SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

**SCHOOL OF COMPUTING**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

# UNIT-I Compiler Design – SCS1303

# UNIT 1-- LEXICAL ANALYSIS

Structure of compiler – Functions and Roles of lexical phase – Input buffering – Representation of tokens using regular expression – Properties of regular expression – Finite Automata – Regular Expression to Finite Automata – NFA to Minimized DFA.

## STRUCTURE OF COMPILER:

Compiler is a translator program that reads a program written in one language -the source language- and translates it into an equivalent program in another language-the target language. As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

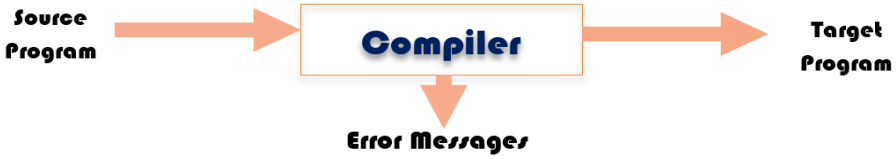



Fig. 1.1 A Compiler

## A LANGUAGE-PROCESSING SYSTEM:

The input to a compiler may be produced by one or more preprocessor and further processing of the compiler's output may be needed before running machine code is obtained.

**Preprocessors:**

Preprocessors produce input to compilers. They may perform the following functions:

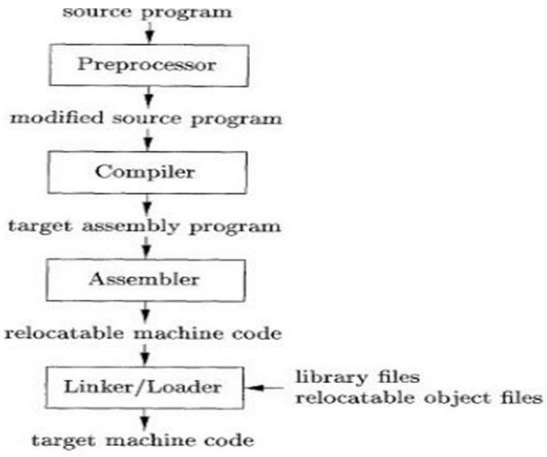



Fig. 1.2. A language-processing system

1. Macro processing: A preprocessor may allow a user to define macros that are shorthand for longer constructs.
2. File inclusion: A preprocessor may include header files into the program text. For example, the C
3. Preprocessor causes the contents of the file <*stdio.h*> to replace the statement
4. *#include* <*stdio.h*>when it processes a file containing this statement.
5. "Rational" preprocessors. These processors augment older languages with more modern flow-of-control and data-structuring facilities. For example, such a preprocessor might provide the user with built-in macros for constructs like while-statements or if-statements, where none exist in the programming language itself.
6. Language extensions. These processors attempt to add capabilities to the language by what amounts to built-in-macros.

**Assemblers:**

Some compilers produce assembly code, which is passed to an assembler for producing a relocatable machine code that is passed directly to the loader/linker editor. The assembly code is the mnemonic version of machine code. A typical sequence of assembly code is:

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

**Loaders and Linker-Editors:**

- A loader program performs two functions namely, loading and link-editing. The process of loading consists of taking relocatable machine code, altering the relocatable addresses and placing the alters instructions and data in memory at the proper locations.
- The link-editor allows making a single program from several files of relocatable machine code.

**THE PHASES OF A COMPILER**

**Analysis-Synthesis Model of Compilation:**

There are two parts to compilation: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation.

The analysis consists of three phases:

1. Linear analysis, in which the stream of characters making up the source program is read from left to right and grouped into tokens that are sequences of character having a collective meaning.
2. Hierarchical analysis, in which characters or tokens are grouped hierarchically into nested collections with collective meaning.
3. Semantic analysis, in which certain checks are performed to ensure that the components of a program fit together meaningfully.

A compiler operates in phases, each of which transforms the source program from one representation to another. The structure of compiler is shown in Fig.1.3.The first three phases form the analysis portion of the compiler and rest of the phases form the synthesis phase.

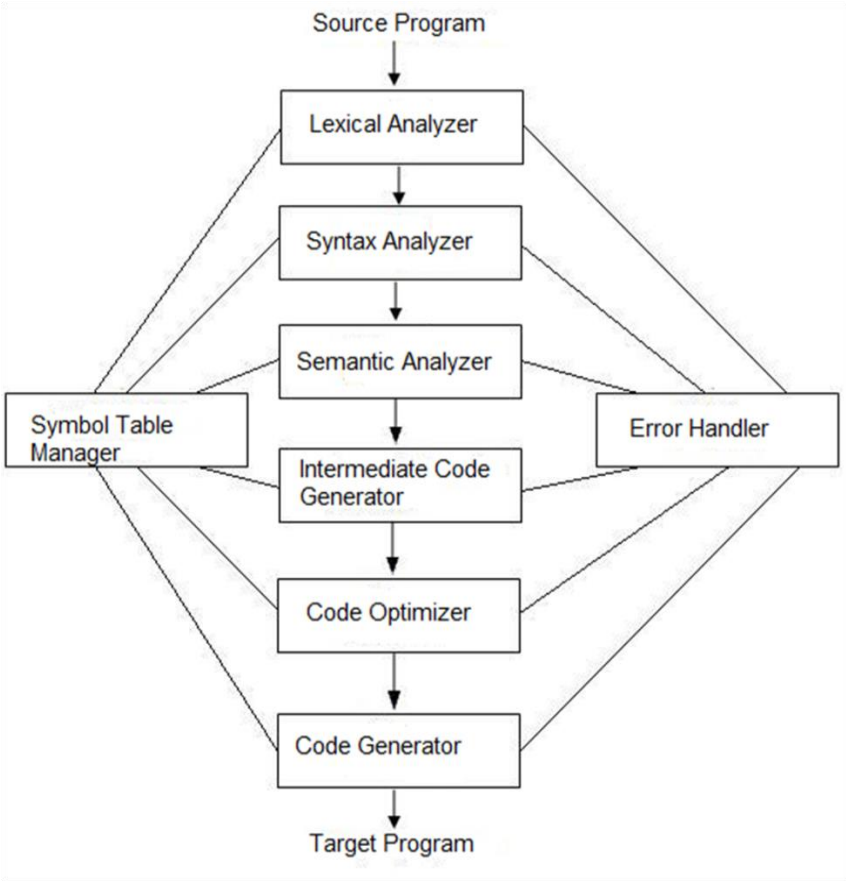



Fig. 1.3. Phases of a compiler

**Lexical Analysis:**

In a compiler, linear analysis is called lexical analysis or scanning.

- It is a first phase of a compiler
- Lexical Analyzer is also known as scanner
- Reads the characters in the source program from left to right and groups the characters into stream of Tokens.
- Such as Identifier, Keyword, Punctuation character, Operator.
- **Pattern:** Rule for a set of string in the input for which a token is produced as output.
- A **Lexeme** is a sequence of characters in the source code that is matched by the Pattern for a Token.


For example, in lexical analysis the characters in the assignment statement,

**position = initial + rate * 60**

would be grouped into the following tokens:

1. The identifier *position*.
2. The assignment symbol=.
3. The identifier *initial*.
4. The plus sign.
5. The identifier *rate*.
6. The multiplication sign.
7. The number60.

The blanks separating the characters of these tokens would be eliminated during this phase.

**Syntax Analysis:**

Hierarchical analysis is called parsing or syntax analysis. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize the output.

The source program is represented by a parse tree as one shown in Fig. 1.5.

- This phase is called the syntax analysis or **Parsing**.
- It takes the token produced by lexical analysis as input and generates a parse tree.
- In this phase, token arrangements are checked against the source code grammar.
- i.e. the parser checks if the expression made by the tokens is syntactically correct.

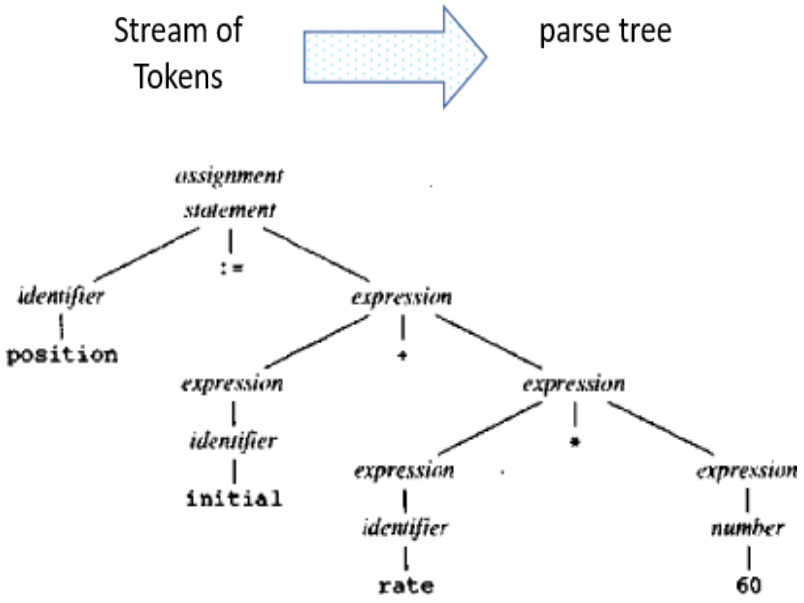



Fig.1.5. Parse tree for position = initial + rate * 60

The hierarchical structure of a program is expressed by recursive rules .i.e by context-free grammars. The following rules define an expression:

1. Any identifier is an expression. i.e. E →id
2. Any number is an expression. i.e. E →num
3. If expression1 ($E^1$) and expression2 ($E^2$)are expressions ,i.e. E → E + E | E * E | ( E)

Rules (1) and (2) are basic rules which are non-recursive, while rule(3) define expression in terms of operators applied to other expressions.

**Semantic Analysis:**

- The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.

- An important component of semantic analysis is type checking. i.e .whether the operands are type compatible.
- For example, a real number used to index an array.

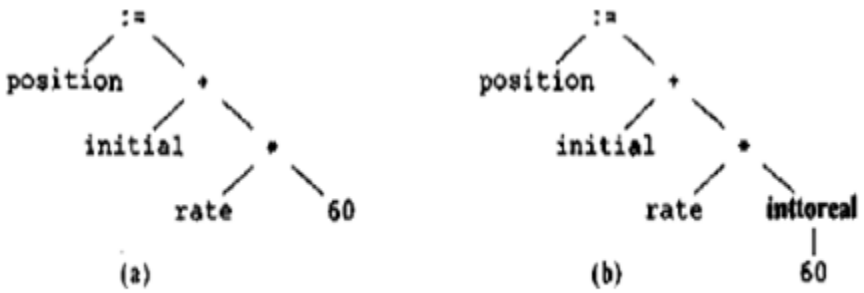



Fig. 1.6. Semantic analysis inserts a conversion from integer to real

**Intermediate Code Generation:**

After semantic analysis, some compilers generate an explicit intermediate representation of the source program. This representation should be easy to produce and easy to translate into the target program. There are varieties of forms.

- Three address code
- Postfix notation
- Syntax Tree

The commonly used representation is three address formats. The format consists of a sequence of instructions, each of which has at most three operands. The IR code for the given input is as follows:

temp1 = inttoreal ( 60 )
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3

**Code Optimization:**

- This phase attempts to improve the intermediate code, so that faster running machine code will result.

- There is a better way to perform the same calculation for the above three address code ,which is given as follows:

  temp1 = id3 * 60.0

  id1 = id2 + temp1

- There are various techniques used by most of the optimizing compilers, such as:
  1. Common sub-expression elimination
  2. Dead Code elimination
  3. Constant folding
  4. Copy propagation
  5. Induction variable elimination
  6. Code motion
  7. Reduction in strength. ..... etc..

**Code Generation:**

- The final phase of the compiler is the generation of target code, consisting of relocatable machine code or assembly code.
- The intermediate instructions are each translated into sequence of machine instructions that perform the same task. A crucial aspect is the assignment of variables to registers.
- Using registers R1 and R2,the translation of the given example is:

  MOV id3 ,R2

  MUL #60.0 , R2

  MOV id2 , R1

  ADD R2 , R1

  MOV R1 , id1

**Symbol-Table Management:**

- An essential function of a compiler is to record the identifiers used in the source program and collect its information.
- A symbol table is a data structure containing a record for each identifier with fields for attributes.(such as, its type, its scope, if procedure names then the number and type of arguments etc.,)
- The data structure allows finding the record for each identifier and store or retrieving

data from that record quickly.

**Error Handling and Reporting:**

- Each phase can encounter errors. After detecting an error, a phase must deal that error, so that compilation can proceed, allowing further errors to be detected.
- Lexical phase can detect error where the characters remaining in the input do not form any token.
- The syntax and semantic phases handle large fraction of errors. The stream of tokens violates the syntax rules are determined by syntax phase.
- During semantic, the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved. e.g. if we try to add two identifiers ,one is an array name and the other a procedure name.

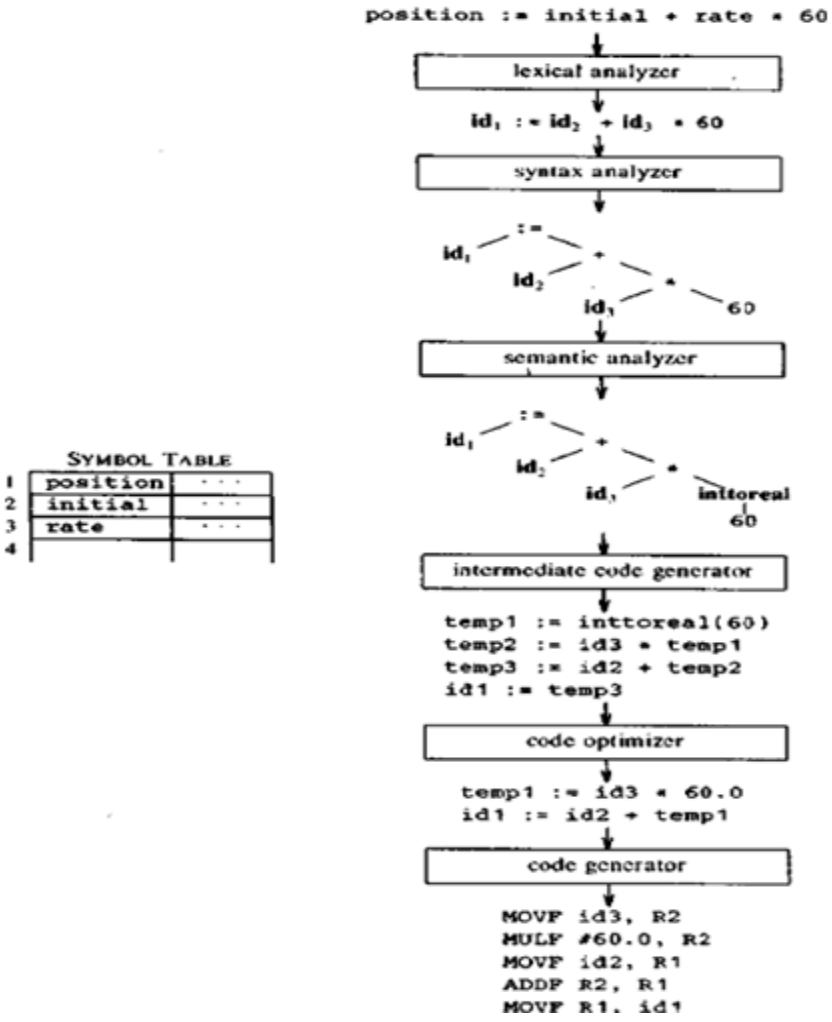



Fig.1.4. Translation of statement

**FUNCTIONS AND ROLES OF LEXICAL PHASE:**

- It is the first phase of a compiler.
- Its main task is to read input characters and produce tokens.
- "get next token "is a command sent from the parser to the lexical analyzer(LA).
- On receipt of the command, the LA scans the input until it determines the next token and returns it.
- It skips white spaces and comments while creating these tokens.
- If any error is present the LA will correlate that error with source file and the line number.

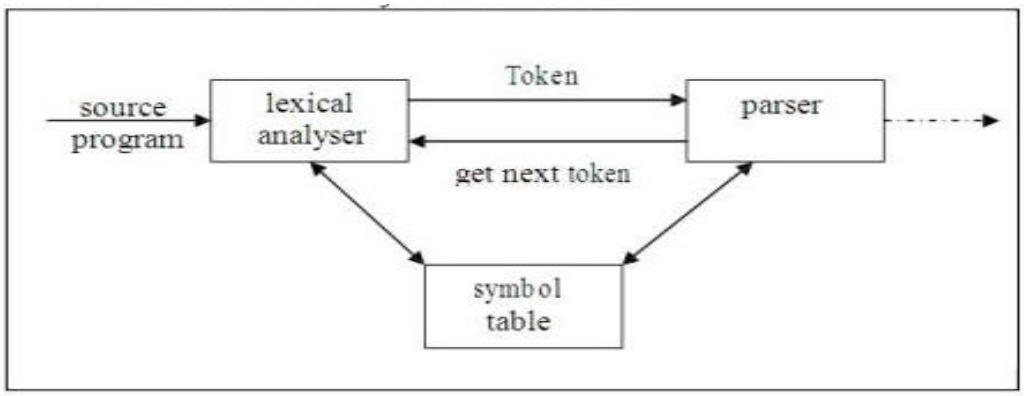



Fig.1.5.Interaction of lexical analyzer with parser

**Issues in Lexical Analysis:**

There are several reasons for separating the analysis phase of compiling into lexical and parsing:

1. Simpler design is the most important consideration.
2. Compiler efficiency is improved. A large amount of time is spent reading the source program and partitioning into tokens. Buffering techniques are used for reading input characters and processing tokens that speed up the performance of the compiler.
3. Compiler portability is enhanced.

**Tokens, Patterns, Lexemes:**

- Token is a sequence of characters in the input that form a meaningful word. In most languages, the tokens fall into these categories: Keywords, Operators, Identifiers, Constants, Literal strings and Punctuation.

- There is a set of strings in the input for which a token is produced as output. This set is described a rule called pattern.
- A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

| TOKEN | INFORMAL DESCRIPTION | SAMPLE LEXEMES |
|---|---|---|
| **if** | characters `i`, `f` | `if` |
| **else** | characters `e`, `l`, `s`, `e` | `else` |
| **comparison** | < or > or <= or >= or == or != | `<=`, `!=` |
| **id** | letter followed by letters and digits | `pi`, `score`, `D2` |
| **number** | any numeric constant | `3.14159`, `0`, `6.02e23` |
| **literal** | anything but ", surrounded by "'s | `"core dumped"` |

Fig.1.6. Examples of tokens

**Attributes of tokens:**

- The lexical analyzer collects information about tokens into their associated attributes. The tokens influence parsing decisions and attributes influence the translation of tokens.
- Usually a token has a single attribute i.e. pointer to the symbol table entry in which the information about the token is kept.

**Example**: The tokens and associated attribute values for the statement given,

| Lexeme | <token, token attribute> |
|---|---|
| E | <id, pointer to symbol table entry for E> |
| = | <assign-op,> |
| M | <id, pointer to symbol table entry for M> |
| * | <mult-op,> |
| C | <id, pointer to symbol table entry for C> |
| ** | <exp-op,> |
| 2 | <number, integer value 2> |
| ; | <separator, > |

**Lexical errors:**

Few errors are discernible at the lexical level alone, because a LA has a much localized view of the source program. A LA may be unable to proceed because none of the patterns for tokens matches a prefix of the remaining input.

**Error-recovery actions are:**

1. Deleting an extraneous character.
2. Inserting a missing character.
3. Replacing an incorrect character by a correct character.
4. Transposing two adjacent characters.

**INPUT BUFFERING**

A two-buffer input scheme that is useful when look ahead on the input is necessary to identify tokens is discussed. Later, other techniques for speeding up the LA, such as the use of "sentinels" to mark the buffer end are also discussed.

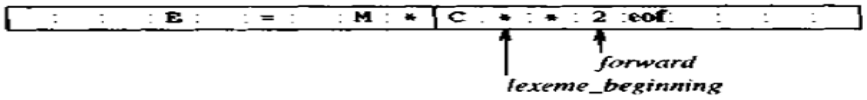
**Buffer Pairs:**

A large amount of time is consumed in scanning characters, specialized buffering techniques are developed to reduce the amount of overhead required to process an input character. A buffer divided into two N-character halves is shown in Fig. 1.7.Typically, N is the number of characters on one disk block, e.g., 1024 or 4096.


Fig. 1.7. An input buffer in two halves.

- **N, input** characters are read into each half of the buffer with one system read command, instead of invoking a read command for each input character.
- If fewer than **N** characters remain in the input, then a special character **eof** is read into buffer after the input characters.(**eof---end of file)**
- Two pointers, forward and lexeme_beginning are maintained. The string of characters between the two pointers is the current lexeme.
- If the forward pointer has moved halfway mark, then the right half is filled with **N** new input characters. If the forward pointer is about to move the right end of the buffer, then

the left half is filled with **N** new input characters and the wraps to the beginning of the buffer. Code for advancing forward pointer is shown in Fig.1.8.




```
if forward at end of first half then begin
        reload second half;
        forward := forward + 1
end
else if forward at end of second half then begin
        reload first half;
        move forward to beginning of first half
end
else forward := forward + 1;
```

Fig. 1.8. Code to advance forward pointer

**Sentinels:**

- With the previous algorithm, we need to check each time we move the forward pointer that we have not moved off one half of the buffer. If so, then we must reload the other half.
- This can be reduced, if we extend each buffer half to hold a sentinel character at the end.
- The new arrangement and code is shown in Fig. 1.9 and 1.10.This code performs only one test to see whether *forward* points to an **eof.**

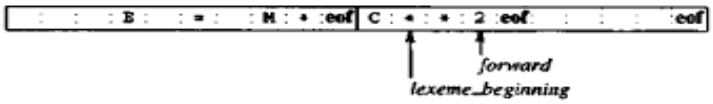



Fig.1.9. Sentinels at end of each buffer half.

```
forward := forward + 1;
if forward↑ = eof then begin
        if forward at end of first half then begin
                reload second half;
                forward := forward + 1
        end
        else if forward at end of second half then begin
                reload first half;
                move forward to beginning of first half
        end
        else  /* eof within a buffer signifying end of input */
                terminate lexical analysis
end
```

Fig. 1.10. Lookahead code with sentinels**.**

**REPRESENTATION OF TOKENS USING REGULAR EXPRESSION:**

Regular expression is an important notation for specifying patterns. The term alphabet or character class denotes any finite set of symbols. Symbols are letters and characters. The set {0,1} is the binary alphabet. ASCII and EBCDIC are two examples of computer alphabets.

- A s*tring* over some alphabet is a finite sequence of symbols drawn from that alphabet. The length of a string *s* is written as **|s|,** is the number of occurrences of symbols in *s*.
- The *empty string, denoted* by ε, is a special string of length zero.
- The term *language* denotes any set of strings over some fixed alphabet.
- The *empty set* {ε}, the set containing only the empty string.
- If *x* and *y* are strings, then the concatenation of *x* and *y* ,written as *xy* ,is the string formed by appending y to x.

Some common terms associated with the parts of the string are shown in Fig. 1.11.

| *TERM* | *DEFINITION* |
|---|---|
| prefix of s | A string obtained by removing zero or more trailing symbols of string s; ban is a prefix of banana. |
| suffix of s | A string formed by deleting zero or more of the leading symbols of s; nana is a suffix of banana. |
| substring of s | A string obtained by deleting a prefix and a suffix from s; nan is a substring of banana. Every prefix and every suffix of s is a substring of s, but not every substring of s is a prefix or a suffix of s. For every string s, both s and e are prefixes, suffixes, and substrings of s. |
| proper prefix, suffix, or substring of s | Any nonempty string x that is, respectively, a prefix, suffix, or substring of s such that s ≠ x. |
| subsequence of s | Any string formed by deleting zero or more not necessarily contiguous symbols from s; baaa is a subsequence of banana. |

Fig. 1.11 Terms for parts of a string

**Operations on Languages**

There are several important operations that can be applied to languages. The various operations and their definition are shown in Fig.1.12.

| OPERATION | DEFINITION |
|---|---|
| *union* of $L$ and $M$ written $L \cup M$ | $L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$ |
| *concatenation* of $L$ and $M$ written $LM$ | $LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$ |
| *Kleene closure* of $L$ written $L^*$ | $L^* = \bigcup_{i=0}^{\infty} L^i$ <br> $L^*$ denotes "zero or more concatenations of" $L$. |
| *positive closure* of $L$ written $L^+$ | $L^+ = \bigcup_{i=1}^{\infty} L^i$ <br> $L^+$ denotes "one or more concatenations of" $L$. |

Fig. 1.12 Definitions of operations on languages.

Let L be theset {A,B,.....,Z,a,b,. .....,z} consisting upper and lowercase alphabets, and D the set {0,1,2,.....,9} consisting the set of ten decimal digits. Here are some examples of new languages created from L and D.

1. $L \cup D$ is the set of letters and digits.
2. $LD$ is the set of strings consisting of a letter followed by a digit.
3. $L^4$ is the set of all four-letter strings.
4. $L^*$ is the set of all strings of letters, including $\epsilon$, the empty string.
5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.
6. $D^+$ is the set of all strings of one or more digits.

**Regular Language**

- A regular language over an alphabet □ is the one that can be obtained from the basic languages using the operations Union, Concatenation and Kleene *.
- A language is said to be a regular language if there exists a Deterministic Finite Automata (DFA) for that language.
- The language accepted by DFA is a regular language.
- A regular language can be converted into a regular expression by leaving out {} or by replacing {} with () and by replacing U by +.

**Regular Expressions**

Regular expression is a formula that describes a possible set of string.

Component of regular expression:

| | |
|---|---|
| **X** | **the character x** |
| **.** | **any character, usually accept a new line** |
| **[xyz]** | **any of the characters x, y, z,…..** |
| **R?** | **a R or nothing (=optionally as R)** |
| **R*** | **zero or more occurrences…..** |
| **R+** | **one or more occurrences……** |
| **R1R2** | **an R1 followed by anR2** |
| **R1 \| R2** | **either an R1 or an R2.** |

A token is either a single string or one of a collection of strings of a certain type. If we view the set of strings in each token class as a language, we can use the regular-expression notation to describe tokens.

Consider an identifier, which is defined to be a letter followed by zero or more letters or digits. In regular expression notation we would write.

**Identifier = letter (letter | digit)***

Here are the rules that define the Regular Expression (RE) over alphabet.

**Definition of RE:**

1. ε is a regular expression, L(ε) = {ε}
2. If ***a*** is a symbol in ∑ then ***a*** is a regular expression, L(a) = {a}
3. Suppose r and s are regular expressions denoting the language L(r) and L(s)
    a. (r) | (s) is a regular expression denoting the language L(r) ∪ L(s)
    b. (r)(s) is a regular expression denoting the language L(r)L(s)
    c. (r)* is a regular expression denoting (L(r))*
    d. (r) is a regular expression denoting L(r)

Unnecessary parentheses can be avoided in regular expressions if we adopt the conventions that:

1. The unary operator * has the highest precedence and is left associative.
2. Concatenation has the second highest precedence and is left associative.
3. **|,** the alternate operator has the lowest precedence and is left associative.

**PROPERTIES OF REGULAR EXPRESSION:**

There are a number of algebraic laws obeyed by regular expressions and these can be used to manipulate regular expressions into equivalent forms. Algebraic properties are shown in Fig. 1.13.

| Axiom | Description |
|---|---|
| r \| s = s \| r | \| is commutative |
| r \| (s \| t) = (r \| s) \| t | \| is associative |
| (r s) t = r (s t) | concatenation is associative |
| r (s \| t) = rs \| rt<br>(s \| t) r = sr \| tr | concatenation distributes over \|<br>( \| = alternation) |
| rϵ = r<br>ϵr = r | ϵ is the identity for concatenation |
| (r \| ϵ)* = r* | |
| r** = r* | * is idempotent |

Fig. 1.13 Algebraic laws of regular expression

**Regular Definitions**

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols.

Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

**Example-1,**

ab*|cd? Is equivalent to (a(b*)) | (c(d?))

Pascal identifier

Letter - A | B | ……| Z | a | b |……| z|

Digits - 0 | 1 | 2 | …. | 9

id - letter (letter / digit)*

**Transition Diagrams for Relational operators and identifier is shown below.**

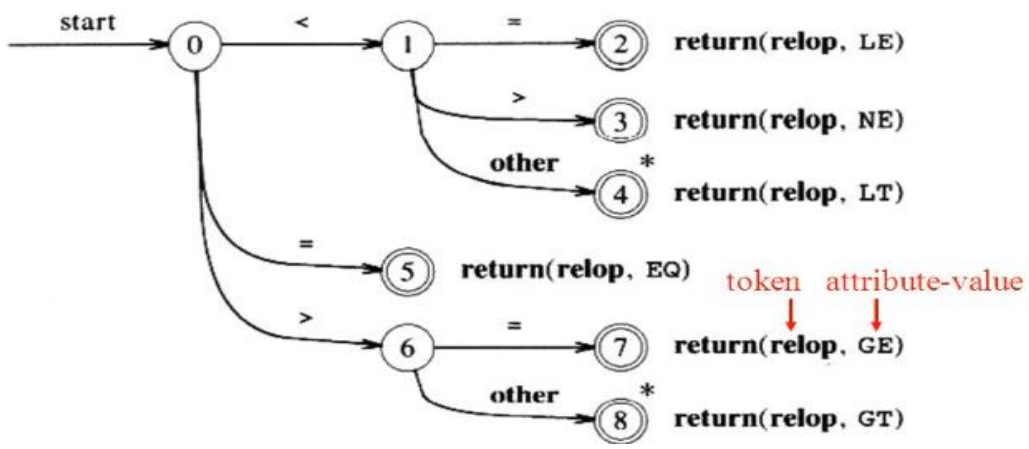



Fig. 1.14 Transition Diagrams for Relational operators

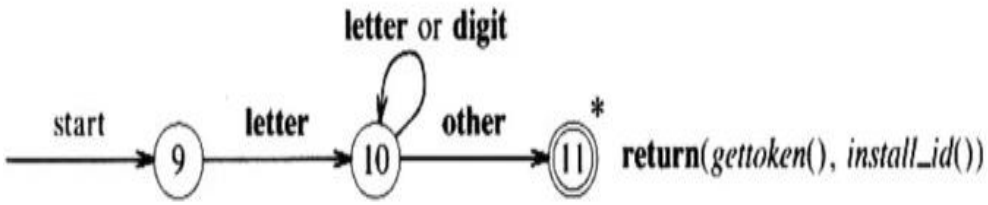



Fig. 1.15 Transition Diagrams for identifier

**FINITE AUTOMATA:**

A recognizer for a language is a program that takes as input a string x and answer "yes" if x is a sentence of the language and "no" otherwise. The regular expression is compiled into a recognizer by constructing a generalized transition diagram called a finite automaton. A finite automaton can be deterministic or non deterministic, where "nondeterministic" means more than one transition out of a state may be possible on the same input symbol.

**Deterministic Finite Automata (NFA)**

A Finite Automata (FA) or Finite State Machine (FSM) is a 5- tuple (Q,∑,q0,A, δ) where,

- Q is the set of finite states
- ∑ is the set of input symbols (Finite alphabet)
- q0 is the initial state

- A is the set of all accepting states or final states.
- δ is the transition function, $Q \times \sum \rightarrow Q$

For any element q of Q and any symbol a in ∑, we interpret δ(q,a) as the state to which the Finite Automata moves, if it is in state q and receives the input 'a'.

**How to draw a DFA?**

1. Start scanning the string from its left end.
2. Go over the string one symbol at a time.
3. To be ready with the answer as soon as the string is entirely scanned.

**Non-Deterministic Finite Automata (NFA) Definition:**

A NFA is defined as a 5-tuple, M=(Q, ∑,q0,A, δ) where,

- Q is the set of finite states
- ∑ is the set of input symbols (Finite alphabet)
- q0 is the initial state
- A is the set of all accepting states or final states.
- δ is the transition function, $Q \times \sum \rightarrow 2^Q$

| DFA | NFA |
|---|---|
| DFA can be understood as one machine. | Multiple small machines computing at the same time. |
| Difficult to construct - Complex structure. | Easier to construct. |
| Rejects the string if not terminated at the accepting state. | Rejects only after multiple checks. |
| Less execution time on input string. | More execution time. |
| All DFA are derived from NFA. | Not all NFA are DFA. |
| DFA requires more space. | Requires less Space. |
| The next possible state is clearly set. | Ambiguity occurs. |

Fig. 1.16 DFA vs. NFA

**Regular expression to NFA (Thompson's construction method):**

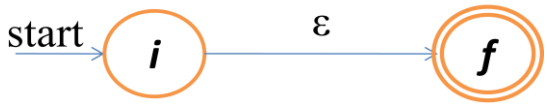
For each kind of RE, define an NFA.

Input: A Regular Expression R over an alphabet ∑

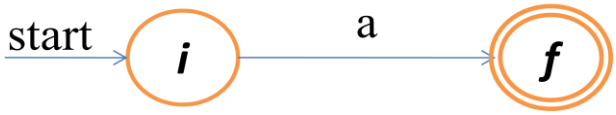
Output: An NFA N accepting L(R)

**Method:**

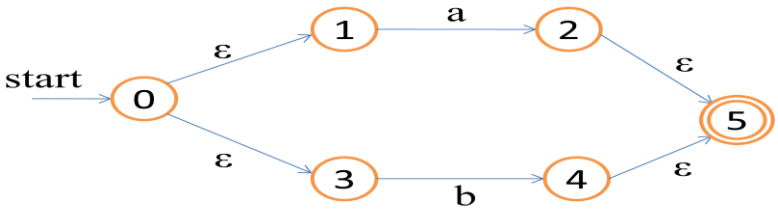
1. R = ε



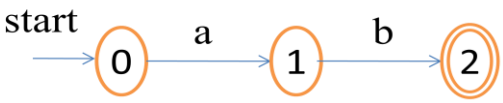

2. R = a



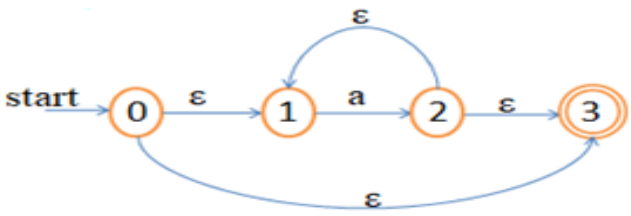

3. R= a | b



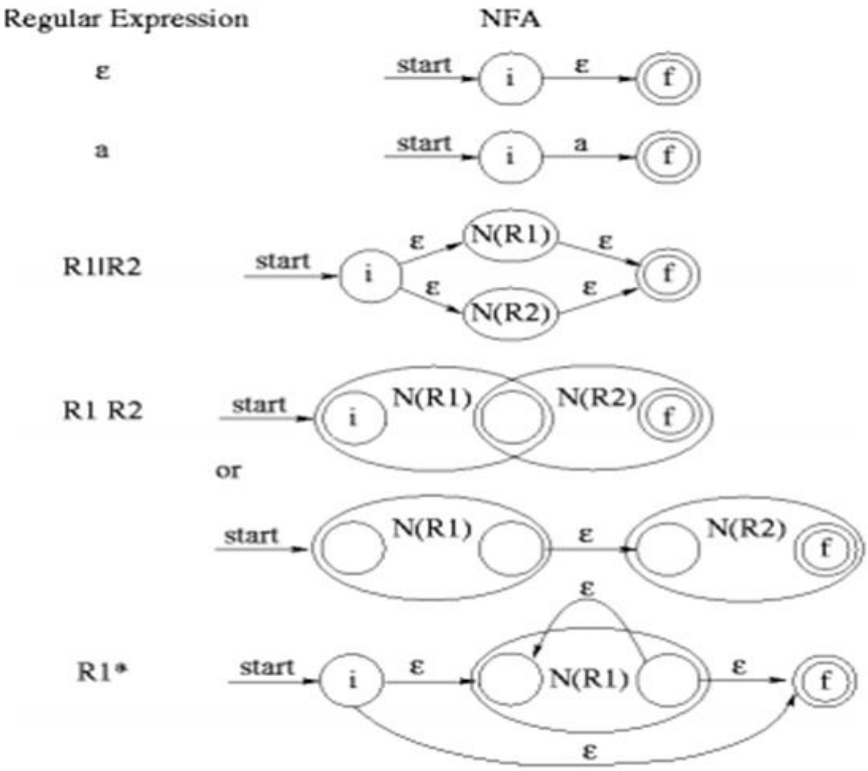

4. R=ab




5. R= a*

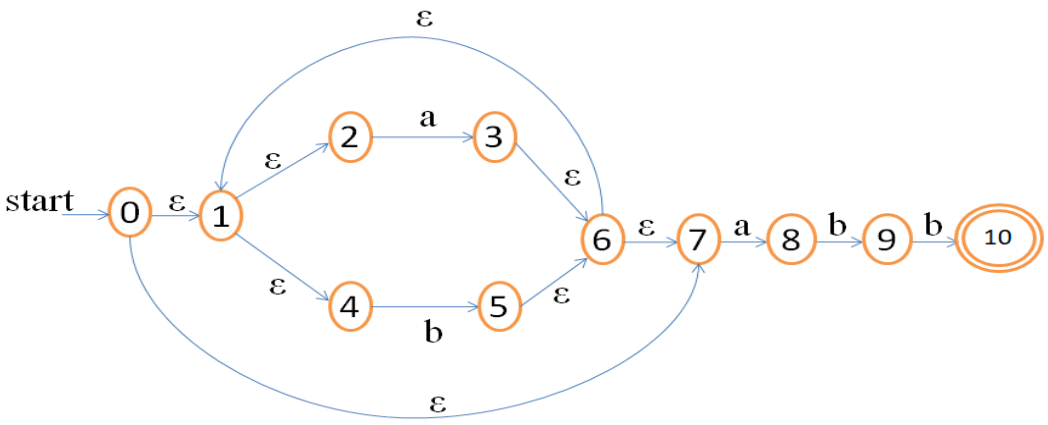


In general the rules if Thompson's construction method is given below:




**Problems:**

1. Construct NFA for: $(a+b)^*abb$




**REGULAR EXPRESSION TO FINITE AUTOMATA – NFA TO MINIMIZED DFA**

**Converting NFA to DFA: The Subset Construction Algorithm**

The algorithm for constructing a DFA from a given NFA such that it recognizes the same language is called subset construction. The reason is that each state of the DFA machine corresponds to a set of states of the NFA. The DFA keeps in a particular state all possible states

to which the NFA makes a transition on the given input symbol. In other words, after processing a sequence of input symbols the DFA is in a state that actually corresponds to a set of states from the NFA reachable from the starting symbol on the same inputs.

There are three operations that can be applied on NFA states:

| OPERATION | DESCRIPTION |
|---|---|
| ε-*closure*(*s*) | Set of NFA states reachable from NFA state *s* on ε-transitions alone. |
| ε-*closure*(*T*) | Set of NFA states reachable from some NFA state *s* in *T* on ε-transitions alone. |
| *move*(*T*, *a*) | Set of NFA states to which there is a transition on input symbol *a* from some NFA state *s* in *T*. |

The staring state of the automaton is assumed to be s0. The ε-*closure*( *s* ) operation computes exactly all the states reachable from a particular state on seeing an input symbol. When such operations are defined the states to which our automaton can make a transition from set *T* on input *a* can be simply specified as: ε-*closure*( *move*( *T*, *a* ))

**Subset Construction Algorithm:**

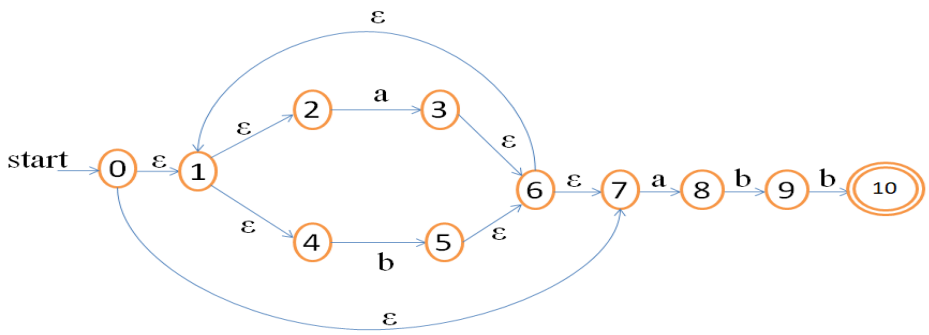
```
initially, ε-closure(s0) is the only state in Dstates and it is unmarked;
while there is an unmarked state T in Dstates do begin
      mark T;
      for each input symbol a do begin
            U := ε-closure(move(T, a));
            if U is not in Dstates then
                  add U as an unmarked state to Dstates;
            Dtran[T, a] := U
      end
end
```

**Algorithm for Computation of ε-*closure*:**

```
push all states in T onto stack;
initialize ε-closure(T) to T;
while stack is not empty do begin
      pop t, the top element, off of stack;
      for each state u with an edge from t to u labeled ε do
            if u is not in ε-closure(T) do begin
                  add u to ε-closure(T);
                  push u onto stack
            end
end
```

**Example:** Convert the NFA for the expression: (a|b)*abb into a DFA using the subset construction algorithm.

Step 1: Convert the above expression in to NFA using Thompson rule constructions.




**Step 2:** Start state of equivalent DFA is **ε-closure(0)**

ε-closure(0) ={ 0,1,2,4,7}

**Step 2.1:** Compute **ε-closure(move(A,a))**

move(A,a) ={3,8}

ε-closure(move(A,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}

ε-closure(move(A,a)) ={1,2,3,4,6,7,8}

Dtran[A,a]=B

**Step 2.2:** Compute **ε-closure(move(A,b))**

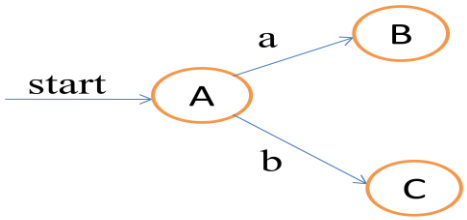
move(A,b) ={5}

ε-closure(move(A,b)) = ε-closure(5) = {5,6,7,1,2,4}

ε-closure(move(A,a)) ={1,2,4,5,6,7}

Dtran[A,b]=C

DFA and Transition table after step 2 is shown below.




| DFA states | Input Symbols | |
|---|---|---|
| | a | b |
| A | B | C |
| B | | |
| C | | |

**Step 3:** Compute Transition from **state B** on input symbol {a,b}

B = {1,2,3,4,6,7,8}

**Step 3.1:** Compute **ε-closure(move(B,a))**

move(B,a) ={3,8}

ε-closure(move(B,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}

ε-closure(move(B,a)) ={1,2,3,4,6,7,8}

Dtran[B,a]=B

**Step 3.2:** Compute **ε-closure(move(B,b))**

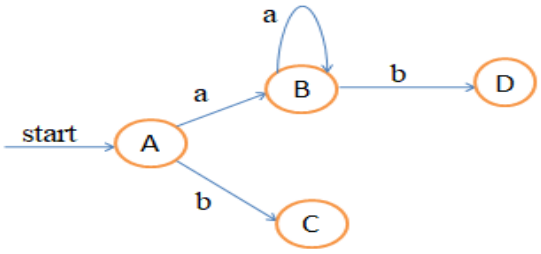
move(B,b) ={5,9}

ε-closure(move(B,b)) = ε-closure(5,9) = {5,9,6,7,1,2,4}

ε-closure(move(B,b)) ={1,2,4,5,6,7,9}

Dtran[B,b]=D

DFA and Transition table after step 3 is shown below.




Transition Table **Dtran**

| DFA states | Input Symbols | |
|---|---|---|
| | a | b |
| A | B | C |
| B | B | D |
| C | | |
| D | | |

**Step 4:** Compute Transition from **state C** on input symbol {a,b}

C = {1,2,,4,5,6,7}

**Step 4.1:** Compute **ε-closure(move(C,a))**

move(C,a) ={3,8}

ε-closure(move(C,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}

ε-closure(move(C,a)) ={1,2,3,4,6,7,8}

Dtran[C,a]= B

**Step 4.2:** Compute **ε-closure(move(C,b))**

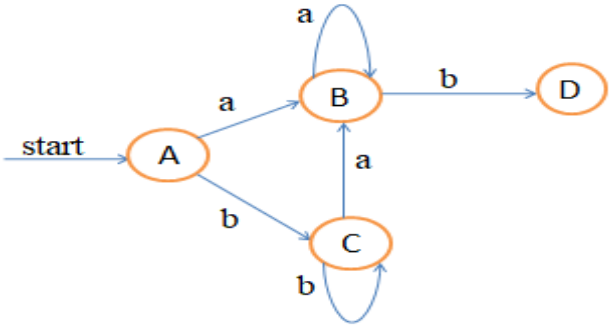
move(C,b) ={5}

ε-closure(move(C,b)) = ε-closure(5) = {5,6,7,1,2,4}

ε-closure(move(C,b)) ={1,2,4,5,6,7}

Dtran[C,b]= C

DFA and Transition table after step 4 is shown below.




Transition Table **Dtran**

| DFA states | Input Symbols | |
|---|---|---|
| | a | b |
| A | B | C |
| B | B | D |
| C | B | C |
| D | | |

**Step 5:** Compute Transition from **state D** on input symbol {a,b}

D = {1,2,,4,5,6,7,9}

**Step 5.1:** Compute **ε-closure(move(D,a))**

move(D,a) ={3,8}

ε-closure(move(D,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}

ε-closure(move(D,a)) ={1,2,3,4,6,7,8}

Dtran[D,a]= B

**Step 5.2:** Compute **ε-closure(move(D,b))**

move(D,b) ={5,10}

ε-closure(move(D,b)) = ε-closure(5,10) = {5,10,6,7,1,2,4}

ε-closure(move(C,b)) ={1,2,4,5,6,7,10}

Dtran[D,b]= E

**Step 6:** Compute Transition from **state E** on input symbol {a,b}

E = {1,2,,4,5,6,7,10}

**Step 6.1:** Compute **ε-closure(move(E,a))**

move(E,a) ={3,8}

ε-closure(move(E,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}

ε-closure(move(E,a)) ={1,2,3,4,6,7,8}

Dtran[E,a]= B

**Step 6.2:** Compute **ε-closure(move(E,b))**

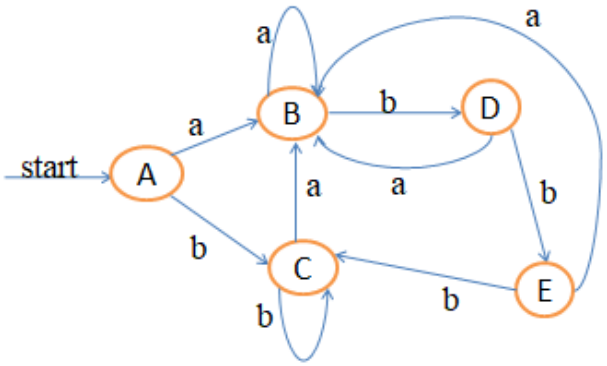
move(E,b) ={5}

ε-closure(move(E,b)) = ε-closure(5) = {5,6,7,1,2,4}

ε-closure(move(E,b)) ={1,2,4,5,6,7}

Dtran[E,b]= C

DFA and Transition table after step 6 is shown below.

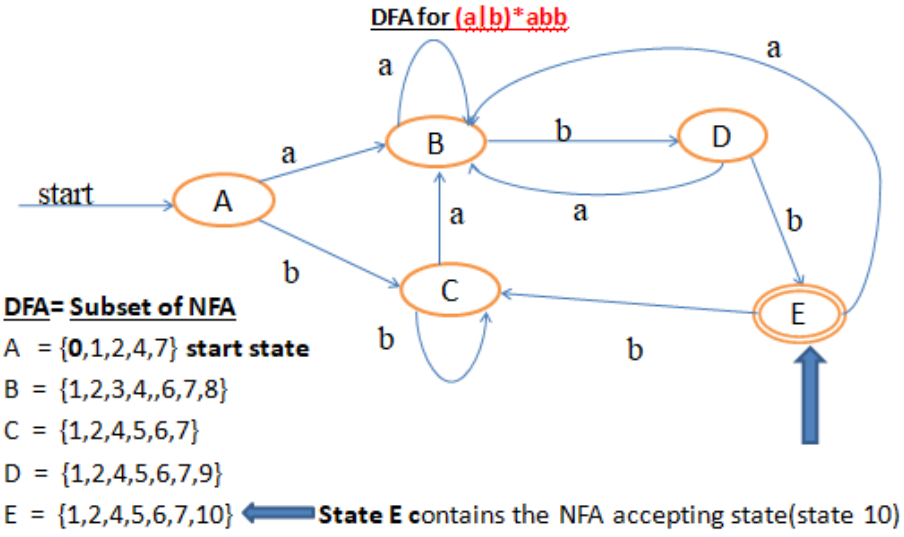



| DFA states | Input Symbols | |
|---|---|---|
| | a | b |
| A | B | C |
| B | B | D |
| C | B | C |
| D | B | E |
| E | B | C |

**Step 7: No more new DFA** states are formed.

**Stop** the subset construction method.

The start state and accepting states are marked the DFA.




**Minimized DFA:**

Convert the above DFA in to minimized DFA by applying the following algorithm.

**Minimized DFA algorithm:**

Input: DFA with ‘s’ no of states

Output: Minimized DFA with reduced no of states.

Steps:

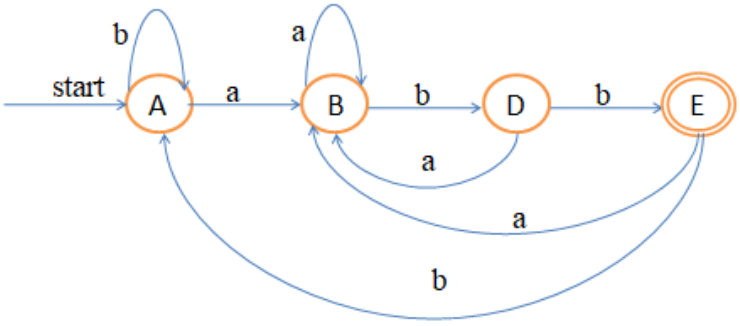
1. Partition the set of states in to two groups. They are set of accepting states and non accepting states.

2. For each group G of π do the following steps until π=π new .
3. Divide G in to as many groups as possible, such that two states s and t are in the same group only when for all states s and t have transitions for all input symbols ‘s’ are in the same group itself. Place newly formed group in π new.
4. Choose representative state for each group.
5. Remove any dead state from the group.

After applying minimized DFA algorithm for the regular expression (a|b)*abb , the transition table for the minimized DFA becomes Transition table for Minimized state DFA :

| DFA states | Input Symbols | |
|---|---|---|
| | a | b |
| A | B | A |
| B | B | D |
| D | B | E |
| E | B | A |

**Minimized DFA:**




**Exercises:**

Convert the following regular expression in to minimized state DFA,

1. (a|b)*

2.(b|a)*abb(b|a)*

3.((a|c)*)ac(ba)*