Unit 2
Lexical Analyzer

…………………………………………………………………………………………………

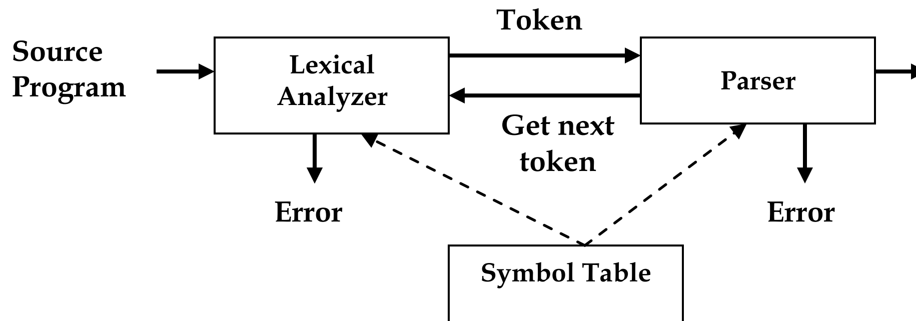
Topics

2.1 Lexical Analysis: The role of the lexical analyzer, Specification and Recognition of tokens, Input Buffer, Finite Automata relevant to compiler construction syntactic specification of languages, Optimization of DFA based pattern matchers

…………………………………………………………………………………………………

Lexical Analysis

The lexical analysis is the first phase of a compiler where a lexical analyzer acts as an interface between the source program and the rest of the phases of compiler. It reads the input characters of the source program, groups them into lexemes, and produces a sequence of tokens for each lexeme. The tokens are then sent to the parser for syntax analysis. Normally a lexical analyzer doesn't return a list of tokens; it returns a token only when the parser asks a token from it. Lexical analyzer may also perform other auxiliary operation like removing redundant white space, removing token separator (like semicolon) etc.



Example: Example of Lexical Analysis, Tokens, Non-Tokens

Consider the following code that is fed to Lexical Analyzer

```
#include <stdio.h>
int Largest(int x, int y)
{
    // This will compare 2 numbers
    if (x > y)
       return x;
    else
       return y;
}
```

Examples of Tokens created

Lexeme	Token
int	Keyword

largest	Identifier
(	Operator
int	Keyword
x	Identifier
,	Operator
int	Keyword
y	Identifier
)	Operator
{	Operator
if	Keyword

Examples of Non-tokens

Type	Examples
Comment	// This will compare 2 numbers
Pre-processor directive	#include <stdio.h>

Example: Let's take an expression as,

newval := oldval + 12

Lexeme	Tokens
Newval	Identifier
=	assignment operator
oldval	Identifier
+	add operator
12	a number

Lexical analyzer put information about identifiers into the symbol table. Regular expressions are used to describe tokens (lexical constructs). A (Deterministic) Finite State Automaton (DFA) can be used in the implementation of a lexical analyzer.

Role of Lexical Analyzer:

- It is the first phase of a compiler
- It reads the input character and produces output sequence of tokens that the Parser uses for syntax analysis.
- Lexical analyzer helps to identify token into the symbol table
- It can either work as a separate module or as a sub-module.
- Lexical Analyzer is also responsible for eliminating comments and white spaces from the source program.
- It also generates lexical errors.
- Lexical analyzer is used by web browsers to format and display a web page with the help of parsed data from JavsScript, HTML, CSS

Advantages of Lexical analysis

- Lexical analyzer method is used by programs like compilers which can use the parsed data from a programmer's code to create a compiled binary executable code
- It is used by web browsers to format and display a web page with the help of parsed data from JavsScript, HTML, CSS
- A separate lexical analyzer helps you to construct a specialized and potentially more efficient processor for the task.

Disadvantage of Lexical analysis

- We need to spend significant time reading the source program and partitioning it in the form of tokens
- Some regular expressions are quite difficult to understand compared to PEG or EBNF rules
- More effort is needed to develop and debug the lexer and its token descriptions
- Additional runtime overhead is required to generate the lexer tables and construct the tokens

Lexical analysis phase errors

The lexical analyzer must be able to cope with text that may not be lexically valid. For example

- A number may be too large
- A string may be too long or identifier may be too long
- A number may be incomplete
- The final quote on a sting may be missing
- The end of a comment may be missing
- A special symbol may be incomplete
- Invalid character may appear in the text
- Compiler directives may be invalid etc.

Tokens, Patterns, Lexemes

Lexemes

A lexeme is a sequence of alphanumeric characters that is matched against the pattern for a token. A sequence of input characters that make up a single token is called a lexeme. A token can represent more than one lexeme. The token is a general class in which lexeme belongs to.

Example: The token "String constant" may have a number of lexemes such as "bh", "sum", "area", "name" etc.

Thus lexeme is the particular member of a token which is a general class of lexemes.

Patterns

Patterns are the rules for describing whether a given lexeme belonging to a token or not. The rule associated with each set of string is called pattern. Lexeme is matched against pattern to generate token. Regular expressions are widely used to specify patterns.

Token

Token is word, which describes the lexeme in source program. It is generated when lexeme is matched against pattern. A **token** is a logical building block of language. They are the sequence of characters having a collective meaning.

Example 1: Example showing lexeme, token and pattern for variables

- **Lexeme**: A1, Sum, Total
- **Pattern**: Starting with a letter and followed by letter or digit but not a keyword.
- **Token**: ID

Example 2: Example showing lexeme, token and pattern for floating number

- **Lexeme:** 123.45
- **Pattern**: Starting with digit followed by a digit or optional fraction and or optional exponent
- **Token**: NUM

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.

A token describes a pattern of characters having same meaning in the source program such as identifiers, operators, keywords, numbers, delimiters and so on. A token may have a single attribute which holds the required information for that token. For identifiers, this attribute is a pointer to the symbol table and the symbol table holds the actual attributes for that token. Token type and its attribute uniquely identify a lexeme. Regular expressions are widely used to specify pattern.

Attributes of Tokens

When a token represents more than one lexeme, lexical analyzer must provide additional information about the particular lexeme. This additional information is called as the attribute of the token. For simplicity, a token may have a single attribute which holds the required information for that token.

Example: the tokens and the associated attribute for the following statement.

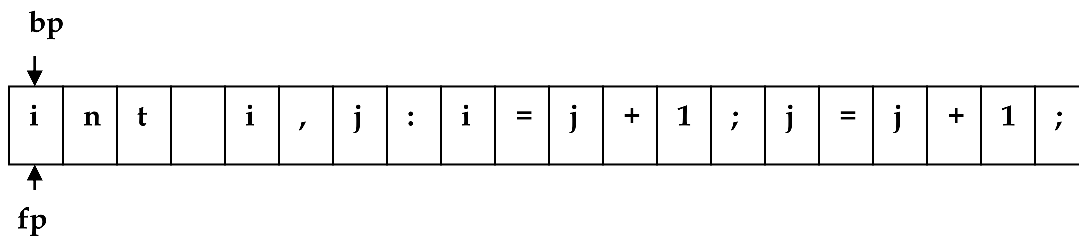
A=B*C+2

<id, pointer to symbol table entry for A>
<Assignment operator>
<id, pointer to symbol table entry for B>
<mult_op>
<id, pointer to symbol table entry for C>
<add_op>
<num, integer value 2>

Input Buffering

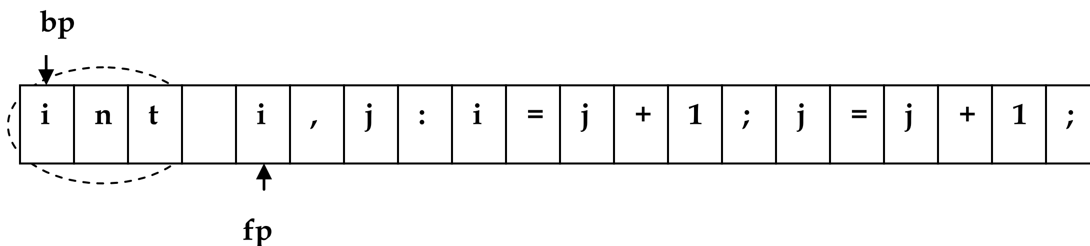
Reading character by character from secondary storage is slow process and time consuming as well. It is necessary to look ahead several characters beyond the lexeme for a pattern before a match can be announced. One technique is to read characters from the source program and if pattern is not matched then push look ahead character back to the source program. This technique is time consuming. 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 begin ptr (bp) and forward to keep track of the pointer of the input scanned. Initially both the pointers point to the first character of the input string as shown below,



The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as ptr (fp) encounters a blank space the lexeme 'int' is identified.

The fp will be moved ahead at white space, when fp encounters white space, it ignores and moves ahead. Then both the begin ptr (bp) and forward ptr (fp) are set at next token. The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. Hence buffering technique is used. A block of data is first read into a buffer, and then second by lexical analyzer. There are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme. These are explained as following below.



1. One Buffer Scheme

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

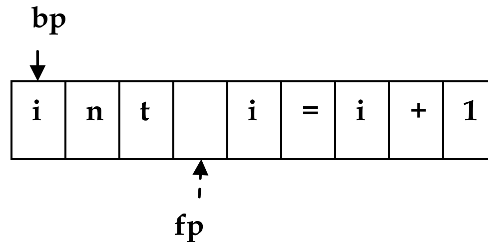


Figure: One buffer scheme storing input string

2. Two Buffer Scheme

To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. The first buffer and second buffer are scanned alternately. When end of 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 moves towards right 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 should be placed at the end first buffer.

Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer. When fp encounters first **eof**, then one can recognize end of first buffer and hence filling up second buffer is started. In the same way when second eof is obtained then it indicates of second buffer. Alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified.

This **eof** character introduced at the end is calling Sentinel which is used to identify the end of buffer.

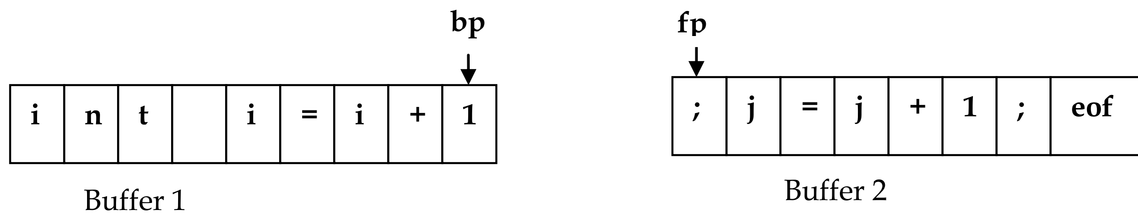


Figure: Two buffer scheme storing input string

Specifications of Tokens

Regular expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions will serve as names for sets of strings. In brief a regular expression is a way to specify tokens. The regular expression represents the regular languages. The language is the set of strings and string is the set of alphabets. Thus following terminologies are used to specify tokens:

a. Alphabets, Strings and Languages
b. Operations on languages
c. Regular expressions
d. Regular definition

Alphabets

The set of symbols is called alphabets. Example 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.

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 'Kanchanpur' is 10 and is denoted by |Kanchanpur|=10. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

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.

Operations on languages

The following are the operations that can be applied to languages:

- Union
- Concatenation
- Kleene closure
- Positive closure

Union

The symbol ∪ is employed to denote the union of two sets. Thus, the set A∪B read "A union B" or "the union of A and B" is defined as the set that consists of all elements belonging to either set A or set B (or both). In regular expression plus (+) symbol is used to represent union operation. Let A and B be two languages, where A = {dog, ba, na} and B = {house, ba} then,

AUB = A+B=A|B = {dog, ba, na, house}

Concatenation

String concatenation is the operation of joining character strings end-to-end. In regular expression dot operator (.) is used to represent concatenation operation.

Example: A.B = {doghouse, dogba, bahouse, baba, nahouse, naba}

Kleene Closure

The Kleene closure, $\sum^*$, is a unary operator on a set of symbols or strings, $\sum$, that gives the infinite set of all possible strings of all possible lengths over $\sum$ including {Φ}.

Mathematically, $\sum^* = \sum^0 \cup \sum^1 \cup \sum^2 \cup \ldots\ldots$ where $\sum^n$ is the set of all possible strings of length n.

Example: If $\sum$ = {a, b} then,

$\sum^*$ = {Φ, a, b, aa, ab, ba, bb, aaa, aba, bab, aab, bba, bbb,…….}

$a^* = \{a^0 \cup a^1 \cup a^2 \cup a^3 \cup \ldots a^n\}$={Φ, a, aa, aaa, aaaa, aaaaa, …………..}

$b^* = \{b^0 \cup b^1 \cup b^2 \cup b^3 \cup \ldots b^n\}$={Φ, b, bb, bbb, bbbb, bbbbb, ………..}

$a.b^*$ = {a}. {Φ, b, bb, bbb, bbbb, bbbbb, ………..}

{a, ab, abb, abbb, abbbb, abbb…….}

$(ab)^* = =\{(ab)^0 \cup (ab)^1 \cup (ab)^2 \cup (ab)^3 \cup \ldots (ab)^n\}$={Φ, ab, abab, ababab,………}

$a^*.b^*$ = {Φ, a, aa, aaa, aaaa, aaaaa, …..}. {Φ, b, bb, bbb, bbbb, bbbbb,..…}

= {Φ, b, bb, bbb, bbbb, bbbbb, a, ab, abb, abbb, abbbb, abbbbb, aa, aab, aabb, aabbb, aaa, aaab, aaabb,………}

$(a+b)^*$ = {Φ, a, b, aa, aaa, aaaa, bb, bbb, bbbb, ab, ba, abababa, bababa, aaaab, baaaa,…………..}

Positive closure

The set $\sum$+ is the infinite set of all possible strings of all possible lengths over $\sum$ excluding {Φ}.

Mathematically, $\sum^+ = \sum^1 \cup \sum^2 \cup \sum^3 \cup \ldots\ldots$ where $\sum^n$ is the set of all possible strings of length n.

Example: If $\sum$ = {a, b} then,

$\sum^+ = \sum^* - \{\Phi\}$

$\sum^+$ = {a, b, aa, ab, ba, bb, aaa, aba, bab, aab, bba, bbb,…….}

$a^+ = \{a^1 \cup a^2 \cup a^3 \cup \ldots a^n\}$={a, aa, aaa, aaaa, aaaaa, …………..}

$b^+ = \{b^1 \cup b^2 \cup b^3 \cup \ldots b^n\}$={b, bb, bbb, bbbb, bbbbb, ………..}

$a.b^+$ = {a}.{b, bb, bbb, bbbb, bbbbb, ………..}

= {ab, abb, abbb, abbbb, abbb…….}

$(ab)^+ = \{(ab)^1 \cup (ab)^2 \cup (ab)^3 \cup \ldots (ab)^n\}$={ab, abab, ababab, abababab………}

$a^+.b^+$ = {a, aa, aaa, aaaa, aaaaa, …….}.{b, bb, bbb, bbbb, bbbbb, ………..}

= {ab, abb, abbb, abbbb, aab, aabb, aabbb, aabbbb, aaab, aaabb, aaabbb…..}

$(a+b)^+$ = {a, b, aa, aaa, aaaa, bb, bbb, bbbb, ab, ba, abababa, bababa, aaaab, baaaa,……}

Regular Expressions

Regular expressions are the algebraic expressions that are used to describe tokens of a programming language. It uses the three regular operations. These are called union/or, concatenation and star. Brackets (and) are used for grouping, just as in normal math.

Examples: Given the alphabet A = {0, 1}

1. 1(1+0)*0 denotes the language of all string that begins with a '1' and ends with a '0'.
2. (1+0)*00 denotes the language of all strings that ends with 00 (binary number multiple of 4)
3. (01)*+ (10)* denotes the set of all stings that describe alternating 1s and 0s
4. (0* 1 0* 1 0* 1 0*) denotes the string having exactly three 1's.
5. 1*(0+ ε)1*(0+ ε) 1* denotes the string having at most two 0's
6. (A | B | C |……… | Z | a | b | c |……… | z | _ |). (A | B | C |……… | Z | a | b | c |……… | z | _ | 1 | 2 |……………| 9)* denotes the regular expression to specify the identifier like programming in C. [TU]
7. (1+0)* 001 (1+0)* denotes string having substring 001
8. 0(0 + 1)* 0 + 1(0 + 1)*1 denotes the RE for the language of all binary strings of length at least 2 that begin and end in the same symbol.
9. ((0 + 1)* 1 + ε) (00)* 00 denotes the RE for the set of all binary strings that end with an even nonzero number of 0's.
10. Regular expression for declaration of valid one dimensional array in programming like C [TU]

    ∑= (0,1…..9,a,b…z, A, B……..Z,_,[,])

RE: (a|b|…..|z|A|B|….|Z|_) (a|b|…..z|A |B|….|Z|_|0|1|…|9)* . [(1|2|….|9) (0|1|2|….|9)*]

11. Regular expression for declaration of valid two dimensional array in programming like C [TU]

    ∑= (0,1…..9,a,b…z, A, B……..Z,_,[,])

RE: (a|b|…..|z|A|B|….|Z|_) (a|b|…..z|A |B|….|Z|_|0|1|…|9)* . [(1|2|….|9) (0|1|2|….|9)*]. [(1|2|….|9) (0|1|2|….|9)*]

12. Regular expression for declaration of valid floating numbs. [TU]

    RE: (0|1|2|….|9)* '.' (0|1|2|….|9)$^+$

Regular Definitions

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*.** Giving names to regular expressions is referred to as a Regular definition. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form,

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

…………….

$d_n \rightarrow r_n$

Where, d_i is a distinct name and r_i is a regular expression over symbols in $\Sigma \cup \{d_1, d_2 \dots d_{i-1}\}$

Σ = Basic symbol and $\{d_1, d_2... d_{i-1}\}$ = previously defined names.

Example 1: 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))*

Example 2: Write regular definition for specifying floating point number in a programming language like C

digit → 0 | 1 | 2 |……………. | 9

num→ digit * (.) (digit)$^+$

Example 3: Write regular definitions for specifying an integer one dimensional array declaration in programming language like in C

Lc → a|b|…..|z

Uc → A|B|…..|Z

Digit →1|2|…….|9

Zero → 0

Us →_

Lb → [

Rb →]

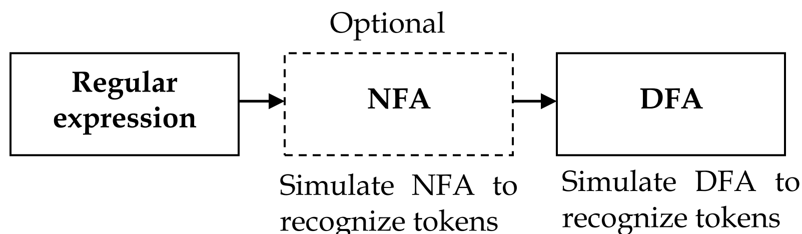
Array → (Lc|Uc|Us) (Lc|Uc|Us|Digit|zero)* Lb. Digit.(Digit|zero)* Rb

Design of a Lexical Analyzer

First, we define regular expressions 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 to NFA, then to DFA)

Algorithm2: Regular Expression → DFA (directly convert a regular expression into a DFA)



The finite automat is used to recognize the token. An automaton with a finite number of states is called a Finite Automaton (FA) or Finite State Machine (FSM).

Formal definition of a Finite Automaton

An automaton can be represented by a 5-tuple (Q, ∑, δ, q_0, F),

Where,

- Q is a finite set of states.
- ∑ is a finite set of symbols, called the alphabet of the automaton.
- δ is the transition function.
- q_0 is the initial state from where any input is processed ($q_0 \in Q$).
- F is a set of final state/states of Q ($F \subseteq Q$).

Finite Automaton can be classified into two types:

- Deterministic Finite Automaton (DFA)
- Non-deterministic Finite Automaton (NDFA / NFA)

Non-Deterministic Finite Automaton (NFA)

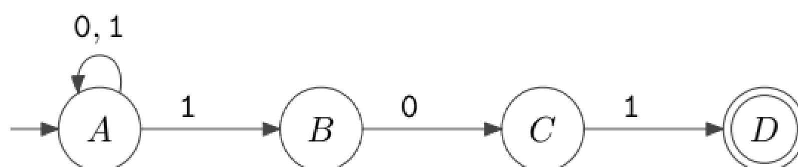
In NFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called Non-deterministic Automaton.

FA is non deterministic, if there is more than one transition for each (state, input) pair. It is slower recognizer but it make take less spaces. An NFA is a 5-tuple (Q, Σ, δ, q_0, F) where,

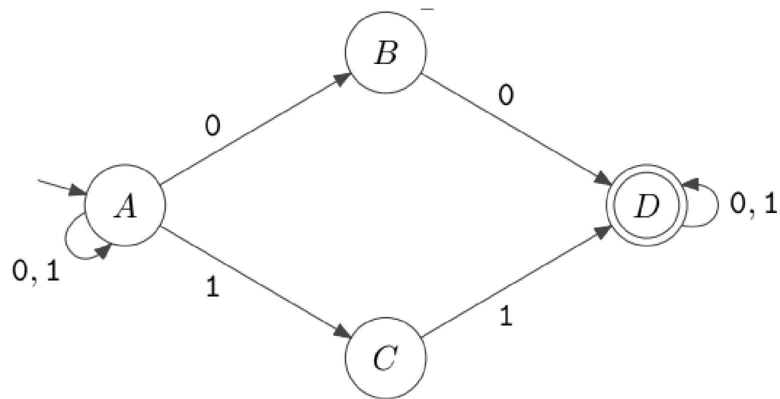
- Q is a finite set of states
- Σ is a finite set of symbols
- δ is a transition function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accepting (or final) states

A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states.

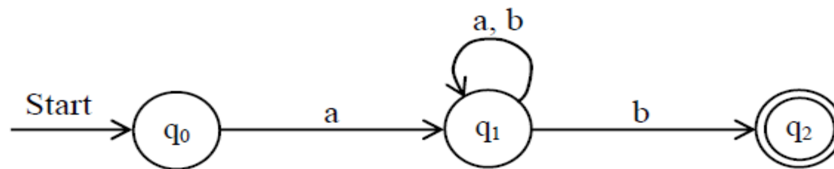
Example 1: An NFA that accepts all binary strings that end with 101.



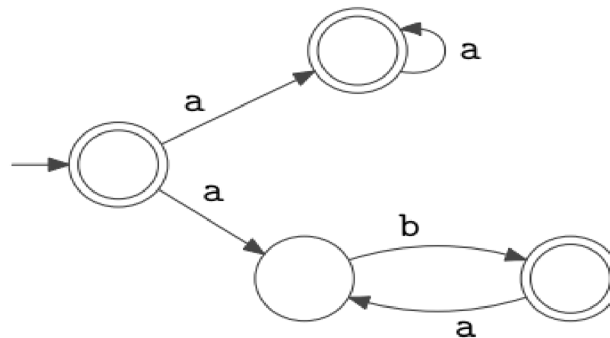
Example 2: An NFA that accepts any binary string that contains 00 or 11 as a substring.



Example 3: NFA over {a, b} that accepts strings starting with a and ending with b.



Example 4: An NFA for a ∗ + (ab) ∗



ε- NFA

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

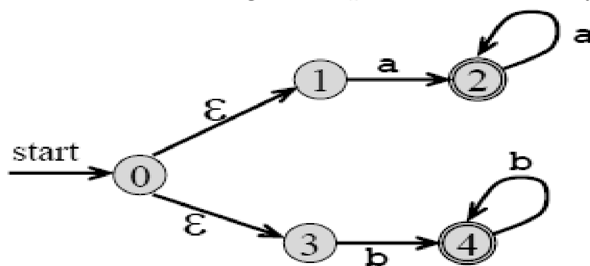
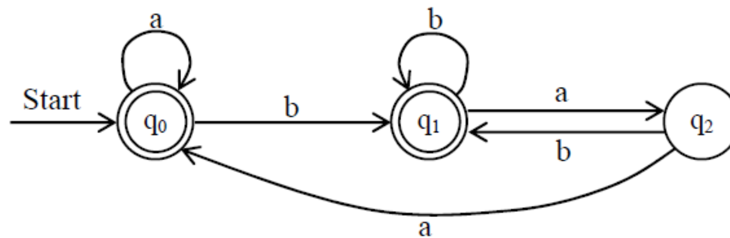


Fig: - ε-NFA for regular expression aa* +bb*

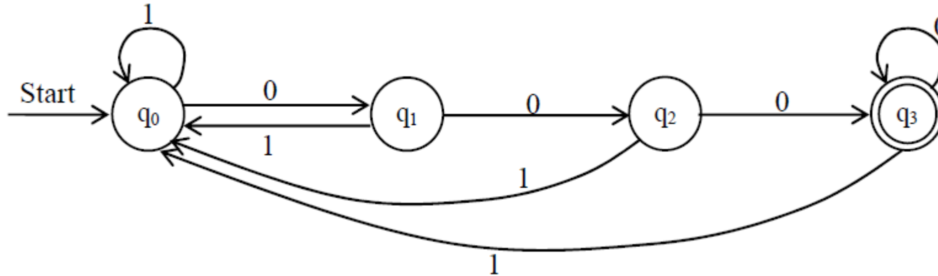
Deterministic Finite Automaton (DFA)

DFA is a special case of NFA. There is only difference between NFA and DFA is in the transition function. In NFA transition from one state to multiple states take place while in DFA transition from one state to only one possible next state take place.

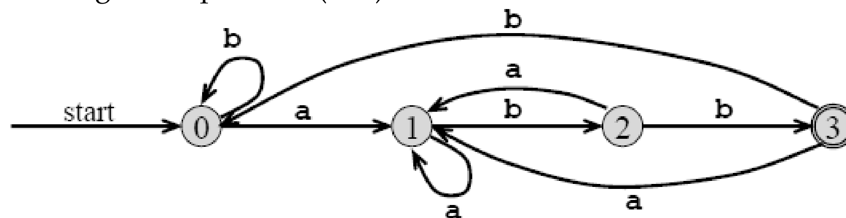
Example 1: DFA that accepts all the strings over Σ = {a, b} that do not end with ba



Example 2: DFA accepting all string over Σ = {0, 1} ending with 3 consecutive 0's.



Example 3: DFA for regular expression (a+b)*abb



Conversion: Regular Expression to NFA

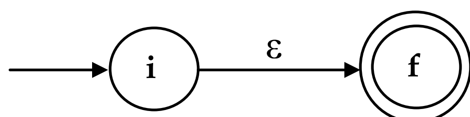
Thomson's Construction

Thomson's Construction is simple and systematic method. It guarantees that the resulting NFA will have exactly one final state, and one start state. It is a process in bottom up manner by creating ε-NFA for each symbol including. Then recursively create for other operations as shown below,

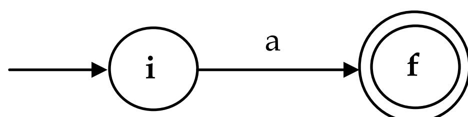
Method

- First parse the regular expression into sub-expressions
- Construct NFA's for each of the basic symbols in regular expression (r)
- Finally combine all NFA's of sub-expressions and we get required NFA of given regular expression.

1. To recognize an empty string ε

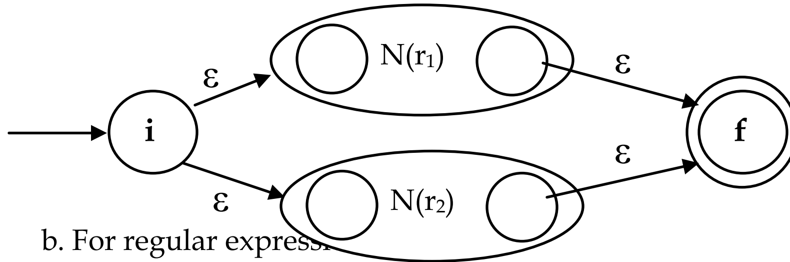


2. To recognize a symbol a in the alphabet Σ

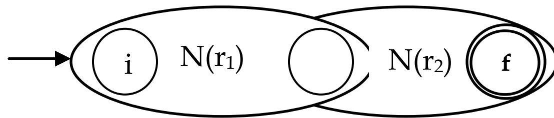


3. If N (r_1) and N (r_2) are NFAs for regular expressions r_1 and r_2

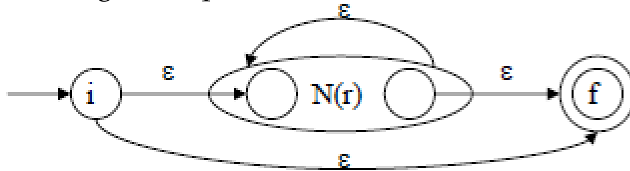
a. For regular expression $r_1 + r_2$



b. For regular express



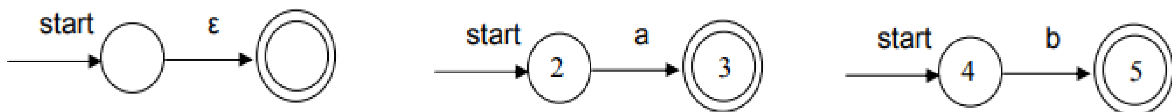
c. For regular expression r*



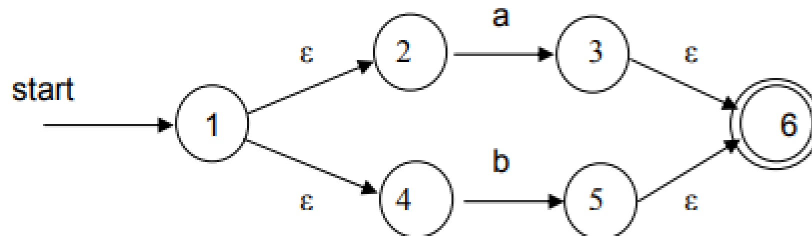
Using rule 1 and 2 we construct NFA's for each basic symbol in the expression, we combine these basic NFA using rule 3 to obtain the NFA for entire expression.

Example 1: NFA construction of RE **(a + b) * abb**

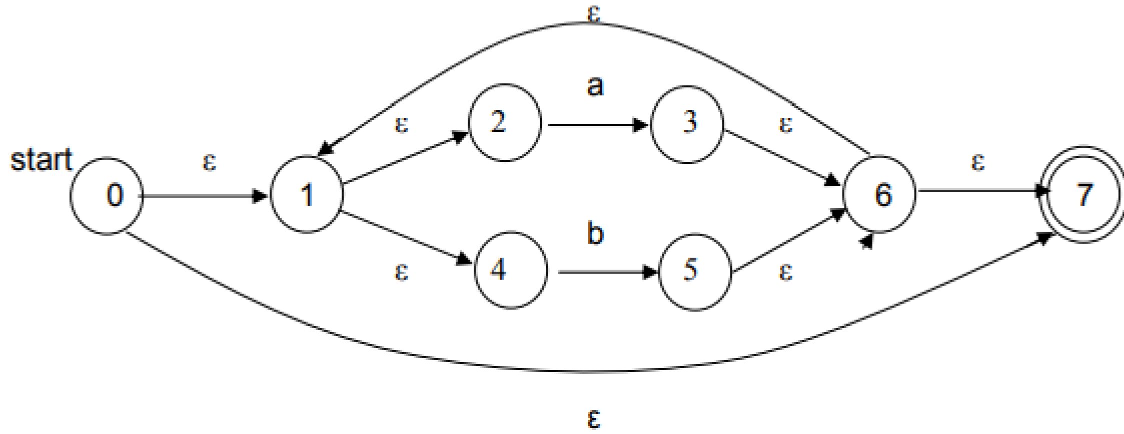
a. the NFA's for single character regular expressions ε, a, b



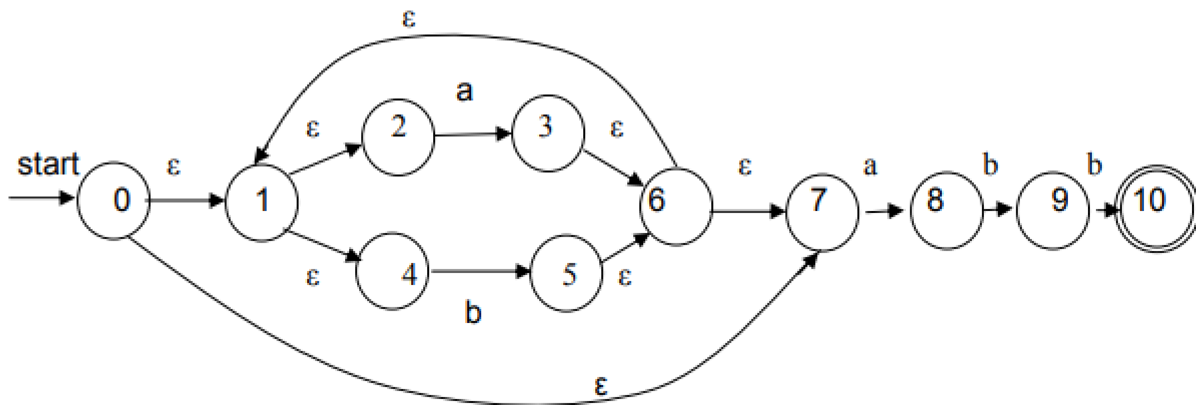
b. The NFA for the union of 'a' and 'b': a|b is constructed from the individual NFA's using the ε NFA as 'glue'. Remove the individual accepting states and replace with the overall accepting state,



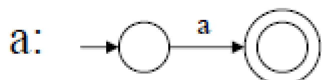
c. Kleene closure on (a|b)*. The NFA accepts ε in addition to (a|b)*

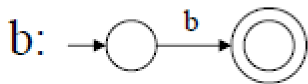


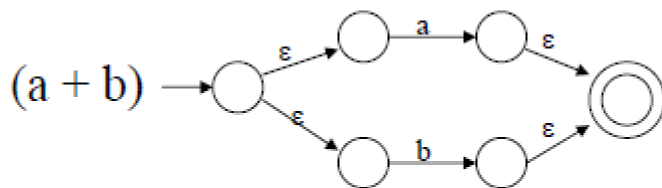
d. concatenate with abb

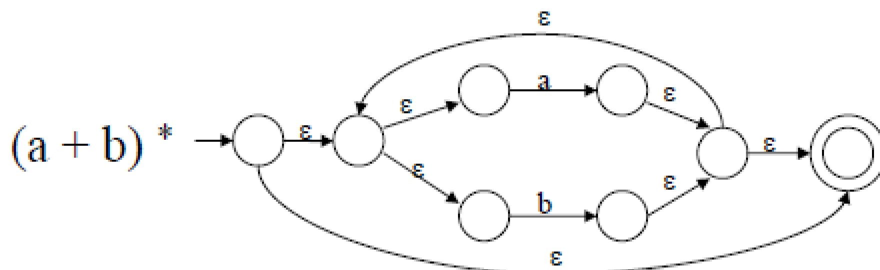


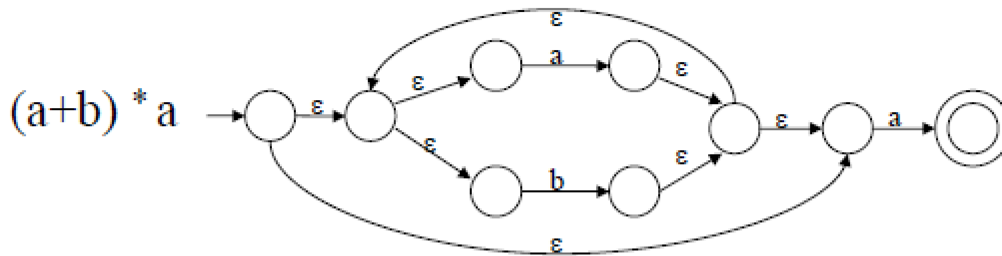
Example 2: NFA construction of RE **(a + b) * a**











Conversion from NFA to DFA

A Deterministic Finite Automaton (DFA) has at most one edge from each state for a given symbol and is a suitable basis for a transition table. We need to eliminate the ε-transitions by subset construction.

Subset Construction Algorithm

Put ε-closure (s_0) as an unmarked state in to Dstates

While there is an unmarked state T in Dstates do

mark *T*

for each input symbol $a \in \Sigma$ **do**

U = ε-closure (move (T, a))

if U is not in Dstates **then**

Add U as an unmarked state to Dstates

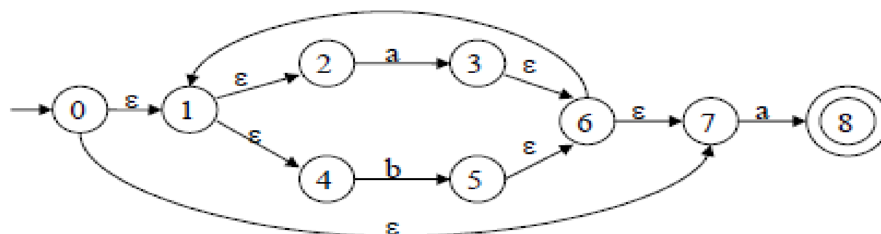
end if

Dtran[T, a] = U

end do

end do

Example 1: At first construct NFA of RE **(a+b)*a** then convert resulting NFA to DFA.
Solution: The NFA of Given RA is given below,



Now convert above NFA to DFA as,

The starting state of DFA = S_0 = ε-closure (Starting state of NFA) = ε-closure ({0})

= {0, 1, 2, 4, 7}

⇨ S_0 = {0, 1, 2, 4, 7}

Mark S_0,

For a: ε-closure (move (S_0, a)) = ε-closure (3, 8) = {1, 2, 3, 4, 6, 7, 8}➔S_1

For b: ε-closure (move (S_0, b)) = ε-closure (5) = {1, 2, 4, 5, 6, 7}➔S_2

Mark S_1,

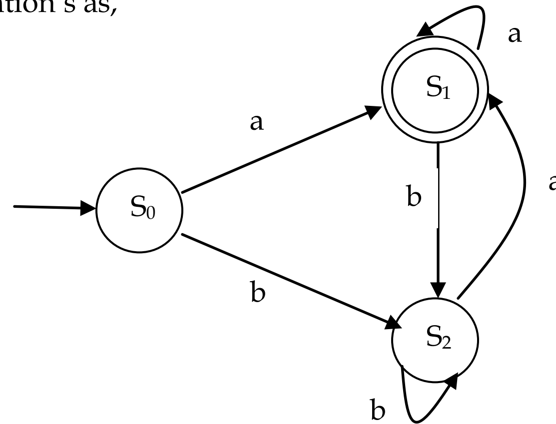
For a: ε-closure (move (S_1, a)) = ε-closure (3, 8) = {1, 2, 3, 4, 6, 7, 8}➔S_1
For b: ε-closure (move (S_1, b)) = ε-closure (5) = {1, 2, 4, 5, 6, 7}➔S_2

Mark S_2,
For a: ε-closure (move (S_2, a)) = ε-closure (3, 8) = {1, 2, 3, 4, 6, 7, 8}➔S_1
For b: ε-closure (move (S_2, b)) = ε-closure (5) = {1, 2, 4, 5, 6, 7}➔S_2

S_0 is the start state of DFA since 0 is a member of S_0= {0, 1, 2, 4, 7}

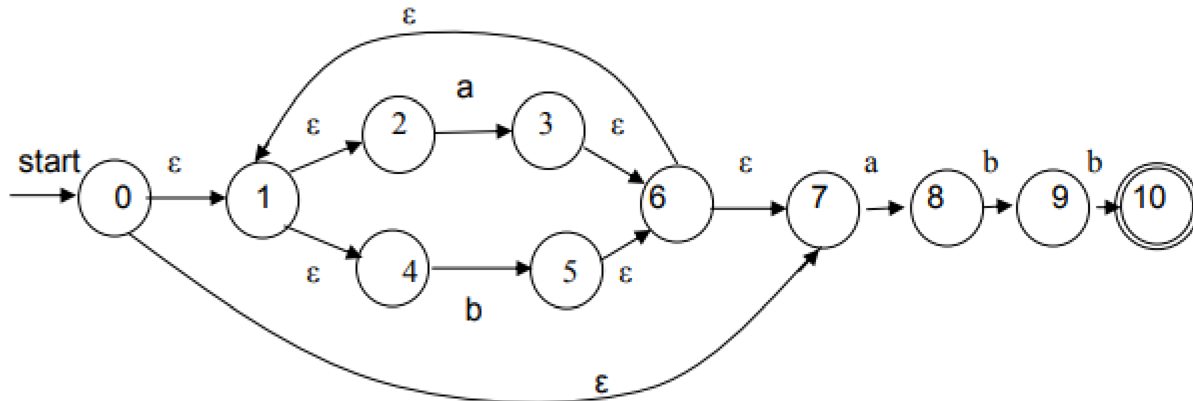
S_1 is an accepting state of DFA since final state of NFA i.e. 8 is a member of S_1 = {1, 2, 3, 4, 6, 7, 8}

Now construct DFA from above information's as,



Example 2: At first construct NFA of RE **(a+b)*abb** then convert resulting NFA to DFA.

Solution: The NFA of Given RA is given below,



Now convert above NFA to DFA as,

The starting state of DFA = S_0 = ε-closure (Starting state of NFA) = ε-closure ({0})

= {0, 1, 2, 4, 7}

⇨ S_0 = {0, 1, 2, 4, 7}

Mark S_0,
For a: ε-closure (move (S_0, a)) = ε-closure (3, 8) = {1, 2, 3, 4, 6, 7, 8}➔S_1
For b: ε-closure (move (S_0, b)) = ε-closure (5) = {1, 2, 4, 5, 6, 7}➔S_2

Mark S_1,
For a: ε-closure (move (S_1, a)) = ε-closure (3, 8) = {1, 2, 3, 4, 6, 7, 8}➔S_1
For b: ε-closure (move (S_1, b)) = ε-closure (5, 9) = {1, 2, 4, 5, 6, 7, 9}➔S_3

Mark S_2,
For a: ε-closure (move (S_2, a)) = ε-closure (3, 8) = {1, 2, 3, 4, 6, 7, 8}➔S_1
For b: ε-closure (move (S_2, b)) = ε-closure (5) = {1, 2, 4, 5, 6, 7}➔S_2

Mark S_3,

For a: ε-closure (move (S_3, a)) = ε-closure (3, 8) = {1, 2, 3, 4, 6, 7, 8}➔S_1

For b: ε-closure (move (S_3, b)) = ε-closure (5, 10) = {1, 2, 4, 5, 6, 7, 10}➔S_4

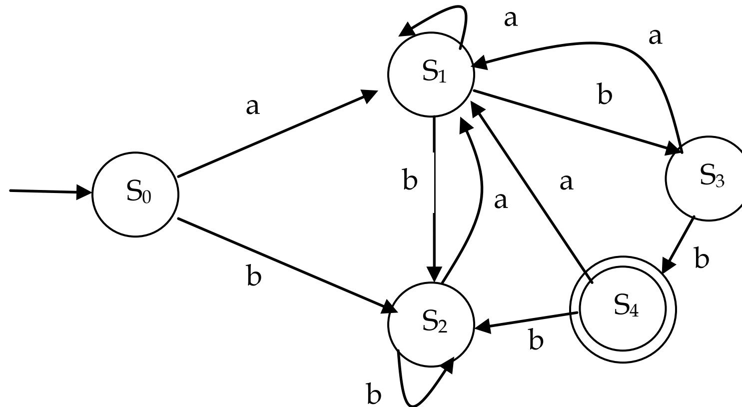
Mark S_4,

For a: ε-closure (move (S_4, a)) = ε-closure (3, 8) = {1, 2, 3, 4, 6, 7, 8}➔S_1

For b: ε-closure (move (S_4, b)) = ε-closure (5) = {1, 2, 4, 5, 6, 7}➔S_2

S_0 is the start state of DFA since 0 is a member of S_0= {0, 1, 2, 4, 7}

S_4 is an accepting state of DFA since final state of NFA i.e. 10 is a member of S_4 = {1, 2, 3, 4, 6, 7, 10}. Now construct DFA from above information's as,



Conversion from RE to DFA Directly

Important States

The state s in ε -NFA is an important state if it has no null transition. In optimal state machine all states are important states. Simply, a state S of an NFA without ε- transition is called the important state if,

Move ({s}, a) ≠ Φ

Augmented Regular Expression

When we construct an NFA from the regular expression then the final state of resulting NFA is not an important state because it has no transition. Thus to make important state of the accepting state of NFA we introduce an augmented character (#) to a regular expression r. This resulting regular expression is called the augmented regular expression of original expression r.

Conversion steps

1. Augment the given regular expression by concatenating it with special symbol #
   I.e. r →(r) #
2. Create the syntax tree for this augmented regular expression
3. 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.

4. Then each alphabet symbol (plus #) will be numbered (position numbers)
5. Compute functions ***nullable, firstpos, lastpos,*** and ***followpos***
6. Finally Construct DFA directly from a regular expression by computing the functions nullable(n), firstpos(n), lastpos(n) and followpos(i) from the syntax tree.
   - **nullable** (n): Is true for * node and node labeled with Ɛ. For other nodes it is false.
   - **firstpos** (n): Set of positions at node ti that corresponds to the first symbol of the sub-expression rooted at n.
   - **lastpos** (n): Set of positions at node ti that corresponds to the last symbol of the sub-expression rooted at n.
   - **followpos** (i): Set of positions that follows given position by matching the first or last symbol of a string generated by sub-expression of the given regular expression.

Rules for calculating nullable, firstpos and lastpos

Node n	nullable *(n)*	firstpos *(n)*	lastpos *(n)*
A leaf labeled Ɛ	True	Ø	Ø
A leaf with position i	False	{i}	{i}
An *or* node $n = c_1 \mid c_2$	Nullable (c_1) or Nullable (c_2)	firstpos (c_1) U firstpos (c_2)	lastpos (c_1) U lastpos (c_2)
A cat node $n = c_1.c_2$	Nullable (c_1) and Nullable (c_2)	If (Nullable (c_1)) firstpos (c_1) U firstpos (c_2) else firstpos (c_1)	If (Nullable (c_2)) lastpos (c_1) U Iastpos (c_2) else lastpos (c_2)
A star node $n = c_1^*$	True	firstpos (c_1)	lastpos (c_1)
A +ve closure node $n = c_1^+$	False	firstpos (c_1)	lastpos (c_1)

Computation of followpos Steps

The position of regular expression can follow another in the following ways:

- If n is a cat node with left child c_1 and right child c_2, then for every position i in lastpos(c_1), all positions in firstpos(c_2) are in followpos(i).
- For cat node, for each position i in lastpos of its left child, the firstpos of its right child will be in followpos(i).
- If n is a star node and i is a position in lastpos(n), then all positions in firstpos(n) are in followpos(i).
- For star node, the firstpos of that node is in followpos of all positions in lastpos of that node.

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(c1) do
                        followpos(i) := followpos(i) ∪ firstpos(c2)
                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
```

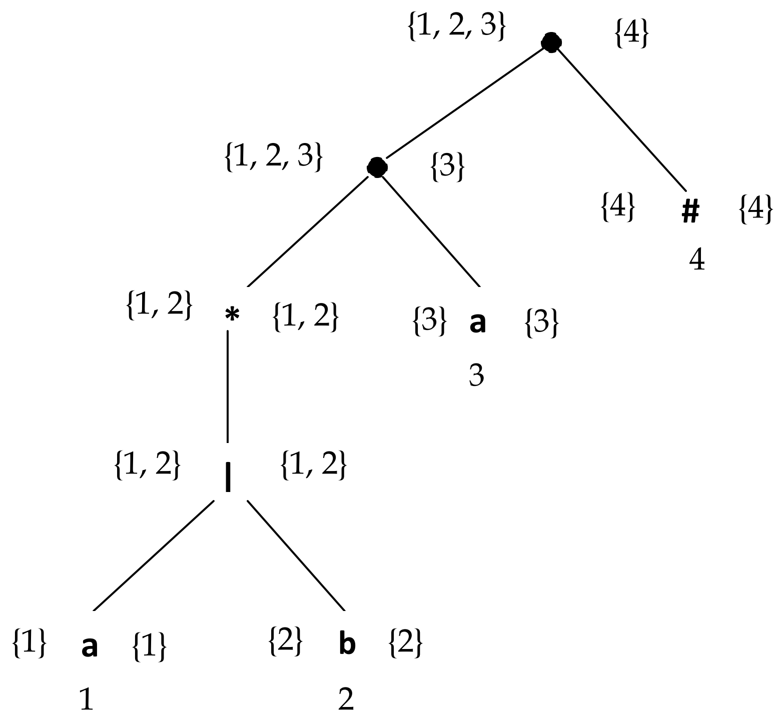
Example 1: Convert the regular expression **(a | b)* a** into equivalent DFA by direct method.

Solution:

Step 1: At first augment the given regular expression as,

(a | b)* . a. #

Step 2: Now construct syntax tree of augmented regular expression as,



Step 3: Compute followpos as,

Followpos(1) = {1, 2, 3}　　　　Followpos(2) = {1, 2, 3}

Followpos(3) = {4}　　　　Followpos(4) = {Φ}

Step 4: After we calculate follow positions, we are ready to create DFA for the regular expression as,

Starting state of DFA = S_1 = Firstpos(Root node of Syntax tree) = {1, 2, 3}

Mark S_1

For a: follwpos(1) U followpos(3) = {1, 2, 3, 4}➔ S_2

For b: followpos(2) = {1, 2, 3} ➔ S_1

Mark S_2

For a: follwpos(1) U followpos(3) = {1, 2, 3, 4}➔ S_2

For b: followpos(2) = {1, 2, 3} ➔ S_1

Now there was no new states occur.

So starting state of DFA = {S_1}

And accepting state of DFA = {S_2}

Note: Accepting states = states containing position of # i.e.

Figure: Resulting DFA of given regular expression

Example 2: Convert the regular expression (a | ε) b c* into equivalent DFA by direct method.

Solution:

Step 1: At first augment the given regular expression as,

(a | ε) b c* #

Step 2: Now construct syntax tree of augmented regular expression as,

Step 3: Compute followpos as,

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

Followpos(3) = {3, 4}   Followpos(4) = {Φ}

Step 4: After we calculate follow positions, we are ready to create DFA for the regular expression as,

Starting state of DFA = S_1 = Firstpos(Root node of Syntax tree) = {1, 2}

Mark S_1

For a: follwpos(1) = {2}➔ S_2
For b: followpos(2) = {3, 4} ➔ S_3
For c: followpos(Φ) = {Φ} ➔ S_4

Mark S_2

For a: follwpos(Φ) = {Φ} ➔ S_4
For b: followpos(2) = {3, 4} ➔ S_3
For c: followpos(Φ) = {Φ} ➔ S_4

Mark S_3

For a: follwpos(Φ) = {Φ} ➔ S_4
For b: followpos(Φ) = {Φ} ➔ S_4
For c: followpos(3) = {3, 4} ➔ S_3

Mark S_4

For a: follwpos(Φ) = {Φ} ➔ S_4
For b: followpos(Φ) = {Φ} ➔ S_4
For c: followpos(Φ) = {Φ} ➔ S_4

Now there was no new states occur.

So starting state of DFA = {S_1}

And accepting state of DFA = {S_3}

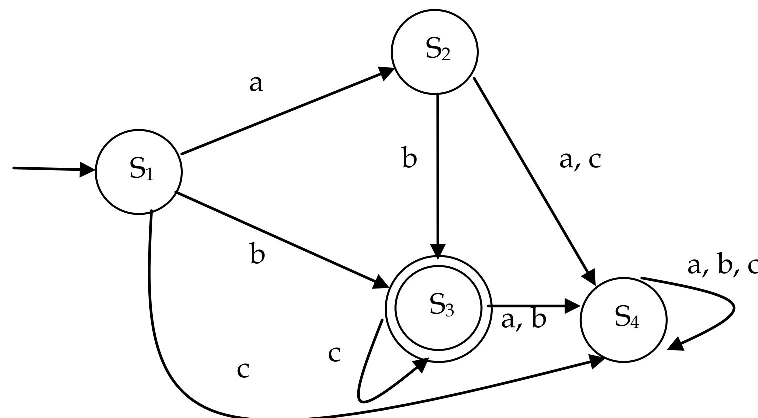


Figure: - DFA for above RE

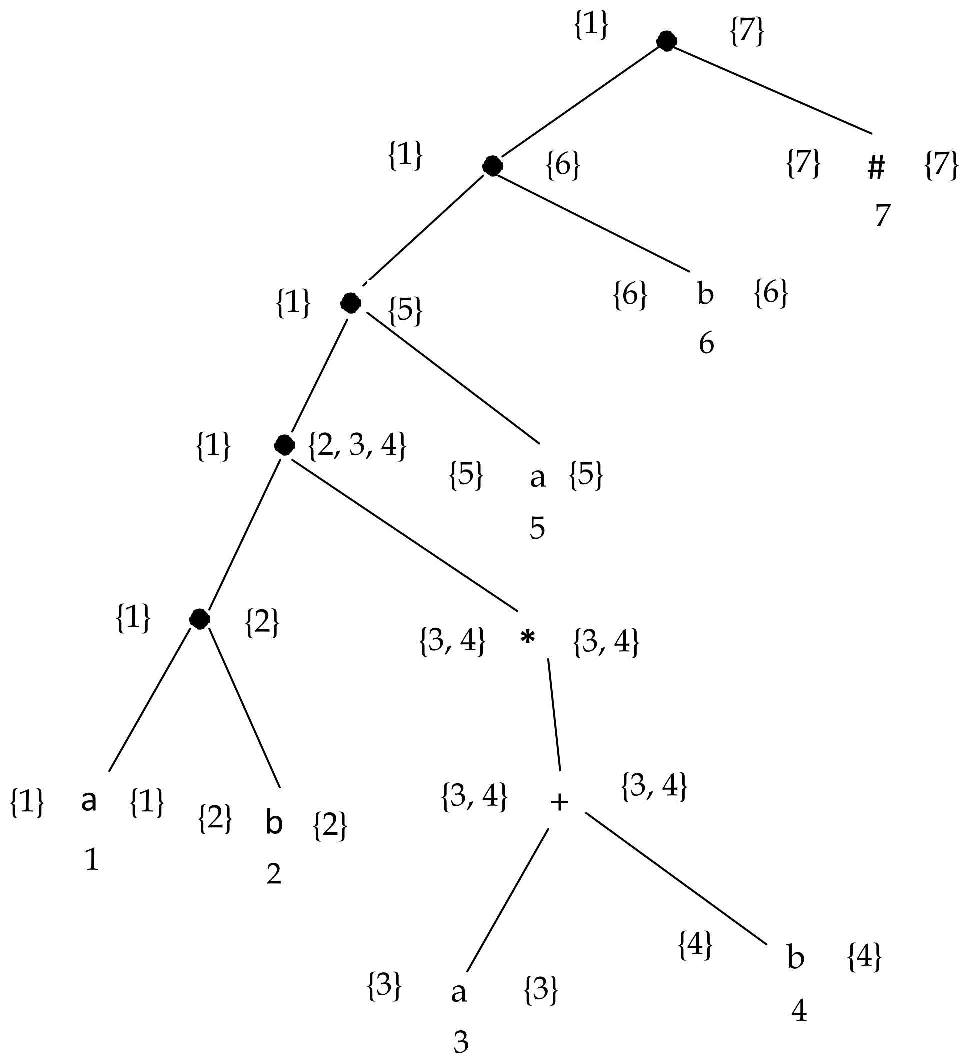
Example 3: Convert the regular expression ba(a+b)* ab into equivalent DFA by direct method.

Solution:

Step 1: At first augment the given regular expression as,

b.a.(a+b)*.a.b.#

Step 2: Now construct syntax tree of augmented regular expression as,



Step 3: Compute followpos as,

b. a. (a+b)*. a. b. #

1 2 3 4 5 6 7

Followpos(1) = {2}

Followpos(2) = {3, 4, 5}

Followpos(3) = {3, 4,5}

Followpos(4) = {3, 4, 5}

Followpos(5) = {6}

Followpos(6) = {7}

Followpos(7) = {Φ}

Step 4: After we calculate follow positions, we are ready to create DFA for the regular expression as,

Starting state of DFA = S_1 = Firstpos(Root node of Syntax tree) = {1}

Mark S_1

For a: follwpos(Φ) = {Φ}➔ S_2
For b: followpos(1) = {2} ➔ S_3

Mark S_2

For a: follwpos(Φ) = {Φ} ➔ S_2
For b: followpos(Φ) = {Φ} ➔ S_2

Mark S_3

For a: follwpos(2) = {3, 4, 5} ➔ S_4
For b: followpos(Φ) = {Φ} ➔ S_2

Mark S_4

For a: follwpos(3) U follwpos(5) = {3, 4, 5, 6} ➔ S_5
For b: followpos(4) = {3, 4, 5} ➔ S_4

Mark S_5

For a: follwpos(3) U follwpos(5) = {3, 4, 5, 6} ➔ S_5
For b: followpos(4) U followpos(6) = {3, 4, 5, 7} ➔ S_6

Mark S_6

For a: follwpos(3) U follwpos(5) = {3, 4, 5, 6} ➔ S_5
For b: followpos(4) = {3, 4, 5} ➔ S_4

Now there was no new states occur.
So starting state of DFA = {S_1}
And accepting state of DFA = {S_6}

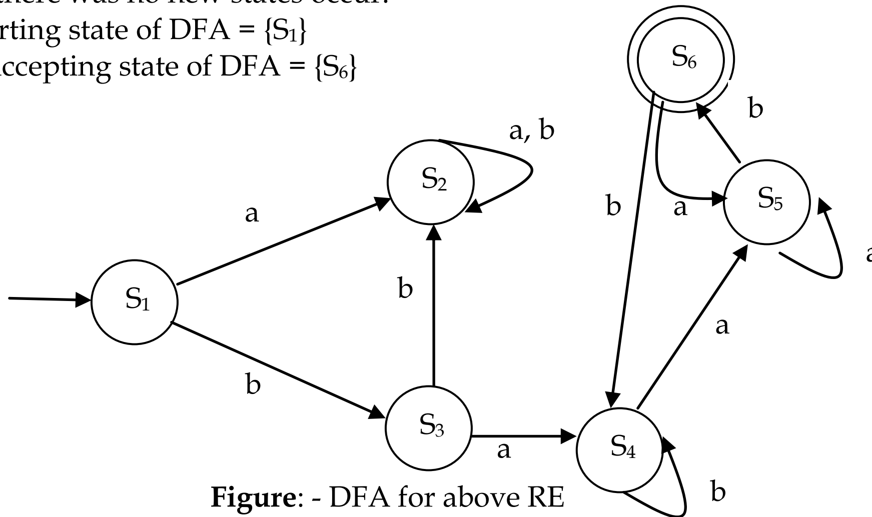


Figure: - DFA for above RE

State minimization in DFA

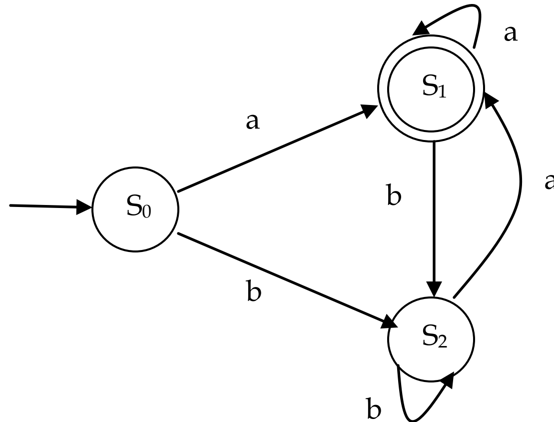
A DFA obtained by the subset construction algorithm is not necessary a DFA with the minimum possible number of states. The process of DFA minimization refers to remaining the inaccessible and indistinguishable states whose presence of absence in a DFA does not affect the language accepted by the automata. Here we use partition method for state minimization in DFA.

Suppose there is a DFA D < Q, Σ, q_0, δ, F > which recognizes a language L. Then the minimized DFA D < Q′, Σ, q_0, δ′, F′ > can be constructed for language L as:

- **Step 1:** We will divide Q (set of states) into two sets. One set will contain all final states and other set will contain non-final states. This partition is called P_0.
- **Step 2:** Initialize k = 1

- **Step 3:** Find P_k by partitioning the different sets of P_{k-1}. In each set of P_{k-1}, we will take all possible pair of states. If two states of a set are distinguishable, we will split the sets into different sets in P_k.
- **Step 4:** Stop when $P_k = P_{k-1}$ (No change in partition)
- **Step 5:** All states of one set are merged into one. Number of states in minimized DFA will be equal to number of sets in P_k.

Example 1: Minimize following DFA by using state partition method,



Solution:

Step 1: P_0 will have two sets of states. One set will contain S_1 which is final state of DFA and another set will contain remaining states S_0, S_2.

So, $P_0 = \{\{S_1\}, \{S_0, S_2\}\}$

Step 2: To calculate P_1, we will check whether sets of partition P_0 can be partitioned or not:

i) For set $\{S_1\}$:

Since we have only one state in this set, it can't be further partitioned.

So, S_1 is not distinguishable.

i) For set $\{S_0, S_2\}$:

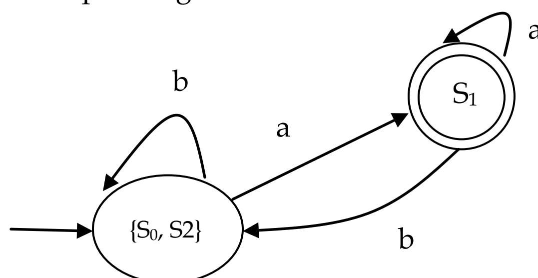
$\delta(S_0, a) = S_1$ and $\delta(S_2, a) = S_1$

$\delta(S_0, b) = S_2$ and $\delta(S_2, b) = S_2$

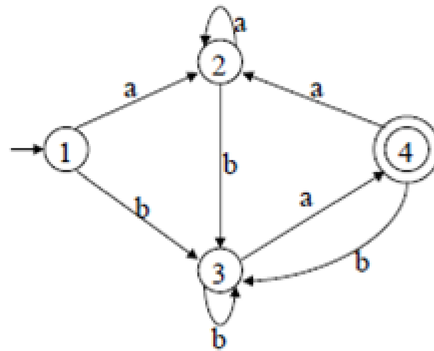
Moves of S_0 and S_2 on input symbol 'a' is S_1 which is in same set in partition P_0. Similarly, Moves of S_0 and S_2 on input symbol b is S_2 which is in same set in partition P_0. So, S_0 and S_2 are not distinguishable. So,

$P_1 = \{\{S_1\}, \{S_0, S_2\}\}$

Minimized DFA corresponding to DFA of above is shown in Figure below as:



Example 2: Minimize following DFA by using state partition method,



Solution:

Step 1: P_0 will have two sets of states. One set will contain 4 which is final state of DFA and another set will contain remaining states 1, 2 and 3.

So, $P_0 = \{\{4\}, \{1, 2, 3\}\}$

Step 2: To calculate P_1, we will check whether sets of partition P_0 can be partitioned or not:

i) For set {4}:

Since we have only one state in this set, it can't be further partitioned.

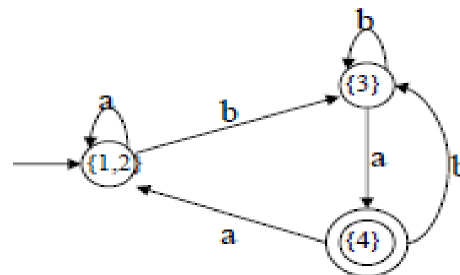
So, {4} is not distinguishable.

i) For set {1, 2, 3}

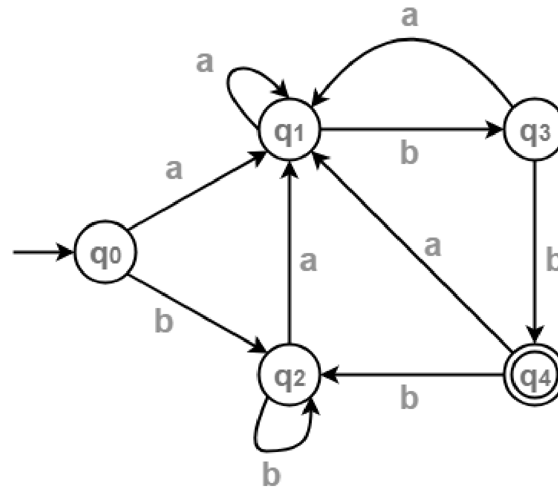
δ	a	b
1	{2}	{3}
2	{2}	{3}
3	{4}	{3}

Since transaction for state 1 and 2 for input symbol 'a' and 'b' are 2 and 3 respectively. So, P1= {{4}, {1, 2}, {3}}

Minimized DFA corresponding to DFA of above is shown in Figure below as:



Example 3: Minimize following DFA by using state partition method,



Solution:

Step 1: P_0 will have two sets of states. One set will contain q_4 which is final state of DFA and another set will contain remaining states q_0, q_1, q_2 and q_3.

So, $P_0 = \{\{q_4\}, \{q_0, q_1, q_2, q_3\}\}$

Step 2: To calculate P_1, we will check whether sets of partition P_0 can be partitioned or not:

i) For set $\{q_4\}$:

Since we have only one state in this set, it can't be further partitioned.

So, $\{q_4\}$ is not distinguishable.

ii) For set $\{q_0, q_1, q_2, q_3\}$

δ	a	b
$\rightarrow q_0$	q_1	q_2
q_1	q_1	q_3
q_2	q_1	q_2
q_3	q_1	q_4

From above transaction table we observe that states q_0 and q_3 are equivalent.

So,

$P_1 = \{\{q_0, q_2\} \{q_1, q_3\} \{q_4\}\}$

ii) For set $\{q_1, q_3\}$

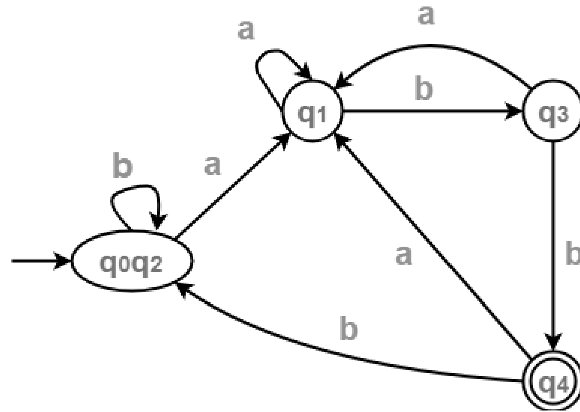
δ	A	B
q_1	q_1	q_3
q_3	q_1	q_4

From above transaction table we observe that states q_1 and q_3 are not equivalent.

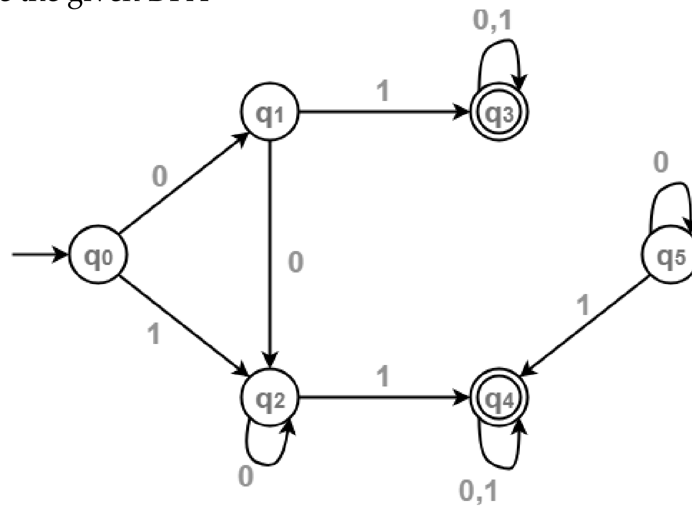
So, $P_2 = \{\{q_0, q_2\}, \{q_1\}, \{q_3\}, \{q_4\}\}$

Since, $\{q_0, q_2\}$ is not distinguishable.

So, Minimized DFA corresponding to DFA of above is shown in Figure below as:



Example 4: Minimize the given DFA-

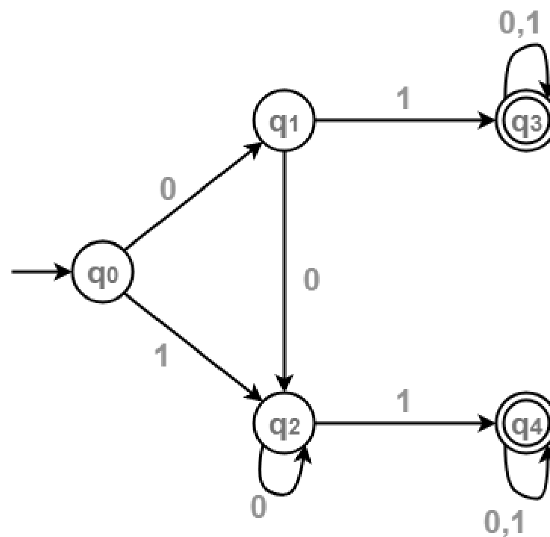


Solution:

Step 1: State q5 is inaccessible from the initial state.

So, we eliminate it and its associated edges from the DFA.

The resulting DFA is,



Step 2: Draw a state transition table

Δ	**0**	**1**
→q0	q1	q2
q1	q2	*q3
q2	q2	*q4
***q3**	*q3	*q3
***q4**	*q4	*q4

Now using Equivalence Theorem, we have-

P0 = {q0, q1, q2} {q3, q4}

P1 = {q0} {q1, q2} {q3, q4}

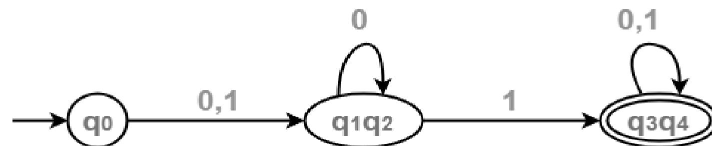
P2 = {q0} {q1, q2} {q3, q4}

Since P2 = P1, so we stop.

From P2, we infer:

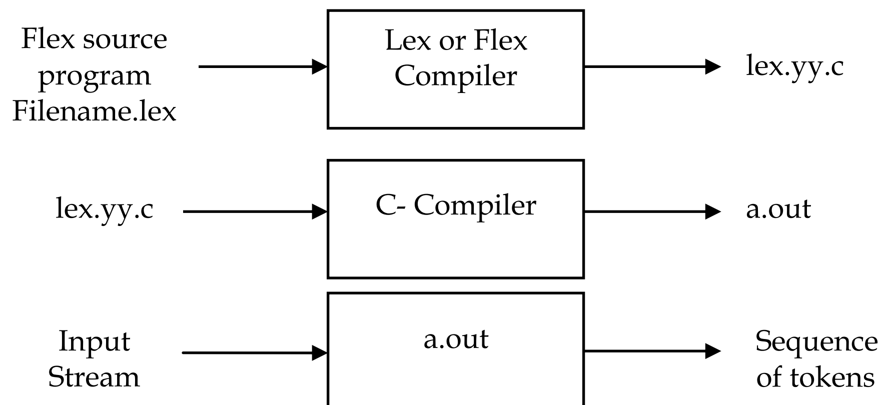
- States q1 and q2 are equivalent and can be merged together.
- States q3 and q4 are equivalent and can be merged together.

So, our minimal DFA is;



Flex: An introduction

Flex is a tool for generating scanners. A scanner is a program which recognizes lexical patterns in text. The flex program reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. Flex generates as output a C source file, 'lex.yy.c' by default, which defines a routine yylex(). This file can be compiled and linked with the flex runtime library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.



Flex specification

A flex specification consists of three parts:

Regular definitions, C declarations in %{ %}

%%

Translation rules

%%

User-defined auxiliary procedures

The translation rules are of the form:

p1 {action1}

p2 {action2}

…………………..

pn {actionn }

In all parts of the specification comments of the form /* comment text */ are permitted.