10/1

COMPILER DESIGN

→ Principles of compiles design by Aho & Ullman → Vaoosa Publication.

INTRODUCTION TO COMPILER :-

- **TRANSLATOR** :- types
  - compiler
  - interpreter
  - assembler
  - preprocessor.

- STRUCTURE OF COMPILER

TRANSLATOR :- It converts 1 language to other language. It is a program that takes a program written in 1 programming language as i/p & produces an o/p in a program written in another language. Input program is called SOURCE PROGRAM & output program is called OBJECT or target program.

It can be shown by :-

I/P source program (1 PL) → [TRANSLATOR] → O/P object Target Program (2 PL)

Translator is a generic (general) term.

DIFFERENT TYPES OF TRANSLATOR :-

1) **COMPILER** :- It is a specific translator which takes a source program written in HLL (C, Fortan) & produces as o/p a program written LLL (assembly or machine).

I/P HLL Program → [COMPILER] → O/P Low Level language program

Source program is first compiled & converted into object program. This object program is linked with different libraries (LINKER) & then loaded in memory (LOADER) for the execution.

2) **INTERPRETER** :- It is a translator that transfers a source program written in HLL into an intermediate form (postfix notation or 3-address code) which can be directly executed by the machines.

I/P HLL program → [Interpreter] → O/P Intermediate form directly executed

Diff b/w INTERPRETER & COMPILER :-

INTERPRETER	COMPILER
1) HLL → Intermediate	1) HLL → LLL
2) It performs line by line translation of code	2) It takes complete program in 1 go.
3) Execution time is slow	3) It is comparatively fast, bigger in size
4) Smaller in size ~~(time~~	4) Bigger in size
5) Better for implementation of complex programming language constructs	6) It is not better than interpreter in case of complex constructs

3) ASSEMBLER :- It is a translator which converts assembly language to machine language.

Assembly language → [Assembler] → Machine language

4) PRE PROCESSOR :- It is a translator that takes a HLL program as input & produces a program in another HLL (changes the representation of code). It performs :-

→ include headers files with source program.
→ copies values of macros in the source program.
→ remove the comments
→ removes the extra space in source program.

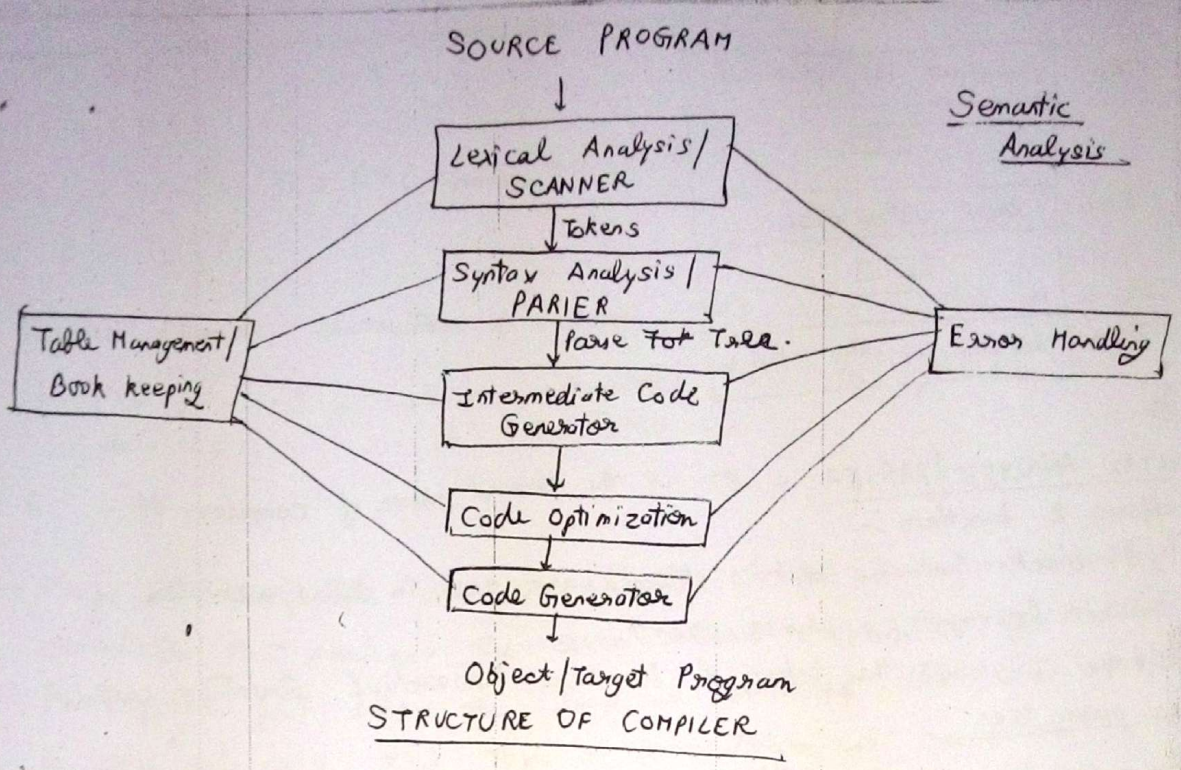
one HLL → [PRE-PROCESSOR] → Another HLL

Ex: Include → preprocessor directive

11/1

STRUCTURE OF COMPILER

The process of compilation is a complex one. It is not possible to complete it in a single step. So it is divided into a series of sub-processes called as PHASES. A phase is a logically cohesive operation that takes as input 1 representation of the source program & produces as o/p another representation of it.



STRUCTURE OF COMPILER

1) LEXICAL ANALYSIS / SCANNER :- It is the first phase of compiler. It reads the source program character by character & groups them into tokens. A token is a sequence of characters that is meaningful & cannot be further broken down. There are generally 5 types of tokens - Keywords, Identifier, Constant, Operator & special symbols.

Ex :- There is a statement in FORTRAN language

if (5 EQ MAX) GOTO 100

There are 8 tokens here if , (, 5 , E Q , MAX ,) , GOTO ~~100~~

The output of LA is a stream of token that is passed on to the next phase. The first 2 phases - LA & SA are often group together in same pass i.e. LA operates either under the control of parser or as a co-routine with the parser. The parser asks LA for the next token whenever it needs it. LA returns a code for the token it has found.

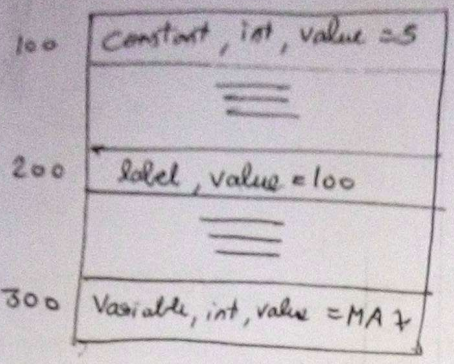
The LA calls a book keeping routine which installs the token in the symbol table if it is not present there. A symbol table is the data structure used to keep a record of all the tokens. For the same FORTRAN statement, the symbol table can be formed as follows :-

if (5 EQ MAX) GOTO 100

5 MAX 100

if ([const, 100] EQ [id, 300]) GOTO [LABEL, 200]

TYPE, Memory address



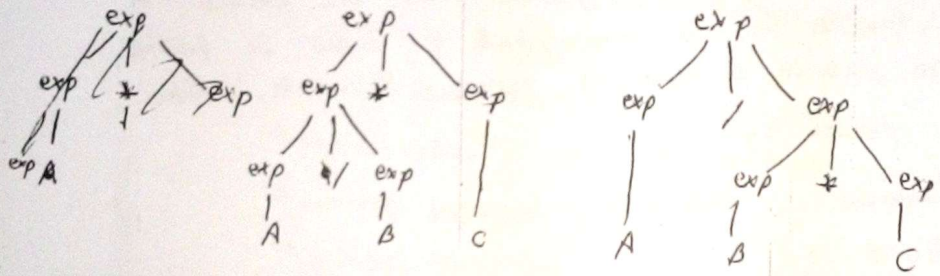
12/1

2.) SYNTAX ANALYSIS / PARSER - It is the second phase of compiler. It performs 2 functions :-

1) It checks that the tokens obtained are acc. to the specification of source language (grammar) or not.

2) If yes it groups the token & forms a heirarchial structure called as parse tree.

Ex :- We have an exp. A/B * C. There can be 2^2 possible parse trees for this expressions :-



The correct parse tree depends on syntactic rules defined in grammars.

→ Parse tree of if statement & while loop in BOOK.

3.) INTERMEDIATE CODE GENERATION :- This phase uses the parse tree & creates a stream of simple estimation in the form of 3 address code.

Ex :- For the same exp. A/B * C, its 3-address code can be written as follows :-

T_1, T_2 are temporary variables $T_1 = A/B$, $T_2 = T_1 * C$.

SMP

4.) CODE OPTIMIZATION (optional) :- This is optional phase designed to improve the intermediate code so that the final object program takes less space & less time. There are 2 types of optimization :-

a) Local optimization b) Loop optimization.

In this method, local transformations are applied on the code Ex :-

Elimination of common ~~sub-p~~ subexpressions. Suppose :-

A = B + C + D

E = B + C + F

We can write $T_1 = B + C$ [redacted] $A = T_1 + D$

$E = T_1 + F$

Another example elimination of jumps over jumps. This is a statement.

if A > B goto L2 → these instructions can be written as :-

goto L3 if A < B goto L3

L2 ——— L2

→ This is optimization done in the loop. In this case, we move a computation that produces the same result everytime to a point in the program just before the loop is enhanced. Then, this computation is done only once. This type of computation is called LOOP INVARIANT

5.) CODE GENERATION :-

This is the final phase of compiler. It produces the object code by deciding on the memory locations of data, selecting the registers etc.

Ex : A = B + C

Assembly :- load B
add C
store A

6.) TABLE MANAGEMENT / BOOK KEEPING :- This part of compiler keeps a track of the names or tokens used in the program & record essential info about them (datatype etc). There is a data structure used here called SYMBOL TABLE. Most of the info is collected in the early phases i.e. LA & SA.

7.) ERROR HANDLING :- This part is invoked whenever there is a flow or errors in source program. It warns the programmer by issuing a diagnostic message. Every phase can detect an errors.

COMPILER WRITING TOOLS

These are the tools that help to generate compilers automatically.

Ex:- Compiler – compilers
Compiler generators
Translators writing system.

These are 2 more tools – LEX & YACC (yet another compiler compilers)- LEX is used to generate LA & YACC is used to generate SA.

The input given to these tools is - a description of lexical & ~~syal~~ syntatic structure of course language, a description of output & a description of target system.

17/1

PASS

A pass is a combination of one or more phases into a single module.
Ex:- The first 2 phases LA & SA work together in the same pass.

A pass reads either the source program or o/p of previous pass makes the transformations as specified by its phases & writes the o/p into an intermediate file which is read by the next pass.

The no. of passes depends upon the structure of source language & the environment in which the compiler has to operate.

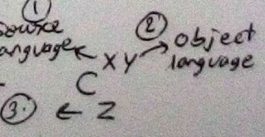
MULTI- PASS COMPILER :- It uses less space bcoz space occupied by 1 pass can be reused by the next pass. It is slower in execution bcoz each pass reads & writes into intermediate files.

BOOTSTRAPPING V.Imp

It is a process of obtaining a compiler for a complete ~~bayo~~ language by using a compiler for a subset of that language. A compiler is characterized by 3 languages :-

1) Source language
2) Object "
3) Language in which compiler itself is written.

These all may be quite different. In this we use following notations :-



CROSS - COMPILER :- It is a compiler which runs on 1 machine & produces object code for some other machines.

Ex :- To implement the concept of bootstrapping, we take eg. of new language L that we want to make available on 2 machines A & B. We assume both A & B are similar to each other from hardware point of view.

First for machine A, we write a small compiler,

subset of L ← SA

C_A

↓

Machine of language A

Then we make C_S^{LA}

After that C_S^{LA} is made to run through the compiler already made.

Equation is :-

$$C_S^{LA} \longrightarrow \boxed{C_A^{SA}} \longrightarrow C_A^{LA}$$

BOOTSTRAPPING

Now suppose we want to produce another compiler for L to run on machine B & produce object code for B $\left(C_B^{LB}\right)$ For this first of all we write C_L^{LB} using the concept of bootstrapping. Then we use following equation :-

$$C_L^{LB} \rightarrow \boxed{C_A^{LA}} \longrightarrow C_A^{LB} \qquad (L\text{-}C)$$

$$C_L^{LB} \rightarrow \boxed{C_A^{LB}} \longrightarrow C_B^{LB}$$

COMPILER DESIGN

CHAPTER 3 : LEXICAL ANALYSIS :-

- lexical analyser also known as Scanner.
- INPUT :- Programming Language i.e. source program.
  OUTPUT :- Tokens

This is the first phase of compiler. It reads the source program character by character & groups them into tokens. In this chapter we will study how to design & implement lexical analysis / analyser.

There are 2 issues in this phase :-

1) We need a method of describing all the possible tokens that can appear in the input program. For this we have a notation called as REGULAR EXPRESSION that describes all the possible tokens.

2) We need a mechanism to recognize these tokens. For this transition diagrams & finite automata are used.

The advantage of using regular expression is that from a regular expression we can automatically construct the token recognizers.

NEED FOR LEXICAL ANALYSIS AS A SEPARATE PHASE :-

The analysis of source program is split into the first 2 phases LA & SA (syntax analysis). This is done :-

→ to simplify the overall design of the compiler.
→ It is easier to specify the structure of token than their syntax.

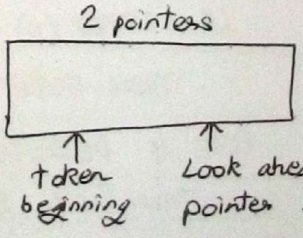
CONCEPT OF INPUT BUFFER :-

It is assumed that L.A. reads its input from an input buffer. This is needed for the following reason :-

→ Buffer is a data structure.

Sometimes it is required by LA to examine many characters beyond the next token, even to determine the next token. Many schemes are used to design the buffers. For ex :-

The following fig. shows an example of it.



The first pointer marks the beginning of token while the second pointer scans the character until the next token is discovered.

Ex :- In PL/I program (language) there is a statement :-

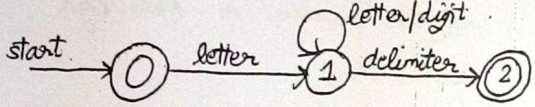
DECLARE(avg1, avg2, . . . avgn)

In this language declare is a keyword & it is also used as an array name. So this is identified by L.A. as an array after scanning it till the right parenthesis. (from the buffer)

DESIGN OF L.A. :

To design a software the simplest method used is a flowchart. To design s we use special kind of flow chart used as TRANSITION DIAGRAMS. In these circles are used to show states & states are connected by arrows called Edges. The labels on the edges indicate the input character that appear in the source program.

Ex :- Transition diagram for an ~~identifying~~ identifier is as follows :-



→ deliniter is a character i.e. neither a digit nor letter & it is defined by programmer.

- An identifier is defined as a letter followed by any number of letter & digits.

19/1

To convert transition diagram into a program, we write code for each state.

```
State 0 :  c = GETCHAR();
           IF LETTER(c) THEN GOTO
              STATE 1;
           ELSE
              FAIL();

State 1 :- c = GETCHAR();
           IF LETTER(c) OR DIGIT(c)
              THEN GOTO STATE 1
           ELSE IF DELIMITER(c)
              THEN GOTO STATE 2
```

ELSE
FAIL();

STATE 2 :- RETRACT();
return (id, INSTALL())

There are some functions used here :-

1) GETCHAR() :- This function returns the next character in the source program and advances the lookahead pointer.

2) LETTER(c) :- This function returns true if c is a letter

3) DIGIT(c) :- This function returns true if c is a digit

4) DELIMITER(c) :- " " " " " " " " delimiter

5) FAIL() :- This function retracts the lookahead pointer & start the next transition program or calls error routine.

6) RETRACT() :- This function retracts the lookahead pointer.

7) INSTALL() :- This function installs the identifier in the symbol table, if it is not present there.

In state 2, Lexical analyser returns to the parser a pair consisting of integer code for that particular token and a value that is the pointer to the particular symbol table returned by ~~the~~ install function.

TOKEN	CODE	VALUE
begin	1	–
end	2	–
if	3	–
Then	4	–
else	5	–
identifier	6	pointer to symbol table
constant	7	pointer to symbol table
<	8	1

<=	8	2
=	8	3
<>	8	4
>	8	5
>=	8	6

REGULAR EXPRESSION :- It is a notation used by lexical analyser for describing tokens in the source program.

SOME IMPORTANT ~~TABLE~~ TERMS :-

1) Alphabet / Character class :- Finite set of symbols / characters.

Ex :- {0,1}

2) String (Word / sentence) :- Finite set of symbols.

Ex :- 001

3) Length of a string (x) is denoted by |x| - It is the no. of symbols present in the string.

4) Empty string ϵ — string of length 0.

5) Operations :-

If x & y are strings then their concatenation is denoted by xy or x.y. It is a string made up of symbols of x followed by symbols of y.

Ex :- abc.de = abcde

$x \cdot \epsilon = x$, $x^2 = x \cdot x$, $x^i = x \cdot x \ldots$ i times

6) Prefix of a string x :- It is a string formed by discarding 0 or more trailing symbols of x.

Ex :- abc is a prefix of abcde.

→ Similarly suffix of string x :- is cde of abcde.

23/1 LANGUAGE :-

It is a set of strings form from a specific alphabet. Ex:-
if $\{0,1\}$ is an alphabet then its language will be :-

$\{0, 10, 0110, \ldots\}$

We use ϕ to denote empty language or it is a set containing empty string. $\phi = \{\varepsilon\}$

OPERATIONS ON LANGUAGE :-

1) **CONCATENATION** :- If L & M are 2 languages then their concatenation is shown as :-

$$L \cdot M = \{ xy \mid x \text{ is in } L \text{ \& } y \text{ is in } M \}$$

$L \cdot M$ concatenation is all set of xy such that $x \in L$ & $y \in M$.

Ex:- $L = \{0, 01\}$

$M = \{1\}$

$L \cdot M = \{011, 0111\}$

Concatenation of L to the power i $L^i = L \cdot L \ldots$ i times

~~Con~~ $L^0 = \phi = \{\epsilon\}$

2) **UNION** :- Union of 2 languages L & M is shown as follows :-

$$L \cup M = \{ x \mid x \text{ is in } L \text{ or } x \text{ is in } M \}$$

3) **CLOSURE** :- Closure operation means concatenation with itself any number of times (from 0 to ∞).
Suppose there is a language L, its closure is denoted by L^*. It means concatenation of L with itself any number of times.

$$\boxed{L^* = \bigcup_{i=0}^{\infty} L^i}$$ $L^* \rightarrow$ Kleene Closure

$$L \cdot L^* = L \cdot \left(\bigcup_{i=0}^{\infty} L^i \right) \rightarrow \text{POSITIVE CLOSURE.}$$

$$L \cdot L^* = \bigcup_{i=0}^{\infty} L^{i+1}$$

$$\boxed{L^+ = \bigcup_{i=1}^{\infty} L^i}$$ $L^+ \rightarrow$ Positive Closure.

The positive closure of language L is concatenation of L with itself from 1 to ∞.

REGULAR EXPRESSION NOTATION for token identifiers is :-

identifier = letter(letter | digit)*

UNION

→ Each regular expression denotes a LANGUAGE.

→ There are some rules :-

1.) If ∈ is a regular expression then it denotes a language ∅.

2.) If a is a regular expression then it denotes a language {a} {a}

3.) If R & S are 2 regular expression denoting languages L_R & L_S respectively.

i.) If R|S is a regular exp. then $L_R \cup L_S$.

ii.) If RS is a regular exp then $L_R \cdot L_S$.

iii.) If R^* is a regular exp then L_R^*.

→ There are some algebraic laws followed by Regular exp :-

1.) R|S = S|R (| is commutative).

2.) R|(S|T) = (R|S)|T (| is associative).

3.) R·(S·T) = (R·S)·T (· is associative)

4.) R·(S|T) = (R·S)|R·T (distributive).

FINITE AUTOMATA

→ A finite automata works like a token recognizer.

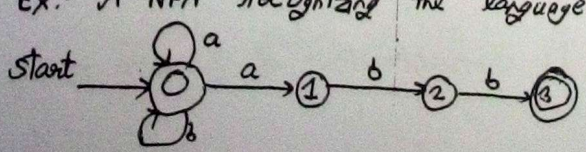
→ A token recognizer for a language L is a program that takes as input a string s & answers yes if s ∈ L & no otherwise.

It is of 2 types :-

- DFA - ~~NFA~~ NFA or NDFA.

NFA :- It is a labelled directed graph. Nodes are called 'states & labelled edges are called TRANSITIONS. Edges can be labelled by ∈ or any other character & some character can label 2 or more transitions out of a single state.

Ex:- A NFA recognizing the language (a|b)* abb



TRANSITION TABLE :-

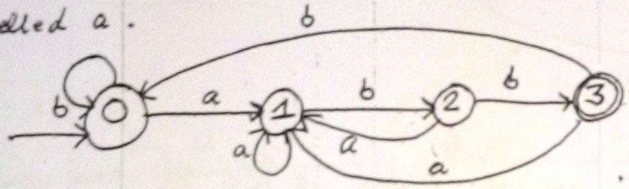
It is a table used to represent F.A. It has a row for each state & column for each input symbol.

States	a	b
0	{0,1}	{0}
1	—	{2}
2	—	{3}

A NFA accepts an input string x iff there is a path from start state to some accepting state such that the labels along that path spell out small x.

DFA :- It has 2 properties :-

1) There are no transitions on input ϵ.
2) For each state s on input symbol a there is atmost 1 edge labelled a.



It is easier to simulate DFA in programming.

HOW TO CONVERT REGULAR EXPRESSION TO NFA (ALGORITHM).

∴ **Given :-** A regular expression R.

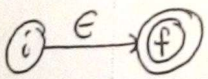
∴ **Output :-** A NFA accepting the language denoted by R.

∴ **Method :-**

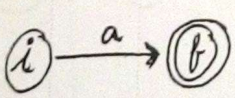
Firstly we decompose R into its basic components. For each component we draw a NFA.

Ex :-

i) **Ist CASE :-** If the component is 'ϵ' then its NFA will be



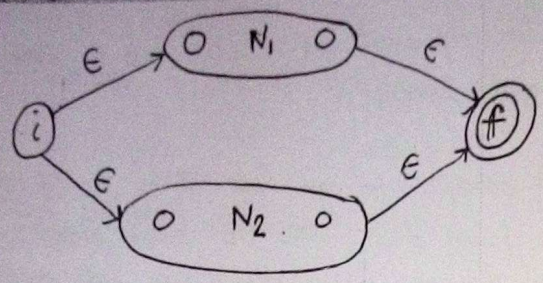
ii) **II CASE :-** If the component is 'a' then its NFA will be :-



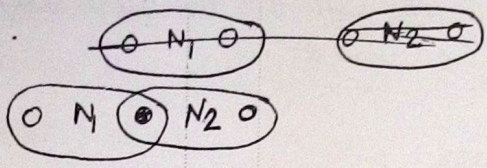
iii) **III CASE :-** After this we combine the NFAs of basic components to produce NFA for complete regular expression.

Now there are 3 possibilities :-

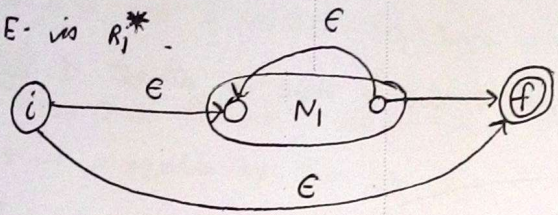
a) If the regular expression is ~~R1, R2~~ 'R1 or R2' / 'R1 ∪ R2' / 'R1 | R2'. The NFA for R_1 is N_1 & for R_2 is N_2 respectively, then their combined NFA will be drawn as follows :—



b) R.E. is $R_1 . R_2$.

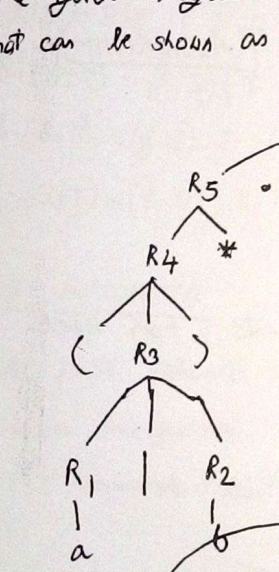


c) If R.E. is R_1^*.



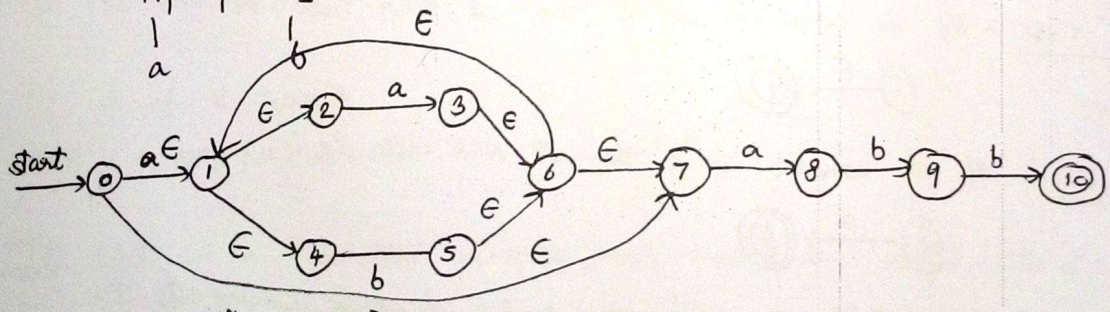
Ex:- If we have R.E. $(a|b)^*abb$.

The given regular expression can be broken up into its basic components that can be shown as follows :-

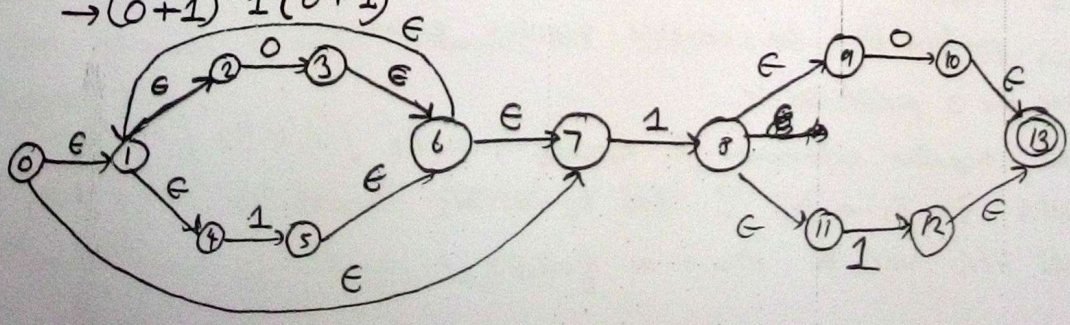


→ For each basic component (R) we draw its NFA.

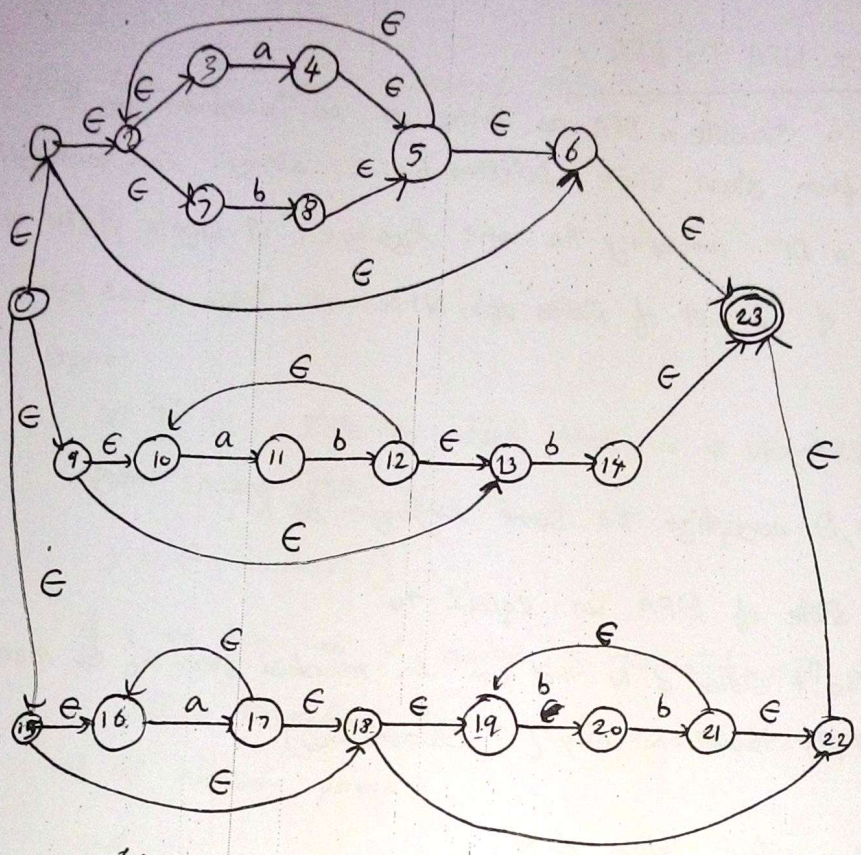
→ The combined NFA for the complete regular expression is drawn as follows :-



→ $(0+1)^*1(0+1)$



$(a|b)^* | (ab)^* b | a^* (bb)^*$



DFA

States	a	b	Token accepted
~~0 1 9 15~~			
0 1 2 3 7 9 10 15 16	4 1 1 17	8	No
4 1 1 17	—	12 20	a
12 20	—	14 21	No
14 21			

ALGO TO CONVERT NFA TO DFA :-

It is easier to simulate a DFA as there is no transition on ϵ. A DFA has atmost 1 path from start state labelled by 1 string. For each NFA, we have to draw a DFA accepting the same language. A single state of DFA will consists of a set of states of NFA.

Given :- A NFA, N

Output :- A DFA, D accepting the same language as N.

Method :- Initial state of DFA is equal to

= S_0 + states of N that can be reached from S_0 by means of empty transitions only (ϵ closure(S_0)).

where S_0 is initial state of N.

ϵ-CLOSURE(S) :-

This function gives us a set of states that can be reached from S by ϵ-transitions (empty transitions).

1.) S itself comes in ϵ-CLOSURE(S)

2.) if t is in ϵ-closure(S) & there is an edge labelled ϵ from t to u then u is also added to ϵ-closure(S).

This step is repeated till no more states can be added.

→ In the beginning, we assume that each state of DFA, D is unmarked.

ALGO:-

While there is ~~con~~ unmarked state $x = \{s_1, s_2, \ldots s_n\}$ of $\mathcal{D}$ do

begin

Mark x;

for each input symbol a do

begin

let 'T' be a state to which there is a transition ~~from~~ on 'a' ~~from~~ source state s_i in x.

$y = \epsilon\text{-closure}(T)$

if y ~~is~~ has not ~~be~~ added to the ~~stat~~ set of states of $\mathcal{D}$.

add a transition from x to y labelled a.

if not already present

end;

end;

Ex:- We take the same NFA, $(a|b)^*abb$.

Initial state of $\mathcal{D} = 0 + \epsilon\text{-closure}(0)$

$= \{0, 1, 2, 4, 7\} = A$

Transition diagram of $\mathcal{D}$:-

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

→ On input symbol a & state A following states have transitions & these next states are {3,8}.

Now ϵ-closure({3,8})

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

Similarly on input symbol of b of state A $= \{5\}$

Now ϵ-closure({5})

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

- On input symbol 'a' of state B, {3,8}

ϵ-closure({3,8})

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

- On input symbol b of state B {5,9}

ϵ-closure({5,9})

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

- On input symbol 'a' of state C {3,8}

ϵ-closure({3,8})

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

- On input symbol 'b' of state C {5}

ϵ-closure({5})

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

- On input symbol a of D {3,8}

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

- On input symbol b of D {5,10}

ϵ-closure({5,10}) $= \{1,2,4,5,6,7,10\} = E$

- On input symbol a of state E {3,8}

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

- On input symbol b of state E {5}

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

MINIMIZING THE NO. OF STATES OF DFA (if possible)

INPUT → A DFA, D with no. of states

OUTPUT → A DFA, D' accepting the same language as D & having as few states as possible.

1.) We construct a partition π of set of states. Initially ϕ consists of 2 groups i.e. final state F & a non-final state Q-F'.

2.) Then we construct a new partition π_{new} by the following methods :-

→ for each group G of π do

begin

partition G into subgroups such that 2 states s & t of G are in the same subgroup if & only if for all input symbol, a state s & t have transitions to the states in the same group of π. Place all subgroups formed in π_{new}.

end

Ex:- For same DFA there were total 5 states :-

(A, B, C, D, E).

$\pi_1 = (ABCD)(E)$

$\pi_2 = (ABC)(D)(E)$

$\pi_3 = (AC)(B)(D)(E)$

Since A & C both gives same transitions. In such a case we select a representative out of the 2 states.

Suppose we select A.

$\therefore \pi_4 = (A)(B)(D)(E)$.

Thus transition table of reduced DFA.