the empty string
- If a is a symbol in Σ then a is a regular expression, $L(a) = \{a\}$
- If r and s are two regular expressions with languages $L(r)$ and $L(s)$, then
 - $r|s$ is a regular expression denoting the language $L(r) \cup L(s)$, containing all strings of $L(r)$ and $L(s)$
 - rs is a regular expression denoting the language $L(r)L(s)$, created by concatenating the strings of $L(s)$ to $L(r)$
 - r^* is a regular expression denoting $(L(r))^*$, the set containing zero or more occurrences of the strings of $L(r)$
 - (r) is a regular expression corresponding to the language $L(r)$

$L(r) = \{a, ab\}$ $L(r^*) = \{\epsilon, a, ab, aa, aab, abab, abba, \dots\}$

So, if you have got say L of r, suppose I have got the two strings a and a b then L of r star L of r star. So, this will have all these all these words all these strings. So, 0 or more occurrences of this language this language words, if I take 0 occurrences. So, that is the null. So, epsilon if I take one occurrence. So, I can get a and a b if I take two occurrences then I can get a a or a a b similarly a b, a b, a b a. So, this way we can think about different words ok, different strings this set is infinite because this is 0 or more occurrence you can just go on augmenting taking more and more strings and connecting them together. In different number of times I have taken only two times, you can take it any number of times because this is a star.

So, this r^* is a regular expression denoting $L(r)$ whole star, the set containing 0 or more occurrences of the string of $L(r)$ ok. So, and each whenever we have taking the string you are free to choose any string. So, that way they can be concatenated in arbitrary fashion.

So, this is a very powerful operator that we have ok, unlike this or said say consecutive occurrence. So, this star is a 0 or more occurrences of the symbol then with in bracket r is a regular expression which corresponding to the language $L(r)$ so it is nothing new. So, this r the regular expression r has got the language $L(r)$. So, if you put it with in bracket also we do not get anything new, we get the same language $L(r)$. So, this way the regular expression, the regular expressions are defined. So, this is definition of regular expression.

(Refer Slide Time: 27:45)

Regular definitions

$d_1 \rightarrow r_1$
 $d_2 \rightarrow r_2$
 \dots
 $d_n \rightarrow r_n$

• Example:
 $\text{letter_} \rightarrow A | B | \dots | Z | a | b | \dots | Z | _$
 $\text{digit} \rightarrow 0 | 1 | \dots | 9$
 $\text{id} \rightarrow \text{letter_} (\text{letter_} | \text{digit})^*$

So, here is some example suppose. So, we have got this thing like regular definitions. So, definition 1 will give me some regular expression d_1 , definition 2 will give me some regular expression d_2 definition n will give me regular expression r_n . So, one such definition is the letter underscore so that means, I am talking about. So, this is a definition which says that all these characters can appear A B capital A capital B upto capitals Z then small a small b up to small z and this underscore. Then this digit is a definition it is 0 1 up to 9 and id is another definition.

So, these are these are the specified in terms of regular expression only, you see this whole thing is a regular expression, similarly this whole thing is a regular expression.

Now I am use these two definitions to define an id how am I defining letter underscore followed by letter underscore or digit and this star. So, I can start; that means, for a valid id for any character string to be called a valid id. So, it may I start with letter or it may start with underscore character.

So, maybe so I can write, I can have a character string like this underscore a b. So, that is valid similarly I can have like a underscore b 2 underscore 3. So, this is also valid. So, they are all id they will all be identified as id. So, this can be done. So, we can have this type of regular expressions formulated by first writing a few definitions and then writing the more complex one, building the more completes one based on the simpler regular expressions.

So, so these are called regular definitions. So, you can we use these definitions to build more and more complex set and. So, you can understand that anything so this is not valid like you cannot select 2 1 underscore 5. So, that is not a valid one or 2 underscore a that is not valid one because it says that it should start with the letter or underscore character, but here for these two cases that is not true so they are not valid.

Whereas these cases they are fine and they can occur any number of time. So, in the first character, even this is going to be a valid string. So, all underscore and this can go up to infinite. So, this is also valid because I am starting with an underscore that follow, that satisfies the first part of the definition and then I am starting with the then I am doing with it is the second underscore and that is satisfying any number of time. So, this also a valid id in the as per in this particular language.