

Principles of Compiler Design
Prof. Y. N. Srikant
Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 2
Lexical Analysis - Part 1

(Refer Slide Time: 00:18)

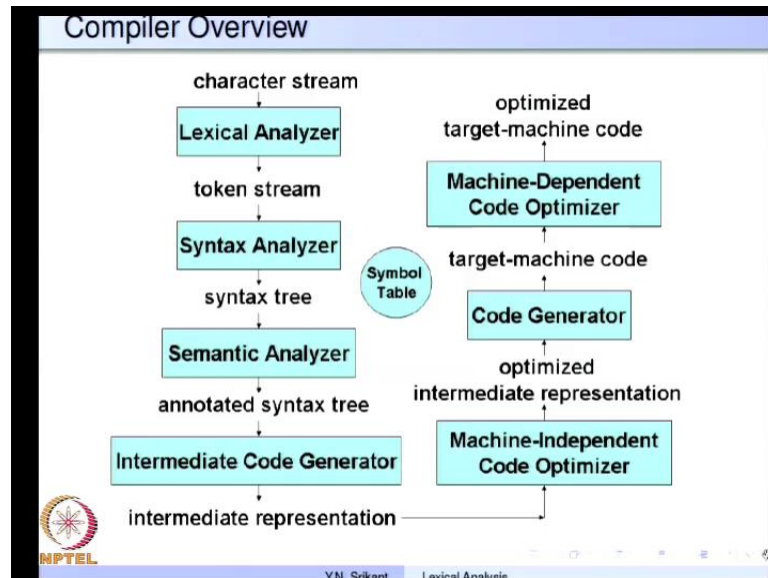
Outline of the Lecture

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

MPTEL
Y.N. Srikant Lexical Analysis

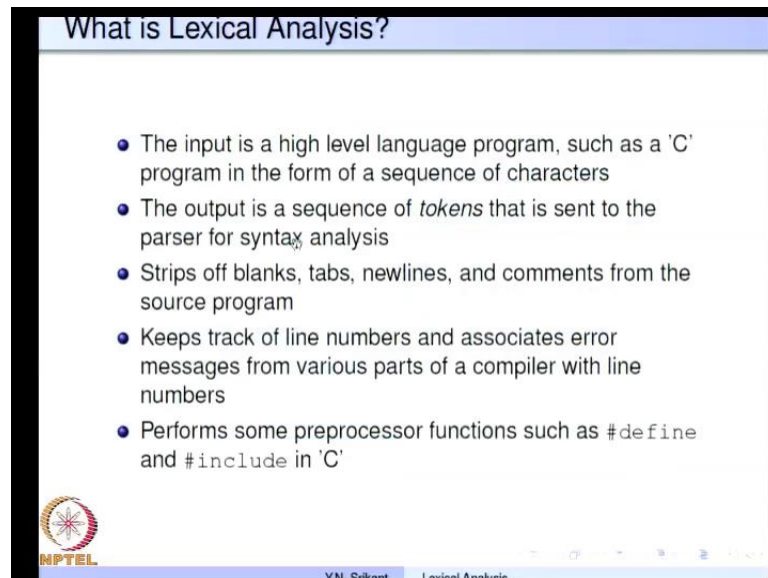
Welcome to the lecture on lexical analysis. In this lecture we will discuss lexical analysis in detail what is lexical analysis? Why should lexical analysis be separated from syntax analysis? What are tokens patterns and lexemes, what are the difficulties in lexical analysis recognition of tokens? So, for this we require fundamentals of finite state automata and transition diagrams. And then for specification of tokens we require regular expressions and regular definitions. And we will study all this and then finally we will take a look at a very effective tool called LEX for lexical analyzer generation.

(Refer Slide Time: 01:09)



To do a bit of recap this is the block diagram of a compiler. So, the lexical analyzer is the first component in a compiler it takes a character stream as input, outputs a token stream which goes into a syntax analyzer. So, this is where the lexical analysis ((Refer Time: 01:34)) I know actually is performed.

(Refer Slide Time: 01:39)



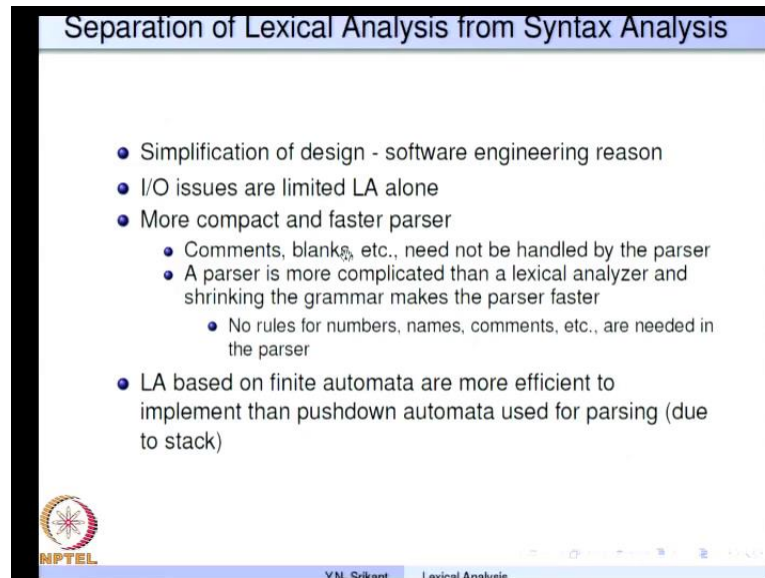
So, let us go to the details. So, what exactly is lexical analysis? The input to a lexical analyzer is a high level language program. So, may be its written in C, it is written in C plus plus or java or any other language. But the common feature of all these is that they

are sequences of characters, there is no other distinction between these programs. They are all sequences of characters, and the output obtained from a lexical analyzer is a sequence of tokens. So, the tokens go into the parser for syntax analysis. The lexical analyzer does a lot of cleaning on the input for example, it strips off the blanks, tabs, new line characters and comments from the source program. Because these are not very important for parsing you know once the token stream is formed these are not at all important.

So, these are all removed by the lexical analyzer and then the tokens are formed. The lexical analyzer keeps track of numbers, the line numbers and associates error messages with the various lines of a source code. The error messages might have actually you know a reason because of syntax analysis or semantic analysis or even lexical analysis, but these errors are all kept track of by the lexical analyzer. And then you know associated with various numbers, source line numbers of the program the lexical analyzer also performs 2 processor functions for example, hash define and hash include in.

So, hash define defines a macro and hash include includes a file. So, hash define whatever back cover is define you know the effect of that macro is nothing but replacing a particular name with a sequence of characters. So, the lexical analyzer actually performs this expansion wherever that appears it expands that name with the sequence of characters mentioned in the hash define macro. And then submits that expanded sequence to the rest of the lexical analyzer hash include simply says now take this particular file it is also a part of the program. So, perform compilation on it. So, hash include simply means start reading from a different file. And then do lexical analysis and parsing etcetera on the rest of the file which is mentioned in the hash include.

(Refer Slide Time: 04:40)



The slide is titled "Separation of Lexical Analysis from Syntax Analysis" and contains the following bulleted list:

- Simplification of design - software engineering reason
- I/O issues are limited LA alone
- More compact and faster parser
 - Comments, blank, etc., need not be handled by the parser
 - A parser is more complicated than a lexical analyzer and shrinking the grammar makes the parser faster
 - No rules for numbers, names, comments, etc., are needed in the parser
- LA based on finite automata are more efficient to implement than pushdown automata used for parsing (due to stack)

The slide also features the NPTEL logo in the bottom left corner and a footer with the text "Y.N. Srikant Lexical Analysis".

So, separation of lexical analysis from syntax analysis, I already mentioned this briefly in the last lecture just to do a recap the first reason is a software engineering reason. It simplifies design a compiler is a very, very large piece of software millions of lines of code. So, making it modular is essential and making the lexical analysis as a separate module helps in, you know reducing the complexity of building a compiler. As I already mentioned the I O issues are limited to lexical analysis alone. The errors and so on reading from different files because of hash include etcetera. And it also makes lexical analysis you know if it is separate it is actually more compact and faster.

You know the parser becomes more compact more you know fast apart from the lexical analyzer itself being very fast, why? The reason is lexical analysis is based on finite state automata. These are much easier to implement in the form of tables rather than implement you know the functions of the lexical analyzer in a push down automata which uses a stack this will become clear as we go on in the course and study parsing as well. So, the comments blanks you know need not be handle by the parsers. So, why not remove them in the lexical analyzer itself? So, that makes the work of parser a little less a parser is; obviously, more complicated And therefore, keeping track of you know number of lines of code names comments etcetera you know it is absolutely unnecessary for the parser. So, the lexical analyzer is a better place to take care of these and this makes both the lexical analyzer and the parser more efficient.

(Refer Slide Time: 06:49)

The slide is titled "Tokens, Patterns, and Lexemes" and contains the following content:

- Running example: `float abs_zero_Kelvin = -273;`
- Token (also called *word*)
 - A string of characters which logically belong together
 - **float, identifier, equal, minus, intnum, semicolon**
 - Tokens are treated as terminal symbols of the grammar specifying the source language
- Pattern
 - The set of strings for which the *same* token is produced
 - The pattern is said to *match* each string in the set
 - `float, l(l+d+)*, =, -, d+, ;`
- Lexeme
 - The sequence of characters matched by a pattern form the corresponding token
 - "float", "abs_zero_Kelvin", "=", "-", "273", ";"

In the bottom right corner of the slide, there is a portrait of a man with glasses and a white shirt. The bottom left corner features the NPTEL logo. The bottom center has the text "Y.N. Srikant Lexical Analysis".

Now, let me define tokens, patterns and lexemes and then go on to the operation of lexical analyzer itself. So, let us take a running example, it is a programming language statement similar to C `float absolute 0 Kelvin equal to minus 273` followed by a semicolon here as we know a float is a reserve word `abs 0 Kelvin` is a name. It is a variable and it could also be seen as a constant it depends on the type of usage that we want for it and `minus 273` is an integer constant. So, now, on this particular running example we are going to show what are tokens, what are patterns and what are lexemes? A string of characters which logically belong together is a token for example, the word `float` and the word `abs 0 Kelvin` are separated by a blank and they are; obviously, two different strings of characters.

So, we can very safely say `float` is a you know token then `abs 0 Kelvin` is another token the `equal to assignment` is one token the `minus sign` is another token the `number two hundred and seventy three` is one more and then the `semicolon` is a last token in this particular sentence. So, once the tokens are identified you know these are actually passed on to the syntax analyzer. So, the tokens are treated as what are known as terminal symbols of the grammar specifying the source language this will become clear as we go on. So, that makes the life of parser a little easier it need not worry about the characters making up the token `float` it can it needs to worry only about a some kind of a number called `float` that is it. So, internally tokens are going to be represented in the form of

integers and that makes a token stream very efficient. Then what is a pattern? the set of strings for which the same token is produced is called as a pattern.

So, we are going to define what are known as regular expressions to define these patterns later, but for the present, let us understand what exactly are patterns. So, in this case the pattern is said to match each string in the set of a strings that it supposed to match. So, for example, in for the running example the word float is a pattern on its own. Because no other string actually matches this particular pattern, but for the identifier or the name we have a general pattern which says letter. So, we have as pattern here l is letter and when d is digit and then we have underscore the plus actually is a form of the notation it the way to read then star is for iteration.

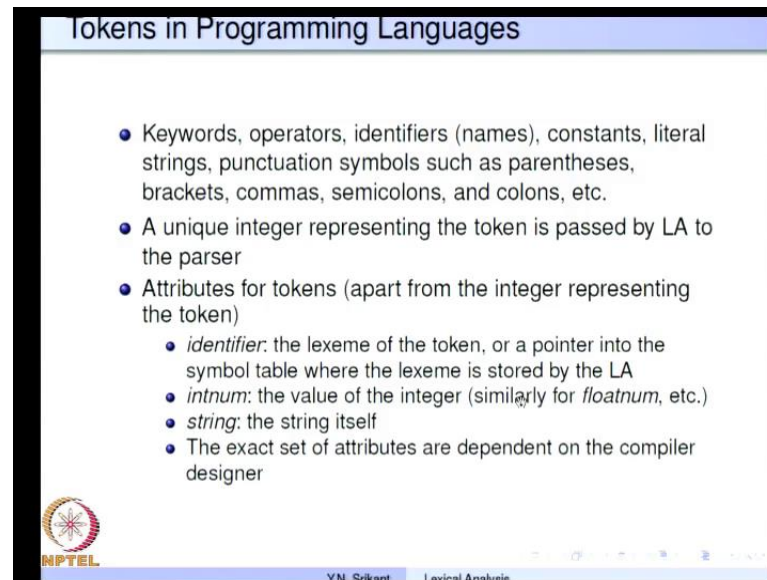
So, let me explain what this pattern is it says letter followed by either letter or digit or underscore any number of times 0 inclusive. So, we can produce for example, the way abs 0 abs underscore 0 underscore Kelvin is produced we have a letter then followed by two more later. So, in this case letter or digit or underscore any number of times is exercised to produce two more letters. And then it is exercised to produce one underscore then four letters another underscore and then five more letters. So, this is the sequence which is actually exercised to produce that particular name. So, that is a pattern for the identifiers or names then equal to star this equal to and minus sorry equal to and minus match are patterns which match themselves and nothing else. And finally, for the integer number constant d plus d is a digit.

So, any number of digits together, but plus says at least 1's you know the digit is used at least 1's. So, the number cannot have 0 number of digits it should have at least one digit. So, using this pattern 3 times d d d will give us 273 and finally, of course, a semicolon is a pattern which matches itself. So, these are the patterns. So, the central theme of specifying lexical analysis would be to come up with a formalism to specify these patterns which cover the entire programming language the set of programming language constructs. So, we are going to define regular expressions which can DO this job very admirably.

Finally, what is lexeme? A lexeme is the sequence of character matched by a pattern to form the corresponding token. So, another words, this is a pattern and that is actually the name identifier. So, the character which form this particularly identifier actually are the,

are sum of lexeme. So, the string float, the string abs underscore 0 underscore Kelvin the string equal to the string minus. And the string 273 are the sequence of characters and they are called the lexeme corresponding to the tokens.

(Refer Slide Time: 13:12)



The slide is titled "Tokens in Programming Languages" and contains the following content:

- Keywords, operators, identifiers (names), constants, literal strings, punctuation symbols such as parentheses, brackets, commas, semicolons, and colons, etc.
- A unique integer representing the token is passed by LA to the parser
- Attributes for tokens (apart from the integer representing the token)
 - *identifier*: the lexeme of the token, or a pointer into the symbol table where the lexeme is stored by the LA
 - *intnum*: the value of the integer (similarly for *floatnum*, etc.)
 - *string*: the string itself
 - The exact set of attributes are dependent on the compiler designer

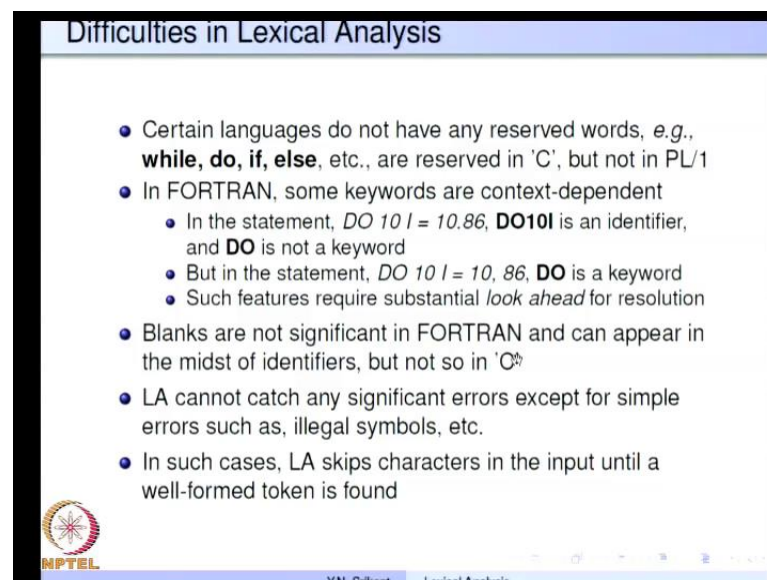
At the bottom left of the slide is the NPTEL logo. At the bottom center, it says "YN. Srikant" and "Lexical Analysis".

So, we have talked about tokens very briefly. So, let us discuss it little more to understand how tokens are actually used for specifying various the parts of a programming language. Keywords, operators, identifiers so we will always say identifiers instead of name then various type of constants, integers constants, floating point constants. Then literal strings these are string constants, punctuation symbols such as parentheses bracket commas semicolon and colons etcetera and many more. So, these are the various types of tokens that are possible in a programming language. And we need to specify patterns for each one of these tokens a unique integer representing the tokens is passed by the lexical analysis to the parser. So, as I told you each token is given a name by the designer.

So, representing; it is very efficient then tokens you know if the number corresponding to a token does not say everything about tokens. You also need extra attributes extra value for these tokens. So, let see what is values are required to specify a tokens completely for names or identifiers. The lexeme of token are the string corresponding to that token or a pointer to into the symbol table where the lexeme is stored by the lexical analyzer that is in summary we require the string corresponding to the name. So, that is also to be

accessed in the, you know while doing parsing or semantic analysis also code generation etcetera. So, that is one of the attributes that we want for that identifier. And then for integer numbers we want the value of the number similarly for floating point numbers float num we want the value of the floating point number in the appropriate you know representation and so on. For strings we need the string itself and the exact set of attributes are dependent on the compiler designer. So, it is possible for example, here it is possible that the token contains the string corresponding to the name itself. In the case of identifier some other designer may say no let me store in the table and provide a pointer to the symbol table.

(Refer Slide Time: 16:02)



Difficulties in Lexical Analysis

- Certain languages do not have any reserved words, *e.g.*, **while, do, if, else**, etc., are reserved in 'C', but not in PL/1
- In FORTRAN, some keywords are context-dependent
 - In the statement, *DO 10 I = 10.86*, **DO10I** is an identifier, and **DO** is not a keyword
 - But in the statement, *DO 10 I = 10, 86*, **DO** is a keyword
 - Such features require substantial *look ahead* for resolution
- Blanks are not significant in FORTRAN and can appear in the midst of identifiers, but not so in 'C'
- LA cannot catch any significant errors except for simple errors such as, illegal symbols, etc.
- In such cases, LA skips characters in the input until a well-formed token is found

MPTel
Y.N. Srikant Lexical Analysis

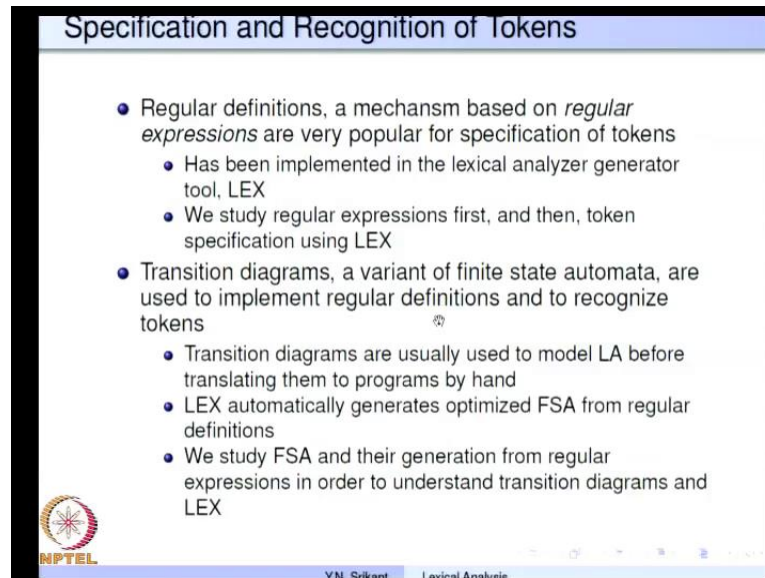
So, that is a description of, what kind of tokens arise in programming languages. And now, let us also discuss the difficulties in lexical analysis. For example, certain languages do not have any reserved words. So, in fact, while, do, if, else etcetera they are reserved in C. But they are not reserved words in programming language P L 1 and in FORTRAN some keywords are context dependent. So, let us take an example take this DO 10 I equals to 10.86 DO 10 I is a identifier and DO is not a key word even though this is supposed to you know its looks like in DO loop in Fortran. Because this is DO 10 I equals to 10.86 now DO 10 I is taken as identifier a name and DO is not separated as keyword.

But if the statement were to be DO 10 I equals to 10 comma 86 then definitely this is a DO loop and DO is a keyword. In fact, the tokens for this particular statement would be DO a reserved word 10 a label name and I another identifier equal to and the integer value 10 a comma and another integer value 86. So, these are going to be various tokens for this particular statement whereas the previous statement we have just one identifier DO 10 I then equal to and then floating point constant 10.86. So, the sequence of tokens for a statement would be very different depending on the fact whether it is a you know DO statement or a different type of assignment statements and etcetera handling such features requires what is known as a look ahead.

So, here until you know when we see DO 10 I it is not possible to determine that it is a variable or the DO statement. In fact, we need to go and parse this 10 find whether it is a comma or a dot if it is a comma then it is a DO statement, but if it is a dot then it is an assignment statement. And only after reaching this comma or dot can we really determine the token sequence even for the previous string of strings you know that previous string. So, whether it would be one variable followed by equal to or it would have to reserved word followed by a label name and then a int variable. So, and then equal to.

So, the token strings are going to be very different and the token strings can be determined only after looking at the dot and or a comma which actually the string of characters 1 0 in this case. Then blanks are not significant FORTRAN and can appear in the middle of identifier, but in C C plus plus and Pascal etcetera it is not. So, blanks actually separate various tokens lexical analyzer cannot catch any significant errors except for very simple 1's like illegal symbols etcetera and rest of the errors are called by the parsers. So, if an error occurs there is very little that the lexical analyzer can do apart from skipping characters in the input until a well formed token is found. It just keeps skipping characters skip one character and try to find a token skip another character try to find token etcetera and till it succeeds and really finds a meaningful token.

(Refer Slide Time: 19:55)



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

- Regular definitions, a mechanism based on *regular expressions* are very popular for specification of tokens
 - Has been implemented in the lexical analyzer generator tool, LEX
 - We study regular expressions first, and then, token specification using LEX
- Transition diagrams, a variant of finite state automata, are used to implement regular definitions and to recognize tokens
 - Transition diagrams are usually used to model LA before translating them to programs by hand
 - LEX automatically generates optimized FSA from regular definitions
 - We study FSA and their generation from regular expressions in order to understand transition diagrams and LEX

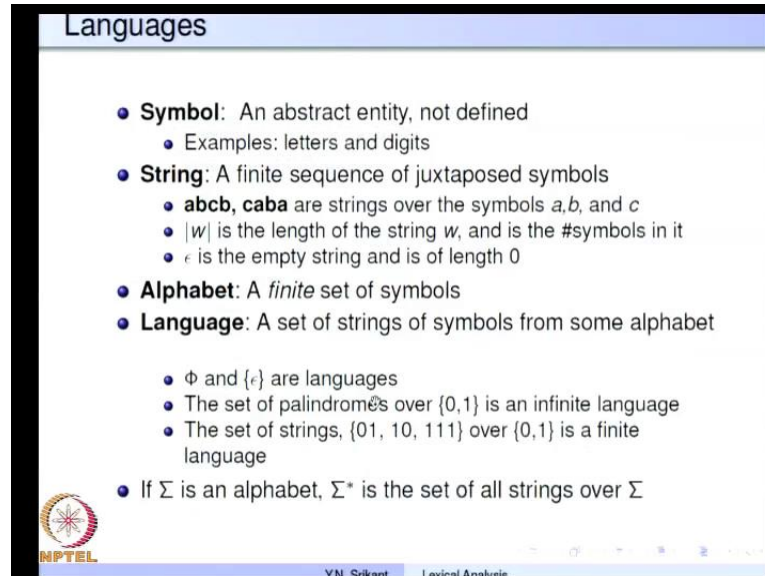
At the bottom left of the slide is the NPTEL logo. At the bottom center, the text "Y.N. Srikant Lexical Analysis" is visible.

Now, specification and recognition of tokens. So, regular definitions a mechanism based on regular expressions are very popular for specification of tokens. So, we will look at the details of regular definitions and regular expressions in the following lectures, the regular definitions have been implemented in a tool called LEX. So, if you write regular expressions then you know automatically the tool LEX produces lexical analyzer. So, we will discuss regular expressions and then token of specification using LEX regular definitions and so on. We also use transition diagrams which are nothing but variants of finite state automata. So, they are used to implement regular definitions and to recognize tokens. Transition diagrams are used to usually used to model the lexical analyzer before translating them to programs by hand.

By the way it is possible to write lexical analyzers by hand as well. In the olden days that is exactly what was really done even today for very small languages for efficiency sake it is a compiler designer sometime write lexical analyzers and parsers by hand. So, it is not as if it is very artificial to think of such a situation. So, when we design regular you know lexical analyzers to be implemented by hand we use transition diagrams to model them. And then translate these transition diagrams by hand to programs where as LEX automatically generates optimized finite state automata from regular definitions it does not require transition diagrams as a method of specification. So, we will first study finite state automata and their generation from regular expressions in order to understand transition diagrams and the working of LEX itself. So, now so for we have look that

tokens we know what tokens are and but we still do not know how to specify tokens. And once we learn how to specify tokens we will see how to translate them to programs.

(Refer Slide Time: 22:29)



The slide is titled "Languages" and contains the following definitions:

- **Symbol:** An abstract entity, not defined
 - Examples: letters and digits
- **String:** A finite sequence of juxtaposed symbols
 - **abcba, caba** are strings over the symbols *a, b*, and *c*
 - $|w|$ is the length of the string *w*, and is the #symbols in it
 - ϵ is the empty string and is of length 0
- **Alphabet:** A *finite* set of symbols
- **Language:** A set of strings of symbols from some alphabet
 - Φ and $\{\epsilon\}$ are languages
 - The set of palindromes over $\{0,1\}$ is an infinite language
 - The set of strings, $\{01, 10, 111\}$ over $\{0,1\}$ is a finite language
- If Σ is an alphabet, Σ^* is the set of all strings over Σ

The slide also features the NPTEL logo in the bottom left corner and the text "Y.N. Srikant Lexical Analysis" in the bottom right corner.

In our study of you know token recognizers or lexical analyzers we require to define languages finite state automata and regular expressions. So, let us go through some of the definitions and understand them a symbol and a symbol is really an abstract entity. And we do not really define it, it is assumed to be known to everybody it is like a set for example, sets are not defined you know mathematically they are just a abstract and everybody suppose to understand them. So, examples of symbols are letters digits etcetera what is a string a finite sequence of a juxtaposed symbols that is symbols placed one after another is a sequence of symbols. So, such a sequence of symbols is called a string.

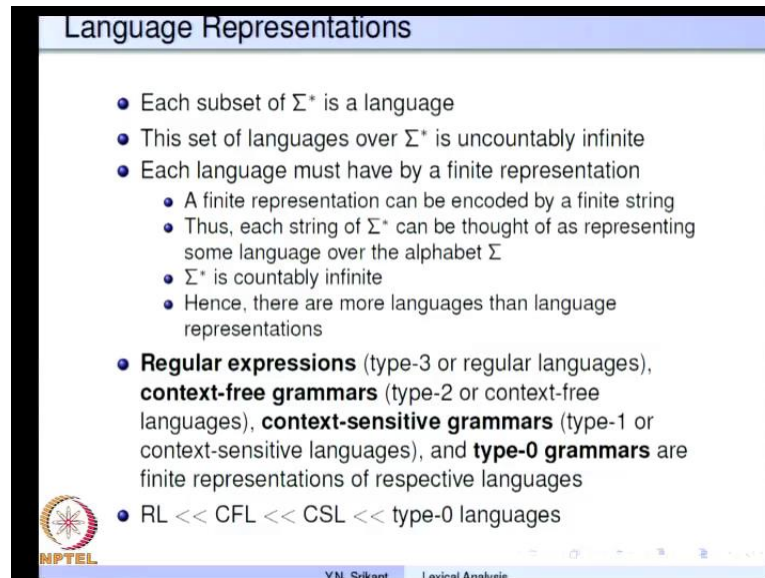
So, if we consider the letters *a b* and *c a b c b c a b a* there are strings over the symbols. Now, well it is very easy to see that you can form any number of symbols any number of strings given these three symbols *a b* and *c* infinite. In fact, if you write $\text{mod } w$ then you know $\text{mod } w$ is the it is stands for the length of the string *w* and is actually the number of symbols in the string ϵ is the empty string and is of length of 0. These become very important for our formal definitions later on and what is an alphabet it is a finite set of these symbols in our case most of the time you know we will be using characters as symbols and set of characters would be our alphabet as appropriate to various

programming languages and what is a language. So, this is not a natural language such as English and nor is it a programming language such as C or Pascal this is mathematical entity called a language, and it is defined as a set of strings of symbols from some alphabet.

So, you take symbols like a b c that is a set of a b c will become an alphabet and then you know you can form any number of strings over it. And if you take some of those strings put them in a set and that set become say language it is not necessary that languages be either finite or infinite all the time there are finite languages and there are infinite languages. So, the null set and the set containing the null string epsilon are both defined as languages that is by definition the set of palindromes over 0 1 is a an infinite language. Because there are infinite number of palindromes which you can form using two symbols 0 and 1 the set of strings 0 1 1 0 1 1 1 only 3 strings over the alphabet 0 comma 1 is a finite language why is it called a finite language.

There are only three strings which is a finite number and the set contains only these three and therefore, it is a finite language. So, if we say sigma is an alphabet then sigma star is the set of all possible strings over sigma. So, if u take a single symbol 0 then starting with epsilon 0 0 0 3 0 4 0 5 0 in general 0 to power n with n greater than equal to 0 this set is an infinite set. And that would be the set of all strings over this sigma which is nothing but a single symbol alphabet containing 0, but similarly you could define sigma as 0 comma 1. Then all possible strings are formed using 0 and 1; obviously, this is an infinite language. So, this is called as sigma star.

(Refer Slide Time: 27:10)



Language Representations

- Each subset of Σ^* is a language
- This set of languages over Σ^* is uncountably infinite
- Each language must have by a finite representation
 - A finite representation can be encoded by a finite string
 - Thus, each string of Σ^* can be thought of as representing some language over the alphabet Σ
 - Σ^* is countably infinite
 - Hence, there are more languages than language representations
- **Regular expressions** (type-3 or regular languages), **context-free grammars** (type-2 or context-free languages), **context-sensitive grammars** (type-1 or context-sensitive languages), and **type-0 grammars** are finite representations of respective languages
- $RL \ll CFL \ll CSL \ll$ type-0 languages

MPTEL

Y.N. Srikant Lexical Analysis

So, having set what sigma star is, sigma star is nothing but all possible strings over a particular alphabet every subset of sigma star is a language. So, that is the formal definition of a language. So, you could have finite subsets of sigma star and; obviously, you can have infinite subsets of the subsets which are infinite sets. So, those are also possible. So, there are infinite languages and finite languages. So, now, there is a tricky you know description, the set of languages over sigma star is uncountably infinite, why? In general if you take a set right and then you define the set of all subsets of that set that is known as a power set.

So, that would have to DO power n elements if n is a the number of elements in the base set that is the finite set will have power set of cardinality to do power n. But once you have, you know sigma star as an infinite set, the number of elements in sigma star is infinite. The number of subsets of sigma star is also infinite, but it becomes what is known as uncountable. So, I am I cannot really spend too much time talking about, what is uncountability, because that is a part of a discrete mathematics. In general set of numbers like 1 2 3 infinite set of this kind is countably infinite. Whereas, the set of subsets of you know infinite sets such as sigma star is uncountably infinite. So, in some sense uncountably infinite is in codes bigger than countably infinite in the mathematical sense. Each language must have now let us look at this carefully each language must have a finite representation otherwise we cannot talk about it.

So, a finite representation; obviously, can be encoded by a finite string. So, any finite representation you know can be encoded in a small string how long that string is that is up to the representation to decide thus if choose a particular sigma And then each string of that sigma star can be thought of as representing some language over the alphabet. So, because you can say each such of each string encodes you know a finite representation and that becomes you know ah some language representing over this alphabet sigma. So, it so happens sigma star itself is countably infinite what I said here is set of languages over sigma star is uncountably infinite. But if you take sigma star it is countably infinite and the set of languages over sigma star is uncountably infinite that is the bigger thing than this sigma star. Hence there are more languages than language representations, the moral of this story is no matter what type of finite representation you come up with for languages.

So, that it can be they can be processed by machines that is compilers interpreters etcetera you know this particular representation will have its limitation. In other words you cannot come up with representations for every language that is possible. There are more languages than language representations so but that is not going to be a big disadvantage for us. Because we are there are more than ample languages available in the representations that we choose and they are more than sufficient for our practical purpose today. So, we are not really worried by the statement that there are more languages than language representations. So, the available representations are sufficient to take care of all the languages that we know of today future. Of course, we have no idea what would happen now there are what are known as type three or regular languages. So, now, we will go into classification of these languages these regular languages can be represented in a finite way using what are known as regular expressions.

So, these languages are infinite, but the representations are finite that is the basic idea of any representation then there are type two or context free languages again these are infinite. But then the grammars context free grammars which we are going to study later form a representation of these languages again grammars are finite objects. So, they are finite representations then we have context sensitive grammars which represent context sensitive languages. Again context sensitive grammars are finite representations of infinite languages and type 0 grammars are finite representations of type 0 languages. So, here is the hierarchy regular languages are weaker than context free languages context

free languages are weaker than context sensitive languages which are in turn weaker than type 0 languages. So, this hierarchy of languages is known as the Chomsky hierarchy based on the, you know to respect the inventor who you know proposed this hierarchy known Chomsky.

(Refer Slide Time: 33:17)

The slide is titled "Examples of Languages" and contains the following text:

Let $\Sigma = \{a, b, c\}$

- $L_1 = \{a^m b^n | m, n \geq 0\}$ is regular
- $L_2 = \{a^n b^n | n \geq 0\}$ is context-free but not regular
- $L_3 = \{a^n b^n c^n | n \geq 0\}$ is context-sensitive but neither regular nor context-free
- Showing a language that is type-0, but none of CSL, CFL, or RL is very intricate and is omitted

The slide also features the NPTEL logo in the bottom left corner and the text "Y.N. Srikant Lexical Analysis" in the bottom right corner.

So, let us look at some examples of the languages. So, sigma let us say is a three symbol set a comma b comma c and that is our alphabet the first language L_1 the set of all you know a's and b's a to the power m b to the power n that is a number of a's followed by a number of b's, but m and n are not related it is just that m and n are greater than or equal to zero. So, you would have epsilon then you would have a b then you would have a b square a cube b 3 etcetera. I just gave examples of strings from this language, but it is an infinite set with no relationship between m and n that is very important L_2 is a similar language, but it says a n b n n greater than equal to 0.

So, the number of a's is equal to the number of b's and the b's follows a's here the number of a's and b's are not related, but the b's follows a's. So, the first language can be represented using regular expressions the second language cannot be represented using regular expressions, but you need context free grammars. Let us look at the third language a to the power n b to the power n, c to the power n n greater than or equal to 0. So, the strings are a number of a's followed by equal number of b's followed by equal number of c's.

So, the difference between these two is the C part this side only a's and b's being equal, but we here add c's also which are in equal in number two number a's and b's. So, once you DO that this language fails to be regular it fails to be context free, but is what is known as a context sensitive language. So, we require context free grammars to represent L 3 context sensitive grammars to represent L 3 context free grammars to represent L 2 and regular expressions to represent L 1 showing a language which is type 0 is outside the scope of this course. But you know very, very intricate you need many more mathematical arguments before we can show that.

(Refer Slide Time: 35:51)

Automata

- Automata are machines that accept languages
 - Finite State Automata accept RLs (corresponding to REs)
 - Pushdown Automata accept CFLs (corresponding to CFGs)
 - Linear Bounded Automata accept CSLs (corresponding to CSGs)
 - Turing Machines accept type-0 languages (corresponding to type-0 grammars)
- Applications of Automata
 - Switching circuit design
 - Lexical analyzer in a compiler
 - String processing (*grep*, *awk*), etc.
 - State charts used in object-oriented design
 - Modelling control applications, e.g., elevator
 - Parsers of all types
 - Compilers

NPTEL

Y.N. Srikant Lexical Anal

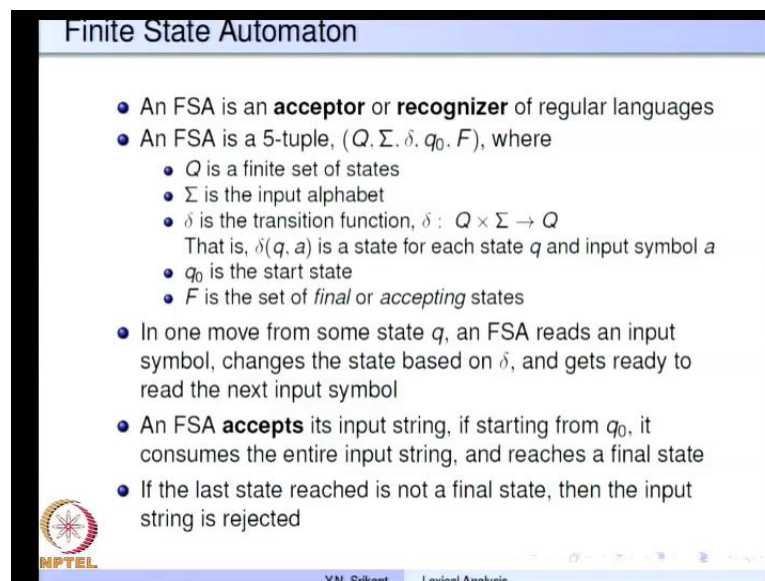
So, I am going to omit it. So, now, what exactly are automata automata are machines. So, what do these machines really do these machines accept languages and those languages you know correspond to the once that we have already defined. For example, finite state automata accept regular languages and they can be specified using regular expressions pushdown automata accept, context free languages. And they can be specified using context free grammars linear bounded automata accept, context sensitive languages and they can be specified using context sensitive grammars. Turing machines accept type 0 languages and they can be specified using type 0 grammars.

So, these are the 4 types of automata which are extremely important in the study of languages and automata theory for our purpose. We restrict ourselves to the finite of state automata and push down automata finite state automata are used for regular for lexical

analysis and push down automata are used for parsing. There are many applications of automata for example; finite state automata have been used extensively in switching circuit design. Of course, I already mentioned its use in lexical analyzer then the Unix tools grep and awk.


They perform string processing and are based on finite state machines object oriented design. For example, UML it uses what are known as state charts they are nothing, but extensions of finite state automata modeling control applications. For example, an elevator operation can be easily specified using finite state automata parsers of all types use push down automata and of course, compilers. You know there are tree automata and so on which are extensions of the finite state automata which are used in compilers for code generation and other purposes.

(Refer Slide Time: 38:15)



Finite State Automaton

- An FSA is an **acceptor** or **recognizer** of regular languages
- An FSA is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, where
 - Q is a finite set of states
 - Σ is the input alphabet
 - δ is the transition function, $\delta : Q \times \Sigma \rightarrow Q$
That is, $\delta(q, a)$ is a state for each state q and input symbol a
 - q_0 is the start state
 - F is the set of *final* or *accepting* states
- In one move from some state q , an FSA reads an input symbol, changes the state based on δ , and gets ready to read the next input symbol
- An FSA **accepts** its input string, if starting from q_0 , it consumes the entire input string, and reaches a final state
- If the last state reached is not a final state, then the input string is rejected

 Y.N. Srikant Lexical Analysis

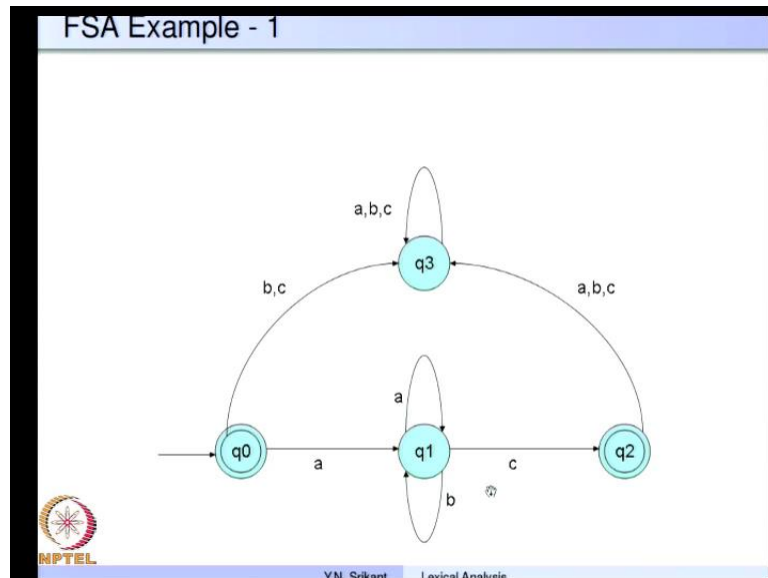
Let us begin our discussion of the finite state automaton a finite state automaton is said to be an acceptor or recognizer of regular languages. I have already this, it is a machine really and it can be programmed. Let us look at the formal definition of a finite state automaton it is defined as a 5 tuple quintuple first is first component is Q which is a set of finite set of states. Then there is a Σ which is the input alphabet for this particular machine. Then we have a δ which is the transition function it requires a little more explanation and I am going to do that after we run through this definition. Then a one of

the states in Q is designated as the start state and it is represented as q_0 and F is a subset of Q and whatever is in F is a final state.

So, now, let us get back to δ as I said the finite state automaton is a machine. So, δ tells you how the machine progresses from one state to another on consuming a particular symbol from the input. So, δ is a transition function it is a mapping between $Q \times \Sigma$ and Q from $Q \times \Sigma$ to Q . So, in other words we write it as $\delta: Q \times \Sigma \rightarrow Q$ something like that. So, it means when the machine is in state q and next input symbol is a , the machine changes the state and goes to the state for which you know which is defined as $\delta(q, a)$. So, that is a state which enters. So, in one move from some state q finite state machine reads an input symbol changes the state based on δ and gets ready to read the next input symbol as when I shown you an example. It will be a very clear an finite state automaton accepts its input string if starting from start state q_0 it consumes the entire input string and reaches a final state.

So, both these conditions are very important it is not enough if it consumes the entire input string. but is in non final state. And it is not enough if reaches a final state and there some more input remaining both must happen and it must start from start state q_0 it cannot start from some other state. So, in such a case that automaton is set to have accepted the input if the last state reached is not a final state then the input string is rejected in other words its reads the entire input. But then enter a non final state then the input is not a part of the language or rather it is not accepted by the finite state automaton.

(Refer Slide Time: 41:41)



So, let us take it as a simple example. So, you start from the start here q0, q1, q2 and q3. These are the set of states of the automaton q0, q1, q2 and q3. q0 is the initial state. And then there are 2 states q1 and q2 which are special which have double circles. So, these are the final states. So, q0 to q1 set of then the delta function is shown by these arcs and the labels. So, when the state delta of q0, a is q1. So, in other words from the state q0 on input a, the machine goes to the state q1 similarly from the state q1 on input c it goes to state q2 etcetera. So, from q0 it goes to a or goes description of this in the next slide.


(Refer Slide Time: 43:02)

FSA Example -1 (Contd.)

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{a, b, c\}$
- q_0 is the start state and $F = \{q_0, q_2\}$
- The transition function δ is defined by the table below

state	symbol		
	a	b	c
q_0	q_1	q_3	q_3
q_1	q_1	q_1	q_2
q_2	q_3	q_3	q_3
q_3	q_3	q_3	q_3

The accepted language is the set of all strings beginning with an 'a' and ending with a 'c' (ϵ is also accepted)

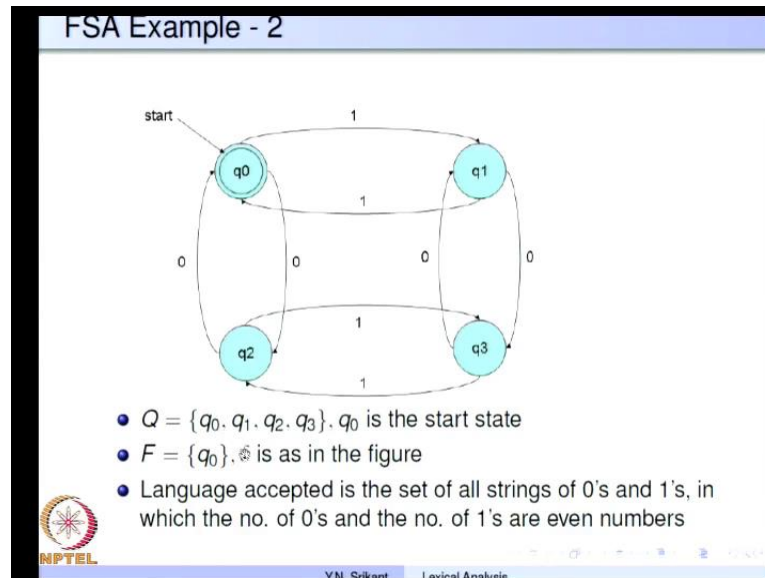


Y.N. Srikant Lexical Analysis

So, as I said it has 4 states q_0, q_1, q_2, q_3 Σ is $\{a, b, c\}$ q_0 is the start state and F is $\{q_0, q_2\}$. So, you can observe that q_0 and q_2 are the final states the transition function is defined by a table below. So, let us look at it from q_0 we already know on a it goes to q_1 on b it goes to q_3 and on c it goes to q_3 etcetera. Now, let us look at the machine again and see if some strings is accepted or rejected let us take the string a b c. So, we start from the start state q_0 the first symbol is a. So, we go to state q_1 , the second symbol is b. So, from q_1 on b the machine actually stays in state q_1 consumes the b and on the final symbol c it goes to state q_2 . So, the input is exhausted and it has reached the final state.

So, the string a b c is definitely accepted by the automaton. So, you can now see that single a fired by any number of a's and b's followed by 1 c. This set of strings is accepted by the machine where as any strings which begins with a b or c it enters the state q_3 from which it is not possible to get out or though if the state remains there. So, if you consider the string b a c so on b it goes to q_3 and on a stays in q_3 and again on b C it stays in q_3 so on b a c is starting from the input stay you know with the start state q_0 it ends in q_3 after exhausting the input which is not a final state therefore, the string b a c is rejected by the automaton. So, the accepted language for this particular set is the set of all strings beginning with an a and ending with a c. Of course, epsilon is also accepted simply, because this particular state the start state without consuming anything will also is final state. So, epsilon the input string with of 0 length is also accepted.

(Refer Slide Time: 45:40)



Another example, so again we have a q_0 , q_1 , q_2 and q_3 as 4 states that is a set Q and the state q_0 is start state as usual. But it is not mandatory to make q_0 as a start state all the time even though that is the notation which is used in every text book it could be q_3 as well. But the designer can use a different notation if he or she desires is just q_0 . So, only q_0 is a final state and all others are non final states. So, in other words if the input you know takes the machine from q_0 to some other state after exhausting the input then the input is not accepted. But if it brings it back to the initial state then the input is accepted δ is here.

So, what is the language? δ is always shown in the form table in or the form of a picture, picture is easier to understand. So, we have used pictures in our case the language accepted is the set of all strings of 0's and 1's in which the number of 0's and the number of 1's are even numbers. So, that you can check you know on a single 0 over here on equal number of 1's you know I keep circulating between these two states right. So, for example, 0 and a 0 right and then let us say go through a single 0 and then on a single one I go here. But then I have to consume another 0 to get to q_1 and finally, I must consume another one to get back to q_0 . So, if I consume only odd number of 0's and or odd number of 1's I will never get back to q_0 I will remain in one of the other state its q_1 , q_2 and q_3 . So, and therefore, so those strings are not in the set of accepted strings.

(Refer Slide Time: 47:56)

Regular Languages

- The language **accepted** by an FSA is the set of all strings accepted by it, i.e., $\delta(q_0, x) \in F$
- This is a **regular language** or a **regular set**
- Later we will define **regular expressions** and **regular grammars** which are **generators** of regular languages
- It can be shown that for every regular expression, an FSA can be constructed and vice-versa

NPTEL

Y.N. Srikant Lexical Analysis



So, we saw an example of finite state automata. Now, the language accepted by a finite state automaton is the set of all strings accepted by it. That is starting from the start state the, you know the string x which is the input string must actually take you to the final state it must belong to the final state. So, this notation is an extension of the notation where second component was a single symbol, but that is a very straight forward extension. So, we can say $\delta(q_0, x)$ must take you to a final state. So, this is the language you know all the strings of this kind which take you to the final state from the start state. Or in the language accepted that by the finite state automaton and this is what is known as a, this is a regular language or a regular set.

So, this is how defined the regular language one of ways in which we defined the regular language. Later we will also define regular expressions and regular grammars which are also you know specification of regular languages. But that is not the task right now of course, it can be shown that for every regular expression a finite state automation can be constructed and for every finite state automaton a regular expression can also be constructed. So, we will look at this briefly a little later.

(Refer Slide Time: 49:40)

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 deterministic FA (DFA), that accepts the same language as the NFA

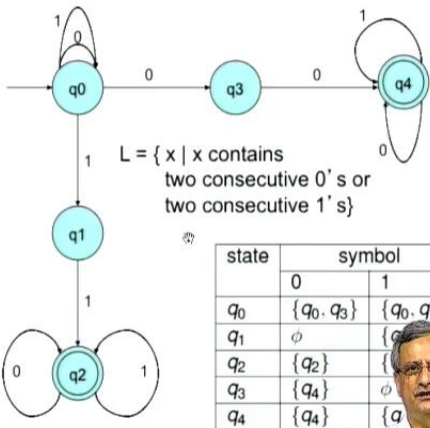


Y.N. Srikant Lexical Analysis

Now, what we have seen so far are finite state automata, but there something very special about it they are what are known as deterministic finite state automata. So, why are they called deterministic? The deterministic automata do not permit more than one transition from any state on a particular symbol whereas, for non deterministic finite state automata. They allow 0 1 or more transitions from state on a given input symbol.



(Refer Slide Time: 50:22)

Nondeterministic FSA Example - 1



$L = \{ x \mid x \text{ contains two consecutive 0's or two consecutive 1's} \}$

state	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	ϕ	$\{q_1, q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	ϕ
q_4	$\{q_4\}$	$\{q_4\}$



Y.N. Srikant Lexical Analysis

So, to show you a simple example on 0 from the straight q_0 I can either remain in the state q_0 or go to the state q_1 similarly on a one I can either remain in state q_0 or go to a

state q_1 . So, in the you know this is an example where the non determinism shows the machine can decide to stay in q_0 on a 0 or it can decided to stay by jump to q_3 on a 0. So, this is exactly, what is non determinism. So, it allows 0 1 or more transitions from a state on a given input symbol. So, the finite state N F A the other one is called as a D F A deterministic finite state of automata and this is known as an N F A. But the transition function δ is different here it is a same 5 tuple, but the transition function is very different.

So, for example, $\delta(q, a)$ it used to be a single set in the case of a D F A, but here it is a set of all states p such that there is a transition labeled a from the state q to the state p . So, it is a set here and one of the elements of the set could be chosen by the automaton at any point in time. So, for example, in this case you know. So, from the state q_0 on a 0 the δ function says it is a set consisting of q_{naught} comma q_3 and for this on a one if the set consisting of q_{naught} comma q_1 . So, either q_{naught} or q_3 can be chosen by the machine. So, δ in the case of a D F A was represented as $q \times \Sigma^2 \rightarrow q$ that is for a given combination of q and symbol a single state was possible. But here a set of states, this is the power set notation a set of states as which is a subset of the set of state is possible.

A string is accepted by is accepted by an N F A, if there exists a sequence of transitions corresponding to the string that leads from the start state to the some final state. So, it is very similar to the previous one you know in the case of a D F A we said it should be go from start state to final state on the input here. It can go from a start state to anyone of the final states it is not necessary that it goes to the same final state every time it can go to anyone of the final states s_0 and then still we say the string is accepted. So, every N F A can be converted to an equivalent deterministic finite state automata, that accepts the same language as the N F A. So, this is a very powerful result which says the non determinism does not really add anything to finite striate automata. So, we are going to look at that result in some detail later on. So, let us look at this example here is non determinism and the language is the set of symbol x such that x contains two consecutive 0's or two consecutive 1's.

So, let me demonstrate how this work. So, it is you know from q_0 . It can consume any number of 0's and 1's, but finally, it should go to q_3 and then to q_4 . That means, it would have at least two consecutive 0's here followed by any numbers of 0's. And once

again in q_4 if it has taken this path it can have any numbers of 0's and 1's in the beginning. But then it must consume at least two 1's enter q_2 and then it can have any number of 0's and once in this state as well. So, two consecutive 0's or two consecutive 1's, that would be the rule to make a string acceptable to this particular automaton. So, we will stop here and continue with non deterministic automata in the next lecture.

Thank you.