

UNIT I

INTRODUCTION TO COMPILERS

1. INTRODUCTION

Computers understand only machine languages. We users can understand only high level languages. Hence it is necessary to convert high level language programs to machine languages. The conversion is done by translators. The translators can be compilers or interpreters. Therefore the role of either the compiler or interpreters is to do this conversion.

What is (1) Translation, (2) Compilation, (3) Interpretation.

1.1 TRANSLATORS

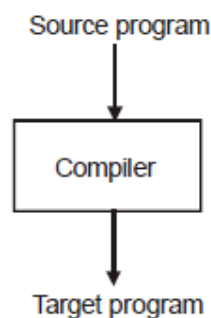
What is a translator?

Translator is a converting tool which is used to translate the high-level language program input into an equivalent machine language program.

Different Types of Translators:

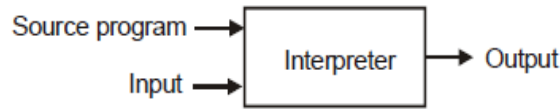
1. Compiler

- Compiler is a translator which is used to convert programs in high level language to low-level language.



2. Interpreter

- It is used to convert programs in high-level language to low-level language.
- It gives better error diagnostics than a compiler.



3. Assembler

- It is used to translate the assembly language code into machine language code.
- It is easier to debug.

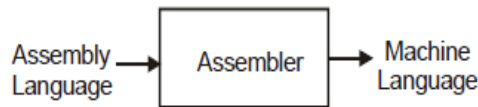


Figure 1.1

1.1.1 COMPILATION AND INTERPRETATION

An interpreter is a computer program that directly executes, i.e. performs instructions written in a programming language, without previously compiling them into a machine language program. An interpreter generally uses one of the following strategies for program execution:

- Parse the source code and perform its behavior directly.
- Translate source code into some efficient intermediate representation and immediately execute this.
- Explicitly execute stored precompiled code made by a compiler which is part of the interpreter system.

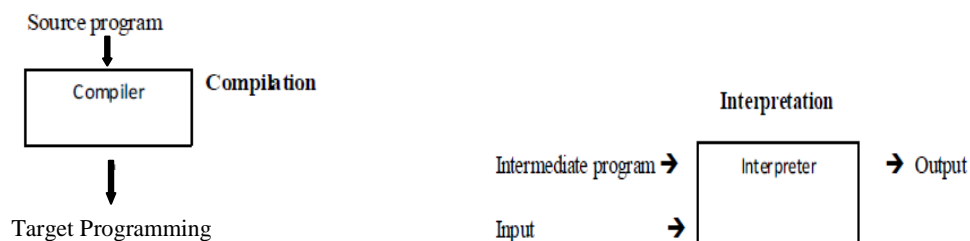
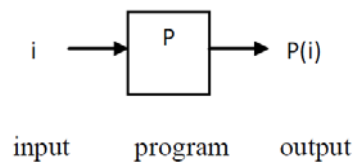


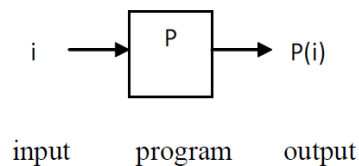
Fig. 1.2 Language Processing System

- Memory management is less.
- Compiler is a program that takes source program as input and produce assembly language as output.
- Compiler take whole point of source program (HLL) converts into machine language. Interpreter takes single line of code as input at a time that will execute each line and its final as error.

A program is something that computes:

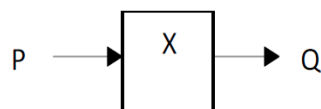


An interpreter is a program whose input is a program P and some input data x ; it executes P on x :



That is, $I(P,x) = P(x)$. The fact that there exists a single interpreter capable of executing all possible programs (Turing-Computable Functions) is a very deep and significant discovery of Turing's.

A translator is a program which takes input as a program in some language P and outputs a program in a language Q such that the two programs have the same semantics.



That is, $\forall x. P(x) = Q(x)$.

Some translators have really strange names:

Assembler

Translates assembly language programs into machine language programs

Compiler

Translates programs in a “high-level” language into programs in a lower-level language

We often translate (compile) source code (often in stages) all the way down to something interpretable, such as machine language.

However, there are also languages that allow you to execute code during compilation time and compile new code at run time.

Advantage

Better error diagnostics than compiler.

Disadvantage

Slower than machine language code directly executed on the machine.

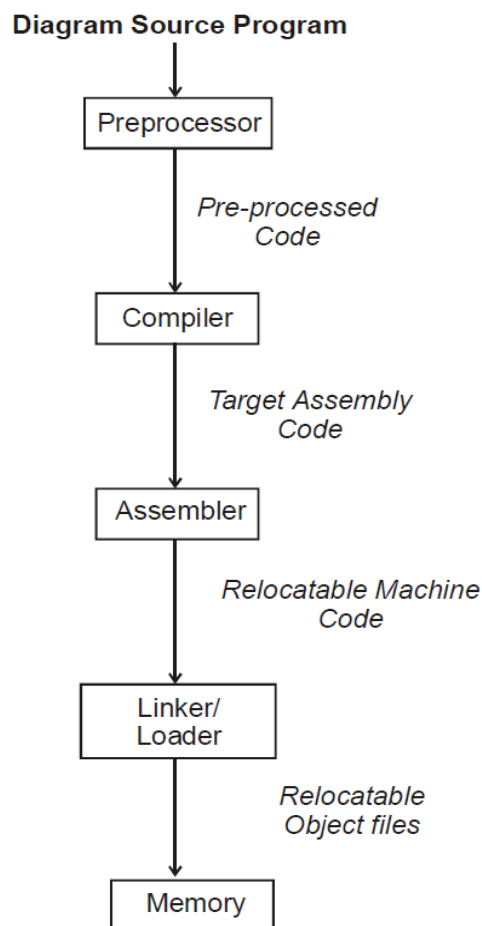


Figure 1.3 Language Processing System

1.1.1.1 Preprocessor

A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation file inclusion, language extension, etc.

1.1.1.2 Compiler

Compiler is a program that takes source program as input and produce assembly language as output.

1.1.1.3 Assembler

An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory. It produces relocatable machine code.

1.1.1.4 Linker / Loader

Linker links all the files referred by the program such as libraries etc.

Loader puts all the executable objects files together and its loads to program into memory.

1.1.1.5 Cross-Compiler

A compiler that runs on platform (A) and is capable of generating executable code for platform B is called a cross-compiler.

1.1.2 PROGRAMMING LANGUAGES & PROCESSING SYSTEM

We know that any computer system is made of hardware and software. The computer understands language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and Operating System components to get the desired code that can be used by the machine. This is known as Language Processing System.

- User writes a program in C language (high-level language).
- The C compiler, compiles the program and translates it to assembly program (low-level language).
- An assembler then translates the assembly program into machine code (object).

- A linker tool is used to link all the parts of the program together for execution (executable machine code).
- A loader loads all of them into memory and then the program is executed.

Before diving straight into the concepts of compilers, we should understand a few other tools that work closely with compilers.

1.2 STRUCTURE OF COMPILER

Definition:

A compiler is a program which translates a program written in a language called the source language to its equivalent in another language called Target language.

During the translation, the compiler returns the errors found in the source program to the user.

The compiler steps are figuratively represented as in Figure 1.4.

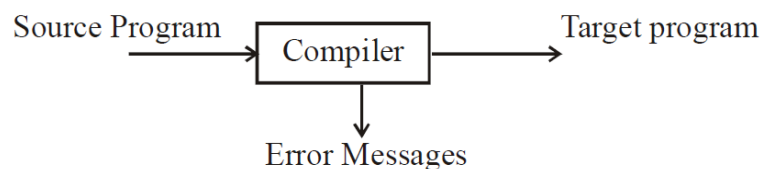


Figure 1.4

Computers understand only machine languages. We users can understand only high level languages. Hence it is necessary to convert high level language programs to machine languages. The conversion is done by translators. The translators can be compilers or interpreters. Therefore the role of either the compiler or interpreters is to do this conversion.

Eg. of source language : C, C++, Java, etc.

Eg. of target language : Another program language or Machine language.

History of Compilers:

The first compiler appeared in 1950. During that period compilers were considered as difficult programs to write.

For example, The first FORTRAN compiler took 18 manual years to implement.

Language programming environment, software tools are available now hence can be considered as a semester project.

1.2.1 MODEL OF COMPILATION

The process compilation has two parts namely:

1. Analysis,
2. Synthesis.

Analysis:

This part breaks the source program into constituent pieces and creates an intermediate representation of the source program.

This information is stored into symbol table.

1.2.2 PHASES OF COMPILATION

(Machine Independent/Language Dependent)

(Machine Dependent/Language Independent)

Phases of A Compiler

The compiler operates in phases and each phase transform the source program from one representation to another. The phases of compiler are as shown in Fig. 1.5.

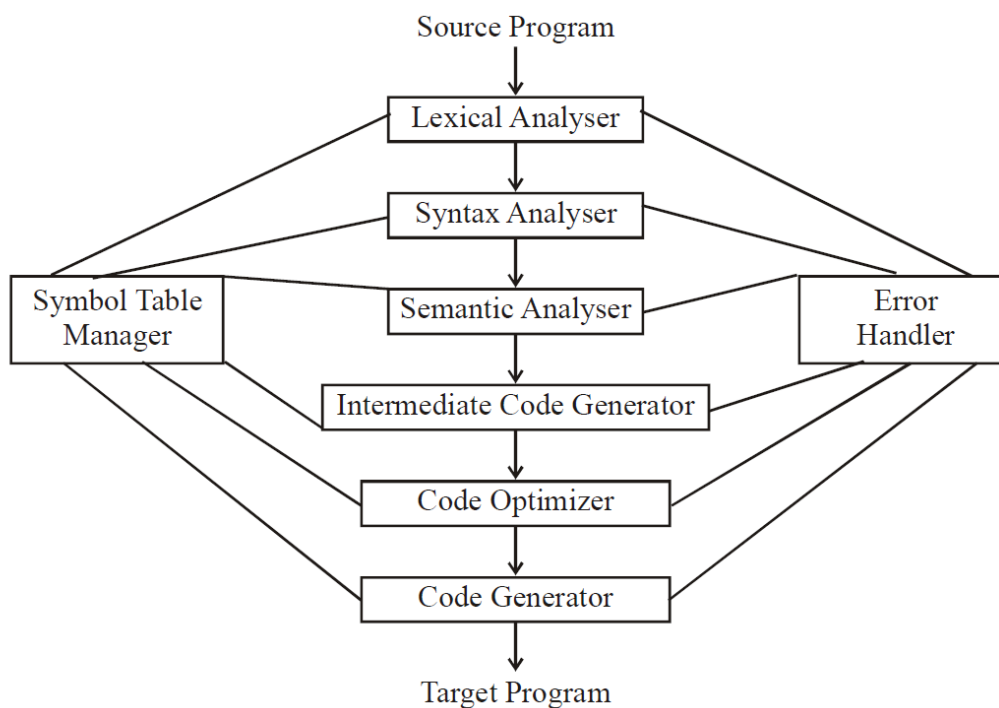


Figure 1.5: Phases of Compiler

Symbol Table

The symbol table is used to store information about to source language constructs. The information is collected during the analysis phase and used during this synthesis phase.

How the lexical analyser interacts with the symbol table?

The symbol table is concerned with saving and retrieving lexemes. When a lexeme is saved the token associated with the lexeme is also saved.

The operations performed on the symbol table are:

insert (s, t)-Return index of a new entry for strings, token t,

lookup (s) -Returns index of the entry for string s, else 0 if s is not in array.

The lexical analyser uses the look up to determine if an entry for the lexeme is found. If not found uses the insert operation to create a record.

Note: Lexeme, Token will be discussed in Unit II.

Symbol Table Implementation

An array called lexemes is used to store the characters forming an identifier. The string is terminated by end of string EOS.

Each entry in the symbol table array 'symtable' has two fields, lexptr pointing to the beginning of a lexeme and token.

The three phase were discussed in the previous section. The last three phases and the other two are discussed in this section.

The symbol table manager and Error Handler communicates with all the six phases.

Symbol Table Management

A symbol table is a data structure, which contains a record for each identifier and its attributes. These attributes provide information about the storage allocated for the identifier, its type, scope for identifier. If incase of procedure names, the details about the number, type of passing parameters and the details about the returning parameters are stored.

The advantages:

- (1) Easy to store or retrieve data quickly.
- (2) Easy to locate

The identifiers are added to the symbol table during the lexical analysis phase. But the attributes are not determined during the lexical phase.

Eg. `int a, b, c ;`

The exact data type of the identifiers a, b, c are not seen by the lexical analyzer. The remaining phases enter about the identifier which is used in various ways.

Semantic analysis and Intermediate Code Generation needs to know the data types. The Code Generation uses the storage details of the identifier.

1.2.3 ERROR DETECTION AND REPORTING

Each phase can encounter errors. In order for the compilation to proceed to discover other errors in the source program after the discovery of an error handling routine takes care.

The syntax and semantic analysis phases handles large fraction of error detected by the compiler. The lexical analysis phase detect error if the left over characters in the input do not form any token of the language.

In the syntax analysis phase errors are discovered when the token stream violates the syntax of the language.

During the semantic analysis the compiler detect the construct that have the correct syntax but without a meaning.

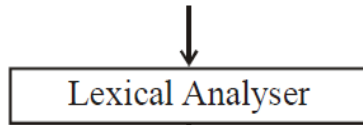
When the lexical analyser reads a letter, it starts saving the letter and digits in a buffer called lex buffer. The collected string is subjected to look up operation in the symbol table. When the look up operation returns '0' the lexeme contains a new identifier, inturn calls the insert operation. P the index of the symbol table is returned.

The integer coding of the character is returned.

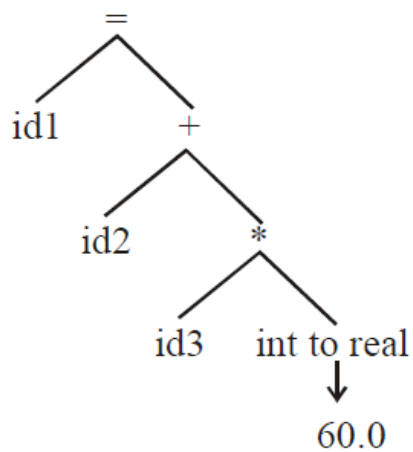
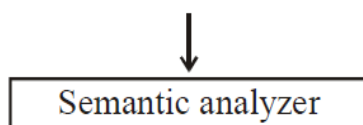
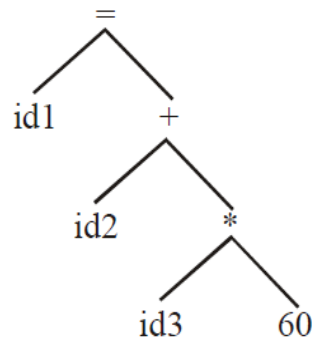
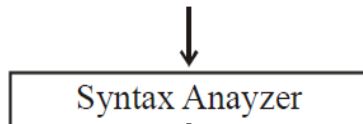
Analysis Phase: Consider the statement

$$\text{Position} = \text{initial} + \text{rate} * 60 . 0;$$

Position = initial + rate * 60



id1 = id2 + id3 * 60



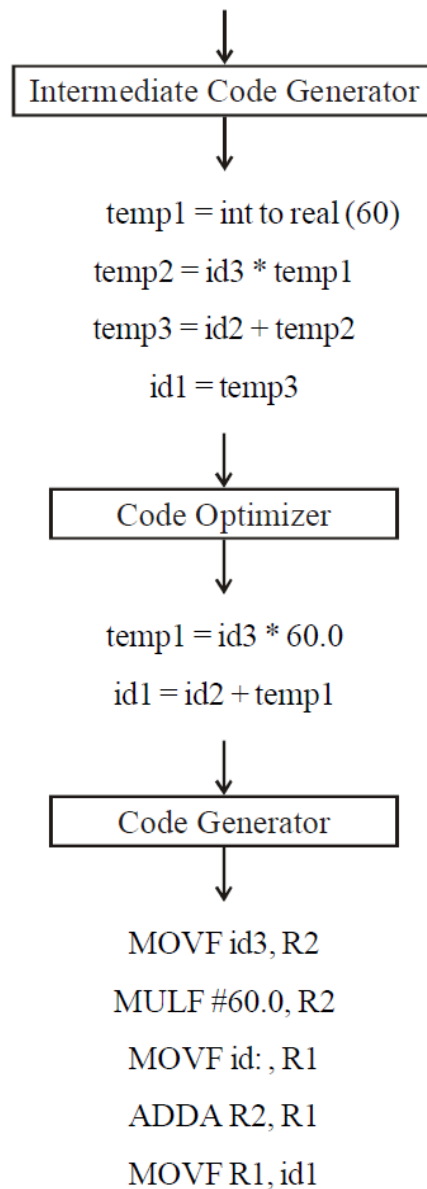


Figure 1.6

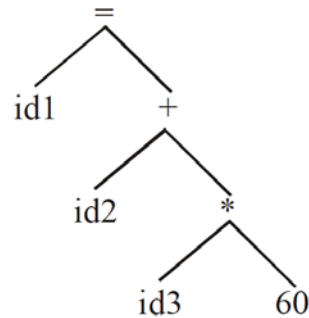
The lexical analyser reads the character in the source program and groups into stream of tokens. The token represents identifier, keyword, operator.

Lexeme : The character sequence forming a token is called as lexeme.

Ex.: When the identifier is found, the lexical analysis considers a token called id and enter the lexeme in the symbol table if not found already.

The lexical value associated with this occurrence of id points to the symbol table entry for the identifier.

Syntax analysis gives hierarchical structure of the token stream. Semantic analysis analyses the meaning.



Intermediate Code Generation

This is an intermediate representation of the source program. The intermediate representation need to satisfy two properties

- (1) Easy to produce
- (2) Easy to translate to target program.

The ICG can be in various form. Most prominent is that of three address code.

The three address codes for the above example is:

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

The three address code has instructions while each instruction take atmost three operands (or) three address (i.e.,) two for operand and one for operator.

Properties of Three Address Code

1. Each three address code has atmost one operator in addition to the assignment.
2. Compiler must generate a temporary name to hold the value computed by each instruction.
3. Some three address instructions may have less than three addresses.

Code Optimization:

This phase tries to improve the intermediate code so that fast running machine code is produced.

Code Generation:

This is the last phase of the compiler. The output is the relocatable machines code (or) assembly code. The memory location are selected for each of the variable used by the program. The intermediate instructions are translated into a sequence of machine instruction. The code for the above said instruction is:

MOVF id3, R2

MULF # 60.0, R2

MOVF id2, R1

ADDA R2, R1

MOVF R1, id1

1.3 LEXICAL ANALYSIS: THIS IS THE FIRST PHASE IN THE COMPILER

Need:

A simple way to build a lexical analyzer is to construct a diagram that illustrate the structure of the token of the source language and then to use the diagram to write program for finding tokens.

1.3.1 THE ROLE OF LEXICAL ANALYZER

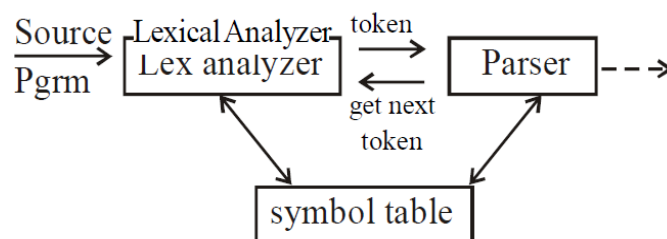


Figure 1.7

The main task of the lexical analyzer is to read the I/P character and produce as O/P a sequence of tokens that the parser uses for syntax analysis.

Upon receiving a “get next token” command from the parser the lexical analyzer reads input character until it can identify the next token.

Functions of Lexical Analyser

- Lexical Analyzer, also performs tasks as stripping out comments, white space in the form of blank, tab and newline character from the source program.
- Another task it performs is it correlates error message from the computer, with the source program. The lexical analyzer keeps track of the number of newline character.
- So that a line number can be associated with error messages.
- If the source program supports macro preprocessor functions, then the preprocessor functions may also be implemented as lexical analysis takes place.
- Lexical analyzer is divided into a cascade 2 phases.
 1. Scanning - does simple task as scanning the input
 2. Lexical analysis - does more complex operations, as separating the given program into tokens.

Issues in Lexical Analysis:

There are several reasons for separating the analysis phase of compiling into lexical analysis and syntax analysis. They are:

1. Simpler design is the most important consideration.
2. Compiler efficiency is improved, specialized, efficient processor for the task may be constructed.
3. Compiler portability is enhanced.

The technical words token, pattern, lexeme are used throughout.

Token: The terminal symbols used in the grammar.

Pattern : The set of strings described by a rule. The pattern is to match each string in the set.

Lexeme : Sequence of characters in the source program that is matched by the pattern for a token.

Eg. for Lexeme \Rightarrow if ($x > y$) “if”, “(”, “ x ”, “ $<$ ”, “ y ” are all lexemes.

If \Rightarrow keyword

(\Rightarrow operator

$y, x \Rightarrow$ Identifier

$< \Rightarrow$ Operator

) \Rightarrow Operator.

Eg:- Compile given program

\Downarrow

Submit to compiler

\Downarrow

Compiler scan program and produce token

\Downarrow

Lexical Analysis (or) Scanner

Source Code

```
int GREAT (int  $x$  , int  $y$ )
{ if ( $x > y$ )
    return  $x$ 
  else return  $y$ 
}
```

Lexeme	Token
int	keyword
GREAT	identifier
(operator
int	keyword
x	identifier

Device specific anomalies can be restricted to the lexical analyzer.

<i>Token</i>	<i>Sample Lexemes</i>	<i>Informal Description of Pattern</i>
1. Const	const	const
2. if	if	if
3. relation	<, <=, =, < >, >, >, >=	< or < = or = or < > or > = or >
4. id	count, p2	letter followed by letters and digits
5. num	3, 4, 0	any numeric constant
6. literal	“core dumped”	any character between “_” except “,”

- Tokens are treated as terminal symbols in the grammar of the source language.
- In most programming language, the following constructs are treated as tokens.
 - (a) keywords,
 - (b) operators,
 - (c) identifier,
 - (d) constants,
 - (e) literal strings,
 - (f) punctuation symbols (, ;

Attributes for Tokens:

The lexical analyzer collects information about tokens into the associated attributes.

Eg.: $E = M * C * * 2$ [MC2]

Explanation:

<exp.op>	<id, pointer to symbol table entry for E>
----------	---

<num, integer value>	<assign-op>
	<id, pointer to symbol table entry for M>
	<mult-op>
	<id, pointer to symbol table entry for C>

A token has usually only a single attribute - a pointer to the symbol table entry in which the information about the token, the pointer becomes the attribute for the token.

LEXICAL ERRORS

Eg: $\overset{\mapsto \text{if keyword error}}{fi}(a = f(x))$

Lexical analyzer has a very localized view of source program. If the string “ fi “ is encountered in a C program in the place of “if” a lexical analyzer cannot tell whether “fi” is misspelling of the keyword “if” or undeclared function identifier, the lexical analyzer must return the token for an identifier and let some other phase of the compiler handle such error.

Error Recovery actions are:

1. Deleting an extraneous character. *Eg.: if [if= if]*
2. Replacing an incorrect character by a correct character. *Eg.: it \Rightarrow int*
3. Swapping two adjacent characters. *Eg.: fi \Rightarrow if*
4. Inserting a missing character. *Eg.: it \Rightarrow int*

Three general approaches for implementation of a lexical analyzer:

1. Use a lexical analyzer generator such as the lex compiler to produce the lexical analyzer. In this case the generator provides routine for reading and buffering the input.
2. Write the lexical analyzer in a conventional systems programming language using the I/O facilities of that language to read the I/P.
3. Write the lexical analyzer in assembly language to read the I/P.

1.4 BUFFER PAIRS: (INPUT BUFFERING)

The lexical analyser needs to look beyond the lexeme for a pattern to match.

A buffer may be divided in to N character halves

“N” is the number of characters on one disk block.

We read ‘N’ I/P characters into each half of the buffer with one system read command rather than invoking a read command for each I/P character. Two pointers are maintained.

They are forward pointer and Lexeme Beginning. The string of characters between the two pointers is the current lexeme. Initially both pointers point to the first character of the next lexeme.

The forward pointer moves forward direction until a match is found. If found the forward pointer is set to the character at its right end. After processing both pointers point to the same character.

If the forward pointer is about to reach the half way mark, the right half is filled with N new input characters. If the forward pointer is about to reach the right half, the left half is filled with N new input characters.

The forward pointer wraps around to the beginning of the buffer.

Limitation:

1. Amount of look ahead is limited
2. Due to this limitation look ahead is may be impossible when the forward pointer max travel more than the length of the buffer.

Eg:- $E = MC^2$

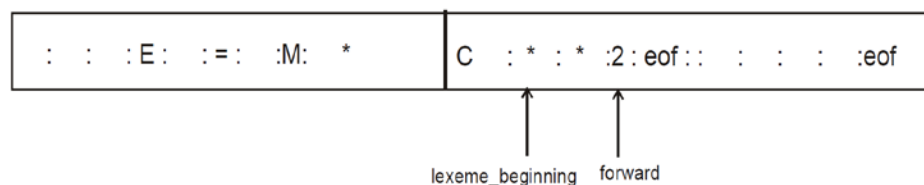


Figure 1.8 Input Buffer in two halves

Sentinels:

When the above method is used, for every move of the forward pointer it must be checked for the movement whether it has moved off one half of the buffer. In order to improve this, sentinals are used in moving characters. Special technique is needed to reduce the overhead to process the character. One such scheme in buffer pairs.

This method needs two tests for each advancement of the forward pointer except for the ends of the buffer halves.

The two tests can be reduced to one if each half has a sentinel character at the end. The sentinel character is a special character which does not form a part of the source program.

Eg. of Sentinel can be eof.

In this method the code needs only one check to see whether the read character is EOF. More tests are needed only when EOF is encountered or end of the characters in source file.

```

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

```

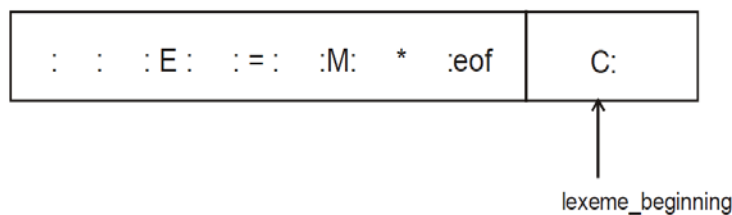


Figure 1.9 Sentinels at end of each buffer half

1.5 SPECIFICATION OF TOKENS

Regular expressions are important notation for supplying pattern. Each pattern matches a string, so regular expressions will serve as names for set of strings.

Strings and Languages

1. Alphabet or character class denotes any finite set of symbols.

(Eg) of symbols are letters and characters.

Note: set $\{0,1\}$ is binary alphabet.

ASCII and EBCDIC are example of computer alphabets.

2. A string over some alphabet is a finite sequence of symbols drawn from that alphabet

Length of the string ' s ' is $|s|$, empty string is denoted by ' ϵ ' length zero.

3. The term language denotes any set of strings over some defined alphabet. Abstract languages like $\{\epsilon\}$ are the set containing only the empty string.
4. If x and y are strings then the concatenation of x and y written xy is the string formed by appending y to x . The empty strings is the identity element under concatenation.

i.e., $s \epsilon \epsilon s = s$.

1.5.1 TERMS FOR PARTS OF A STRINGS

1.	Prefix of s	A string obtained by removing zero or more trailing symbols of string (Eg) Ban is a prefix of Banana.
2.	Suffix of s	A string formed by deleting zero or more of the leading symbols of s . Eg. nana is a suffix of banana
3.	Substring of s	A string obtained by deleting a prefix and a suffix s (eg) nan is a substring of banana.
4.	Proper prefix or proper suffix or substring of s	Any nonempty string x ie respectively a prefix, suffix or substring of s such that $s \neq x$
5.	Subsequence of s	Any string formed by deleting zero, or more not necessarily contiguous symbols from s (Eg) baaa.

1.5.2 OPERATIONS ON LANGUAGES

There are several important operations that can be applied to language. Lexical analysis operations are:

- Union
- Concatenation
- Closure

$$L^0 = \{ \epsilon \}$$

$$L^i = L^{i-1} L \Rightarrow L^i \text{ is } L \text{ concatenated } i-1 \text{ times.}$$

Example : Let $L = \{A, B, \dots Z, a, b, \dots z\}$

$$D = \{0, 1, \dots, 9\}$$

L – Set of alphabets

D – Set of digits

New language are created from L and P by applying the operators defined in the following table.

<i>Operations</i>	<i>Definition</i>
Union of L and M ($L \cup M$)	$L \cup M = \{s s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M (LM)	$LM = \{st s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L (L^*)	$L^* = \bigcup_{i=0}^{\infty} L^i$ <p>L^* zero or more concatenation of L.</p>
Positive closure of L (L^+)	$L^+ = \bigcup_{i=1}^{\infty} L^i$ <p>L^+ denote one or more concatenation of L.</p>

Examples of new languages created from L and D by applying the preceding operators, where

L - Letters

D - Digits

1. LUD is set letters and digits.
2. LD is the set of strings with letter followed by a digits.
3. L4 is set of all 4 letter strings.
4. L^* is set of all strings of letters including ϵ .
5. $L(LUD)^*$ is the set of all strings of letters & digits beginning with a letter.
6. D^+ is the set of all strings of one or more digits.

1.6 RECOGNITION OF TOKENS

We shall see how to recognize token.

Consider the following grammar fragment:

Statement	→	if expr then statement if expr then statement else statement ϵ
expr	→	term relop term term
term	→	id num

terminals → if, then, else, relop, id and num.

generates sets of strings given by the following regular definitions:

if	→	if
then	→	then
else	→	else
relop	→	< <= = <> > =
id	→	letter (letter digit)*
num	→	digit ⁺ (. digit ⁺) ? (E -) ? digit ⁺ ?
digit	→	0 1 2 3 4 5 6 7 8 9
letter	→	A B C Z a b z.

- For this language fragment, the lexical analyzer will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id, and num.
- Lexemes are separated by white space, consisting of non null sequence of blanks, tabs and newlines. The lexical analyzer will recognize white space. It will do so by comparing a string against the regular definition *ws*, below.

$$\text{delim} \rightarrow \text{blank} \mid \text{tab} \mid \text{newline}$$

$$\text{ws} \rightarrow \text{delim}^+.$$

If the match for *ws* is found, the lexical analyzer does not return a token to the passer instead it proceeds to find a token following the white space and returns that to the passer. A lexical analyzer that will isolate the lexeme for the next token in the input buffer and produce as output a pair consisting of the appropriate token and attribute value using the following table:

Regular Exp	Token	Attribute Value
<i>ws</i>	–	–
if	if	–
then	then	–
else	else	–
id	id	Pointer to table entry
num	num	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Transition Diagrams:

As an intermediate step in the construction of a lexical analyzer we just produce a flowchart called a transition diagram. Transition diagrams indicate the actions that takes place when a

lexical analyzer is called by the parser to get the next token. It is used to keep track of information about the characters that are seen while scanning.

Positions in a transition diagram are drawn as circles and are called states. The states are connected by arrows, called edges. Edges leaving state's 'S' have labels indicating the input characters that can next appear after the transition diagram has reached state 'S' the label **other** refers, to any character that is not indicated by any of the other edges leaving 'S'.

Transition diagrams are deterministic (i.e) no symbol can match the labels of two edge leaving one state.

Start state is the initial state of the transition diagram, where control resides when we begin to recognize a token.

Transition diagram for the pattern `>=` and `>`

- Starting state is state 0
- goto state 6 if the input character is `'>'`

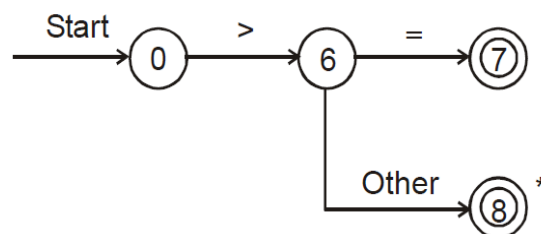


Figure 1.10

One reaching state '6' we read the next input character

Goto state 7 if the input character is `'='`.

Otherwise the edge labeled "other" indicate that we are to go to state 8.

Double Circle indicate the accepting state.

- If failure occur in all transition diagrams then a lexical error has been detected and we invoke an error recovery routine.

A transition diagram for the token `relop` is shown is the following Figure 2.5.

Note: The symbol table is checked to see if the lexeme is found and marked as key word the `instal_id ()` refer as 0. If lexeme is found but a variable, the `instal_id ()` returns a pointer to the

symbol table. Else the entry is added to the symbol table inserted as variable and a pointer is returned to the necessary is returned.

Transition diagram for relational operators in Pascal.

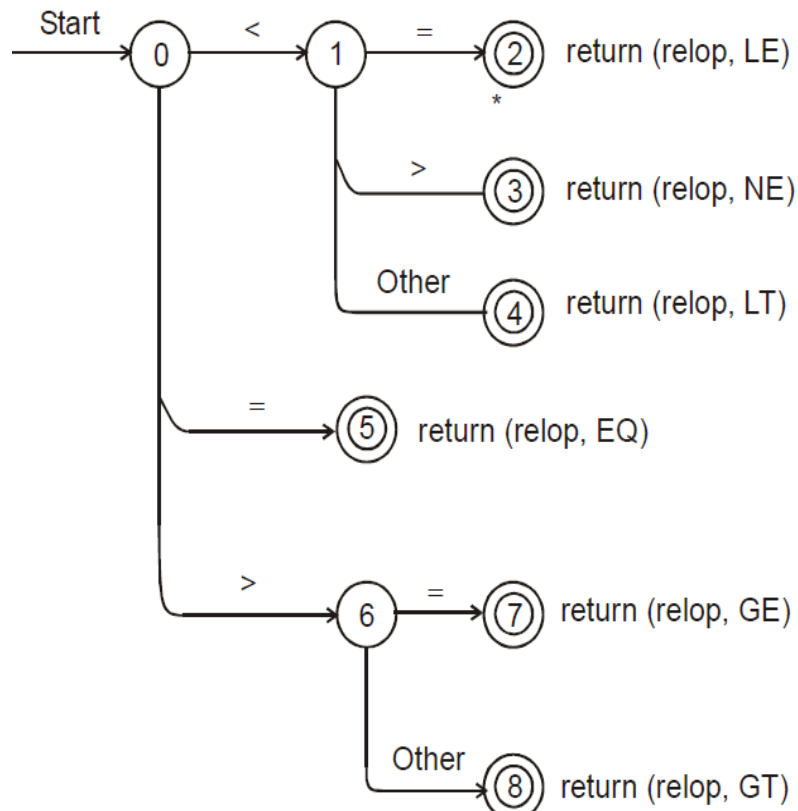


Figure 1.11

Transition Diagram for Identifier and Keywords

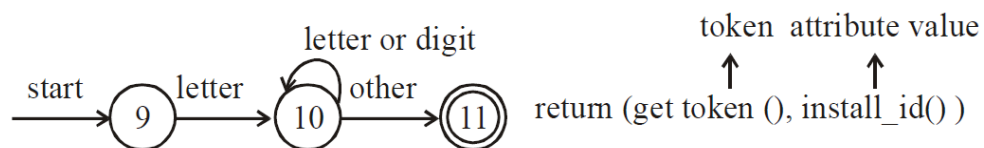


Figure 1.12

The return statement next to the accepting state in the Figure uses `gettoken()` and `install_id()` to obtain the tokens and attribute values respectively.

Note:

The `get token ()` checks for the lexeme in the symbol table. If the lexeme is a key word returns the token else id is returned.

Regular definition of unsigned numbers

$$\text{num} \rightarrow \text{digit}+ (. \text{digit}+) ? (E (+|-) ? \text{digit} +) ?$$
1.7 LEX

Several tools have been built for constructing lexical analyzer from notations based on regular expressions.

A tool called “Lex”, has been widely used to specify lexical analyzer for a variety of languages. We refer to the tool as the lex compiler.

Lex is generally used in the following manner:

1. A specification of a lexical analyzer is prepared by creating a program `lex.l` in the lex language.
2. `lex.l` is run through the lex compiler to produce a C program `lex.yy.c` which consists of tabular representation of a transition diagram constructed from the regular expressions of `lex.l`.
3. Finally `lex.yy.c` is run through the ‘C’ compiler to produce an object program `a.out` which is the lexical analyzer that transforms an input stream into a sequence of tokens.

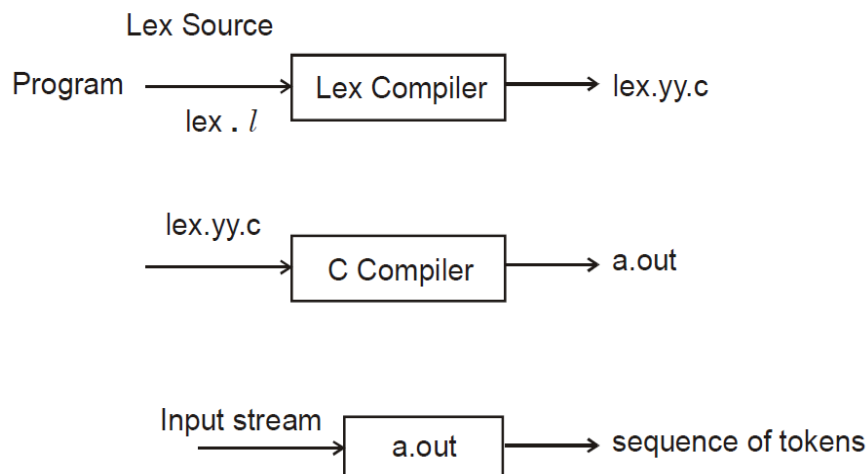


Figure 1.13

Lex Specification

The Lex program consists of three parts:

declaration

%%

transition rules

%%

auxiliary procedures

declaration section includes

- declaration of variables manifest constant.
- regular declaration of identifiers that is declared to represent constants.

Example: int pi = 3.14.

- regular definition components of the regular expressions appearing in the transition rules.

The transition rules of Lex program are statements of the form

P_1 {action 1}

P_2 {action 2}

⋮

P_n {action n }.

- where each P_i is regular expression.
- action i describes what action the lexical analyzer should take when pattern P_i matches a lexeme.

When activated by the parser the lexical analyzer begin reading the remaining input one character at a time until it has found the longest prefix of the input that is matched by one of the regular expressions P_i . Typically action will return control to the parser, with a token P_i .

Example: statement

DO 5 I = 1, .25

DO 5 I = 1, 25

Suppose that all removable blanks are stripped before lexical analyzer begins.

The above statements then appear to the lexical analyzer as

DO5I=1.25

DO5I=1,25.

- In the 1st statement we cannot tell until we see the decimal point that the string DO is part of the identifier DO5I.
- In the second statement DO is a keyword by itself. Lex specifies the keyword

$DO|(\{letter\}|\{digit\})^* = (\{letter\} | \{digit\})^*.$

With the specification the lexical analyzer look ahead is its input buffer for the sequence of letters and digits followed by an equal sign followed by letters and digits followed by a comma to be sure that it did not have an assignment statement.

Example 2:

IF [I, J] = 3 → Here IF is an array name.

IF (condi) statement → Here IF is a keyword.

Note : In this situation to confirm that IF is a keyword, seen forward for right parentheses followed by a letter in the statement before seeing a newline character.

1.7.1 DESIGN OF LEXICAL ANALYZER GENERATION

Lexical compilers are programs for constructing Lexical analyzers. The specification of lexical analyzer may be of the form

$P_1\{ \text{action 1} \}$

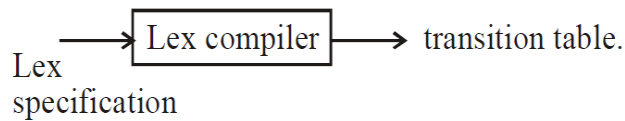
$P_2\{ \text{action 2} \}$

$P_n\{ \text{action n} \}$

P_n - regular expression

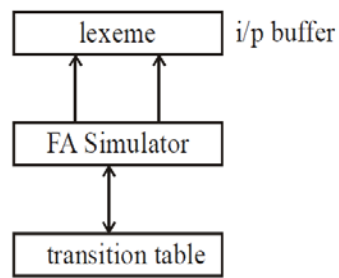
n - program fragment that is to be executed whenever a i/p matches P_i

- If more than one pattern matches, the recognizer choose the longest lexeme matched.
- If there are two or more patterns that match the longest lexeme, the first listed matching pattern is chosen.

Lex Compiler**Figure 1.14**

The Lex compiler constructs a transition table for finite automaton from the regular expression patterns in the lex specification.

The Lexical Analyzer consists of finite automaton simulator that uses this transition table to check if the regular expression in the input buffer matches with any of the patterns, defined.

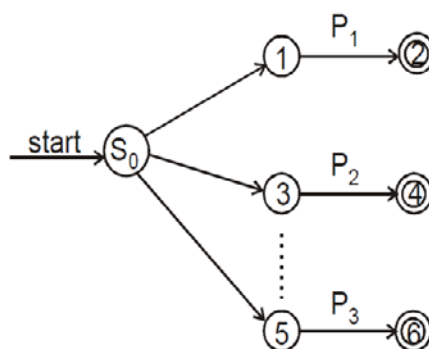
Semantic Lexical Analyzer**Figure 1.15**

Implementation of a Lexical analyzer can be based on either non-deterministic or deterministic automata.

The transition table of NFA is considerably smaller than DFA. But DFA can recognize patterns faster than NFA.

Pattern Matching based on NFA:

The NFA for the pattern $p_1 \mid p_2 \mid \dots \mid p_n$.

**Figure 1.16**

We can modify the algorithm to stimulate NFA by making it recognize the longest prefix of the input that is matched by a pattern.

According to this algorithm even if we find a set of states that contains an accepting state, to find the longest match simulate NFA until it reaches termination (i.e) the set of state from which there is no transitions on the current i/p symbol.

The lex specification is designed to fill the input buffer without having NFA reach termination.

Compiler puts some restriction on the length of an identifier and violation of this limit will be detected when the input buffer overflows.

If the 3 tokens a , abb , a^*b^+ are recongnized by the automata of the following figure.

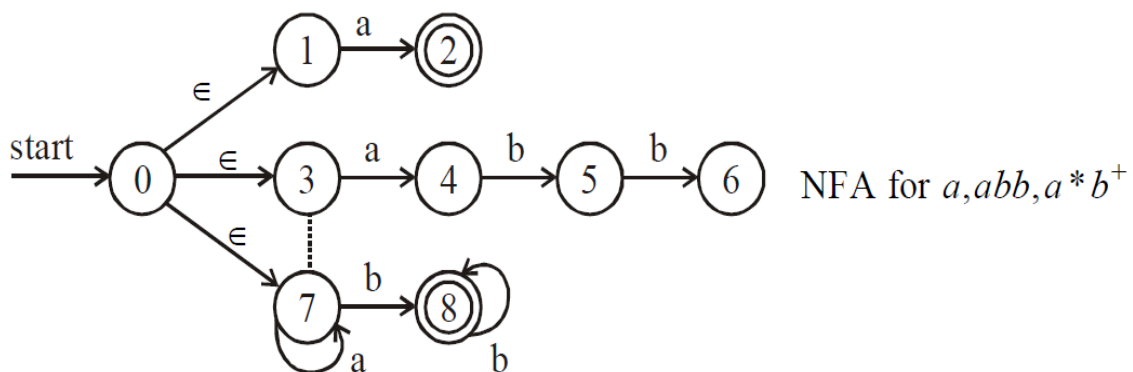


Figure 1.17

The following figure shows the set of states is patterns that match as each character of the i/ p aaba is processed.

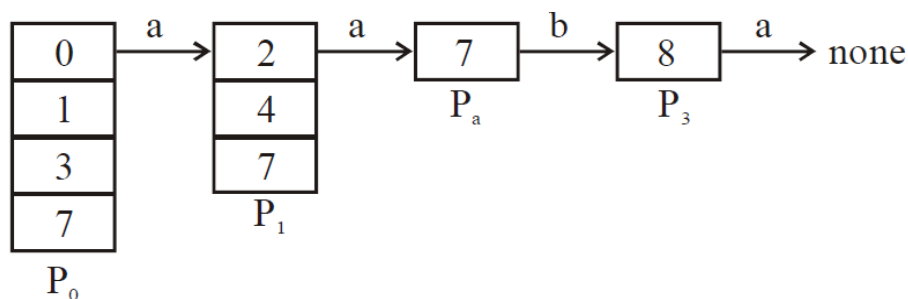


Figure 1.18

Eventhough '2' is an accepting state to recognize the 1st character 'a' there is a transition from state 7 to state 7 on the second i/p character, So we much continue making transitions.

Since the last match occurred after we read the 3rd i/p character. We report that the 3rd pattern has matched the lexeme aab.

Thus action is only executed, if P_i turns out to be the pattern yielding the longest match.

Another approach to the construction of a Lexical analyzer is to use DFA or perform pattern matching:

State	I/p symbol		Pattern Announced
	a	b	
0137	247	8	none
247 ←	7	58	a
8 ←	-	8	a*b+
7 ←	7	8	none
58 ←	-	68	a*b+
68 ←	-	8	abb

Ex: In 247 only '2' is accepting and it is the accepting state of the automaton for regular expression 'a'. Thus 247 DFA state recognizes pattern 'a'. In 0137, none are accepting state. string abb matches a^*b^+ and abb. In such a situation the accepting state corresponding to the pattern listed first in the Lex specification. Refer fig. 2.30. So we declare that abb has been found in DFA state 68.

0137 \xrightarrow{a} 247 \xrightarrow{b} 58 \xrightarrow{a} no state
 \searrow
 has a accepting state (8)

Thus 58 announces that the pattern a^*b^+ has been recongnized.

Implementing the look ahead operator

The lookahead operator / is necessary in some situations. When converting a pattern with / NFA we can treat '/' as if it were E so that we do not look for '/' on the i/p.

NFA recognizing IF

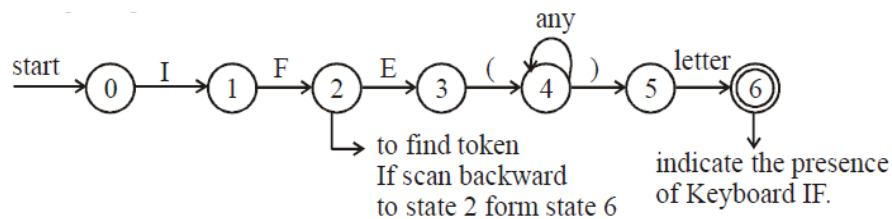


Figure 1.19

Construct RE to DFA

This section explains how to construct DFA directly from NFA using an augmented R.E (r) #.

Step 1: Construct syntax tree T for (r) #.

Step 2: Compute 4 functions nullable, first position, last position, and follow position.

Step 3: Construct the DFA from follow position.

Rules to compute nullable and first position..

Node r	Nullable (r)	First position (n)
n is a leaf labeled ϵ	True	ϕ
N is a leaf labelled with position i	True	$\{i\}$
	Nullable (C_1) (or) Nullable (C_2)	first posn. (C_1) \cup first posn. (C_2)
	Nullable (C_1) (or) Nullable (C_2)	If Nullable (C_1) then first posn. (C_1) \cup first posn. (C_2) else first posn (C_1)

	True	First posn(C_1)
---	------	---------------------

First position (r) gives the set of position that can match the first symbol of a string generated by the sub expression root at r .

Last position (r) gives the set of positions that can match the last symbol in the string.

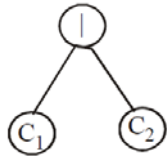
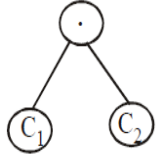
To calculate the first position and last position the need to know which nodes are the root of sub expression that generate empty string. Such nodes are called nullable.


Nullable (r) is true if node n is nullable false otherwise.

Last position

The rules for last posn.(r) are same that of firstpos (r) but C_1 and C_2 are reversed.

Rule:

Node r	Nullable (r)	First position (n)
n is a leaf labeled ε	True	ϕ
N is a leaf labelled with position i	True	$\{i\}$
	Nullable (C_1) (or) Nullable (C_2)	last posn. (C_1) \cup last posn. (C_2)
	Nullable (C_1) (or) Nullable (C_2)	If Nullable (C_2) then last posn.(C_1) \cup last pos.(C_2) else last posn $r(C_1)$

	True	last posn(C_1)
---	------	--------------------

<i>Node</i>	<i>First pos</i>	<i>How to calculate</i>
1	{1}	According to Rule 2 Node 1 is a leaf with position 1
2	{2}	Rule 2 Node 2 is a leaf with position 2
3	{3}	Rule 2
4	{4}	Rule 2
5	{5}	Rule 2
6	{6}	Rule 2

Example: Consider the RE $(a|b)^*abb$.

Solution: Make the grammar as augmented R.E. by attaching # at the end.

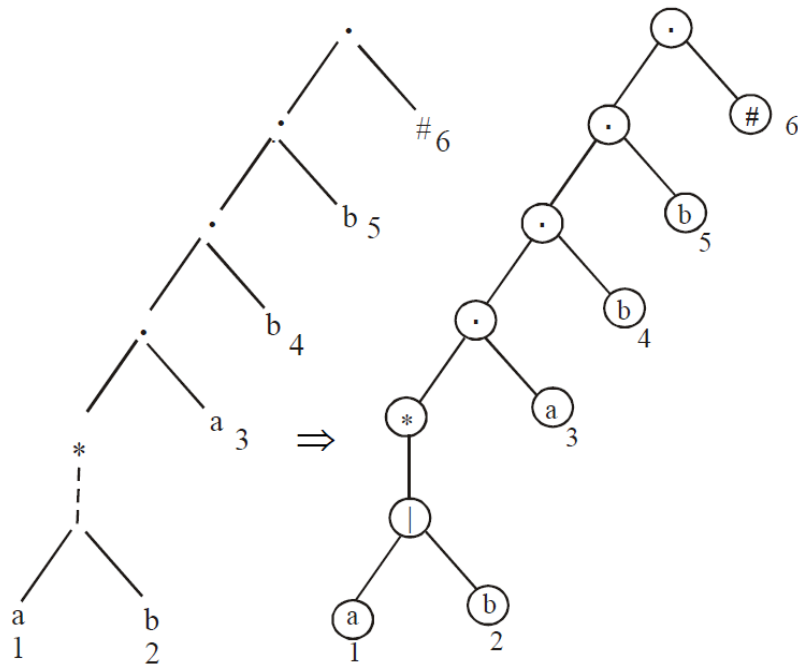
Hence the R.E. becomes $(a|b)^*abb\#$.

Step 1: Construct the syntax tree for $(a|b)^*abb\#$. To draw the tree the operators are * and · in this R.E.

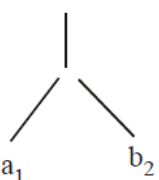


The · called as cat is concatenation operation which is hidden in the R.E. It is implied *. The R.E. can be expanded as $(a|b)^* \cdot a \cdot b \cdot b$.

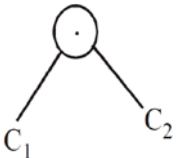
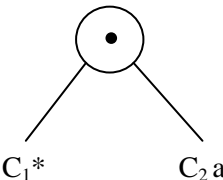
Hence the tree is as shown in Figure. Label the leaf nodes as 1, 2, 3, 4, 5, 6.

Using the rules given calculate the nullable, first position, last position and follow position.

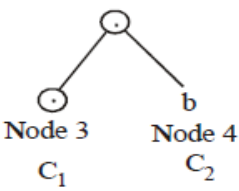


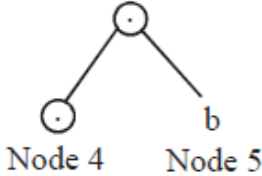
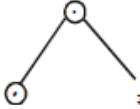
Calculate for the intermediate nodes.

Node	First position	Reason
	{1, 2}	 <p>Rule 3 $(\text{first posn. } C_1) \cup (\text{first posn. } C_2)$</p>
*	{1, 2}	<p>Rule 5: In our example, b is</p>  <p>Hence first position of $$ is first position of $*$.</p> 

.	{1,2,3}	<p>Rule 4</p>  <p>If nullable (C_1) then First posn. (C_1) \cup first posn. (C_2) else first posn. (C_1) To calculate nullable (C_1), refer Table For our example</p>  <p>Check if * is nullable C_1 (Refer Table) C_1 is * C_2 is a V is nullable. Hence first posn. (.) is first posn. (C_1) \cup first posn. (C_2). \therefore First posn(R) is first posn(*) \cup first posn (a) = {1,2} \cup {3} = {1,2,3}</p>
---	---------	--

Calculate the first position

Node	First pos	How to calculate
.	{1,2,3}	 <p>Rule 4: if nullable (C_1) then first posn. (C_1) \cup First pos (C_2)</p>

		<p>else first posn. (C_1)</p> <p>Nullable (\cdot)</p> <p>= Nullable C_1 and Nullable C_2</p> <p>= Nullable (\cdot) and Nullable (b)</p> <p>Recursive Nullable (*) and false True and False \Rightarrow False</p> <p>\therefore First posn. of A is first posn. node C_1 which is \cdot hence b is $\{1,2,3\}$.</p>
•	$\{1,2,3\}$	 <p>The above said rule is applicable.</p> <p>Rule 4.</p> <p>Nullable (\cdot) = Nullable (C_1) and Nullable (C_2)</p> <p>= Nullable (\cdot) and Nullable (b)</p> <p>Recursive = Nullable (\cdot) and false</p> <p>Recursive = Nullable (*) and false</p> <p>= True and False</p> <p>= False</p> <p>\therefore First posn. (\cdot) is First posn. (C_1)</p> <p>= First posn. (\cdot)</p>
•	$\{1,2,3\}$	 <p>Do as above said.</p>

Enter in the tree to the left of the node to denote as “firstpos”.

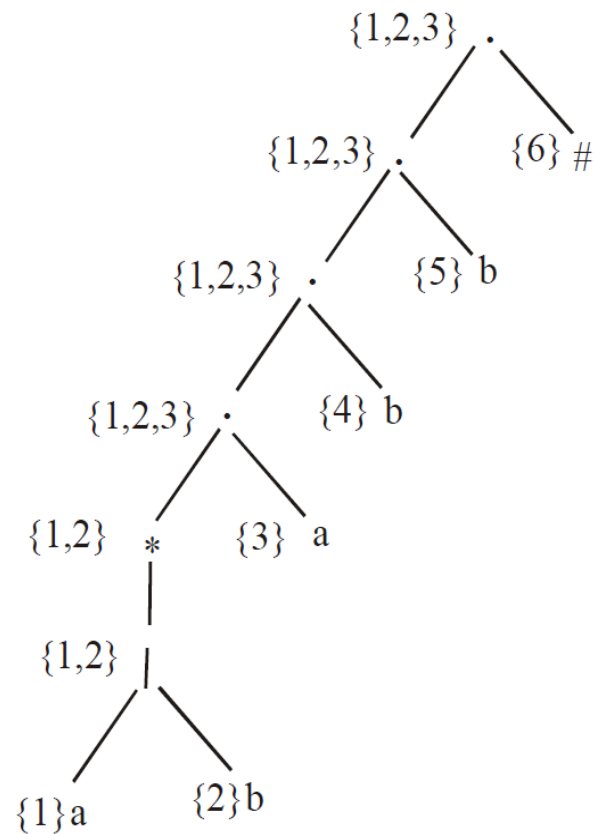
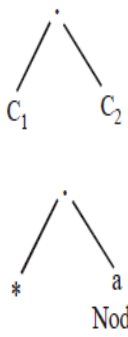
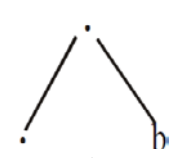
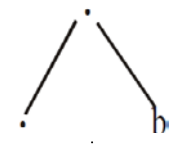



Figure 1.20

Calculate Last Position of the Node.

Node a	Nullable (a)	Last position (a)	Reason
1	False	{1}	Since Node 1 is a leaf labeled with position 1.
2	False	{2}	Node is a leaf labeled with position 2
3	False	{3}	The above said rule.
4	False	{4}	
5	False	{5}	
6	False	{6}	

For Intermediate Nodes.

Node	Last position	Reason
	{1,2}	To calculate nullable (C_1), refer Table. Last posn. (C_1) \cup Last posn. (C_2) Last posn. (C_1) \cup Last posn. (C_2) $\{1\} \cup \{2\}$ $\{1,2\}$
*	{1,2}	Last posn. (1)
.	{3}	 <p>If available (C_2) is true.</p> <p>Nullable (a) is False.</p> <p>\therefore Last para (\cdot) = Last posn. (3) = {3}</p>
.	{4}	 <p>Nullable (6) = False</p> <p>Last posn. (\cdot) = Last posn. (6) = {5}</p>
.	{5}	 <p>Nullable (6) = False</p> <p>Last posn. (\cdot) = Last posn. (6) = {5}</p>
.	{6}	 <p>Nullable (*) False</p> <p>\therefore Last posn. (#) = {6}</p>

Insert the last position the node in the augmented tree to the right of the node.

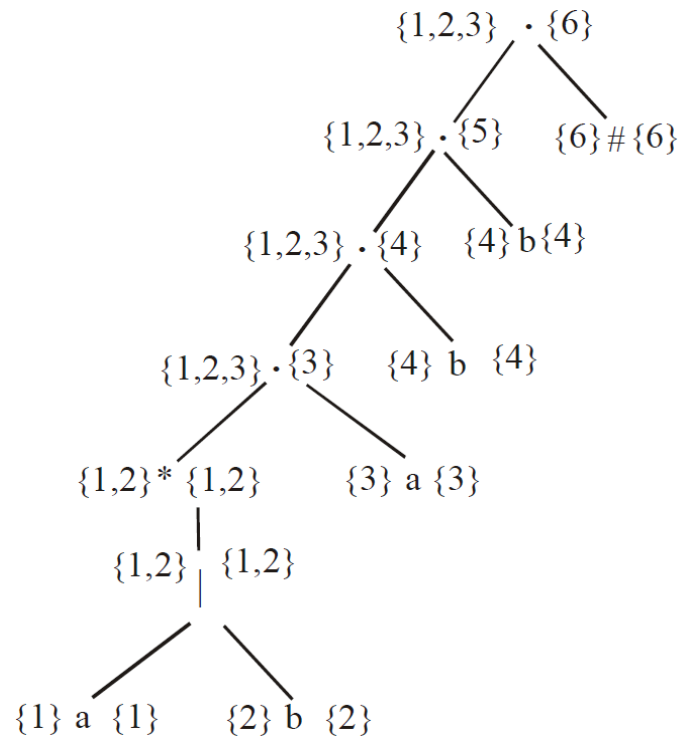


Figure 1.21

To calculate the followpos

The steps followed are;

- (i) If n is a cat node with left child C_1 and right child C_2 and i is a position in last position (C_1) then all the position in first position (C_1) are in follow position (i).
- (ii) If n is a star node and P is a position in last position (n) then all position in first position (n) are in follow position (i).

Initial state \Rightarrow firstpos(Root)

$$A = \{1,2,3\}$$

$$\text{Dtran}[A,a] = \text{followpos}(\{1,3\})$$

$$= \text{followpos}(1) \cup \text{followpos}(3)$$

$$= \{1,2,3\} \cup \{4\}$$

$$= \{1,2,3,4\} \Rightarrow B$$

$$\text{Dtran}[A,b] = \text{followpos}(\{2\})$$

$$= \{1,2,3\} \Rightarrow A \quad \text{Dtran}[B,a] = \text{followpos}(\{1,3\})$$

$$= \{1,2,3,4\} \Rightarrow B \quad \text{Dtran}[B,b] = \text{followpos}(\{2,4\})$$

$$= \text{followpos}(2) \cup \text{followpos}(4)$$

$$= \{1,2,3,4\} \cup \{5\}$$

$$= \{1,2,3,4,5\} \Rightarrow C$$

$$\text{Dtran}[C,a] = \text{followpos}(\{1,3\})$$

$$= \text{followpos}(1) \cup \text{followpos}(3)$$

$$= \{1,2,3\} \cup \{4\}$$

$$= \{1,2,3,4\} \Rightarrow B$$

$$\text{Dtran}[C,b] = \text{followpos}(\{2,5\})$$

$$= \{1,2,3,4\} \cup \{6\}$$

$$= \{1,2,3,6\} \Rightarrow D$$

$$\text{Dtran}[D,a] = \text{followpos}(\{1,3\})$$

$$= \text{followpos}(1) \cup \text{followpos}(3)$$

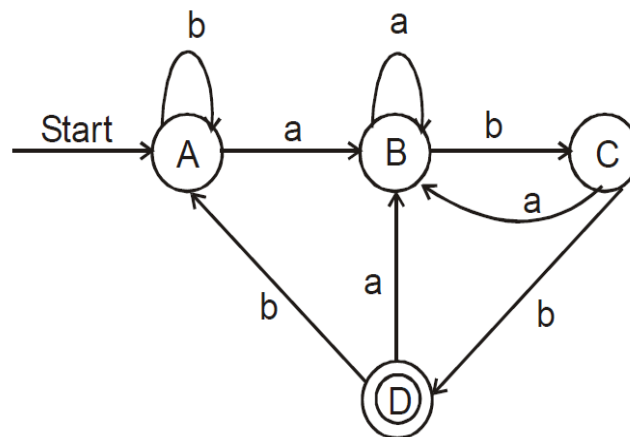
$$= \{1,2,3\} \cup \{4\}$$

$$= \{1,2,3,4\} \Rightarrow B \quad \text{Dtran}[D,b] = \text{followpos}(\{2\})$$

$$= \{1,2,3\} \Rightarrow A$$

Transition table

Q	a	b
Ⓐ	B	A
B	B	C
C	B	D
Ⓓ	B	A

Transition diagram**1.8 FINITE AUTOMATON**

A recognizer for a language is a program that takes as input a string x and answers “yes” if x is a sentence of the language and “no” otherwise.

We compile a regular expression into a recognizer by constructing a generalized transition diagram called a finite automaton. A finite automaton can be deterministic or non-deterministic means that more than one transition out of the state may be possible on the same input symbol.

Both of them can recognize, exactly what regular expressions can denote.

Deterministic finite automata can lead to faster recognizers than non-deterministic finite automata.

Non Deterministic Finite Automata (NFA)

A Non Deterministic Finite Automata is a mathematical model that consists of

1. a set of state Q
2. a set of input symbols Σ
3. a transition function move that maps state-symbol pair to sets of states ($\delta = 2^Q$).
4. a state is distinguished as the start states (q_0).
5. a set of states is distinguished as the accepting states, F .

NFA can be represented by a labeled directed graph called a transition graph in which the nodes are the states and the labeled edges represent the transition function. The graph looks

like a transition diagram but the same character can label two or more transition out of one state and edges can be labeled by the special symbol Σ as well as by input symbols.

The transition graph for an NFA that recognizes language $(a|b)^* abb$

The set of states of the NFA is $\{0,1,2,3\}$ and the input symbol alphabet is $\{a,b\}$.

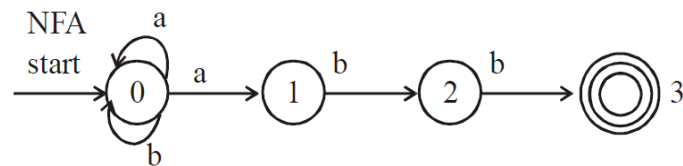


Figure 1.22

1.8.1 TRANSITION TABLE

In a computer the transition function of an NFA can be implemented in several different ways. The easiest implementation is the transition table.

State	Input symbol	
	a	b
0	{0, 1}	{0}
1	~ -	{2}
2	~ 0	{3}

Advantages :

- Provides fast access to the transition of a given state on a given character.

Disadvantages :

- It can take up a lot of space, when the input alphabet is large and most transitions are to the empty set.

Adjacency List: Representations of the transition function provide more compatible implementations.

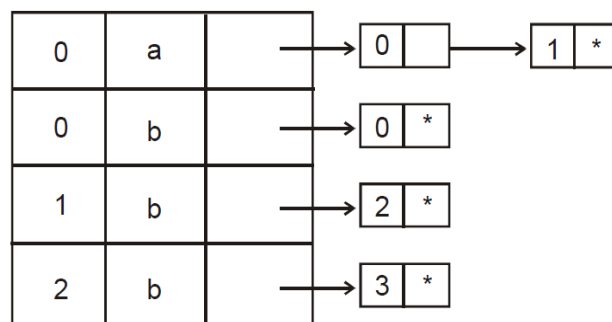


Figure 1.23

Disadvantages:

- Access to a given transition is slower.

An NFA accepts an input

- String x if and only if there is some path in the transition graph from the start state to some accepting state, the NFA accepts the i/p strings. (eg) $aabb$ is accepted by the path from '0' following the edge labeled 'a' to state '0' again then to state 1, 2 & 3 via edges labeled a,b, & 'b' respectively

A path can be represented by sequence of state transitions called moves

The following diagram shows the moves made in accepting input string $aabb$

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

$aa^*|bb^*$

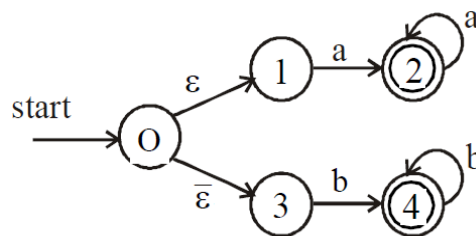


Figure 1.24

The several other, sequence of moves may be made on the input string $aabb$, but none of the others happen to end in an accepting state (e.g.) another sequence of moves $aabb$ keeps reentering the non-accepting state '0'

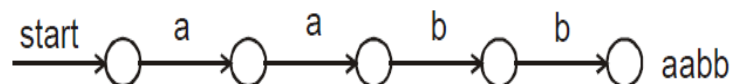


Figure. 1.25

Deterministic Finite Automata

A deterministic finite automaton is a special case of a non-deterministic finite automaton in which

1. No state has an ϵ transition (i.e.,) NO transition on input ϵ

2. For each state δ and input symbol 'a' there is at most one edge labeled 'a' leaving S

A deterministic finite automation has at most one transition from each state on any input. If we are using a transition table to represent the transition function of a DFA, then each entry in the transition table is a single state.

Advantage

- It is easy to determine whether deterministic finite automation accepts as input string since, there is at most one path from the start state labeled by that string.

Transition graph of a deterministic finite automation accepting the language $(a/b)^* abb$

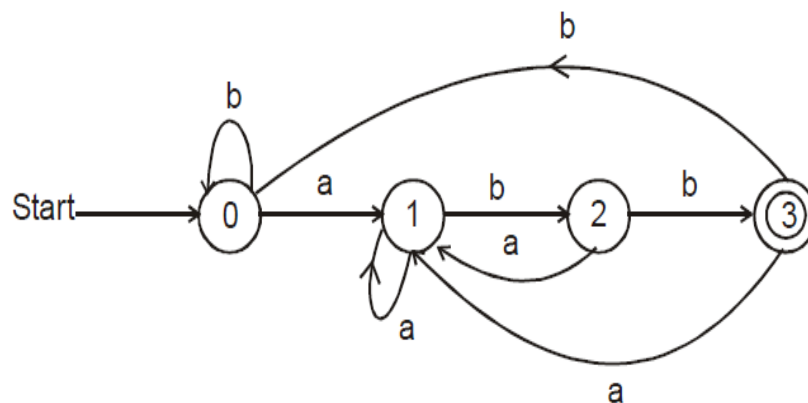


Figure 1.26

Conversion of an NFA into a DFA

The algorithm to convert NFA to DFA is often called the subset construction. The general idea behind the NFA-DFA construction is that each DFA state corresponds to a set of NFA states. The DFA uses its state to keep track of all possible states the NFA can be in after reading each input symbol.

Step 1 :

To construct NFA from Regular Expression using Thompson's Construction.

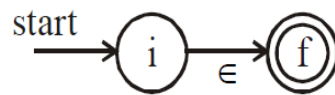
Rules for constructing NFA

Input : RE r over alphabet Σ

Output : NFA N accepting $L(r)$

Method :

When input is \in the NFA is

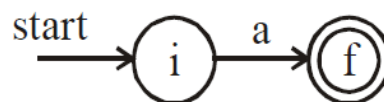
**Figure 1.27**

i is the new start state

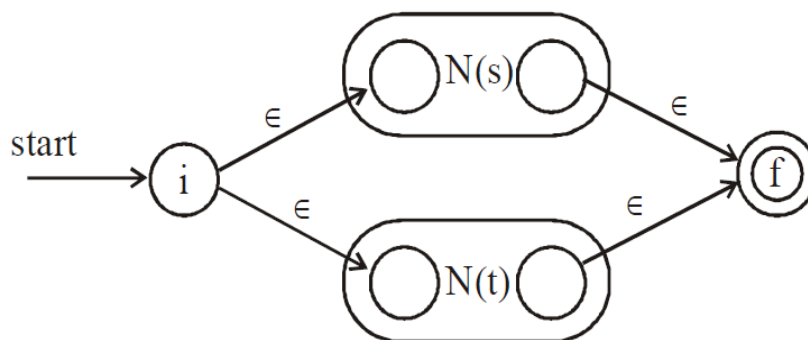
f is the new accepting state

The above NFA recognizes $\{\epsilon\}$

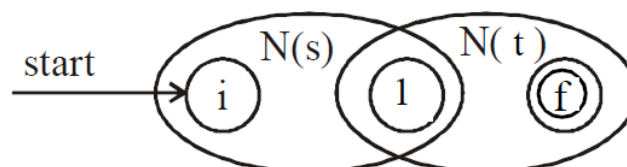
When RE input is 'a', NFA is

**Figure 1.28**

When the RE is 's|t' the NFA is

**Figure 1.29**

When the RE is 'st' the NFA is N(st)

**Figure 1.30**

When the RE is s^* the NFA (s^*) is

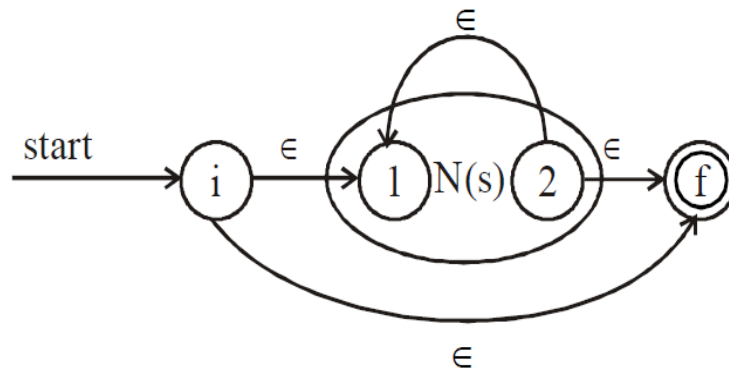
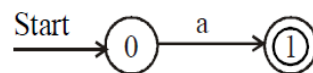


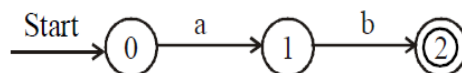
Figure 1.31

Example :

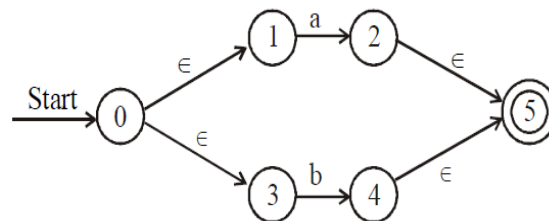
1) $RE = a$



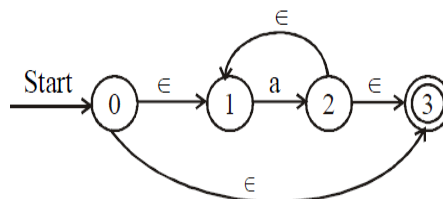
2) $RE = a . b$



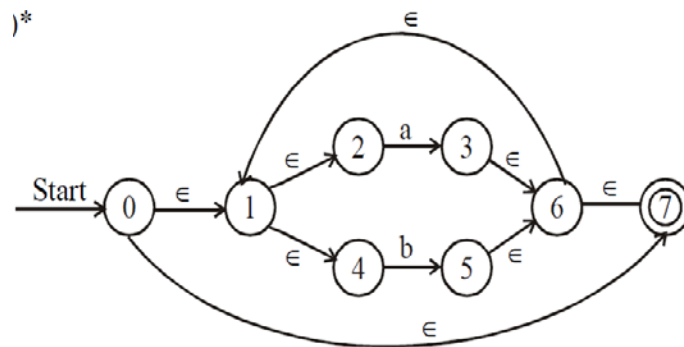
3) $RE = a|b$



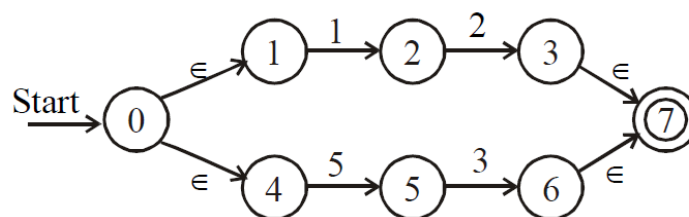
4) $RE = a^*$ [*means 0 or more occurrences]



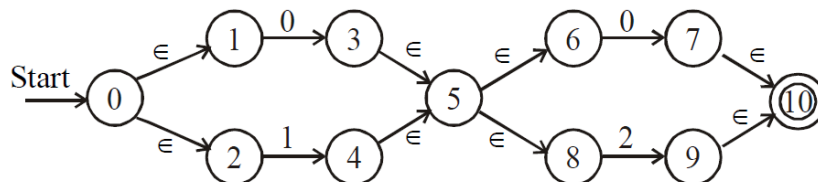
5) $RE = (a|b)^*$



6) RE = 12/53



7) RE = (0|1) (0|2)



Algorithm:

Input: An NFA N.

Output: A DFA D accepting the same language.

Method: Algorithm constructs a transition table Dtran for D. We construct Dtran so that

D will simulate “in parallel” all possible moves N can make, on a given input string.

The operations in the following table keep tracking sets of NFA states.

STEP 1: Operations on NFA States:

Operation	Description
(1) ϵ -Closure (S)	Set of NFA states reachable from, NFA state
(2) ϵ -Closure (T)	‘S’ on ϵ transition alone.

(3) Move (T, a) (or) $\delta(T, a)$	Set of NFA states to which there is a transition on input symbol 'a' from some NFA status in T.
--	---

STEP 2: THE SUBSET CONSTRUCTION

Initially ϵ -closure (S0) is the only state in D states and it is unmarked. While there is an unmarked state T in D states do begin.

mark T;

for each input symbol a do begin

$U := \epsilon\text{-closure}(\text{move}(T, a));$

if U is not in D states then

add U as an unmarked state to D states;

$D\text{tran}[T, a] = U$

end

end

STEP 3: COMPUTATION OF ϵ -closure

push all state in T onto stack

initialize ϵ -closure (T) to T;

While stack is not empty do begin

pop t;

for each state u with an edge from i to u labeled ϵ do

if it is not its ϵ -closure (T) do begin

add u to ϵ -closure (T);

push u onto stack

end

end

Simulating NFA:

```

S :=  $\epsilon$ -closure ([S0]);
a := next char;
while a  $\neq$  eof of do begin
  S :=  $\epsilon$ -closure move ( $\delta(s,a)$ );
  a := next char
end
if  $S \cap F \neq \emptyset$  then
  return "yes";
else
  return "no";

```

Difference between NFA & DFA

NFA:- It should have one start state It can have one or more accepting or final state. There can be ϵ transitions on a particular input symbol.

DFA; It should have one start state. It should have one or more accepting state. There cannot be ϵ transitions. There can be only one transition of the DFA

i - initial state

N - any NFA state

D - only DFA state

a - input symbol

Procedure:

Step 1 :- Find ϵ -closure (i) and let $N = \epsilon$ -closure (i) and make the state as the first DFA state D

Step 2 :- Find ϵ -closure ($\delta(D,A)$) and check if it is already a marked DFA state, if not then name it as D1 Also marked $DTrans (D,a) = D$, If it is already available then marked $DTrans (D,a) =$ already available state.

Step 3:- Step 2 must be done for all n input symbols for D and find DFA state $D_1 D_2$ if possible

Step 4:- For any DFA state D_1, D_2, \dots, D_n do step 2 for all n Input symbols till we find no new DFA states.

$DTran(D, a)$ represents DFA transition of any DFA state D on the Input a

$DTran(D, a) = D_1$ means DFA state D reaches D_1 on 'a'

1.9 REGULAR EXPRESSIONS (RE)

RE allows to define the set of new language that can be built using the operators.

- An identifier is a letter followed by zero or more letters or digits with this notation. Identifier is defined as $\text{letter}(\text{letter} \mid \text{digit})^*$ is a regular expression built out of (LUD).
- A regular expression is built u_p out of simpler regular expressions using a set of defining rules.
- Each regular expression ' r ' denotes a language $L(r)$ is formed by combining in various ways. The language is denoted by subexpression of ' r '.

Rules that define the regular expression over alphabet Σ

1. ϵ is a regular expression that $\{\epsilon\}$ the set with empty string.
2. If ' a ' is a symbol in Σ , then ' a ' is a RE that denotes $\{a\}$ the set containing string ' a '
3. Suppose ' r ' & ' s ' are REs denoting language $L(r)$ & $L(s)$, then.
 - a) $(r) \mid (s)$ is a RE denoting $L(r) \cup L(s)$
 - b) $(r)(s)$ is a RE denoting $L(r)L(s)$
 - c) $(r)^*$ is a RE denoting $(L(r))^*$
 - d) (r) is a RE denoting $L(r)$

A language denoted by a RE is said to be a Regular Set. The specifications of a RE is an example of recursive definition.

Rules (1) & (2) are the basis of definition.

Unnecessary parenthesis can be avoided in RE if:

1. The unary operator '*' has highest precedence and its left associative.
2. Concatenation has second highest precedence and is left associative.
3. '|' has lowest precedence and is left associative.

Ex: $((a) | (b) * (c))$ is equivalent to $a|b*c$

Ex: Let $\Sigma = \{a, b\}$

1. The RE $a|b$ denotes the set $\{a, b\}$.
2. The RE $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$.
3. The RE a^* denotes the set of all strings of zero or more a's
 $\{ \epsilon, a, aa, aaa, \dots \}$.
4. The RE $(a|b)^*$ denotes set of all string containing the set of zero or more instances of an a's or b's.
5. The RE $a|a^*b$ denotes the set containing the string 'a' and all string consisting of zero or more 'a' followed by b.





Note: If two Regular Expressions ' r ' & ' s ' denote same language, we say ' r ' & ' s ' equivalent.

Ex: $(a|b) = (b|a)$

There are number of algebraic laws obeyed by Regular Expressions and these can be used to manipulate the regular expressions into equivalent forms.

Algebraic Laws that hold for regular expressions r , s and t .

Algebraic Properties of Regular Expressions.

Axion	Perception
$r s = s r$	$ $ is commutative
$r (s t) = (r s) t$	$ $ is associative
$(rs)t = r(st)$	concatenation is associative
$r(s t) = rs rt$ 	concatenation distributive over $ $
$(s t)r = sr tr$ 	
$\epsilon r = r$ 	ϵ is identity element for concatenation
$r\epsilon = r$ 	
$r^* = (r \epsilon)^*$	relation between $*$ and ϵ is idempotent
$r^{**} = r^*$	

Regular Definition

For notational convenience, we give names to REs.

Ex. Consider sequence of definition which follows:

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

:

:

$$d_n \rightarrow r_n$$

$d_i \rightarrow$ distinct name

$r_i \rightarrow$ RE over the symbols

Eg. of Regular definition for pascal identifiers is

$$\text{letter} \rightarrow A|B|\dots\dots\dots|Z|a|b|\dots\dots\dots|z$$

$$\text{digit} \rightarrow 0|1|\dots\dots\dots|9$$

$$\text{id} \rightarrow \text{letter} (\text{letter}|\text{digit})^*$$

Note: For pascal identifiers is the set of strings of letters and digits beginning with a letter.

(Eg) Consider the pascal string such as 5.280, 39.3-7, 6,336E4,

The following regular definition provides precise specification for the preceeding class of strings.

digit	→	0 1 9
digits	→	digit digit*
optional_fraction	→	digits ∈
optional_exponent	→	(E (+ - ∈) digits) ∈
num	→	digits optional_fraction optional_exponent

Notational Short Hand:

Certain constructors occur so frequently in RE, that it is convenient to introduce notational shorthands for them.

1. One or more instance: ‘+’ means “one or more instance of”

If ‘ r ’ is a RE that denote language $L(r)$ then $L(r)^+$ is a RE that denote the language r^+

Note: $r^* = r^+ | ∈$.

2. Zero or one instance:

The unary postfix operator ‘?’ means “zero or one instance of”

Note: r is a short hand for $r | ∈$

If ‘ r ’ is a RE then (r) is a RE that donates the language $L(r) \cup \{ ∈ \}$ Using ‘+’ and ‘?’ num can be redefined as following:

digit	→	0 1 9
digit	→	digit+
optional_fraction	→	(.digits) ?
optional_exponent	→	(E (+ -) ? digits) ?
num	→	digits optional_fraction optional_exponent

3. Character Class

Notation $[a\ b\ c]$ where a, b, c are alphabet symbols denote RE $a|b|c$

Character class $[a-z]$ denote the RE $a|b|.....|z$

Non-Regular Sets.

Some language cannot be described by any RE.

Eg: of programming language constructs that cannot be described by regular expressions

- Regular expressions can't be used to describe balanced or nested constructs.

Eg: Set of all string of balanced parenthesis can't be described by RE, but it can be specified by a Context Free Grammar (CFG)

- Repeating Strings cannot be described by Regular Expression.

The set $\{ w cw \mid w \text{ is string of } a's \text{ and } b's \}$ cannot be denoted by either RE or a CFG.

- Two arbitrary numbers can't be compared to see whether they are the same.

Note: RE can be used to denote only a fixed no of repetitions or an unspecified no of repetitions of a given construct.

1.9.1 TRANSITION DIAGRAM FOR UNSIGNED NUMBERS

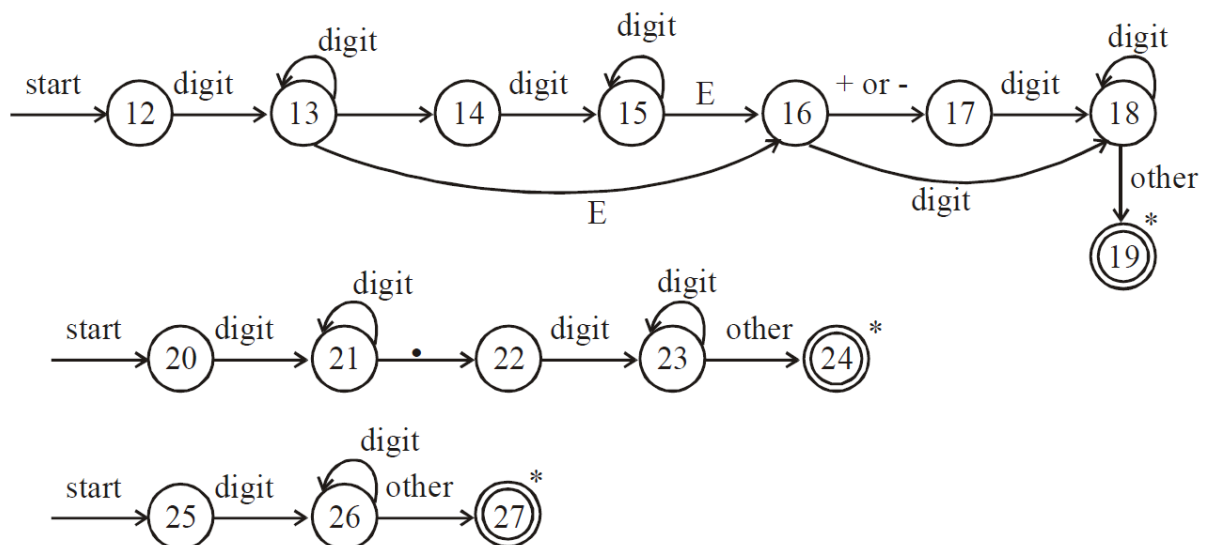


Figure 1.32

Note: The definition $\text{num} \rightarrow \text{digit}^+ (. \text{digit}^+)? (E(+|-)? \text{digit}^+)?$ is of the form.

digits fraction? exponent?

Where fraction and exponent are optional. The lexeme for a given token must be the longest possible.

Example: lexical analyzer must not stay with 12 (or) 12.3 when the input is 12.3 E4.

When state **19** , **24** or **27** is reached the procedure `install_num` enters lexeme into a table of numbers and return a pointer to the created entry.

A transition diagram for recognizing ws

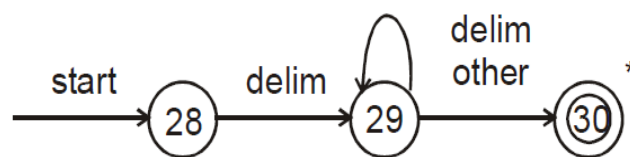


Figure 1.33

Implementing a Transition Diagram:

A sequence of transition diagrams can be converted into a program to look for the tokens specified by the diagram.

Each state gets a segment of code. If there are edges leaving a state. Its code reads a character and selects an edge to follow. A function `nextchar()` is used to read the next character from the input buffer.

If the input character read, matches none of the transition diagram, an error recovery routine is called two variables “state” and “start” keeps track of the present state and the starting of the current transition diagram respectively.

1.10 MINIMIZATION OF DFA

Collection of DFA states are named as C

Step 1 : Divide C into two group CS and CF where CF will have DFA states which are having final or accepting state. CS will be having DFA states having non final states of DFA.

Step 2 :

- 2.1. Divide any group C such that two states S1, S2 are in the same group of S1 and S2 have same transition for all the input symbols.
- 2.2. Repeat 2.1 for all new sub groups obtained in step1.

Step 3 : After partitioning into subgroups for CS and CF representative from each group may be selected as follows. If (ABC) is a group and A,B,C have same transition on all input symbols. A may be selected as a representative and A will replace all B,C occurrences.

Step 4 : A dead state may be removed. A state is dead if that state is not accepting any transitions and has no transition on all the IP symbols.

Step 5 : Remove all the state that are not reachable from the state.

Example 1: NFA N for $(a/b)^* abb$.

Step 1: Draw the NFA.

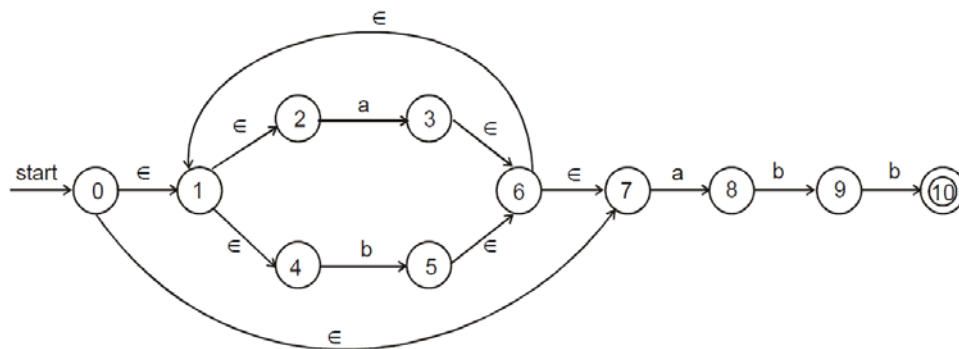


Figure 1.34

Step 2: Find the ϵ -closure of the states.

Set of states reachable from state '0' (start state) on ϵ -transition is:

$$\epsilon\text{-closure}(0) = \{0, 1, 2, 4, 7\} = A.$$

Set of states reachable from the states in set 'A' on 'a' input symbol is:

$$Aa = S(A, a) = \{3, 8\}. \text{ As is move } (A, a).$$

Set of states reachable from the states in set 'Aa' on ϵ -transition is

$$\in\text{-closure (move (A,a))} = \in\text{-Closure (3, 8)} = \{3, 8, 6, 7, 1, 2, 4\}$$

(Note: The states in A_a is also included is the set of \in closure of A_a)

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

Write in ascending order of states = { 1, 2, 3, 4, 6, 7, 8} = B

Set of states reachable from the states on set 'A' on 'b' input symbol is

$$Ab = \delta(A, b) = \{5\}$$

$$\text{Set of is } \in\text{-closure (Ab)} = \{5, 6, 7, 1, 2, 4\} = C$$

$$\text{Consider } B = \{1, 2, 3, 4, 6, 7, 8\}$$

$$\delta(B, a) = (Ba) = \{3, 8\}$$

$$\begin{aligned} \in\text{-closure } (Ba) &= \{3, 8, 6, 7, 1, 2, 4\} \\ &= \{1, 2, 3, 4, 6, 7, 8\} = B \end{aligned}$$

$$Bb = \{9, 5\}$$

$$\begin{aligned} \in\text{-closure } (Bb) &= \{9, 5, 6, 7, 1, 2, 4\} \\ &= \{1, 2, 4, 5, 6, 7, 9\} = D \end{aligned}$$

$$\text{Consider } C = \{1, 2, 4, 5, 6, 7\}, \quad \text{then } Ca = \{3, 8\}$$

$$\begin{aligned} \in\text{-closure } (Ca) &= \{3, 8, 6, 7, 1, 2, 4\} \\ &= \{1, 2, 3, 4, 6, 7, 8\} = B, \quad \text{Now } Cb = \{5\} \end{aligned}$$

$$\begin{aligned} \in\text{-closure } (Cb) &= \{5, 6, 7, 1, 2, 4\} \\ &= \{1, 2, 4, 5, 6, 7\} = C \end{aligned}$$

$$\text{Consider } D = \{1, 2, 4, 5, 6, 7, 9\}, \quad \text{then } Da = \{8, 3\}$$

$$\begin{aligned} \in\text{-closure } (Da) &= \{3, 8, 6, 7, 1, 2, 4\} \\ &= \{1, 2, 3, 4, 6, 7, 8\} = B. \quad Db = \{10, 5\} \end{aligned}$$

$$\begin{aligned} \in\text{-closure } Db &= \{10, 6, 5, 7, 1, 2, 4\} \\ &= \{1, 2, 4, 5, 6, 7, 10\} = E \end{aligned}$$

$$Ea = \{3, 8\}$$

$$\in\text{-closure } (Ea) = B \quad Eb = \{5\}$$

$$\in\text{-closure } (Eb) = C$$

States are repeated hence stop.

Step 3: Draw the Transition Table.

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

are repeated. [A also has the same state as that of C, E, but cannot be minimized since it is the starting state].

Step 4: Minimized Table

	a	b
A	B	A
B	B	D
D	B	E
E	B	A

E - Accepting State

A - Starting State

Step 5: Draw the DFA.

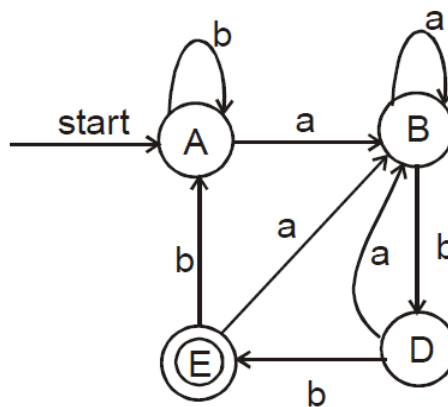
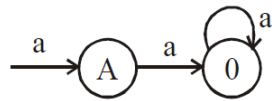
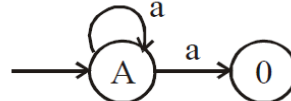


Figure 1.35

Note:

$$a^* = aa^*$$

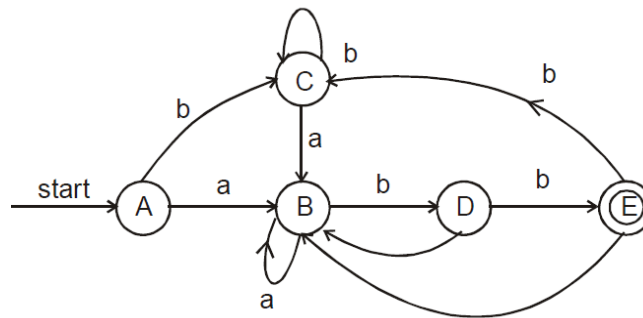
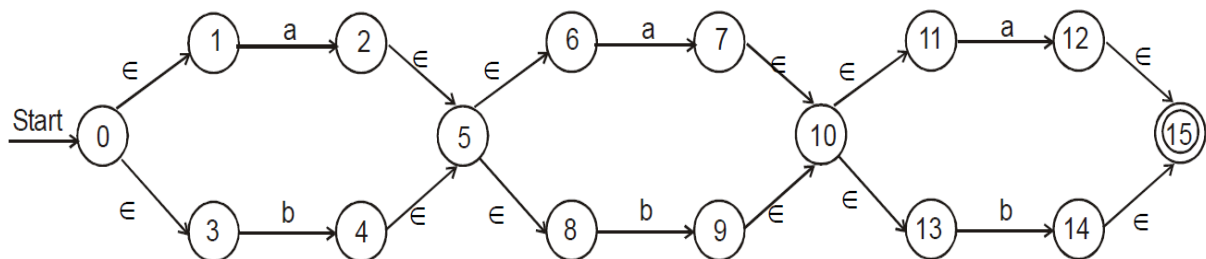
DFA a^+ (or) aa^* **Figure 1.36****NFA** a^+ (or) aa^* **Figure 1.37**

$$aa^* = \{a, aa, \dots\}$$

$$a^* a = \{a, aa, \dots\}.$$

Step 5 and 6 Minimization of DFA.

	a	b	
→ A	B	A	ABCDE
B	B	D	(AC)BDE
D	B	E	AABDE
E	B	A	ABDE

**Figure 1.38****Example 2:** RE = $(a \mid b)(a \mid b)(a \mid b)$ 

Step 2: Find the ϵ -closure of all the states.

Step 1 : ϵ -closure (0) = {0, 1, 3} = A

Step 2 : move (A, a) = {2}

ϵ -closure (2) = {2, 5, 6, 8} - B

Step 3 : move (A, b) = {4}

ϵ -closure (4) = {4, 5, 6, 8} - C

Step 4 : move (B, a) = {7}

ϵ -closure (7) = {7, 10, 11, 13} - D

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

ϵ -Closure (9) = {9, 10, 11, 13} - E

Step 6 : move (C, a) = {7}

ϵ -Closure (7) = D

Step 7 : move (C, b) = {9}

ϵ -Closure (9) = E

Step 8 : move (D, a) = {12}

ϵ -Closure (12) = {12, 15} - F

Step 9 : move (D, b) = {14}

ϵ -Closure (14) = {14, 15} - G

Step 10 : move (E, a) = {12}

ϵ -Closure (12) = F

Step 11 : move (E, b) = {14}

ϵ -Closure (14) = G

Step 12 : move (F, a) = $\{\phi\}$

Step 13 : move (F, b) = $\{\phi\}$

Step 14 : move (G, a) = $\{\phi\}$

Step 15 : move (G, b) = $\{\phi\}$

Repeated states hence stop.

Now draw the Transition Table,

→ A	B	C
B	D	E
C	D	E
D	F	G
E	F	G
F	-	-
G	-	-

Note: Initial State to be marked as ○

Final State as 0 (or) ○*.

Do the minimization

G_1 G_2

Step 1: Sub group (A B C D E) (F G) [(i.e., Final States and Non-Final States)]

Step 2: Check each subgroup for equivalence,

G_3 G_4

$G_1 = (A) (D E) (B C)$

$G_2 = (FG)$

Step 3: Now check, G_3 and G_4 for equivalence.

$G_3 = (A) (D E)$

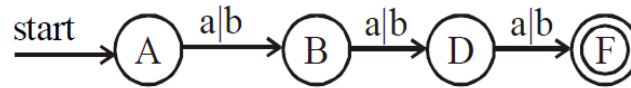
$G_4 = (B C)$

Step 4: We conclude 'B' and 'C' states are equivalent and F, G are equivalent and DE states too.

Step 5: Minimized Table is given as (D E), (F G) and (B C) are equivalent.

	a	b
→ A	B	B
B	D	D
D	F	F
F	-	-

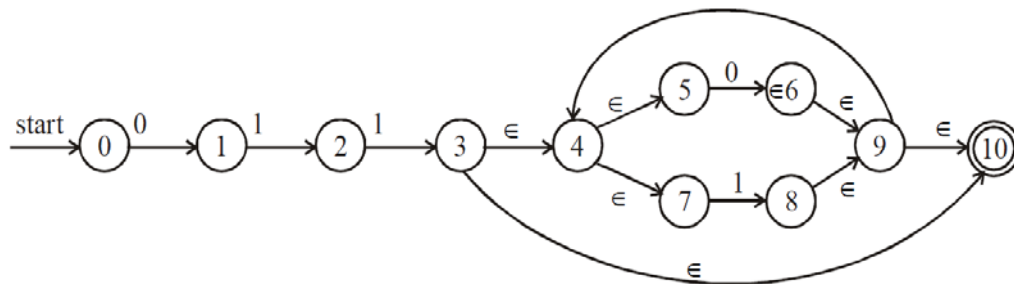
Step 6: DFA is



Example 3: Determine the minimized DFA for RE

011 (01)*

Step 1:



Step 2:

1. ϵ -closure (0) = {0} = A

$$A_0 = d(A, 0) = \{1\}$$

$$\epsilon\text{-closure}(d(A, 0)) = \{1\} = B$$

$$A_1 = d(A, 1) = \{\phi\}$$

$$\epsilon\text{-closure}(A_1) = \{\phi\}$$

2. $B_0 = d(B, 0) = \{\phi\}$

$$\epsilon\text{-closure}(d(B, 0)) = \{\phi\}$$

$$B_1 = d(B, 1) = \{2\}$$

$$\epsilon\text{-closure}(B_1) = \{2\} = C$$

3. $C_0 = d(C, 0) = \{\}$

$$\epsilon\text{-closure}(C_0) = \{\}$$

$$C_1 = \{3\}$$

$$\epsilon\text{-closure}(C_1) = \{3, 4, 5, 7, 9, 10\} = D$$

4. $D_0 = \{6\}$

$$\epsilon\text{-closure}(D_0) = \{3, 4, 5, 7, 9, 10\} = \{E\}$$

$D1 = \{8\}$

$\in\text{-closure}(D1) = \{4, 5, 7, 8, 9, 10\} = F$

5. $E0 = \{6\}$

$\in\text{-closure}(E0) = \{6, 4, 5, 7, 9, 10\} = E$

$E1 = \{8\}$

$\in\text{-closure}(E1) = F$

6. $E0 = \{6\}$

$\in\text{-closure}(F0) = E$

$F1 = \{8\}$

$\in\text{-closure}(F1) = F$

Repeats hence stop.

	0	1
A	B	-
B	-	C
C	-	D
D	E	F
E	E	F
F	E	F

	0	1
A	B	-
B	-	C
C	-	D
D	D	D

Figure DFA

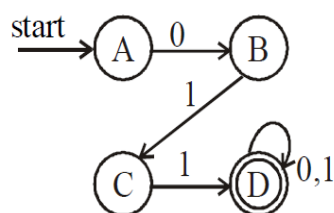
Figure: Minimized DFA

Step 3: ABCDEF

(A) (B) (C) (DEF)

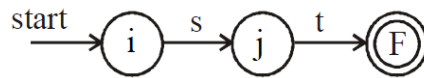
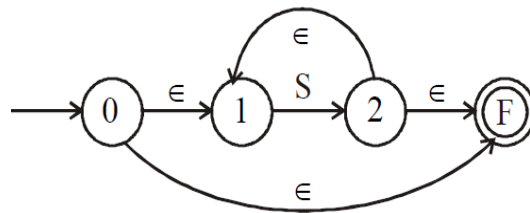
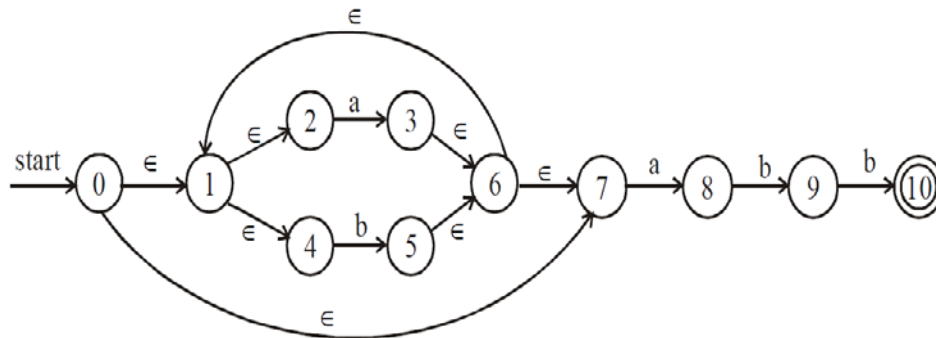
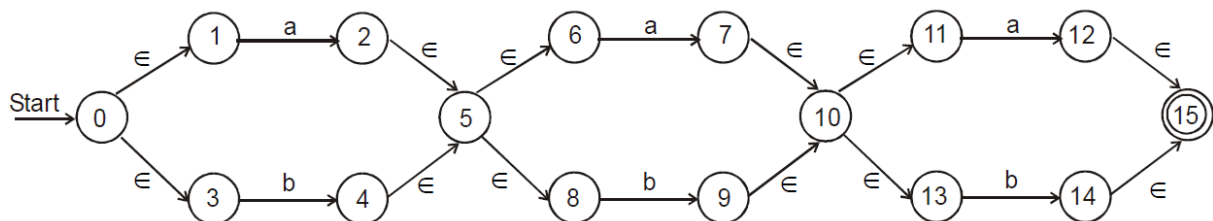
(A) (B) (C) (D)

Note : 'D' is the representation which replaces all occurrence of 'E' & 'F'



Exercise 1:

NFA for st

**Exercise 2:**NFA for S^* **Exercise 3:**NFA for $(a|b)^*abb$ **Exercise 4:**RE $(a|b)(a|b)(a|b)$ 

Each intermediate NFA produced during the course of the construction corresponds to a subexpression of ' r ' has secured important properties which follows.

- has exactly one final state
- no edge enter the starting state
- No edge leave the final state

Exercise 5:

$(a|b)^+$ (1(or) more time)

