Unit-2

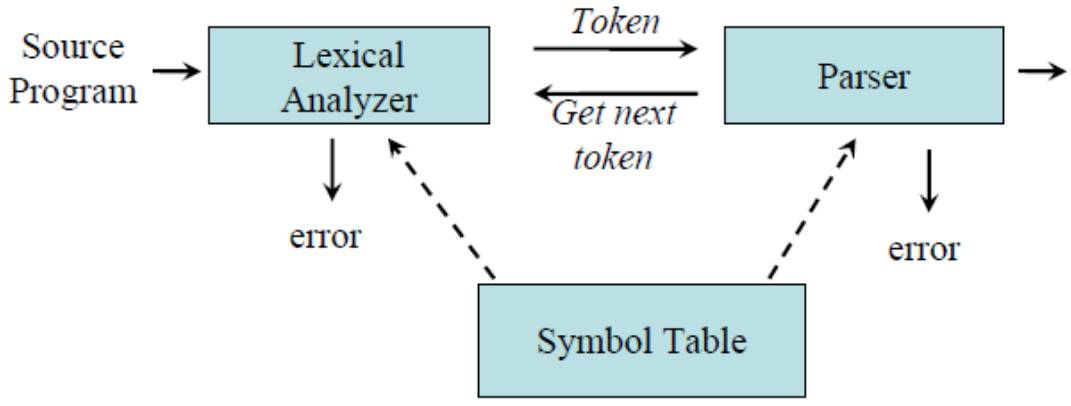
Unit-2

2.1 Lexical Analyzer

2.1.1 Role of Lexical Analysis

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser. Another main task of the lexical analyzer is to strip out white spaces and other unnecessary character from the source program while generating the tokens. Last but not the least another main task is to correlate error message from the compiler with the source program.

These interactions are suggested in Figure. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the getNextToken command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.



Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input). Another task is correlating error messages generated by the compiler with the source

program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line

number with each error message. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions. If the source program uses a macro – preprocessor, the expansion of macros may also be performed by the lexical analyzer.

Why separate Lexical Analyzer is needed?

a) Simplicity of design is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks.
b) Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing.
c) Specialized buffering techniques for reading input characters can speed up the compiler significantly.
d) Compiler portability is enhanced. Input – Device – Specific peculiarities can be restricted to the lexical analyzer.

2.1.2 Tokens, Lexemes and Patterns

A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

2.1.2.1 Attribute of Tokens

When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched. For example, the pattern for token number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program. Thus, in many cases the lexical analyzer

returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token; the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.

The token name and associated attribute values for the following statement.

A=B*C+2

<id, pointer to symbol table entry for A>

<assignment_op>

<id, pointer to symbol table entry for B>

<mult_op>

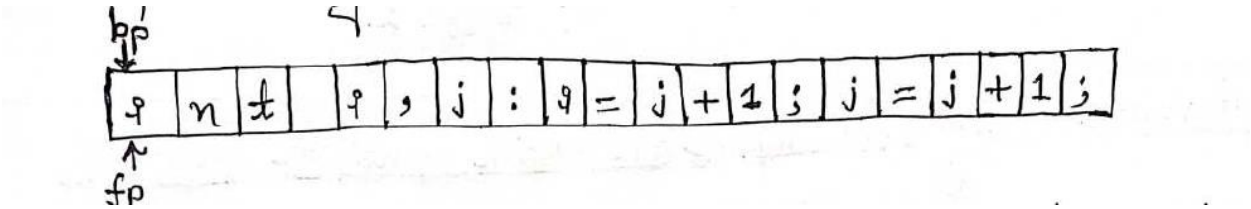
<id, pointer to symbol-table entry for C>

<add_op>

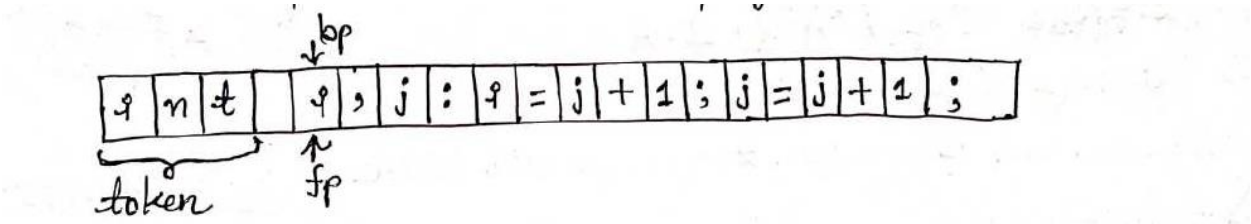
<number, interger value 2>

2.1.3 Input Buffering

Reading character by character from secondary storage is slow process and time consuming, so we use buffer technique to eliminate this problem and increase efficiency. The lexical analyzer scans the input from left to right one character at a time. It uses two pointers" bp" (begin pointer) and" fp" (forward pointer) to keep track of the pointer of input scanned. Initially both the pointers point to the first character of the input string.



The forward pointer moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of the lexeme. In above example as soon as fp encounters a blank space the lexeme 'int' is identified. Then both bp and fp are set at next token as shown in the figure below. This process will be repeated for the whole program.



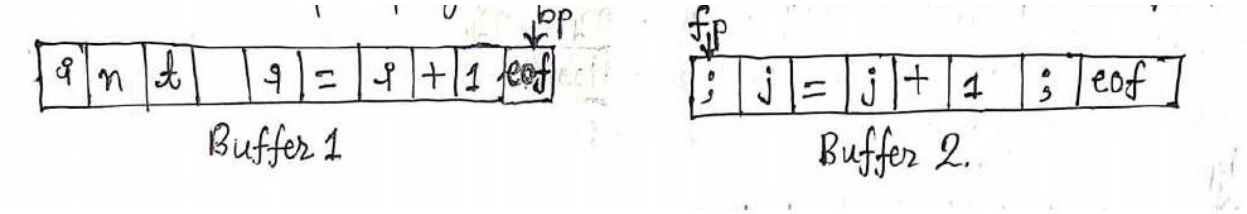
2.1.3.1 One Buffer Scheme:

In this scheme, only one buffer is used to store the input string. The problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan the rest of the lexeme the buffer must be refilled.

2.1.3.2 Two Buffer Scheme:

In this scheme two buffer are used to store input string and they are scanned alternately. When end of the current buffer is reached the other buffer is filled. The only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely.

Initially both the bp and fp are pointing to the first character of first buffer. Then the fp move forward in search of end of lexeme. As soon as blank character is recognized, the string between bp and fp is identified as corresponding token. To identify, the boundary of first buffer end of buffer character (eof) should be placed at the end of the first buffer, similarly for second buffer also. Alternatively, both the buffers can be filled up until end of the input program and stream of tokens is identified.



2.1.4 Specification of Tokens

2.1.4.1 Alphabet:

An alphabet is a finite, nonempty set of symbol. Conventionally, we use the symbol Σ for an alphabet. Common alphabet includes:

Σ ={0,1}, this is the binary alphabet.

Σ ={a,b,….z}, this is the set of all lower-case letters.

Power of an alphabet

If Σ is an alphabet, we can express the set of all strings of a certain length from that alphabet using an exponential notation. We define Σk to be the set of strings of length k, each of whose symbol is in Σ

For example: if {a,b,c}

$\Sigma^0 = \in$

Σ^1={a,b,c}

The set of all string over an alphabet is conventionally denoted by Σ*

Example Σ={0,1}

Σ*={0,1}*={∈,0,1,00,01,10,11,……}

Sometimes, we wish to exclude the empty string from the set of strings. The set of nonempty string from alphabet Σ is denoted by Σ^+.

Concentration of String

Let x and y be strings. Then xy denotes the concentration of x and y, that is the string formed by making a copy of x and following it by a copy of y.

Example: let x=1101 and y=0011 then xy=11010011

2.1.4.2 Strings:

A string is a finite sequence of symbols chosen from some alphabet.

For example, 01101 is a string from the binary alphabet Σ ={0,1}

- **Empty String**

The empty string is the string with zero occurance of symbol. It is denoted by ∈

- **Length of String**

The number of positions in a string is called its length. The notation for the length of a string S is |S|. For example, |01101|=5

2.1.4.3 Language

A language is a specific set of strings over some fixed alphabet Σ

Example:

a) ϕ the empty set is language.
b) {ε} the set containing empty string is a language
c) The set of well-formed c program is a language.
d) The set of all possible identifier is a language.

Operation on languages:

- Concatnation: L1L2={s1s2|s1 ∈ L1 and s2 ∈ L2}
- Union: L1UL2={s|s ∈ L1 or s ∈ L2}
- Exponential: L^0={∈} L^1=L L^2=LL
- Kleen Closure: L*= $\bigcup_{i=0}^{\infty} Li$

- Positive Closure $L^{+}=\bigcup_{i=1}^{\infty} Li$

2.1.4.4 Regular Expression

- Regular expression are the algebric expressions that are used to describe token of a programming language.
- Let Σ be an alphabet, the regular expression over the alphabet Σ are defined inductively as follows:
- **Basic Step:**
- ϕ is a regular expression representing the empty language i.e L(ϕ)= ϕ
- ∈ is a regular expression representing the language of empty string {∈} i.e; L(∈)={∈}
- If 'a' is a symbol in Σ, then 'a' is a regular expression representing the language {a} i.e; L(a)={a}

Example:

- 1(1+0) *0 denotes the language of all string that begins with a '1' and ends with a '0'.
- (1+0) *00 denotes the language of all strings that ends with 00.

2.1.4.5 Regular Definition

To write regular expression for some languages can be difficult because their regular expressions can be quite complex. In those cases, we may use *regular definitions*. The regular definition is a sequence of definitions of the form,

d1 → r1

d2 → r2

…….

dn → rn

Where di is a distinct name and ri is a regular expression over symbols in Σ∪ {d1, d2... di-1}

Where, Σ = Basic symbol and {d1, d2... di-1} = previously defined names.

Example:

Regular definition for specifying identifiers in a programming language like C.

letter → A | B | C |……| Z | a | b | c |……| z

underscore → '_'

digit →0 | 1 | 2 |……. | 9

id → (letter | underscore). (letter | underscore | digit) *

If we are trying to write the regular expression representing identifiers without using regular definition, it will be complex.

(A | B | C |……| Z | a | b | c |……| z | _ |). ((A | B | C |……| Z | a | b | c |……| z | _ |) (1 | 2 |……. | 9)) *

2.1.5 Recognition of Tokens

To recognize tokens lexical analyzer performs following steps:

A. Lexical analyzers store the input in input buffer.
B. The token is read from input buffer and regular expressions are built for corresponding token
C. From these regular expressions finite automata is built. Usually NFA is built.
D. For each state of NFA, a function is designed and each input along the transitional edges corresponds to input parameters of these functions.
E. The set of such functions ultimately create lexical analyzer program.

2.1.6 Finite Automata

Finite Automata (FA) is the simplest machine to recognize patterns. Finite Automata is a model of a computational system, consisting of a set of states, a set of possible inputs, and a rule to map each state to another state, or to itself, for any of the possible inputs. Formal specification of machine is {Q, ∑, q, F, δ } where

Q: Finite set of states,

∑: set of Input Symbols,

q: Initial state,

F: set of Final States

δ: Transition Function

Finite automata come in two flavors:

- non-deterministic finite automata (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and #, the empty string, is a possible label.
- Deterministic finite automata (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

2.1.6.1 DFA

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called Deterministic Automaton. As it has a finite number of states, the machine is called Deterministic Finite Machine or Deterministic Finite Automaton.

A DFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where.

- Q is a finite set of states.
- ∑ is a finite set of symbols calling the alphabet.
- δ is the transition function where δ: Q × ∑ → Q
- q0 is the initial state from where any input is processed (q0 Q). ∈ Q).
- F is a set of final state/states of Q (F Q). ⊆ Q).

Example

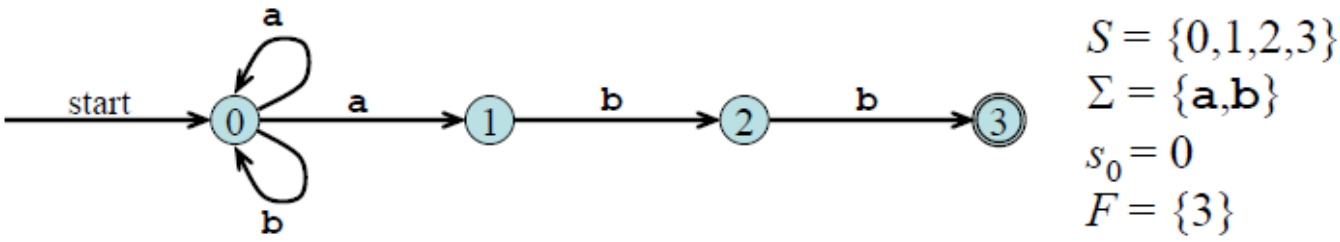


Fig: - NFA for regular expression (a + b)*a b b

2.1.6.2 ε- NFA

In NFA if a transition made without any input symbol is called ε-NFA. Here we need ε-NFA because the regular expressions are easily convertible to ε-NFA.

A ε -NFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where.

- Q is a finite set of states.
- ∑ is a finite set of symbols calling the alphabet.
- δ is the transition function where δ: Q × (∑U ε)→2^Q
- q0 is the initial state from where any input is processed (q0 Q). ∈ Q).
- F is a set of final state/states of Q (F Q). ⊆ Q).

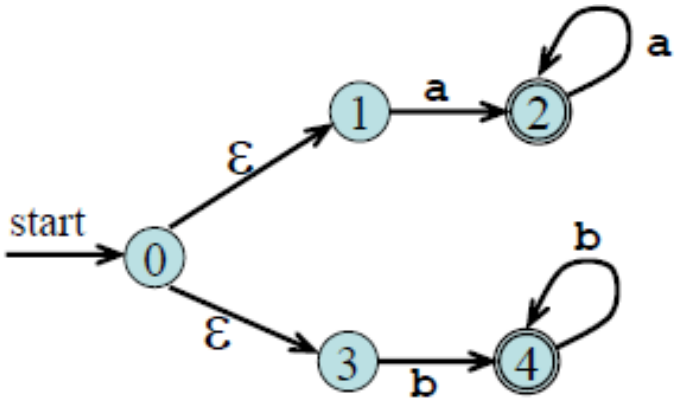


Fig: - ε-NFA for regular expression aa* +bb*

2.1.6.3 NFA

In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called Non-deterministic Automaton. As it has finite number of states, the machine is called Non-deterministic Finite Machine or Non-deterministic Finite Automaton.

A NFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where.

- Q is a finite set of states.
- ∑ is a finite set of symbols calling the alphabet.
- δ is the transition function where δ: Q × ∑ →2^Q
- q0 is the initial state from where any input is processed (q0 Q). ∈ Q).
- F is a set of final state/states of Q (F Q). ⊆ Q).

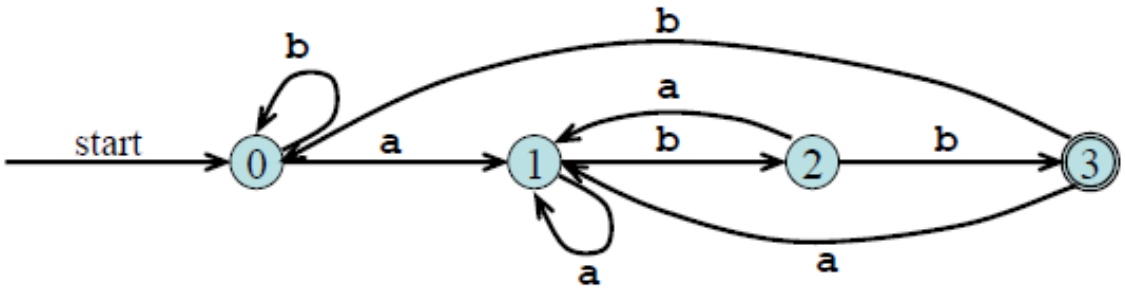
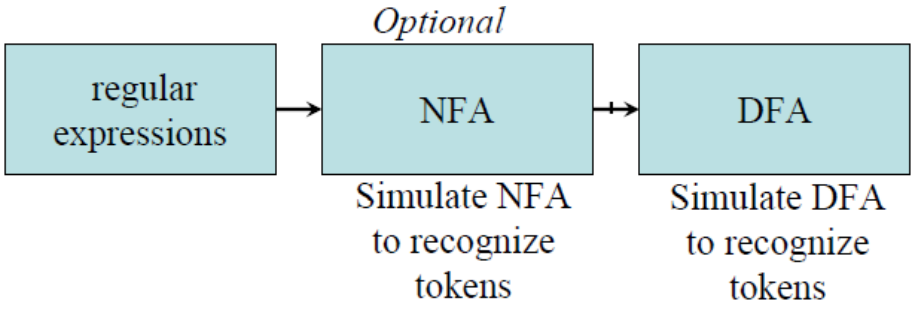


Fig:-DFA for regular expression **(a+b)*abb**

2.1.7 Design of Lexical Analyzer

First, we define regular expression for tokens, then we convert them into a DFA to get a lexical analyzer for our tokens.



Algorithm1:

- Regular Expression→ NFA → DFA (two steps: first NFA, then to DFA) **(Self-Study)**

Algorithm2:

- Regular Expression → DFA (Directly convert a regular expression into a DFA)

2.1.7.1 Conversion from RE to DFA Directly

To construct a DFA directly from a regular expression, we construct its syntax tree and then compute four functions: nullable , firstpos , lastpos , and followpos , defined as follows. Each definition refers to the syntax tree for a particular augmented regular expression (r) #.

A. nullable (n) is true for a syntax-tree node n if and only if the subexpression represented by n has # in its language. That is, the subexpression can be made null or the empty string, even though there may be other strings it can represent as well.
B. firrstpos (n) is the set of positions in the subtree rooted at n that correspond to the first symbol of at least one string in the language of the subexpression rooted at n.
C. lastpos (n) is the set of positions in the subtree rooted at n that correspond to the last symbol of at least one string in the language of the subexpression rooted at n.
D. followpos (p), for a position p , is the set of positions q in , the entire syntax # tree such that there is some string x = a1 a2 ..... an in L (r)# , such # that for some i , there is a way to explain the membership of x in L (r)# by matching ai to position p of the syntax tree and ai +1 to position q.

Conversion steps:

a) Augment the given regular expression by concatenating it with special symbol # i.e. r →(r) #
b) Create the syntax tree for this augmented regular expression

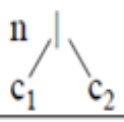
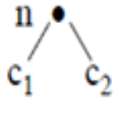
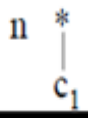
In this syntax tree, all alphabet symbols (plus # and the empty string) in the augmented regular expression will be on the leaves, and all inner nodes will be the operators in that augmented regular expression.

c) Then each alphabet symbol (plus #) will be numbered (position numbers)

d) Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*

e) Finally construct the DFA from the *followpos*

Rules for calculating nullable, firstpos and lastpos:

node n	nullable(n)	firstpos(n)	lastpos(n)
is leaf labeled ε	true	Φ	Φ
is leaf labeled with position i	false	{i} (position of leaf node)	{i}
	nullable(c_1) or nullable(c_2)	firstpos(c_1) ∪ firstpos(c_2)	lastpos(c_1) ∪ lastpos(c_2)
	nullable(c_1) and nullable(c_2)	**if** (nullable(c_1)) **then** firstpos(c_1) ∪firstpos(c_2) **else** firstpos(c_1)	**if** (nullable(c_2)) **then** lastpos(c_1) ∪ lastpos(c_2) **else** lastpos(c_2)
	true	firstpos(c_1)	lastpos(c_1)

Algorithm to evaluate followpos

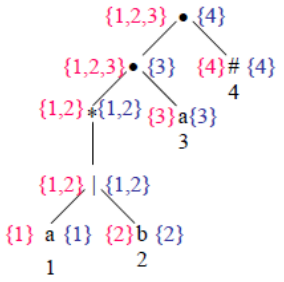
for each node *n* in the tree **do**
    if *n* is a cat-node with left child *c*1 and right child *c*2 **then**
        for each *i* in *lastpos*(*c*1) **do**
            followpos(*i*) := *followpos*(*i*) ∪ *firstpos*(*c*2)
        end do
    else if *n* is a star-node
        for each *i* in *lastpos*(*n*) **do**
            followpos(*i*) := *followpos*(*i*) ∪ *firstpos*(*n*)
        end do
    end if
end do

Example
Convert regular expression (a | b) * a into DFA.

Step-1: Its augmented regular expression is.

(a|b) *a#

Step-2: Construct syntax tree and calculate firstpos and followpos:



Step-3: Now we calculate followpos,

followpos(1)={1,2,3}
followpos(2)={1,2,3}
followpos(3)={4}
followpos(4)={}

Step-4: Compute State of a DFA with a help of followpos.

S_1=*firstpos(root)={1,2,3}*
mark S_1
for a: followpos(1) ∪ *followpos(3)={1,2,3,4}=*S_2 $move(S_1,a)=S_2$
for b: followpos(2)={1,2,3}=S_1 $move(S_1,b)=S_1$
mark S_2
for a: followpos(1) ∪ *followpos(3)={1,2,3,4}=*S_2 $move(S_2,a)=S_2$
for b: followpos(2)={1,2,3}=S_1 $move(S_2,b)=S_1$

Here Start state is S1 and the final state is S2.

Step-5: Design DFA

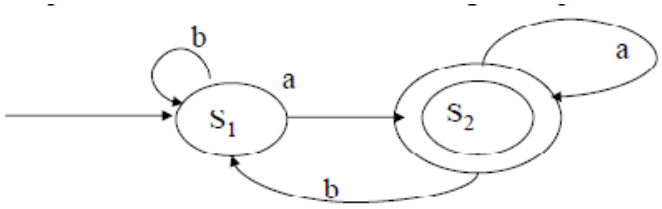


Fig: Resulting DFA of given regular expression

2.1.8 State Minimization in DFA

DFA minimization is the task of transforming a given deterministic finite automaton (DFA) into an equivalent DFA that has a minimum number of states. Here, two DFAs are called equivalent if they

recognize the same regular language. Several different algorithms accomplishing this task are known and described in standard textbooks on automata theory. For each regular language, there also exists a minimal automaton that accepts it, that is, a DFA with a minimum number of states and this DFA is unique (except that states can be given different names). The minimal DFA ensures minimal computational cost for tasks such as pattern matching.

Partition the set of states into two groups:

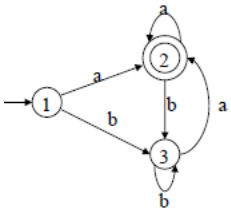
- G1: set of accepting states
- G2: set of non-accepting states

For each new group G:

partition G into subgroups such that states s1 and s2 are in the same group if for all input symbols a, states s1 and s2 have transitions to states in the same group.

- Start state of the minimized DFA is the group containing the start state of the original DFA.
- Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.

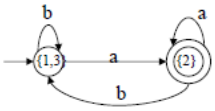
Example



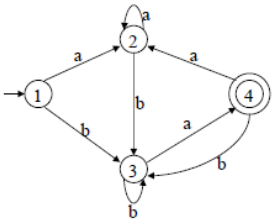
$G_1 = \{2\}$
$G_2 = \{1,3\}$
G_2 cannot be partitioned because

move(1,a)=2	move(1,b)=3
move(3,a)=2	move(2,b)=3

So, the minimized DFA (with minimum states)



Example 2:



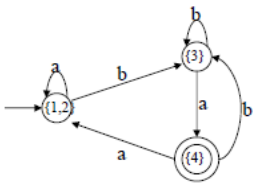
Groups: {1,2,3} {4}

{1,2} {3}

no more partitioning

a	b
1->2	1->3
2->2	2->3
3->4	3->3

So, the minimized DFA (with minimum states)


b
{3}
a
b
{1,2}
a
b
a
{4}