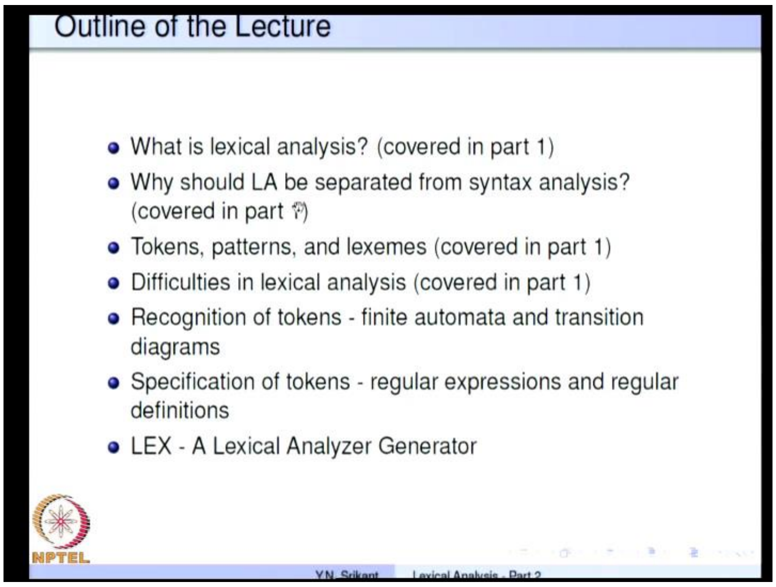
Principles of Complier Design

Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 3
Lexical Analysis - Part 2

(Refer Slide Time: 00:24)



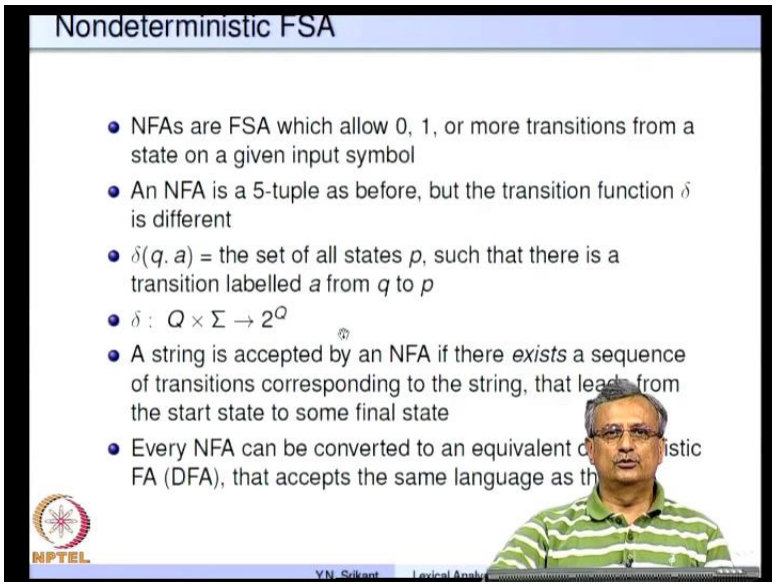
Outline of the Lecture

- What is lexical analysis? (covered in part 1)
- Why should LA be separated from syntax analysis? (covered in part 1)
- Tokens, patterns, and lexemes (covered in part 1)
- Difficulties in lexical analysis (covered in part 1)
- Recognition of tokens - finite automata and transition diagrams
- Specification of tokens - regular expressions and regular definitions
- LEX - A Lexical Analyzer Generator

Y.N. Srikant   Lexical Analysis - Part 2

Welcome to the second part of lexical analysis. So, in this lecture, we are going to continue our discussion on lexical analysis. In the last lecture, we studied lexical analysis, its purpose now why should lexical analysis be separated from syntax analysis, what exactly our tokens, patterns, and lexemes and also the difficulties in lexical analysis, and I gave an introduction to finite automata. Today we will continue the discussion on finite automata, study what are the transition diagram regular expressions etcetera, etcetera.

(Refer Slide Time: 01:02)



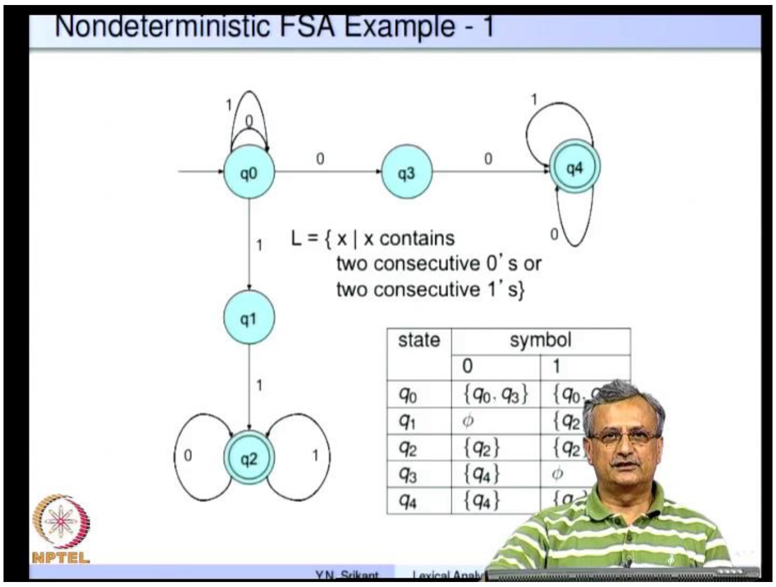
Nondeterministic FSA

- NFAs are FSA which allow 0, 1, or more transitions from a state on a given input symbol
- An NFA is a 5-tuple as before, but the transition function δ is different
- $\delta(q, a)$ = the set of all states p, such that there is a transition labelled a from q to p
- $\delta : Q \times \Sigma \rightarrow 2^Q$
- A string is accepted by an NFA if there *exists* a sequence of transitions corresponding to the string, that leads from the start state to some final state
- Every NFA can be converted to an equivalent [illegible]istic FA (DFA), that accepts the same language as th[illegible]

Y.N. Srikant Lexical Analy[illegible]

To do a bit of a recap, let me go through the material on non deterministic finite state of the automata.

(Refer Slide Time: 01:15)



Nondeterministic FSA Example - 1

L = { x | x contains two consecutive 0's or two consecutive 1's}

state	symbol	
	0	1
q_0	$\{q_0, q_3\}$	$\{q_0,$ [illegible]
q_1	ϕ	$\{q_2$ [illegible]
q_2	$\{q_2\}$	$\{q_2$ [illegible]
q_3	$\{q_4\}$	ϕ
q_4	$\{q_4\}$	[illegible]

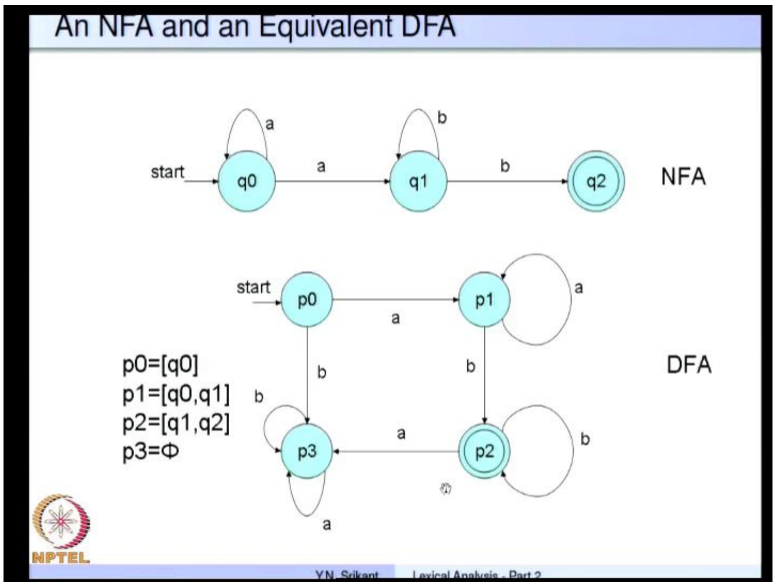
Y.N. Srikant Lexical Analy[illegible]

So, here is an example. Typically a deterministic automaton has exactly one transition on every symbol of the alphabet from each state, whereas in the case of an NFA there will be more than one transition possibly even 0 number of transitions on a particular symbol in the alphabets. So, for example here from the state 0, we have transitions on 0 here, and also here. So, 2 transitions on 0, 2 transitions on 1 as well. But from the state q 3 we have a transition on 0, but there is no transition on 1. Similarly from the state q 1, we have a transition 1, but no transition on 0.

So, any of these combinations are permitted in the case of a non deterministic finite state automate. So, therefore what happens is, because there may be more than one transition on a particular symbol from a state, the transition function cannot be captured very simply, you know as q cross sigma to q, but it becomes actually a mapping from q cross sigma to the power set of q that is we provide a set of states for each symbol. So, for example, from the state q 0 on 0, we go to either state q 0 or q 3, similarly on symbol 1 we go to state q 0 or q 1 etcetera, etcetera. And if there is no state on a particular symbol for example, from q 1 on 0 there is nothing. So, we indicated as phi.

So, this is the difference between the deterministic finite state automata and non deterministic finite state automata. And the basic theorem of automata theory is that NFA, every NFA can be converted to an equivalent DFA that excepts the same language as the NFA.

(Refer Slide Time: 03:18)


An NFA and an Equivalent DFA

p0=[q0]
p1=[q0,q1]
p2=[q1,q2]
p3=Φ

Y.N. Srikant Lexical Analysis - Part 2

So, now let us do how exactly an NFA can be converted to an equivalent DFA. In other words, here is an example, so this is an NFA and we should construct a DFA, which excepts exactly the same language. So, the NFA here as two transitions on symbol a, its alphabet is a comma b. From the state q 1 it has two transitions on symbol b and from the state q 2 there are no transitions.

So, from q 0 there is no transition on b and from q 1 there is no transition on a. The way we proceed to construct the DFA is a fairly straight forward demand driven approach. So, the start state you know is q 0 and every state of the deterministic finite state automaton is indicated in square brackets here. So, as we already saw the, it is possible to make a transition to a set of states from each state of NFA. So, quite logically each state of the DFA would corresponds to a subset of the total set of states. So, for example, P 0 has just one state q 0 of the NFA whereas, P 1 has two states, q 0 comma q 1 from the NFA, P 2 has q 1 and q 2 from NFA and P 3 has nothing. So, it is a phi state that is actually an error state.

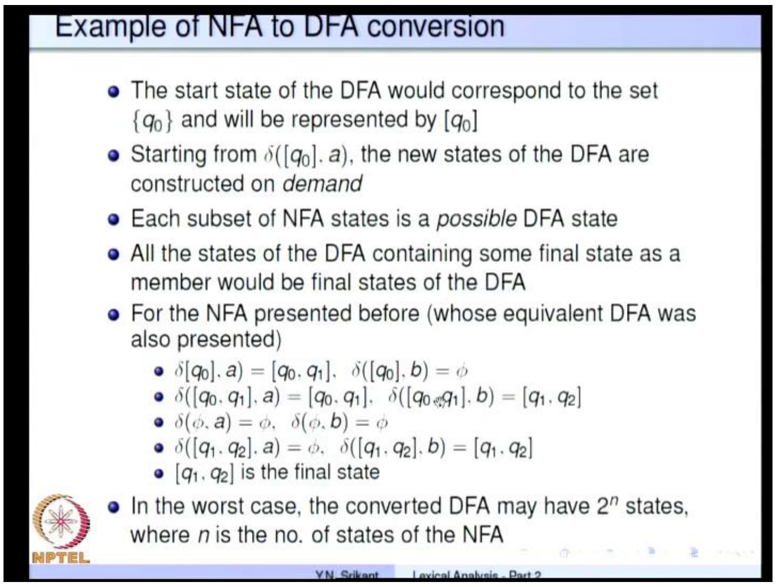
So, we always begin with the state q 0 of the NFA and that is regarded as a start state of the DFA as well. So, let us say P 0 is q 0 the notation is very clear, it consist of the state q 0 of the NFA and from each of the states in this set we find out the transitions that the NFA would make. So, in this case from the state q 0 the NFA makes a transition to either state q 0 or to state q 1. So, the state q 0 comma q 1 in a combine fashion becomes the new state of a DFA. So, from P 0 on a, we go to P 1, which actually is a combination of q 0 and q 1. Now, from P 0 on b let us see what happens? So, from q 0 on b there is no transition; that means, there is an error. So, from P 0 on b we go to an error state, which is called as phi. So, P 3 is phi. So, we mark that as the error state and that transition every time, we make a transition to phi state we actually make a transition to P 3.

So, the P 0 to P 3 transition will be label b now. So, we have now two states P 0 and P 3. So, and we have covered you know all the transitions from P 0. So, let us see what happens to the P 2 mind you has not been construct yet, but P 1 has been constructed. So, from the state P 1 let us see what happens to the DFA on the symbols a and b. So, to capture the effect of the DFA on symbol a from the state P 1, we will look at the transitions of the NFA from the both states q 1 and q 0. So, from q 0 on a it remains either in q 0 or close to q 1 that is, it can actually remain in the state P 1 itself.

So, that is the effect of q 0 and from q 1 there is no transition on a. Therefore, this particular thing does not add any extra to this the transition of delta P 1 you know on a. Next, what about b? So, from q 0 on b, it has nothing and from q 1 on b it either remains in q 1 or goes to q 2, this state does not exist. So, we clear a new state combining q 1 and q 2 and let us call it as P 2. So, there is a new state P 2 and the transition from P 1 on b is to the state P 2. So, quite understandably, you know the transitions of P 2 on b would P 2 itself. So, q 1 on b goes to q 1 and q 2. So, and q 2 does not add anything to the transitions. So, therefore, P 2 on b remains in the state P 2 and on a from P 2. Let us see what happens q 1 goes to error state and q 2 also goes to error state on a. So, therefore, from P 2 we have a transition to P 3, which is the error state.

And now, from the error state either on a or on b be remain in the error state. So, we have self lobes for both a and b. So, these are the only possible state that we need to construct that is not necessary to construct all the subsets states of the NFA. For example, here q 0, q 1, q 2 together give rise to eight combinations each of, which is a possible DFA state, but in this case this DFA as only four states. It is possible to find you know, a examples where 2 to the power n states would be constructed.

(Refer Slide Time: 09:08)



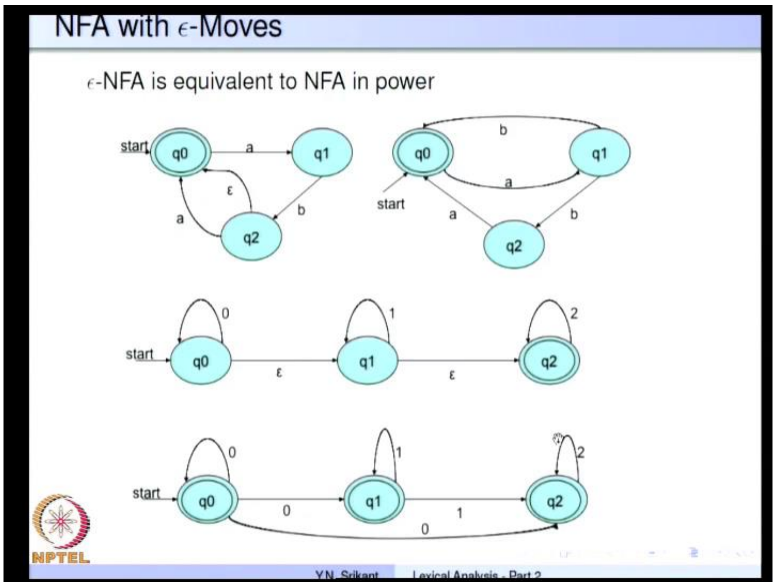
Example of NFA to DFA conversion

- The start state of the DFA would correspond to the set $\{q_0\}$ and will be represented by $[q_0]$
- Starting from $\delta([q_0], a)$, the new states of the DFA are constructed on *demand*
- Each subset of NFA states is a *possible* DFA state
- All the states of the DFA containing some final state as a member would be final states of the DFA
- For the NFA presented before (whose equivalent DFA was also presented)
  - $\delta[q_0], a) = [q_0, q_1]$, $\delta([q_0], b) = \phi$
  - $\delta([q_0, q_1], a) = [q_0, q_1]$, $\delta([q_0, q_1], b) = [q_1, q_2]$
  - $\delta(\phi, a) = \phi$, $\delta(\phi, b) = \phi$
  - $\delta([q_1, q_2], a) = \phi$, $\delta([q_1, q_2], b) = [q_1, q_2]$
  - $[q_1, q_2]$ is the final state
- In the worst case, the converted DFA may have 2^n states, where n is the no. of states of the NFA

Y.N. Srikant Lexical Analysis - Part 2

So, just to summarize what I told you so far. The start state of the DFA would correspond to the set q 0 and will be represented by square bracket q 0 that is the set of the DFA. Starting from delta of q 0 comma a, that is we try to find out the transitions of the new automaton on a. The new states of the DFA are constructed on demand. Each subset of the NFA states is a possible DFA state, but it is not necessary that all of them would be required in the DFA. And how about the final states so if you look at the picture it is easy to see that q 2 is final state here. So, all the subsets, which actually include P 2 in this case it is only q 1 q 2 which corresponds to P 2. So, those would we marked as final states. So, all the states of the DFA containing some final state as a member would be final states of the DFA. So, and here is a description of all exactly we arrived at it, it is notion. So, I would request you to go through it in detail and understand it. In the worst case the converted DFA may have a total power n states where n is the number of states of the NFA.

(Refer Slide Time: 10:33)



NFA with ϵ-Moves

ϵ-NFA is equivalent to NFA in power

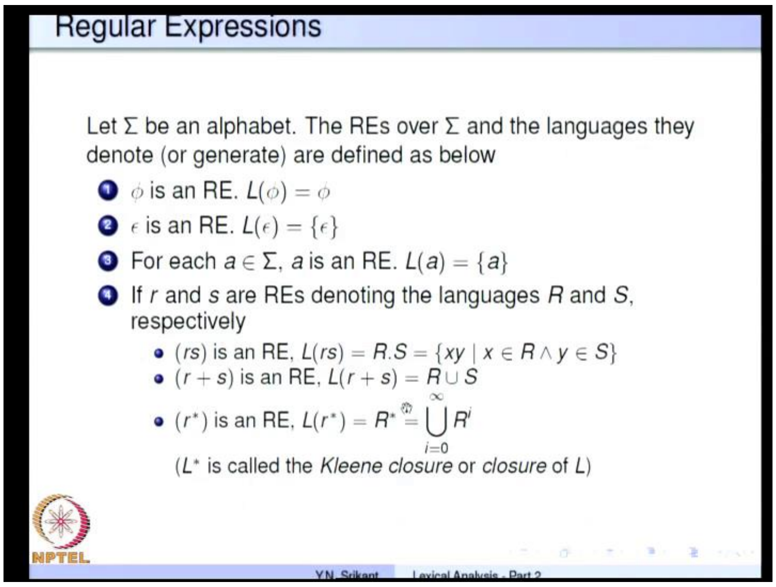
There is another variety of a non deterministic finite state automaton, which is called as epsilon NFA; epsilon is the empty string as we know. So, for example, there is a transition from q 2 to q naught on epsilon and here is a transition from q naught to q 1 on epsilon, q 1 to q 2 on epsilon, etcetera. So, this is silent transition and it does not consume any input. So, in this particular case q naught as 1 transition on a to q 1 another transition on b to from q 1 to q 2 and 1 more transition on a from q 2 to q zero.

But there is a silent epsilon transition also possible from q 2 to q zero. So, the implication is once the automaton arrives in the state q 2. It can make a silent transition without consuming any input to the state q 0 or it can consume by input and then go to state q zero. So, let us see how that is represented in the NFA, there is a algorithm, which can convert epsilon NFA to equivalent NFAs and DFAs, but this is not exactly required our discussion. So, I would give examples and then continue. So, from q 0 on a its goes to q 1, which is as in the original NFA and from q 1 on b we arrive at q 2, but q 2 can also go to q 0 on a silent transitions so; that means, effectively from q 1 on b we may either stop it q 2 or go to q 0. So, we add transitions from q 1 to q 2 on b.

And we would also add transitions from q 1 to q 0 on b, to capture the effect of this epsilon transition. Finally, the epsilon transition is removed and we have a just transition from q 2 to q 0. So, let us see, how it works in this example so from q 0, we have a transition on epsilon to q 1 the implication is on any number of zeros the NFA can remain in q 0 or it can make a silent transition to q 1 and finally, it can also continue to make a silent transition to q 2. In other words on consuming a sequence of zeros the automaton can remain in q 0 or in q 1 or in q 2 and since this happens we would actually you know, we can also consume just epsilon in each case takes it to q 1 and then to q 2.

So, the string epsilon is also accepted we make both all the three q naught, q 1 and q 2 as final states to indicate that epsilon is accepted and then 0 is of course, accepted by making 2 epsilon transitions. And then you know it can make a transition from q 1 again to q 2 on any number of ones. So, from q 0 we add a transition on 0 to q 1, the epsilon transition is removed and from q 1 we add a transition on 1 to q 2 and we also add a transition from q naught to q 2 on 0 indicating that, there is a possibility of epsilon transition all the way to q 2. So, this particular converted NFA accepts the same language as this n epsilon NFA, but sometimes epsilon NFAs are easier to constructs specially in an algorithm make way which, we are going to see other relater.

(Refer Slide Time: 14:25)



Regular Expressions

Let Σ be an alphabet. The REs over Σ and the languages they denote (or generate) are defined as below

1. ϕ is an RE. $L(\phi) = \phi$
2. ϵ is an RE. $L(\epsilon) = \{\epsilon\}$
3. For each $a \in \Sigma$, a is an RE. $L(a) = \{a\}$
4. If r and s are REs denoting the languages R and S, respectively
    - (rs) is an RE, $L(rs) = R.S = \{xy \mid x \in R \wedge y \in S\}$
    - $(r+s)$ is an RE, $L(r+s) = R \cup S$
    - (r^*) is an RE, $L(r^*) = R^* = \bigcup_{i=0}^{\infty} R^i$

    (L^* is called the *Kleene closure* or *closure* of L)

Y.N. Srikant Lexical Analysis - Part 2

So now, So far, we studied finite state automata and looked at some of their properties conversion of NFA to DFA, what would be language accepted or recognized by DFAs etcetera. As I said there is the DFAs and NFAs are machines, whereas it is possible to have a finite representation of regular languages in the form of know as regular expressions. So, regular expressions are specifications and we are going to use regular expressions to specify lexical analyzers and that is our interest in these regular expressions. So, let us first define regular expression, this is really an inductive definition. So, we have three base classes phi.

So, for example, sigma lets say is an alphabet, the regular expressions over sigma and languages they generate or denote are defined as below. So, the null set is a regular expression by definition and the language of phi is also a null set phi. The empty string epsilon is also a regular expression by definition and language of, language generated by the empty string is the set containing the empty string. Please note the difference between phi and the set containing epsilon. Phi contains nothing whereas, this set contains the null string epsilon and for each symbol of the alphabet sigma let us say a, the symbol a itself is a regular expression by definition. So, the languages of that single symbol you know the regular expression is the set containing that particulate symbol a itself.

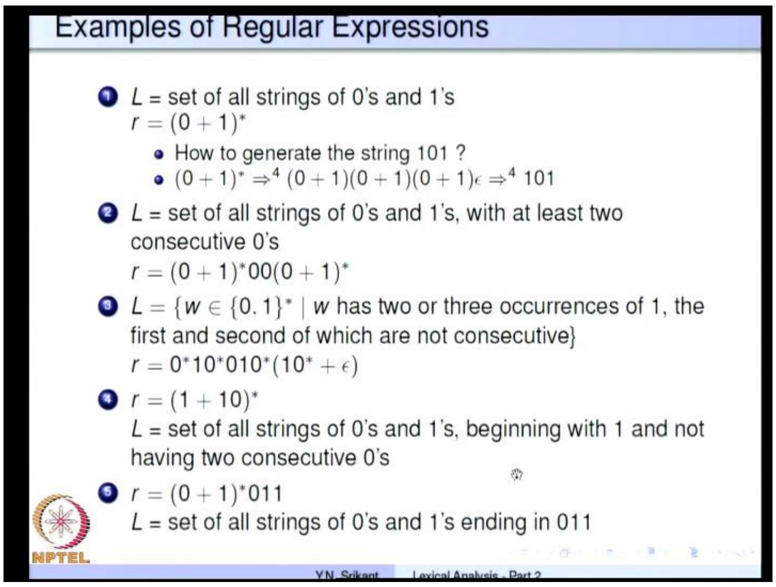
So, these are the three base classes and now, what fallows are the inductive classes. So, now, let us say we are given two regular expressions r and s and these represent the regular languages capital r and capital s. So, first combination mechanism is concatenation. So, R concatenated with S. So, R S is a regular expression and the language of the combined regular expression concatenated regular expression R S is also the concatenated set R dot S. So, you concatenate R and S, what does it mean you take one element from R another element from S and concatenate them. So, in the case of strings it is string concatenations. So, x, y such that x in R and y in S so this is our new language for the concatenated regular expression R S. Now, another operator is the plus which corresponds to union, in the case of languages. So, R plus is a regular expression.

So, and the language of the regular expression r plus s is the union of R and S, R union S. So, expression sentences are strings generated by either or R S or both in the regular expression language of the regular expression r plus s. And finally, the third way of generating regular expressions using compositions is r star or the Kleene closure. So, this operator star is nothing but you know applying concatenating r a large number of times including zero. So, epsilon is 1 and then r, r dot r, r dot, r dot, r dot r, etcetera, etcetera,

etcetera any number of times. So, all this entire you know all these regular expressions are combined with a plus. So, in other words epsilon plus r plus r dot r plus r dot r dot r etcetera, etcetera.

So, this entire sequence is our new regular expression. So, that is that the language corresponding to this new regular expression is r star, which is defined as the union infinite union i equal to 0 to infinity of R to the power i. So, you have R to the power of 0 and then R to the power 1, R square, R cube, R four, etcetera, etcetera. So, we have so many of these you know strings generated by r star. So, it gives us an infinite string really infinite sorry, language really. So, whereas, these 2 if r and s are generate only finite languages r dot s and r plus s generate only finite languages whereas, even if r generates a finite languages this r star really generates an infinite languages. So, L star is known as the Kleene closure or the closure of the L to put it simply.

(Refer Slide Time: 19:53)



Examples of Regular Expressions

1. L = set of all strings of 0's and 1's
   $r = (0+1)^*$
   - How to generate the string 101 ?
   - $(0+1)^* \Rightarrow^4 (0+1)(0+1)(0+1)\epsilon \Rightarrow^4 101$
2. L = set of all strings of 0's and 1's, with at least two consecutive 0's
   $r = (0+1)^*00(0+1)^*$
3. $L = \{w \in \{0,1\}^* \mid w$ has two or three occurrences of 1, the first and second of which are not consecutive$\}$
   $r = 0^*10^*010^*(10^* + \epsilon)$
4. $r = (1+10)^*$
   L = set of all strings of 0's and 1's, beginning with 1 and not having two consecutive 0's
5. $r = (0+1)^*011$
   L = set of all strings of 0's and 1's ending in 011

Y.N. Srikant Lexical Analysis - Part 2

So, now takes examples of regular expressions. So far, we only looked at the way we combine regular expressions. So, if you consider the regular expression 0 plus 1 star this corresponds to the language set of all strings of zeros and ones, let us see how? So, how does 1 generate the string 1 0 1. So, you what you have is 0 plus 1 star. Now, you unroll this star four times. So, I never words you have 0 plus 1 followed by 0 plus 1 followed by another 0 plus 1 and finally, epsilon. So, all this is permitted because this is r star. So, I can concatenate r any number of times. So, I have concatenated it four times.

So, this r, first r generates 0 plus 1, second r generate 0 plus 1, this is a third r and fourth r generates epsilon. So, in each of these, this regular expression is 0 plus 1. So, I can generate the string either 0 or 1 from each of these. So, let me make this generate 1 this can generate a 0 this can generate 1 and this of course, is an epsilon. So, concatenating all the strings we really get 1 0 1, epsilon of course, gets absorbed into the strings 1 0 1 itself. So, in this way since, you can replicate 0 plus 1 any number of times we have done it four times here, you can generate forty, you can do it forty times or four hundred times or any number of them, we can generate strings all strings of zeros and ones.

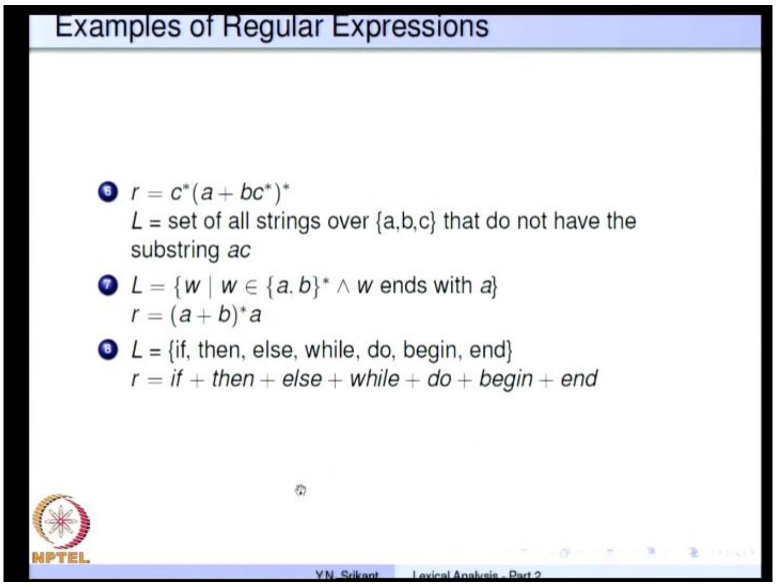
Each of them, each of these replication can generate a 0 or 1 thereby we can generate every 1 of the strings possible from the set combination of zeros and ones. The next example, the same 0 plus 1 star, but then it is followed by 2 zeros and then again we have a 0 plus 1 star. So, this 0 plus 1 star as before can generate any strings of zeros and ones, this can also generate any string of zeros strings of zeros and ones. So, the only property that is true for all the strings is that there are 2 zeros together in the middle. So, this can generate only once and this can generate only once, but these 2 strings, you know 0 and 0 will remain in the middle.

So, the only property that we can states for this language is the set of all strings of zeros and ones with at least 2 consecutive zeros. Mind you, this can generate any number of zeros as well therefore, we can only say at least 2 consecutives 0 because there may be zeros here and zeros here as well. The third example there is a 0 star 1, 0 star 0 1, 0 star followed by 1 0 star plus epsilon. So, again the only property we can state for each of the strings generated by this regular expression is w has 2 or 3 occurrences of 1 the first and second of which are not consecutive. So, how can it generate 2 of them? Definitely 2 because 0 star 1 and then 0 star so 0 star can generate possibly even epsilon that the worst case.

So, in that case still 1 is generated this 0 star can also generate epsilon in which case this is a 0 which is definitely generated. So, far we have 1 0 and then we have another 1 let us say even this 0 star generates epsilon. So, we still have just 1 0 1 followed by 1 0 star plus epsilon. So, if we assume that just this epsilon is active and 1 0 star is not used in the generation. We have our string 1 0 1 which has 2 ones, if we instead of using epsilon we use 1 0 star, we would have generated an extra one so that is third one. So, what it says is the first and second are not consecutive simply because there is a 0 which is being generated here. So, this is the mechanism using which, we can generate strings from regular expressions.

So, this the fourth example 1 plus 1 0 star its says set of all strings of zeros and ones beginning with a 1, which is very easy because both alternatives generates strings starting with 1 and not having consecutive zeros. So, we generate a 0 and then in the next instances 1 plus 1 0 we again generate a 1. Therefore, the 0 gets separated by another 0 with a at least 1 1 therefore, consecutive zeros are not possible. And fifth example is 0 plus 1 star 0 1 1. So, this is very easy any strings of zeros and ones ending with a 0 1 1.

(Refer Slide Time: 25:16)



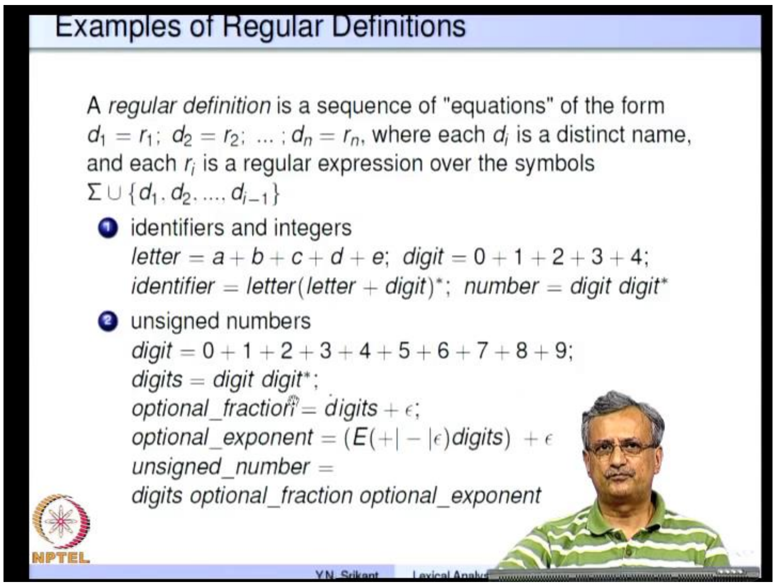
Examples of Regular Expressions

6. $r = c^*(a + bc^*)^*$
   L = set of all strings over {a,b,c} that do not have the substring *ac*
7. $L = \{w \mid w \in \{a.b\}^* \wedge w \text{ ends with } a\}$
   $r = (a + b)^* a$
8. L = {if, then, else, while, do, begin, end}
   $r = if + then + else + while + do + begin + end$

Y.N. Srikant   Lexical Analysis - Part 2

So, some more examples from be, let us they where are the previous examples for all from zeros and ones. So, you have c star followed by a plus b c star whole star. So, you can only say the strings do not have any substring a c because again a plus b c star make sure that a c is not a substring. And this is r equal to a plus b star a so that means, its ends with a and then we have the reserved words of a simple language. So, if then else while do begin end to write a regular expression generating this set L, you just attach all the strings with plus as that gives us a regular expression corresponding to this particular set. This is the way we are going to write regular expressions for various reserved words of our languages.

(Refer Slide Time: 26:13)



Examples of Regular Definitions

A *regular definition* is a sequence of "equations" of the form $d_1 = r_1;\ d_2 = r_2;\ \dots;\ d_n = r_n$, where each d_i is a distinct name, and each r_i is a regular expression over the symbols $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$

1. identifiers and integers
   $letter = a + b + c + d + e;\ \ digit = 0 + 1 + 2 + 3 + 4;$
   $identifier = letter(letter + digit)^*;\ \ number = digit\ digit^*$
2. unsigned numbers
   $digit = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9;$
   $digits = digit\ digit^*;$
   $optional\_fraction = digits + \epsilon;$
   $optional\_exponent = (E(+|-|\epsilon)digits) + \epsilon$
   $unsigned\_number =$
   $digits\ optional\_fraction\ optional\_exponent$

Y.N. Srikant

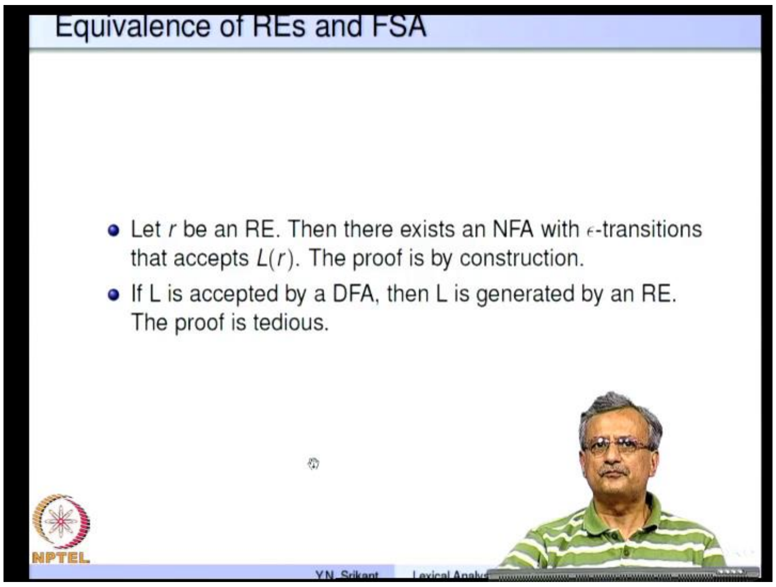
So, 1 level above the regular expression is a regular definition. A regular definition is really sequence of equations and this is more you know used as a short hand for writing specifications of lexical analyzers.

So, let us understand what these are. So, we have definitions of the form d 1 equal to r 1, d 2 equal to r 2, d n equal to r n, which each d I is a distinct name and r I is a regular expression. And each of the r 2, r 3 etcetera, can use the previous d I in general. So, r I is a regular expression over the symbols sigma union d 1, d 2 up to d I minus 1, an example we will make it very clear. So, let us define identifiers, which are nothing but names used in programming languages. So, that is an integers. A letter let us say, we consider only five letter a b c d e, we can define it as a regular expression a plus b plus c plus d plus e, which generates the language a comma b comma c comma d comma e. Then digit in similarly with just the 0 1 2 3 4 can be defined as 0 plus 1 plus 2 plus 3 plus 4.So, these are too very simple regular expressions extended to become regular definitions headed by letter and digit.

So, what is an identifier? So, this is nothing but a name. So, all names must begin with letters followed by either letter or digit the star. So, combination of letters and digit following a letter and a number would be any number of digits so not epsilon of course. So, digit followed by any number of digits. So, this makes the 2 definitions identifier and number a little more understandable instead of writing this number as 0 plus 1 plus 2 plus 3 plus 4 followed by 0 plus 1 plus 2 plus 3 plus 4 star. So, that it would be very difficult to understand in a large definition without using shorthand of these regular definitions.

So, that becomes even more clear in the following example here say digit is 0 to 9 combined with a plus then digits would be digit followed by digit star an optional fraction is digits then plus epsilon. And optional exponent would be e followed by plus or minus or epsilon followed by digits and of course, epsilon indicates that there may not be any exponent as well. And unsigned number it is very clear now, there are digits there is optional fractions and optional exponent. So, the fraction and exponent may be absent there may be still have the integers and if you have digits followed by the fraction being present optional fraction being present you would have fix at point number and if you have the exponent then the exponent part will also be present. So, these are the various possibilities, but it is very clear that readability is enhanced by using these regular definitions.

(Refer Slide Time: 29:41)

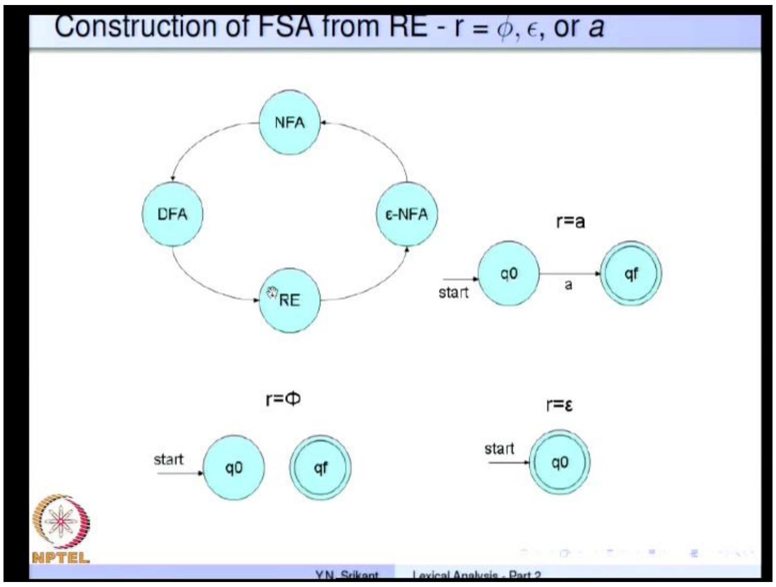


Equivalence of REs and FSA

- Let r be an RE. Then there exists an NFA with ϵ-transitions that accepts $L(r)$. The proof is by construction.
- If L is accepted by a DFA, then L is generated by an RE. The proof is tedious.

So, what about the equivalence of regular expressions and finite state of automata. So, here, we have a very fundamental theorem, the second one, let r be a regular expression then there exists a non deterministic finite state automaton with epsilon transitions that accepts a language L of r. So, this is a very profound a theorem and this is the basis of constructing NFA from specifications of regular definitions. So, the proof is by construction and we will consider the construction in detail. The converse is also to if L is accepted by a DFA then L is generated by a regular expression. This proof is steamily tedious and does not yield any insights into the complications process. So, we will skip that and if the readers are interested they can refer to the text book.

(Refer Slide Time: 30:48)

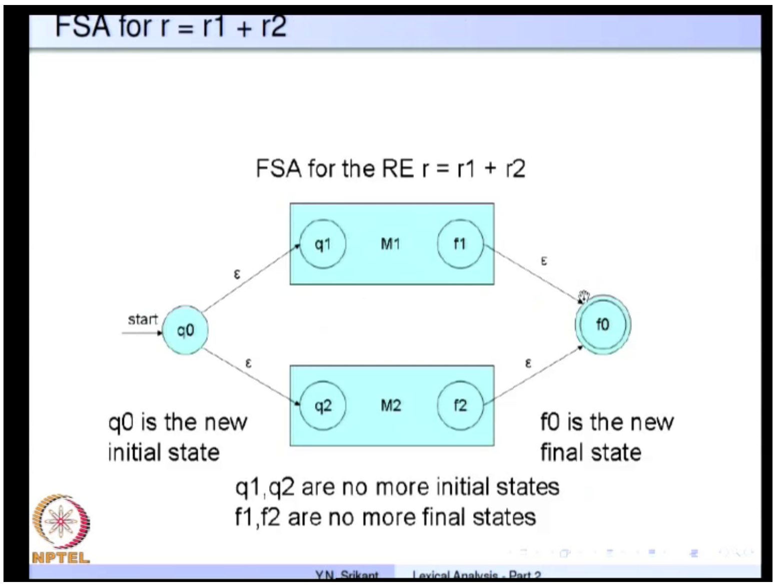


Construction of FSA from RE - r = ϕ, ϵ, or a

So, this is the way the theorem really works. So, let us see here, you have a DFA and you can actually generate a regular expression from it. Regular expressions can be converted to epsilon NFAs, epsilon NFAs can be converted into NFAs and NFAs can be converted to DFAs. So, in other words, if you are given a DFA you build the regular expression from it and then convert it to NFA and back to DFA you actually get the same DFA. So, this is a very powerful result. So, let us see how it works. So, the proof is also based on the inductive definition of regular expressions. So, for r equal to phi, this is finite state machine. So, if it accepts a null language, mind you there is nothing null set, the finite state cannot be reached. So, that the reason why we have shown a start state and a final state, but there is no arc between them so this particular finite state automaton accept the phi language.

What about the epsilon? The state q 0, if it does nothing accepts the strings epsilon. So, we have made it a final state that is the construction for the regular expression r equal to epsilon. What about the regular expression r equal to a, see the symbol a. So, there is a start state and there is a final state and there is a transition on a and q 0 actually get transform to q f. So, and the string a is accepted. So, this is the construction for the three basic cases phi, epsilon and a.

(Refer Slide Time: 32:42)



FSA for r = r1 + r2

q0 is the new initial state

f0 is the new final state

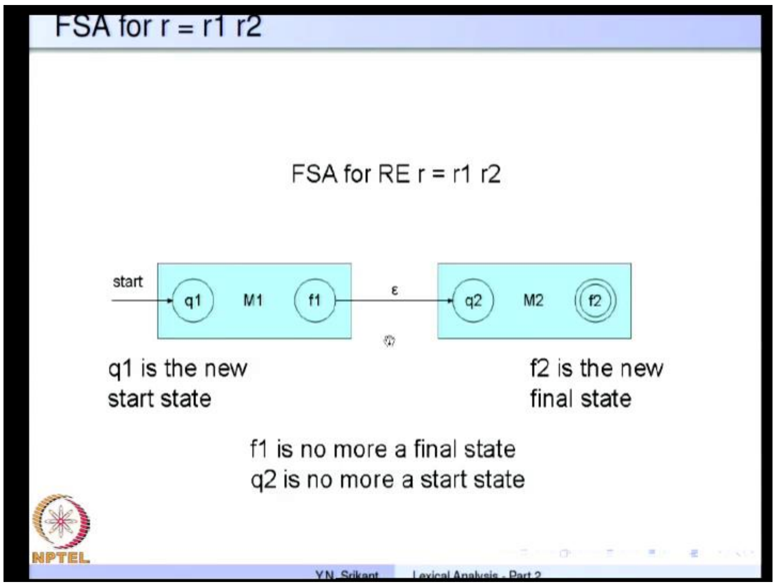
q1,q2 are no more initial states

f1,f2 are no more final states

Now, the next construction is for the operator plus, which produces the regular expression r 1 plus r 2 given the regular expressions r 1 and r 2. So, if you are given r 1 so mind you this is inductive so r 1 is smaller, r 2 is also small than you know smaller than r 1 plus r 2. So, these are the 2 components they really have less number of operators compare to the total of the operators in r 1 plus r 2. Therefore, inductively we can assume that there is a machine M 1 for accepting the language of r 1, similarly another machine M 2 for accepting the language r 2. So, this is the inductive hypothesis. Given these two so q 1 is the start state and f 1 is the final state of M 1, q 2 is the start state and f 2 is the final state of M 2. So, we take these 2 machines add a new start state and a final state.

So, from the new start state q 0 add an epsilon transition to q 1 add another epsilon transition to q 2. Make the 2 final states of f 1 and f 2 as of the 2 machines M 1 and M 2 as non final states. Mind you there are no double circles here and add transitions on epsilon from each of the states to the new final state f 0. So, q 0 is the new initial state, f 0 is a new final state and q 1 and q 2 are no more the initial states and f 1 and f 2 are no more the final states. So, how does this machine accept r 1 plus r 2? If the sentence is from generated from r 1 then quietly the NFA takes this path and accepts the string generated by r 1. Similarly for the string generated by r 2 it takes this path and accepts the strings generated by r 2. It is very clear that there is nothing else possible. So, this is the automaton which accepts the string generated by r 1 plus r 2.

(Refer Slide Time: 35:07)



FSA for r = r1 r2

FSA for RE r = r1 r2

q1 is the new start state

f2 is the new final state

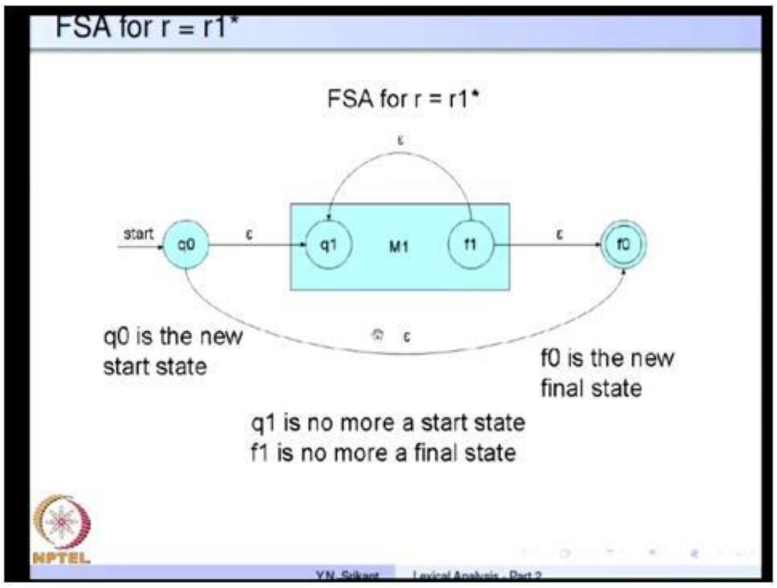
f1 is no more a final state
q2 is no more a start state

Y.N. Srikant   Lexical Analysis - Part 2

I give you an example of all this after the construction process is over. So, let us consider the concatenation operator r 1 r 2, the dot is traditionally not written. And we now, again have two small regular expressions r 1 and r 2 combined with a concatenation operation. So, inductively we can assume that M 1 recognizes the string generated by r 1 and M 2 recognizes the string generated by r 2.

So, the start state of M 1 is the new start state and the final state of M 2 is the new final state. The f 1 of M 1 is strip of its final state nature and similarly the start state of M 2 gets striped of its start state nature and between these 2 there is an epsilon transition. So, this you know now, the machine this combination accepts the strings generated by r 1 r 2 it is very simple, you start with the start state. Now, the r 1 part is taken care of by this and then we make a silent transition to the next machine the r 2 part is taken care of by this and then the machine halts. So, which part if r 1 which part is r 2 why this is a non deterministic automaton. So, it automatically can make transitions you know, which take care of acceptance of a strings corresponding to r 1 and r 2 appropriately.

(Refer Slide Time: 36:57)



FSA for r = r1*

FSA for r = r1*

q0 is the new start state

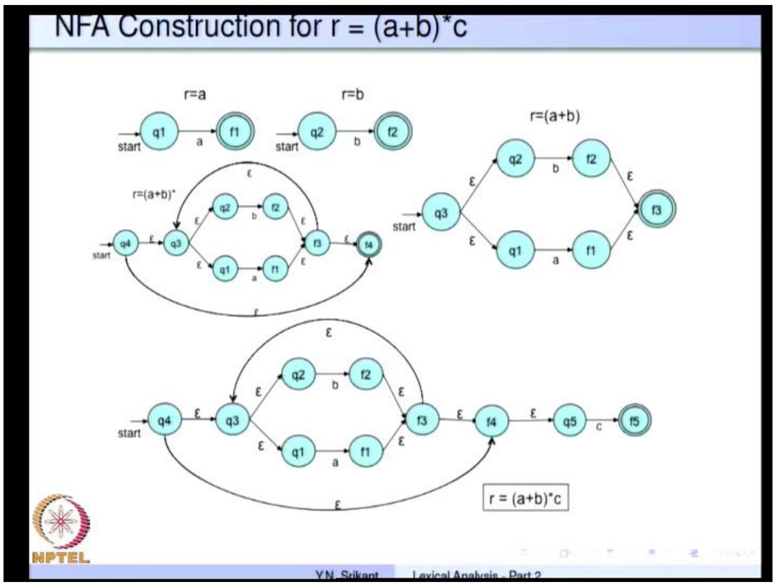
f0 is the new final state

q1 is no more a start state
f1 is no more a final state

We last construction case is that of r 1 star. So, this is the closure operator. So, again r 1 is smaller than r 1 star. So, we have a machine for r 1 by the inductive hypothesis. So, q 1 and f 1 are the start and final states of M 1. Now, f 1 is no more final please, observe that, we add a new start state and new final state as before, we connect q 0 to q 1 and f 1 to f 0 by epsilon transitions. So, far we did not increase the power. How do we actually make it generate r 1 star? So, that r 1 star implies epsilon also you know 0 number of times here. So, to take care of that we add an epsilon arc from q 0 to f 0, there by the new machine accepts epsilon.

Now, what about multiple instances of r 1, r 1 many times so it comes here this machine accepts r 1. Then if there is another instance, it is actually make to you know go to state q 1 by an epsilon transition consume the next instance of r 1 and so on and so for. So, we have added back arc and which is labeled as epsilon to take care of the iterations of r 1 and we have added an arc from q naught to q f naught to take care of the epsilon. So, this is the new machine which takes care of r 1 star. So now, it is time to look at a good example.

(Refer Slide Time: 38:39)

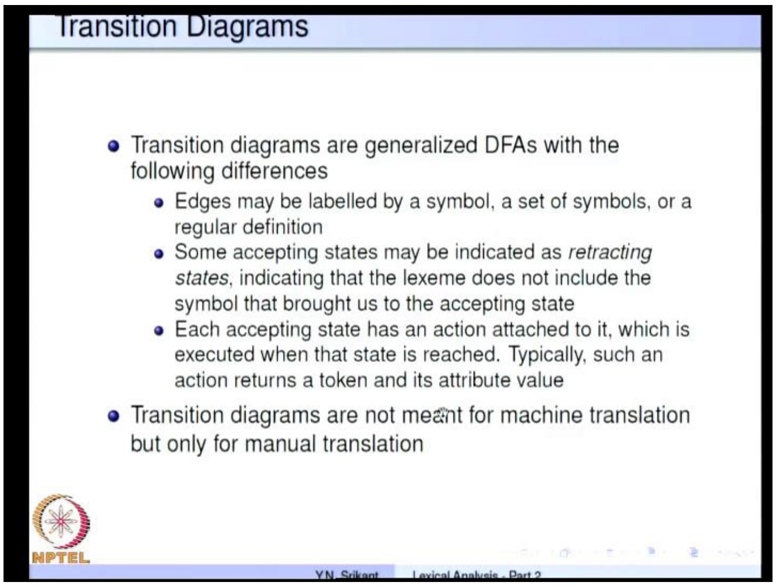


NFA Construction for r = (a+b)*c

So here, the regular expression is a plus b star dot c. So, it has all the operators that we have seen you know. So, the base classes correspond to a, b and c and then there is a plus and then there is a star and then the concatenation. So, r equal to a and r equal to b, I did not write the r equal to c part of it because it is very trivial. So, these are the two corresponding machines for a and b. Now, what about plus b? So, the construction says take the machine for a, take the machine for b attach q, you know q 3 and f 3, which are the new start and final state and add epsilon arc. So, this machine corresponds to a plus b as such. what about a plus b star. So, this is the machine corresponding to this part here to here you know q 3 to f 3 with no back arc is the machine corresponding to a plus b. So, we add a epsilon back arc and another epsilon that is from q 4 to f 4 and this is our machine a plus b whole star.

So, what about a plus b star into c or dot c? So, this is the machine for c, this entire things is the machine for a plus b star connects these two using epsilon make q 4 the new start state and f 5 the new final state. And this is our new machine corresponding to a plus b star c. So, you can see that there are many epsilon arcs here. So, the construction of DFA from NFA can take care of removal of epsilon arcs and you know the non determinism and finally, converted it into deterministic finite automata.

(Refer Slide Time: 40:47)



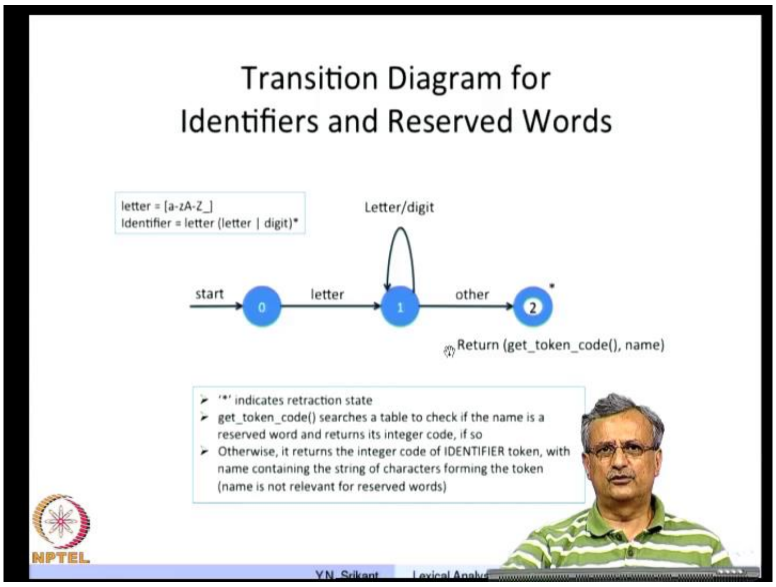
Transition Diagrams

- Transition diagrams are generalized DFAs with the following differences
  - Edges may be labelled by a symbol, a set of symbols, or a regular definition
  - Some accepting states may be indicated as *retracting states*, indicating that the lexeme does not include the symbol that brought us to the accepting state
  - Each accepting state has an action attached to it, which is executed when that state is reached. Typically, such an action returns a token and its attribute value
- Transition diagrams are not meant for machine translation but only for manual translation

Y.N. Srikant   Lexical Analysis - Part 2

So, now we switch back to the finite state automata and their generalizations called transition diagrams. So, let me give you an example before we go further.

(Refer Slide Time: 41:05)



Transition Diagram for Identifiers and Reserved Words

letter = [a-zA-Z_]
Identifier = letter (letter | digit)*

- '*' indicates retraction state
- get_token_code() searches a table to check if the name is a reserved word and returns its integer code, if so
- Otherwise, it returns the integer code of IDENTIFIER token, with name containing the string of characters forming the token (name is not relevant for reserved words)

Y.N. Srikant   Lexical Analy

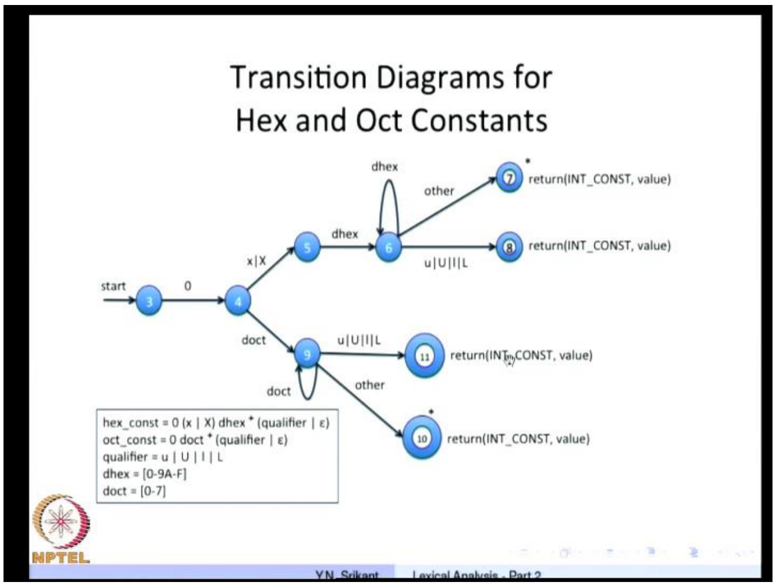
So, here is an example of a transition diagram. So, this is a transition diagram to recognize identifiers and reserved words. So, transition diagrams the most important part is of course, they have states as in machines, final state machines, but the labor is can be either symbols or they can be complete regular expressions. So, for example, letter is a regular expression, which consist of any of the letters of the alphabet a to z or a to z and this arc is written as letter r digit. So, letter r digit implies you know digit is similarly 0 to nine. So, this is a to z and a to z.

So, we have actually put down you know alternates the single symbols and also regular expressions as labels of you are allow to put them as labels of the arcs in a transition diagram. So, their generalized DFAs edges may be labeled by a symbol, a set of symbols or regular definition. Some accepting states may be indicated as what are known as retracting states, indicating that the lexeme does not include the symbol that brought us to the accepting state. This will be clear; when you go through the example, each accepting state has an action attached to it, which is executed when the state is really reached. So, typically you know this action is use to.

For example, evaluate string of bits or digits to produce its value or it can be use to such a table of reserved words and produce its code etcetera, etcetera. Transition diagrams are not meant for machine translation, but they are only for manual translations. So, let us look at some examples. So, here starting from the start state a single letter. So, all identifiers begin with letters, similarly all reserved words also begin with letters. So as for as, you know, if we do not look at the table of reserved words, we cannot distinguish between identifiers and reserved words. So, let us say on letter we go to state 1, on letter r digit we stay in state 1. And on any other symbol that is neither letter nor digit it, could be an operator for example it, I will go to another state called 2.

This is mark with a star indicating that the symbol which brought us from 1 to 2 will not be consumed and is not a part of the token that is produce. And there is a action here get token, retune get token, code comma name. So, this searches a table finds out whether it is a reserved word or an ordinary name identifier gets the corresponding code for it and comes out. So, this is the action part.

(Refer Slide Time: 44:36)



Transition Diagrams for Hex and Oct Constants

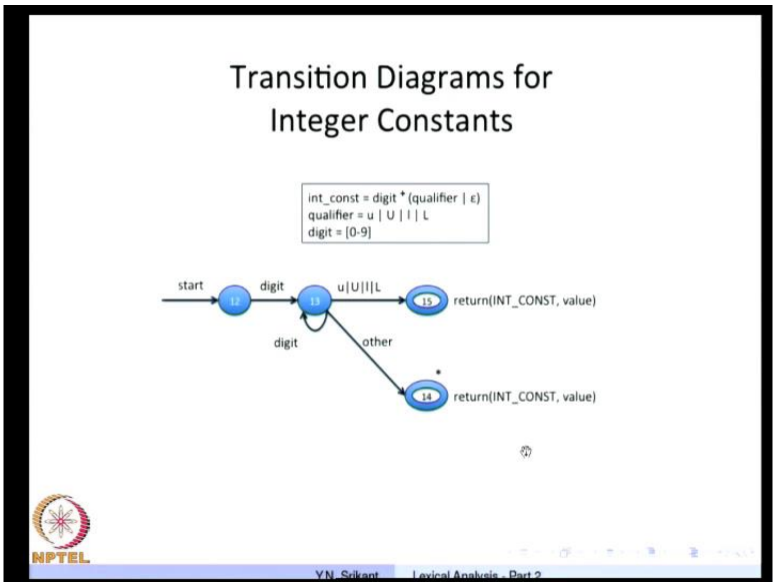
```
hex_const = 0 (x | X) dhex + (qualifier | ε)
oct_const = 0 doct + (qualifier | ε)
qualifier = u | U | l | L
dhex = [0-9A-F]
doct = [0-7]
```

So, let us check another example, how about hex constants and octal constants, hexadecimal and octal constants. So, the syntax is very similar to that of c. So, hexa constant is 0 followed by either a small x or a capital x followed by hexadecimal digits that is 0 to nine and a to a b c d e f all these any number of times.

An hexadecimal digit any number of times followed by you know either a qualifier or nothing. A qualifier simply says it is unsigned or long u or capital u, small l or long l, big l. Similarly octal constant starts with a 0, it has the octal digits from 0 to seven any number of times followed by either qualifier or nothing. So, this is our definition of these are the regular definitions for hexa constant and octa constant. So, let us see, you know how it works in a transition diagram. So, starting from this state 3 on a 0, we go to state 4 now, we branch. It is a hexa constant so there is an x or it is a digit they an octal digit 0 to 7, then we go to straight 9, so either 5 or 9.

So, if it is a an octa constant in straight 9, we consume all the octal digits and on getting a qualifier we go to straight eleven or on getting some other operator or symbol like that we go to straight ten. So, this is a retracting state; that means, this symbol brought us from nine to ten is not a part of the token. So, in both these cases ten and eleven, we return, you know token int const and value as well. So, the string that we have accumulated the lexeme, we have accumulated in entering either state ten or state eleven is actually the string of digits in the constant. So, that part would be taken then evaluated and value is returned as the value of the integer constant. So, I whether it is a hex constant or octal constant it is still an integer and that is the reason why return the token int const.

(Refer Slide Time: 47:06)

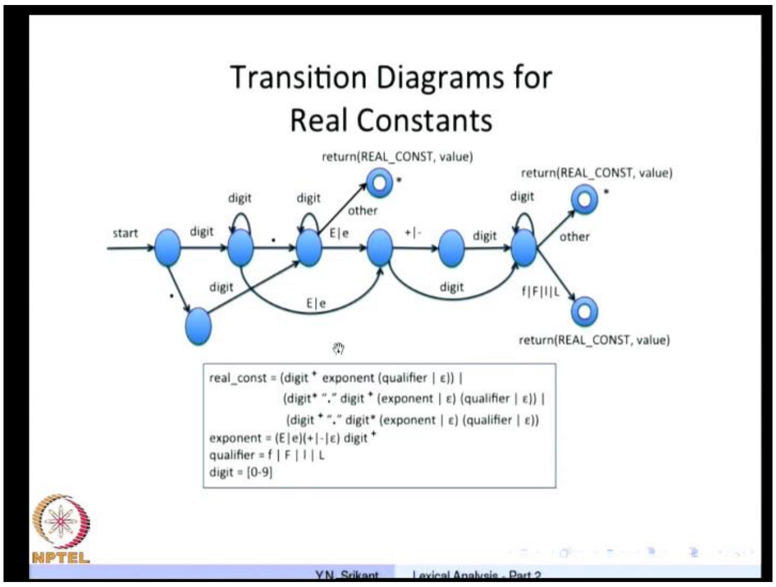


Transition Diagrams for Integer Constants

int_const = digit $^+$ (qualifier | ε)
qualifier = u | U | l | L
digit = [0-9]

How about normal integers? So, normal integer constant is any number of digits followed by qualifier. So, digit is either 0 1, 2, 3, etcetera, 9. And qualifier as usual is unsigned or long, so this very simple start. On a digit go to we since, we have you know digits here. So, we must see that consume a digit and run any number of digits here. So, we really have to say this should have been digit any number of digit plus sorry, this is not digit start, this is digit plus. So, any number of digits here and then either unsigned or other character. In both cases we written integer constant and its value. So, this is a very simple transition diagram for integers.

(Refer Slide Time: 48:02)



Transition Diagrams for Real Constants

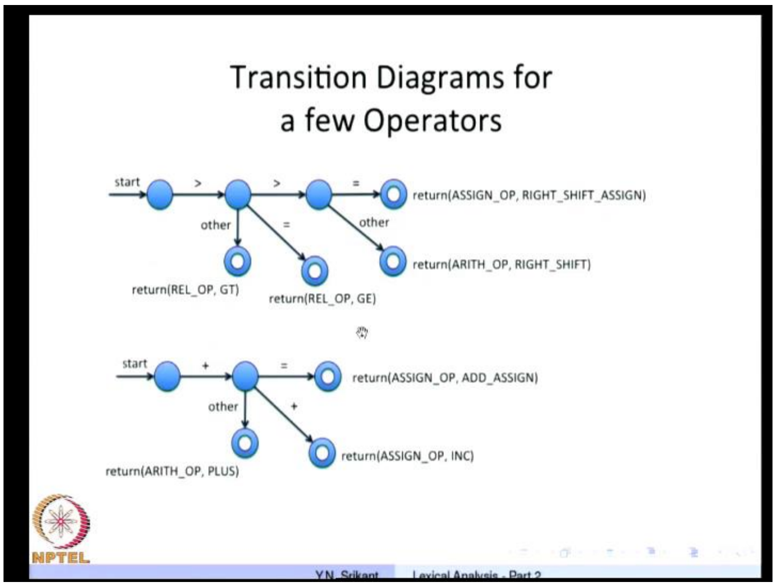
return(REAL_CONST, value)
*
return(REAL_CONST, value)
*
start
digit
digit
digit
.
other
E|e
+|-
digit
digit
other
.
digit
E|e
digit
f|F|l|L
return(REAL_CONST, value)

```
real_const = (digit + exponent (qualifier | ε)) |
               (digit* "." digit + (exponent | ε) (qualifier | ε)) |
               (digit + "." digit* (exponent | ε) (qualifier | ε))
exponent = (E|e)(+|-|ε) digit +
qualifier = f | F | l | L
digit = [0-9]
```

Now, something more complicated. So, this is for real constants. So, star you know really constant can be of any of these three types. So, they could have any number of digits, digit plus followed by an exponent and then followed by a qualifier or nothing or there could be any number of digits followed by a dot, followed by a digit, followed by a exponent or qualifier and or qualifier. Finally, third type you know here, in this case you could have possibly you know no digit prefer the dot, but once you have dot you must have at least 1 digit. And n you have any number of digits followed by dot, followed by any number of digits, followed by exponent or you know qualifier.

So, to permit to optional fractions you know some jugglery of this kind would be necessary. And exponent has either big e or small e followed by the sign of the exponent and then the digits. So, qualifier is either small f capital f l or big l. So, this transition diagram again similar on a digit it goes to this state and remains in it to consuming all the digits on a dot it goes to this state consumes all the digits again and any other symbol goes to this state where we written real constant with value. If there is an exponent it continuous all the way consumes the exponent digits and final returns the aerial constant value. And if we have we do not have any integer part when, we come here and again follow this particular path. So, you can go through this with a couple of examples to convince you that it catches all the real constants it is a slightly more complicated example then before.

(Refer Slide Time: 50:10)



Transition Diagrams for a few Operators

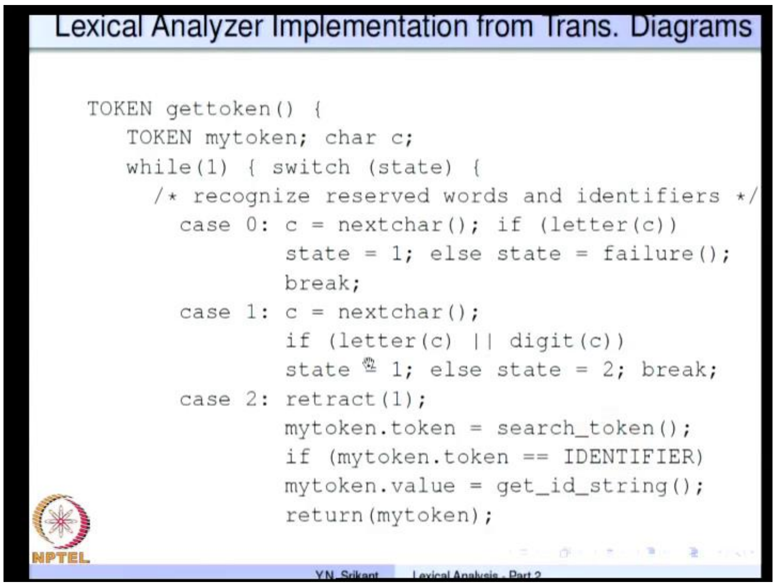
$$\begin{tikzcd}
\arrow[r, "start"] & \bigcirc \arrow[r, ">"] & \bigcirc \arrow[r, ">"] \arrow[d, "other"] \arrow[dr, "="] & \bigcirc \arrow[r, "="] \arrow[dr, "other"] & \circledcirc \; \text{return(ASSIGN\_OP, RIGHT\_SHIFT\_ASSIGN)} \\
& & \circledcirc \; \text{return(REL\_OP, GT)} & \circledcirc \; \text{return(REL\_OP, GE)} & \circledcirc \; \text{return(ARITH\_OP, RIGHT\_SHIFT)}
\end{tikzcd}$$

$$\begin{tikzcd}
\arrow[r, "start"] & \bigcirc \arrow[r, "+"] & \bigcirc \arrow[r, "="] \arrow[d, "other"] \arrow[dr, "+"] & \circledcirc \; \text{return(ASSIGN\_OP, ADD\_ASSIGN)} \\
& & \circledcirc \; \text{return(ARITH\_OP, PLUS)} & \circledcirc \; \text{return(ASSIGN\_OP, INC)}
\end{tikzcd}$$

Then so far, we have seen constants of various kind right, integer constant, hexadecimal constants, octal constants, we have also seen floating point constants, now an identifiers right. So, now, let us see how operators are taken care of in transition diagrams. So, we have many operators, this is greater than and then you know, if it is not a greater than symbol or you know, if it is a just an ordinary greater than, then we must return rel op greater than, if it is a greater than followed by another greater than and then followed by some other symbol then we have a arithmetic shift right, right shift.

And if you have a greater than followed by equal to then we written greater than or equal to. We have greater than and then equal to assign this is actually a right shift assign type of operator. So, this is one of the transition diagrams, here is one more. So, we have a plus followed by equal to which is the add assign operator. It is just plus then we written a op plus and the you know see here either arith op or assign op or you know these are the two types of operators and accordingly there is a code within that called add assign or right shift or shift assign etcetera, etcetera, etcetera. So, the various the arithmetic operators are all handled here. Similarly, we would have you know op, you know the logical operators being handled and so on and so forth. So, it would be repetitive thing without giving extra benefit for the student.

(Refer Slide Time: 52:12)



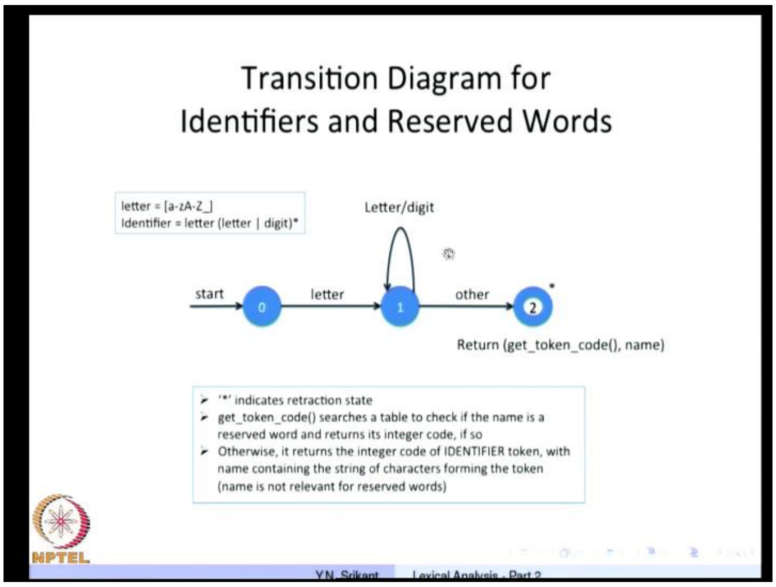
Lexical Analyzer Implementation from Trans. Diagrams

```
TOKEN gettoken() {
   TOKEN mytoken; char c;
   while(1) { switch (state) {
     /* recognize reserved words and identifiers */
       case 0: c = nextchar(); if (letter(c))
               state = 1; else state = failure();
               break;
       case 1: c = nextchar();
               if (letter(c) || digit(c))
               state = 1; else state = 2; break;
       case 2: retract(1);
               mytoken.token = search_token();
               if (mytoken.token == IDENTIFIER)
               mytoken.value = get_id_string();
               return(mytoken);
```

Y.N. Srikant   Lexical Analysis - Part 2

So, now what about generation of a lexical analyzer from transition diagrams. as I already mentioned transitions diagrams are useful only for writing you know, lexical analyzers by hand. So, let us see how the transitions diagrams are implemented in code,

(Refer Slide Time: 52:40)



Transition Diagram for Identifiers and Reserved Words

letter = [a-zA-Z_]
Identifier = letter (letter | digit)*

- '*' indicates retraction state
- get_token_code() searches a table to check if the name is a reserved word and returns its integer code, if so
- Otherwise, it returns the integer code of IDENTIFIER token, with name containing the string of characters forming the token (name is not relevant for reserved words)

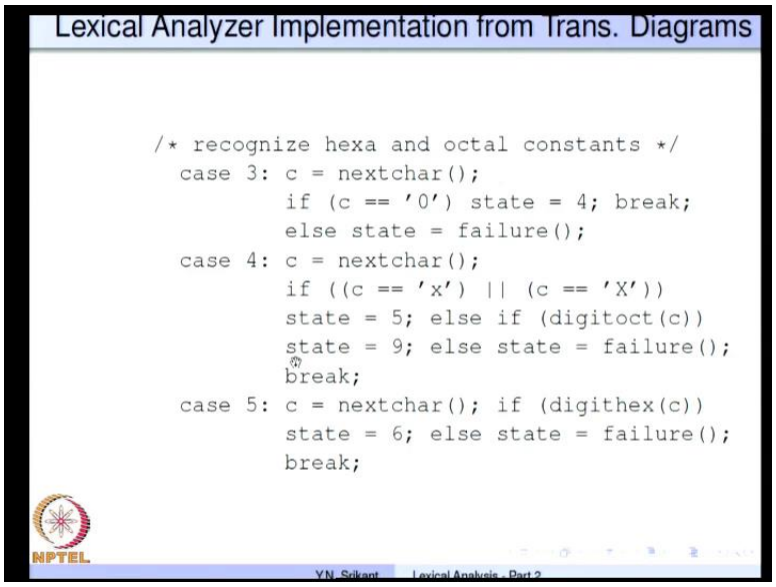
Y.N. Srikant   Lexical Analysis - Part 2

So, the first example would be this. So, we have 0, 1, 2, 3 states. From 0 to 1, we go on letter and in 1, we lean on letter or digit and on any other symbol. We go to 2 and which returns a token. So, let us follow that. So, this is a function get token which returns a token and token as 2 parts, it can contain the token value, token and value ok.

So, 2 parts in the token itself and there is a char c. So, here this is a while loop which keeps going forever until we exhaust the input. So, switch state. So, read the next character this is state 0 if it is a letter go to state 1 otherwise state equal to failure. So, it goes back to the start state and then there is a break. If it is case 1, if it is a letter or digit it remains in state 1, otherwise it goes to state 2 and breaks. Case 2, this is a retraction as I said this is really a retraction. So, in state 2 we return a token and we do not consume the symbol that brought us to the state. So, my token dot token we search the token array we know rather the words array find out what type of token it is either a name or reserved word.

So, if my token dot token equal to identifier then the value is the string corresponding to that identifier, otherwise we no need the string and we return the token. So, then the hexa and octal constants are very similar. So, for example, here let us trace the octal part on 0 we come to 4 and then the doct, we come to 9. And then on the qualifiers we go to eleven or any other symbol, we go to ten if we keep getting octal digits we still remain in state nine.

(Refer Slide Time: 54:45)



Lexical Analyzer Implementation from Trans. Diagrams

```
/* recognize hexa and octal constants */
  case 3: c = nextchar();
          if (c == '0') state = 4; break;
          else state = failure();
  case 4: c = nextchar();
          if ((c == 'x') || (c == 'X'))
          state = 5; else if (digitoct(c))
          state = 9; else state = failure();
          break;
  case 5: c = nextchar(); if (digithex(c))
          state = 6; else state = failure();
          break;
```

Y.N. Srikant   Lexical Analysis - Part 2

So, here in state three we read a character it must be a zero. So, otherwise it is not a hexa or octal constants. So, we go to state four, otherwise it is a illegal character. In state four, it is corresponds to you know either x or x; that means, it is a hexadecimal constant, if it is not x or x then is it a digital, you know the octal digit yes we go to state nine, otherwise it is a failure you know, it is not a correct type of constant.

So, let us look at state nine. So, state nine says if I know symbol we get is a octal digit then we remain in the same state and if we get a qualify u or u or l or l we go to state number eleven, otherwise we go to state ten and stop. So, this is our state ten here. So, state ten is here, this is corresponds to the other symbol part. This corresponds to the octal numbers part and in case ten we retract and this is the fall through to case eleven, which assigns the token as int const. And the value as after evaluating the octal number whatever value we get and returns that particular token. So, we will stop here and continue the lexical analyzer discussion in the next lecture.

Thank you.