MCA Part III

Paper : 19 (Compiler Design)

Topic: LEXICAL ANALYSIS

Prepared by: Dr. Kiran Pandey

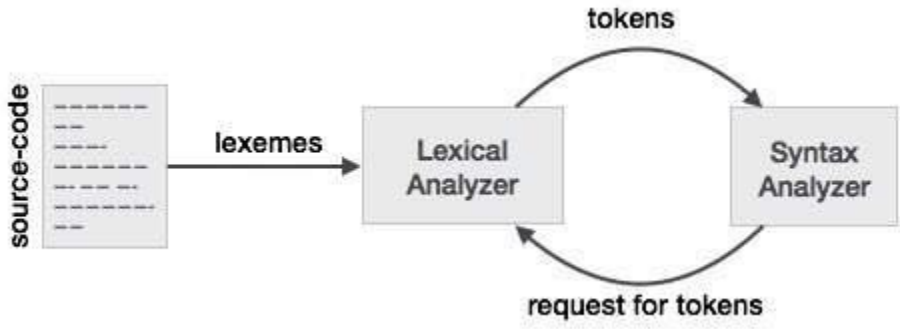
(School of Computer Science)

Email-id: kiranpandey.nou@gmail.com

INTRODUCTION

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



SPECIFICATION AND RECOGNITION OF TOKENS

In this section we will discuss the concept of tokens and understand how token are specified.

Tokens

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

int value = 100;

contains the tokens:

int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).

Specifications of Tokens

Let us understand how the language theory undertakes the following terms:

(i) Alphabets

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

(ii) Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string nou is 3 and is denoted by |nou| = 3. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

(iii)Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=

Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifier	&
Logical	&, &&, |, ||, !
Shift Operator	>>, >>>, <<, <<<

(iv)Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

(v)Longest Match Rule

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

For example:

int intvalue;

While scanning both lexemes till ‘int’, the lexical analyzer cannot determine whether it is a keyword *int* or the initials of identifier int value.

The lexical analyzer also follows **rule priority** where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

REGULAR EXPRESSION AND THEIR ROLE IN LEXICAL ANALYZER

The lexical analyzer scans and identifies only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as **regular grammar**. The language defined by regular grammar is known as **regular language**.

Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition. There are a number of algebraic laws that are obeyed by regular expressions, which can be used to manipulate regular expressions into equivalent forms.

Operations

The various operations on languages are:

- Union of two languages L and M is written as

  L U M = {s | s is in L or s is in M}

- Concatenation of two languages L and M is written as

  LM = {st | s is in L and t is in M}

- The Kleene Closure of a language L is written as

  L* = Zero or more occurrence of language L.

Notations

If r and s are regular expressions denoting the languages L(r) and L(s), then

- **Union** : (r)|(s) is a regular expression denoting L(r) U L(s)
- **Concatenation** : (r)(s) is a regular expression denoting L(r)L(s)
- **Kleene closure** : (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denoting L(r)

Precedence and Associativity

- *, concatenation (.), and | (pipe sign) are left associative
- * has the highest precedence
- Concatenation (.) has the second highest precedence.
- | (pipe sign) has the lowest precedence of all.

Representing valid tokens of a language in regular expression

If **a** is a regular expression, then:

- **a*** means zero or more occurrence of a.

  i.e., it can generate {e, a,aa,aaa,aaaa …}

- **a+** means one or more occurrence of a.

  i.e., it can generate {a,aa,aaa,aaaa, …} or aa*

- **a?** means at most one occurrence of a

  i.e., it can generate either {a} or {e}.

  [a-z] is all lower-case alphabets of English language.

  [A-Z] is all upper-case alphabets of English language.

  [0-9] is all natural digits used in mathematics.

Representing occurrence of symbols using regular expressions

letter = [a – z] or [A – Z]

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]

sign = [+ | -]

Representing language tokens using regular expressions

Decimal = (sign)$^{?}$(digit)$^{+}$

Identifier = (letter)(letter | digit)*

The only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification.

FINITE STATE MACHINES

Finite state machine or automata is a state machine that takes a string of symbols as input and changes its state accordingly. Finite automata is a recognized for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each literal. If the input string is successfully processed and the automata reaches its final state, it is accepted, i.e., the string just fed was said to be a valid token of the language in hand.

The mathematical model of finite automata consists of:

- **Finite set of states (Q)**
- **Finite set of input symbols (Σ)**
- **One Start state (q0)**
- **Set of final states (qf)**
- **Transition function (δ)**

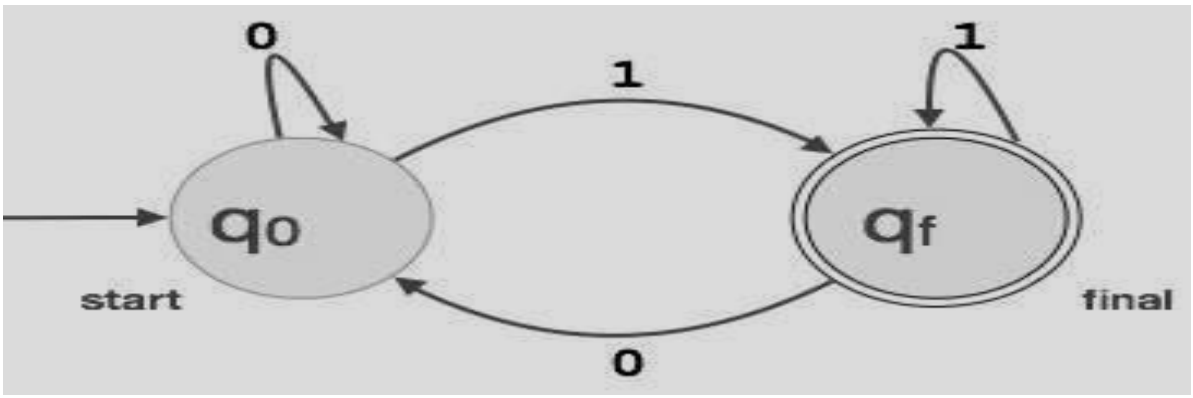
The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), $Q \times \Sigma \rightarrow Q$

Finite Automata Construction

Let **L(r)** be a regular language recognized by some finite automata (FA).

- **States**: States of FA are represented by circles. State names are written inside circles.
- **Start state**: The state from where the automata starts, is known as the start state. Start state has an arrow pointed towards it.
- **Intermediate states**: All intermediate states have at least two arrows; one pointing to and another pointing out from them.
- **Final state**: If the input string is successfully parsed, the automata is expected to be in this state. Final state is represented by double circles. It may have any odd number of arrows pointing to it and even number of arrows pointing out from it. The number of odd arrows are one greater than even, i.e. **odd = even+1**.
- **Transition**: The transition from one state to another state happens when a desired symbol in the input is found. Upon transition, automata can either move to the next state or stay in the same state. Movement from one state to another is shown as a directed arrow, where the arrows points to the destination state. If automata stays on the same state, an arrow pointing from a state to itself is drawn.

Example: We assume FA accepts any three digit binary value ending in digit 1. FA = $\{Q(q_0, q_f), \Sigma(0,1), q_0, q_f, \delta\}$



IMPLEMENTATION OF LEXICAL ANALYSER USING NFA, DFA,

Implementation of Lexical analyzer either by hand or automated tools mainly involve two steps:

(i) Describe rules for token using regular expressions

(ii) Design a recognizer for such rules, that is, for tokens. Designing a recognizer corresponds to converting regular expressions to Finite Automata. The processing can be speeded if the regular expression is represented in Deterministic Finite Automata. This involves the following steps:

(a) Convert regular expression to NFA with ε
(b) Convert NFA with ε to NFA without ε
(c) Convert NFA to DFA.

An NFA can have zero, one or more than one move from a given state on a given input symbol. An NFA can also have NULL moves (moves without input symbol). On the other hand, DFA has one and only one move from a given state on a given input symbol.

Conversion from NFA to DFA

Suppose there is an **NFA** $N < Q, \sum, q0, \delta, F >$ which recognizes a language L.

Then the **DFA** $D < Q', \sum, q0, \delta', F' >$ can be constructed for language L as:

Step 1: Initially Q' = ϕ.

Step 2: Add q0 to Q'.

Step 3: For each state in Q', find the possible set of states for each input symbol using transition function of NFA. If this set of states is not in Q', add it to Q'.

Step 4: Final state of DFA will be all states with contain F (final states of NFA)

Example

Consider the following NFA shown in Figure 1.

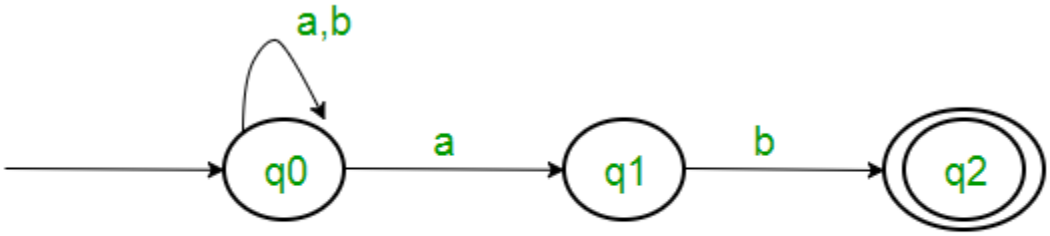


Figure 1

Following are the various parameters for NFA.
Q = { q0, q1, q2 }
∑ = (a, b)
F = { q2 }
δ (Transition Function of NFA)

State	a	b
q0	q0,q1	q0
q1		q2
q2		

Step 1: Q' = ϕ

Step 2: Q' = {q0}

Step 3: For each state in Q', find the states for each input symbol.

Currently, state in **Q**' is **q0**, find moves from **q0** on input symbol **a** and **b** using transition function of NFA and update the transition table of DFA.

δ' (Transition Function of DFA)

State	a	b
q0	{q0,q1}	q0

Now **{ q0, q1 }** will be considered as a single state. As its entry is not in **Q'**, add it to **Q'.**
So **Q' = { q0, { q0, q1 } }**

Now, moves from state { q0, q1 } on different input symbols are not present in transition table of DFA, we will calculate it like:

δ' ({ q0, q1 }, a) = δ (q0, a) ∪ δ (q1, a) = { q0, q1 }

δ' ({ q0, q1 }, b) = δ (q0, b) ∪ δ (q1, b) = { q0, q2 }

Now we will update the transition table of DFA.

δ' (Transition Function of DFA)

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}

Now **{ q0, q2 }** will be considered as a single state. As its entry is not in **Q'**, add it to **Q'**.
So **Q' = {q0, {q0, q1}, {q0, q2} }**

Now, moves from state **{q0, q2}** on different input symbols are not present in transition table of DFA, we will calculate it like:

δ' ({ q0, q2 }, a) = δ (q0, a) ∪ δ (q2, a) = { q0, q1 }

δ' ({ q0, q2 }, b) = δ (q0, b) ∪ δ (q2, b) = { q0 }

Now we will update the transition table of DFA.

δ' (Transition Function of DFA)

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}
{q0,q2}	{q0,q1}	q0

As there is no new state generated, we are done with the conversion. Final state of DFA will be state which has q2 as its component i.e., **{q0, q2}**

Following are the various parameters for DFA.

Q' = { q0, { q0, q1 }, { q0, q2 } }

∑ = (a, b)

F = { { q0, q2 } } and transition function δ' as shown above. The final DFA for above NFA has been shown in Figure 2.

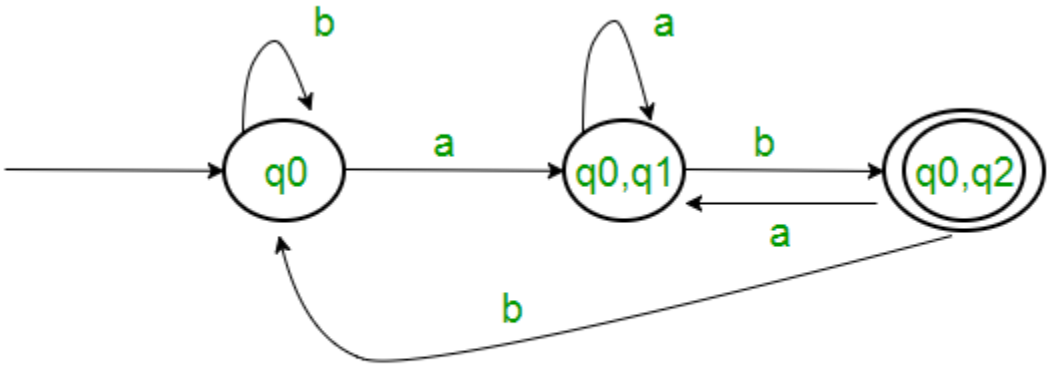


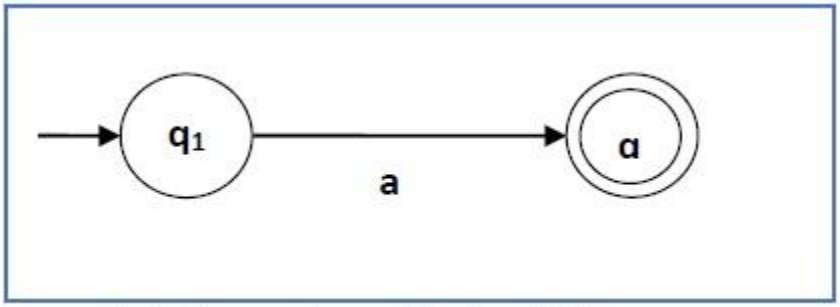
Figure 2

Note: Sometimes, it is not easy to convert regular expression to DFA. First you can convert regular expression to NFA and then NFA to DFA.

SOME EXAMPLES

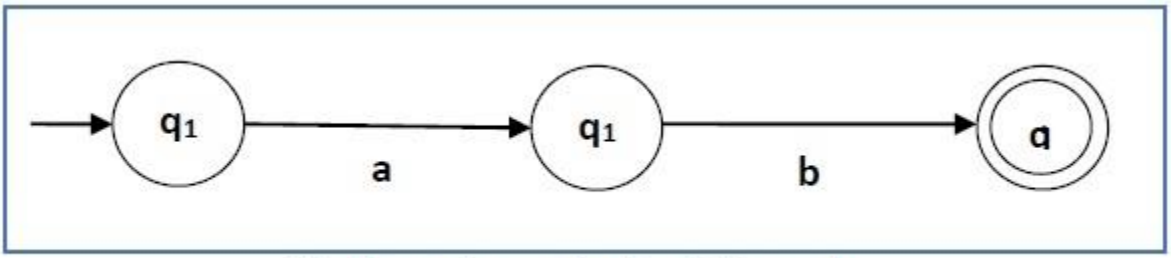
Some basic RA expressions are the following –

Case 1 – For a regular expression 'a', we can construct the following FA –



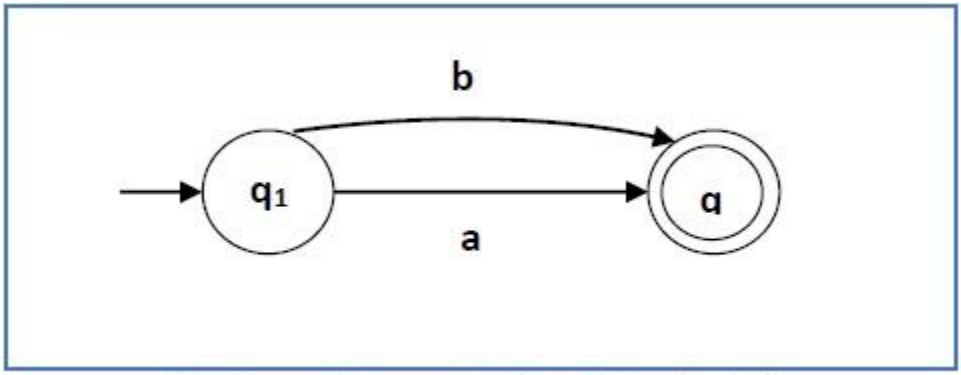
Finite automata for RE = a

Case 2 – For a regular expression 'ab', we can construct the following FA –



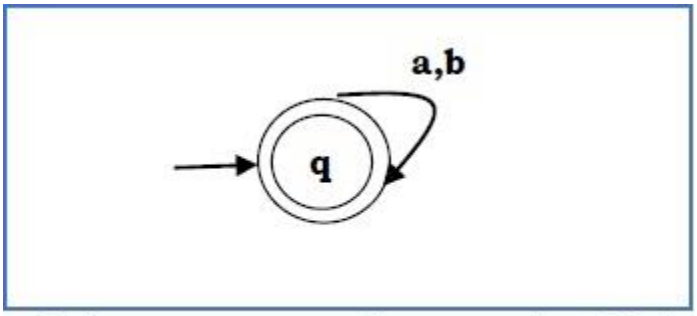
Finite automata for RE = ab

Case 3 – For a regular expression (a+b), we can construct the following FA –



Finite automata for RE= (a+b)

Case 4 – For a regular expression (a+b)*, we can construct the following FA –



Finite automata for RE= (a+b)*

Method

Step 1 Construct an NFA with Null moves from the given regular expression.

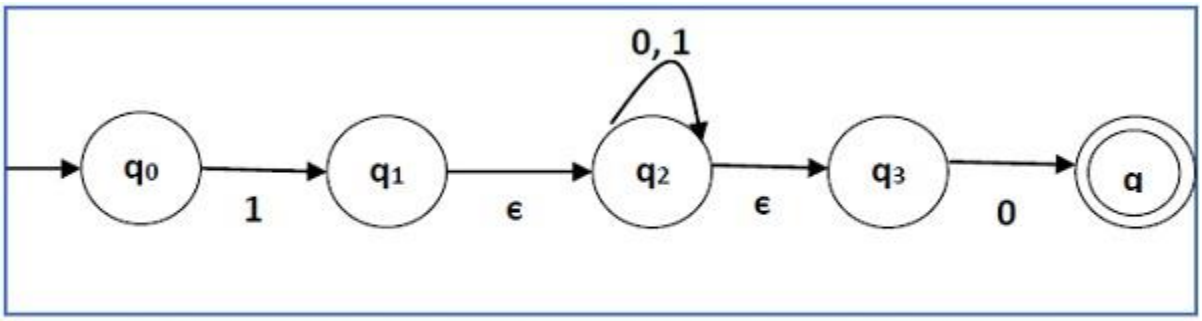
Step 2 Remove Null transition from the NFA and convert it into its equivalent DFA.

Problem

Convert the following RA into its equivalent DFA − 1 (0 + 1)* 0

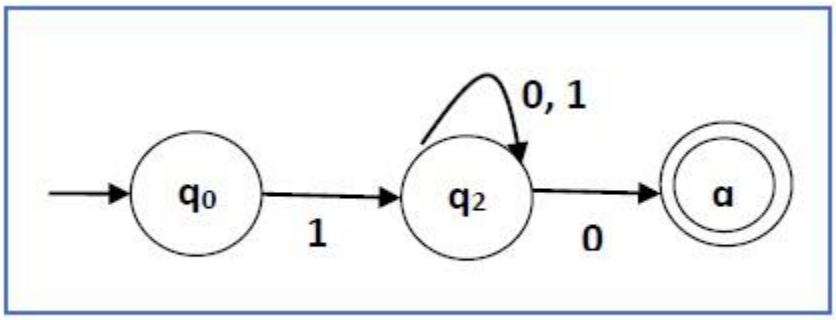
Solution

We will concatenate three expressions "1", "(0 + 1)*" and "0"



NDFA with NULL transition for RA: 1 (0 + 1)* 0

Now we will remove the **ε** transitions. After we remove the **ε** transitions from the NDFA, we get the following −



NDFA without NULL transition for RA: 1 (0 + 1)* 0

It is an NDFA corresponding to the RE − 1 (0 + 1)* 0. If you want to convert it into a DFA, simply apply the method of converting NDFA to DFA discussed in Chapter 1.

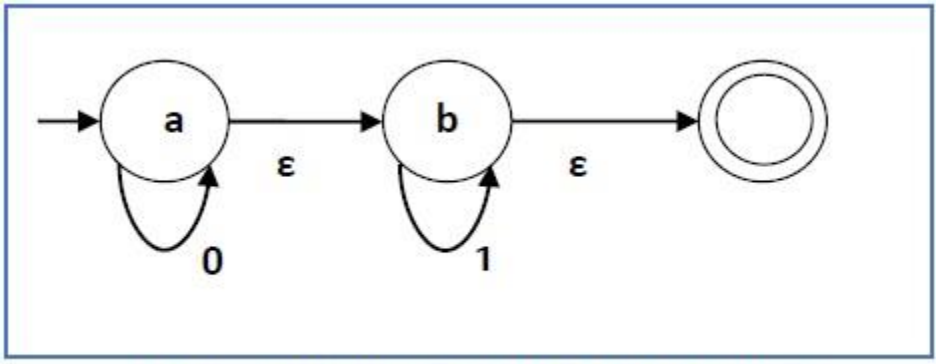
Finite Automata with Null Moves (NFA-ε)

A Finite Automaton with null moves (FA-ε) does transit not only after giving input from the alphabet set but also without any input symbol. This transition without input is called a **null move**.

An NFA-ε is represented formally by a 5-tuple (Q, ∑, δ, q_0, F), consisting of

- **Q** − a finite set of states

- **∑** − a finite set of input symbols
- **δ** − a transition function $\delta : Q \times (\sum \cup \{\varepsilon\}) \rightarrow 2^Q$
- **q_0** − an initial state $q_0 \in Q$
- **F** − a set of final state/states of Q ($F \subseteq Q$).



Finite automata with Null Moves

The above **(FA-ε)** accepts a string set − {0, 1, 01}

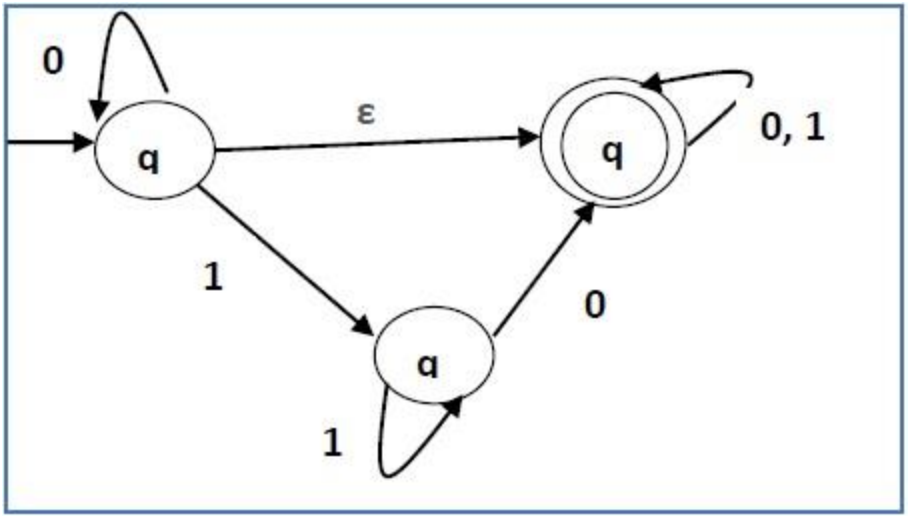
Removal of Null Moves from Finite Automata

If in an NDFA, there is ϵ-move between vertex X to vertex Y, we can remove it using the following steps −

- Find all the outgoing edges from Y.
- Copy all these edges starting from X without changing the edge labels.
- If X is an initial state, make Y also an initial state.
- If Y is a final state, make X also a final state.

Problem

Convert the following NFA-ε to NFA without Null move.



Solution

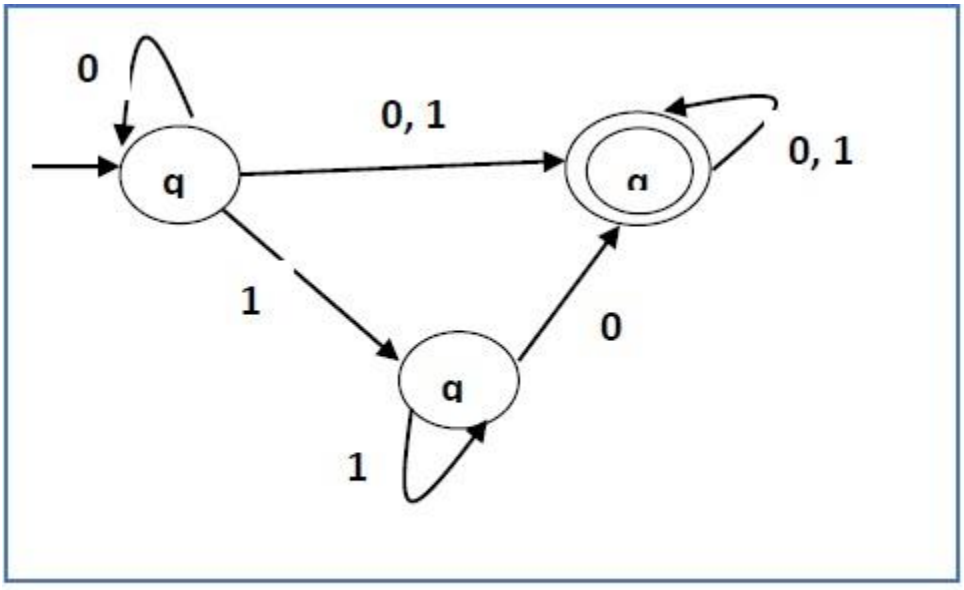
Step 1 –

Here the ε transition is between $\mathbf{q_1}$ and $\mathbf{q_2}$, so let $\mathbf{q_1}$ is **X** and $\mathbf{q_f}$ is **Y**.

Here the outgoing edges from q_f is to q_f for inputs 0 and 1.

Step 2 –

Now we will Copy all these edges from q_1 without changing the edges from q_f and get the following FA –

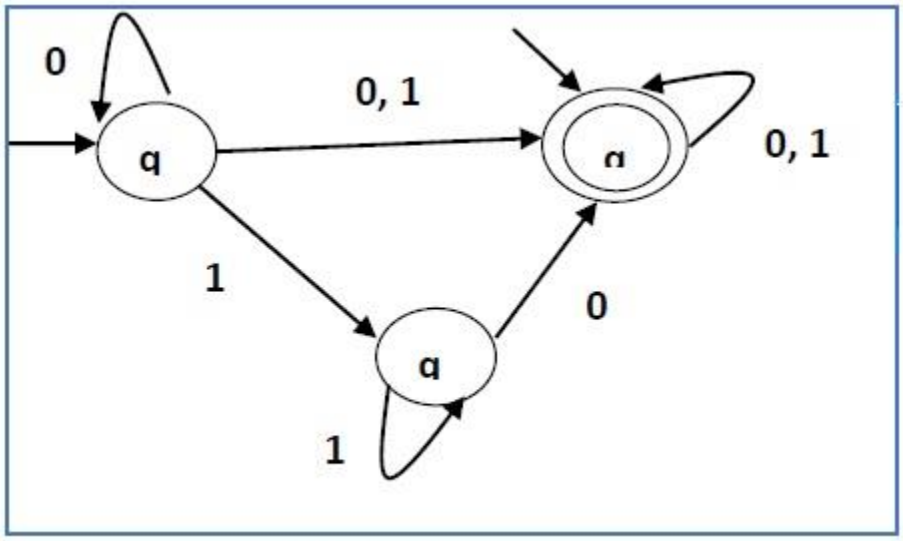


NDFA after step 2

Step 3 –

Here q_1 is an initial state, so we make q_f also an initial state.

So the FA becomes –

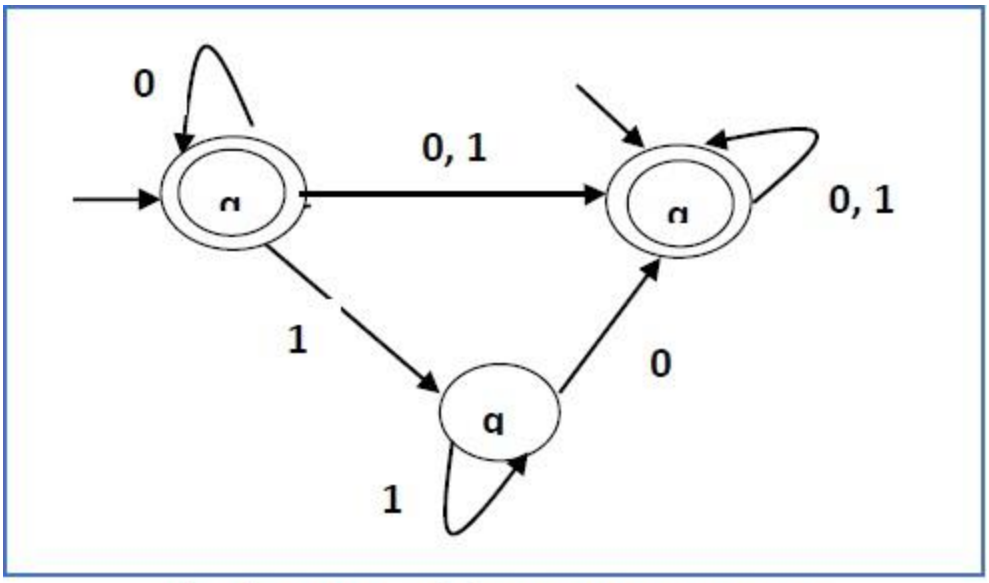


NDFA after Step 3

Step 4 –

Here q_f is a final state, so we make q_1 also a final state.

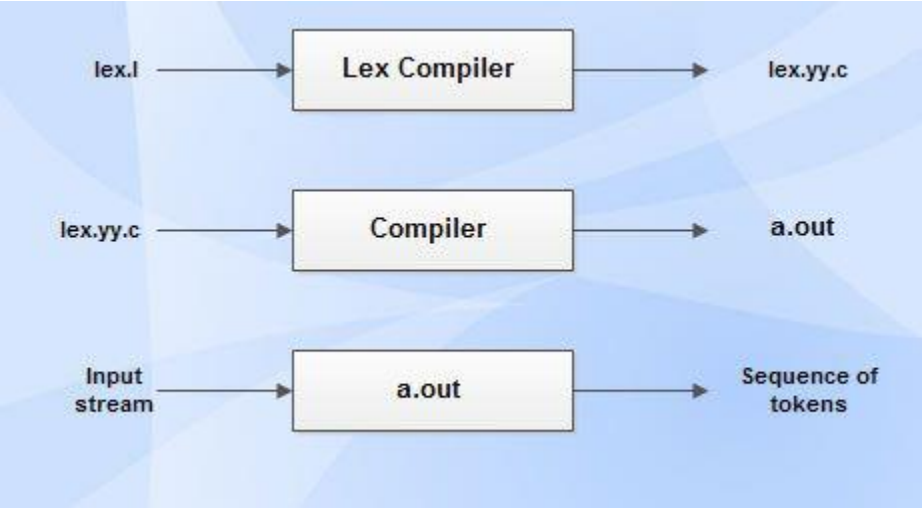
So the FA becomes –



Final NDFA without NULL moves

Lex TOOL

Lex is a tool in lexical analysis phase to which recognizes tokens using regular expression. Lex Tool itself is a lex compiler.



lex.l is an a input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms *lex.l* to a C program known as *lex.yy.c.*

• ***lex.yy.c*** is compiled by the C compiler to a file called *a.out.*

• The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.

• ***yylval*** is a global variable which is shared by lexical analyzer and parser to return the name and an attribute value of token.

The attribute value can be numeric code, pointer to symbol table or nothing.

• Another tool for lexical analyzer generation is Flex.

Structure of Lex Programs

Lex program will be in following form

declarations

%%

translation rules

%%

auxiliary functions

Declarations This section includes declaration of variables, constants and regular definitions.

Translation rules It contains regular expressions and code segments.

Form : Pattern {Action}

Pattern is a regular expression or regular definition.

Action refers to segments of code.

Auxiliary functions: This section holds additional functions which are used in actions. These functions are compiled separately and loaded with lexical analyzer. Lexical analyzer produced by lex starts its process by reading one character at a time until a valid match for a pattern is found. Once a match is found, the associated action takes place to produce token. The token is then given to parser for further processing.

Conflict Resolution in Lex

Conflict arises when several prefixes of input matches one or more patterns. This can be resolved by the following:

• Always prefer a longer prefix than a shorter prefix.

• If two or more patterns are matched for the longest prefix, then the first pattern listed in lex program is preferred.

Lookahead Operator

• Lookahead operator is the additional operator that is read by lex in order to distinguish additional pattern for a token.

• Lexical analyzer is used to read one character ahead of valid lexeme and then retracts to produce token.

• At times, it is needed to have certain characters at the end of input to match with a pattern. In such cases, slash (/) is used to indicate end of part of pattern that matches the lexeme.

(eg.) In some languages keywords are not reserved. So the statements

IF (I, J) = 5 and IF (condition) THEN

results in conflict whether to produce IF as an array name or a keyword. To resolve this the lex rule for keyword IF can be written as,

IF/\ (.* \) {

letter }

Note: A Lexical analyzer can either be generated by NFA or by DFA. DFA is preferable in the implementation of lex.

QUESTIONS

1. What are the tasks performed by lexical analyzer?

2. What is a regular expression? Write some regular expressions.

3. Explain the rules for writing the regular expression.

4. What is a finite state machine?

5. Differentiate between DFA and NFA.

6. What are the number of states in the minimal deterministic finite automaton corresponding to the regular expression (0 + 1)* (10)?

7. Find the number of tokens in the following code:

```
if(x>=y)z = 0;
```

8. Describe lex Tool.