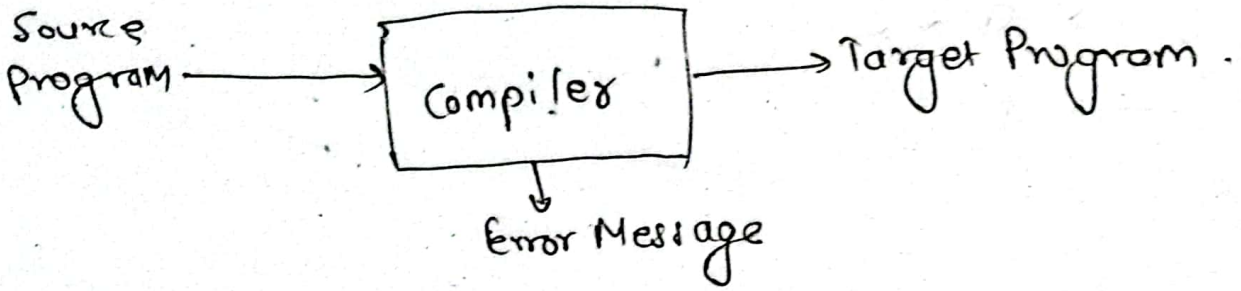
Unit-1: Overview of Compilation

*Compiler:

↳ It is a ~~com~~program that reads a program written in source language and translate it into target language.

↳ It also shows the errors in the source program.



↳ There are two parts in compilation

i) Analysis

ii) Synthesis

↳ In Analysis, source program is broken down into constituent pieces and creates an intermediate representation of src program.

↳ In synthesis, desired target program is constructed from intermediate representation.

↳ To create an executable program, along with ~~mean~~ compiler, other programs are also required as shown in fig (i).

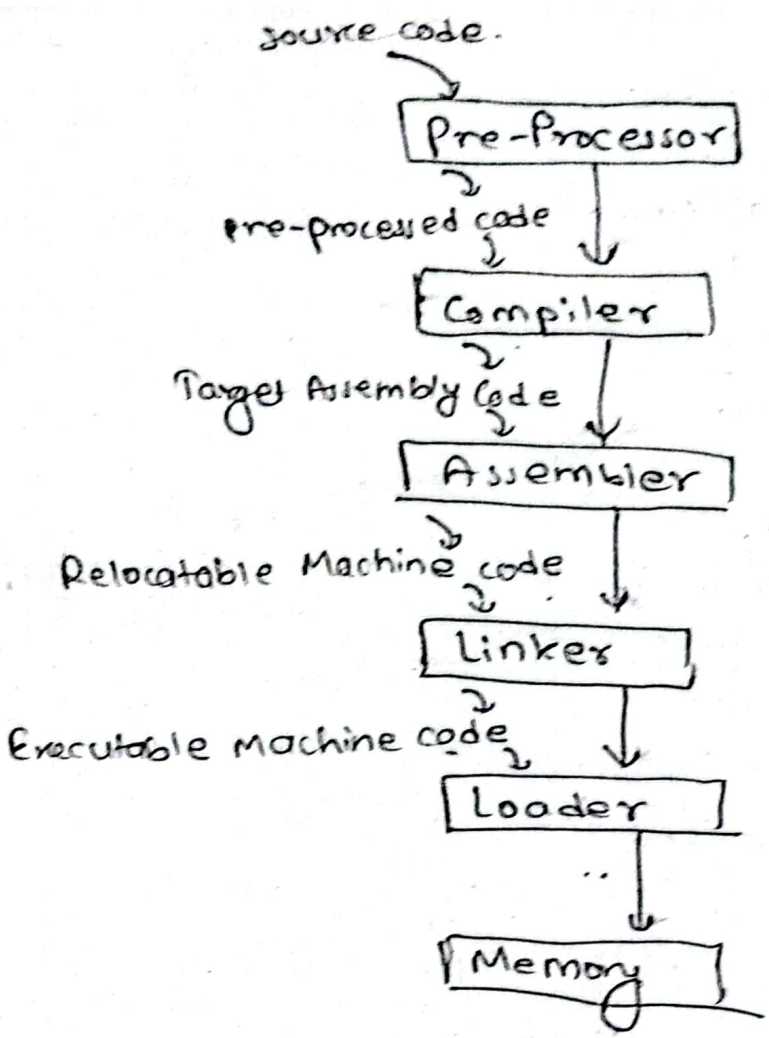


fig (1).

↳ Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another as in fig (2).

05-10
CD

VVI

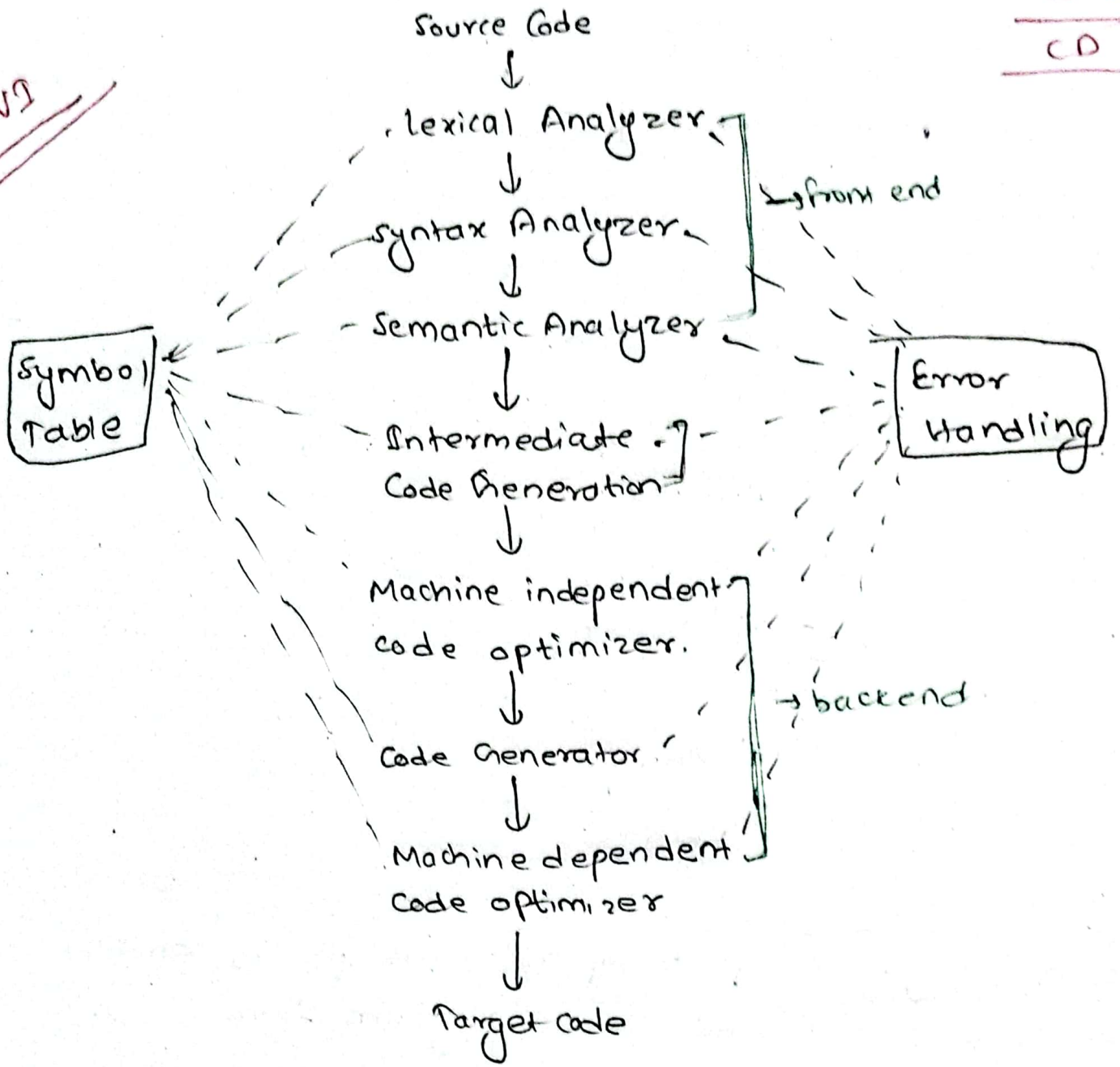


fig (2).

↳ First phase of a compiler is called lexical analyzer or scanning.

↳ Lexical analyzer reads the stream of characters of source code and groups them into meaningful sequence called lexemes.

↳ For each lexeme, lexical analyzer produce as o/p a token of form <token name, attribute value>.

↳ Second phase of a compiler is called syntax analyzer or parsing.
- It takes the token produced by lexical analysis as i/p and generates a parse tree.
- Here, token arrangements are checked against source code grammar.

↳ Next phase is semantic analysis which checks whether the parse tree constructed follows the rules of language.
- It keeps track of identifiers, their types and expression, whether identifiers are declared before use or not, etc.

↳ In next phase, compiler generates intermediate code of source code for target machine.
- It is between high level language and machine language.
- It should be generated in such a way that it make it easier to be translated into target machine code.

↳ Next phase is code optimization of intermediate code.
- It includes removing unnecessary code line, arranging sequence of statements, etc.

↳ Code generation phase takes optimized intermediate code and map to target machine language.
- Code generator translates the intermediate code to sequence of relocatable machine code.

↳ symbol table store information about keyword and tokens found during lexical analysis. It is computed in almost all phase of compiler.

01-10
CD

↳ Error handler handles all errors encountered.

Eg:

$x = a + b \times c;$

⇓

Lexical Analysis

⇓

$id = id + id \times id;$

⇓

syntax analysis

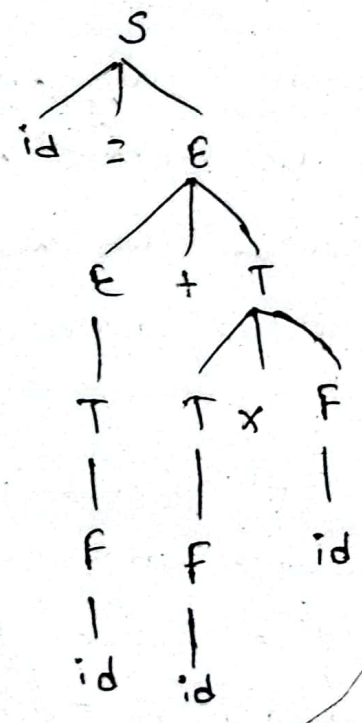
⇓

$S \rightarrow id = E$

$E \rightarrow E + T / T$

$T \rightarrow T \times F / F$

$F \rightarrow id.$



⇓

Semantic analysis

⇓

Parse tree semantically verified

⇓

Intermediate code generation.

⇓

$t1 = b \times c$

$t2 = a + t1$

$x = t2$

⇓

Code optimization

⇓

$t1 = b \times c$

$x = a + t1$

⇓

Target code generation

⇓

```
MUL R1, R2
ADD R0, R2
MOV R2, X
```

Unit-2:

Scanner

*Lexical Analysis

↳ initial part of reading and analyzing the program text.
↳ the text is read and then divided into tokens, each of which corresponds to a symbol in programming language. eg:- variable name, keyword, or number, etc.
↳ lexical analyzer or lexer will discard comments and white spaces.
↳ A lexical analyzer also called as scanner has the following functionality and characteristics.

1) Its primary function is to convert from a sequence of characters into sequence of tokens.
2) it must identify and categorize specific character sequence into tokens.
3) handle lexical errors (illegal characters) by reporting them intelligently to users.
4) token categories can be specified using regular expression.
5) lexical analyzers can be written by hand or implemented automatically using finite automata.

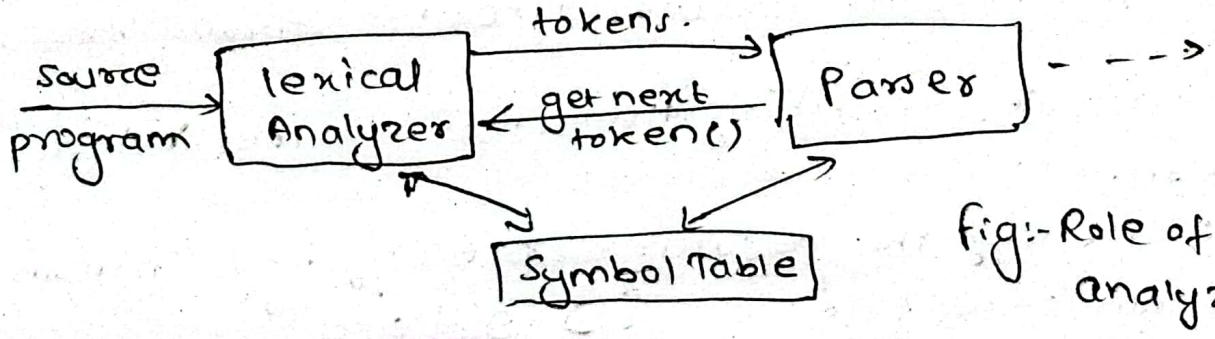


fig:- Role of lexical analyzer.

*Roles

↳ lexical analyzer read the i/p stream one character at a time and translate it into valid tokens.
↳ the lexical analyzer works in lock step with the parser
↳ the parser request lexical analyzer for the next token whenever it requires one using getnexttoken().
↳ lexical analyzer may perform other operations like removing redundant white spaces, removing tokens separators, removing comments, providing line number to parser for error reporting.

*Tokens, patterns, lexemes

↳ A token is a single word of source code input.
↳ When a string representing a program is broken into sequence of substrings, such that each substring represents a constant, identifier, operator, keyword, etc of the language, these substrings are called tokens.
↳ they are building block of programming language.
eg:- if, else, identifier, etc.
↳ lexemes are the actual string matched as token.
—they are the specific characters that make up of a token. eg:- abc, 123.
↳ A token can represent more than one lexemes i.e, token intnum can represent lexemes like 123, 4460, 78129, etc.
↳ Patterns are rules describing the set of lexemes belonging to a token.
- this is usually the regular expressing. eg:- intnum token can be defined as [0-9][0-9]*

Token	Informal Definition	Sample lexemes
if	characters i, f	if
else	characters e, l, s, e	else
Comparison	< or > or <= or >= or == or !=	<=, !=
id	letters followed by letters & digits.	Pi, score, D2
number	any numeric constant	3.1415, 0, $6.02e^{23}$
literal	anything but surrounded by " "	"Hello"

fig:- Examples of tokens.

E.g:

```
printf("Total = %d", score);
```

Here,

↳ printf and score are lexemes matching pattern of for token id

↳ "Total = %d", is a lexeme matching token literal.

↳ In many programming languages, the following classes are cover most or all of the tokens.

i) one token for each keyword. here pattern for a keyword is same as keyword itself.

ii) tokens for the operators either individually or in a classes such as token comparison is maintained.

01-15
CD

iv) one or more tokens representing constants, such that numbers and literal strings.

v) Token for each punctuation symbol such as left and right paranthesis, comma, etc.

* Attributes of Tokens

↳ when more than one lexemes can match a pattern, the lexical analyzer must provide additional information about particular lexeme that matched.

↳ i.e, if there is more than one lexeme for a token, we need to put extra information about token which is called attribute of token.

↳ for eg:- token number matches both 0 and 1 but it is extremely important for code generator to know which lexeme was found in source program. Hence, in this case, lexical analyzer mwt be able to represent 0 and 1 differently.

↳ for token id, information about an identifier is kept in the symbol table so appropriate attribute value for an identifier is a pointer to symbol table entry for that identifier.

eg:- Sum = val1 + val2.

Here,

token and associated attributed values are written as

<id, pointer to symbol table entry for sum>

<assign_op>

<id, pointer to symbol table entry for val1>.

<add-op>
<id, pointer to symbol table entry for val2>

01-16
CD

082-01-16

*Lexical errors

↳ Lexical analyzer, without help of other component cannot easily say there is error.

Let in C, we have

'fi(a==b)-----'

Here,

lexical analyzer can't tell whether fi is misspelling of keyword if or an ~~an~~ undeclared function identifier.

- For fi, the lexical analyzer return token id.

↳ Error in this phase is found when there is no matching string found as given by the pattern.

- When error occurs, the lexical analyzer must not halt the process so it can print the error message and continue.

*Specification of tokens

↳ Regular expressions are an important notation for specifying lexeme patterns.

↳ Each pattern matches a set of strings, so regular expression will serve as name for set of strings.

Some defn:

*Alphabets

↳ An alphabet A is a set of symbols.

*Strings:

↳ A string is a finite sequence of characters from alphabet A.

↳ The length of string 'w' is denoted by |w| i.e. number of characters in w.

*Kleene Closure:

↳ Kleene closure over an alphabet A denoted by A^* is set of all strings of any length (0 also) possible from A.

$$A^* = A^0 \cup A^1 \cup A^2 \cup \cdots$$

*Language:

↳ Language L is set of strings such that $L \subseteq A^*$.

Operation	Definition & Notation
Union of L & M.	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$.
Concatenation of L & M.	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of L	$L^* = \bigcup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

fig:- Defn of operations on language.

*Regular Expression (RE):

↳ is used to describe the tokens of programming language.

↳ has the capability to express finite languages by defining a pattern for finite strings of symbols.

↳ The grammar defined by RE is known as Regular Grammar. 01-16-22

↳ The language defined by Regular Grammar is called Regular Language.

↳ If r and s are regular expression, denoting languages L(r) and L(s) respectively, then

i) r+s is a regular expression denoting L(r) ∪ L(s).

ii) rs is a regular expression denoting L(r).L(s).

iii) r^* is a regular expression denoting $(L(r))^*$

* Properties of regular expressions

i) r+s = s+r ; + is commutative.

ii) r+(s+t) = (r+s)+t ; + is associative.

iii) r(st) = (rs)t ; concatenation is associative.

iv) r(s+t) = (rs)+(rt) ; concatenation distributes over +

(s+t)r = sr + tr

v) ∈r = r∈ = r ; ∈ is identity element for concatenation.

vi) $r^* = (r+\in)^*$; ∈ is guaranted a closure.

vii) $r^{**} = r^*$; * is idempotent.

* Recognition of tokens :

↳ A recognizer for a language is a program that takes string 'w' and answer 'yes' if w is a string of that language, otherwise 'NO'.

↳ The tokens that are specified by RE are recognized by using FA.

↳ Here, transitions are defined and we start from start state of FA using transition diagram.
- if the transition leads to the accepting state, then the $ token is matched and lexeme is returned.
- Otherwise, other transition diagram are tried out until failure is detected.

↳ Recognizer of tokens take the language L and string S as input and try to verify whether $s \in L$ or not.

* Recognizing words:

↳ is done by character by character formulation.

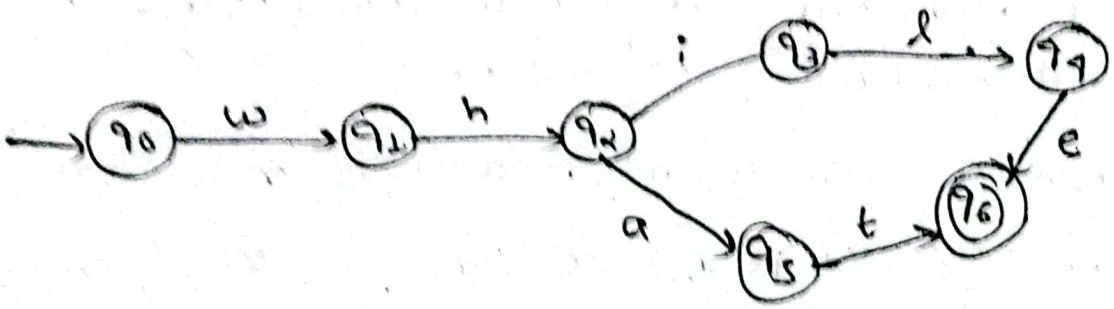
↳ Assume Nextchar() returns a character, sample code for recognizing the keyword 'while' is like below.

```
c ← Nextchar();
if (c='w')
  then begin;
    c ← Nextchar();
if (c='h')
  then begin;
    c ← Nextchar();
if (c='i')
  then begin;
    c ← Nextchar();
if (c='l')
  then begin;
    c ← Nextchar();
if (c='e')
  then success;
  else failure;
  end;
  else failure;
end;
  else failure;
end;
  else failure;
end;
  else failure;
```

~~end;~~
~~else failure;~~

↳ To recognize multiple words, we can create multiple edges, that leave a given state.

↳ eg- one recognizer for both 'while' and 'what' be



* Formalism for recognizer

↳ is through finite automata where FA is 5 tuple

$(Q, \Sigma, \delta, q_0, f)$

↳ for above eg. of 'while' or 'what' recognizer FA is like.

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$

$\Sigma = \{w, h, i, l, e, a, t\}$.

$q_0 = \{q_0\}$

$f = \{q_6\}$

δ :

$\delta(q_0, w) = q_1$

$\delta(q_1, h) = q_2$

$\delta(q_2, i) = q_3$

$\delta(q_2, a) = q_5$

$\delta(q_3, l) = q_4$

$\delta(q_4, e) = q_6$

$\delta(q_5, t) = q_6$

Transition table

Q/Σ	w	h	i	l	e	a	t
→ q_0	q_1	-	-	-	-	-	-
q_1	-	q_2	-	-	-	-	
q_2	-	-	q_3	-	-	q_5	-
q_3	-	-	-	q_4	-	-	-
q_4	-	-	-	-	q_6	-	-
q_5	-	-	-	-	-	-	q_6
* q_6	-	-	-	-	-	-	-

01-16-22
CD

(9) How does FA accept strings?

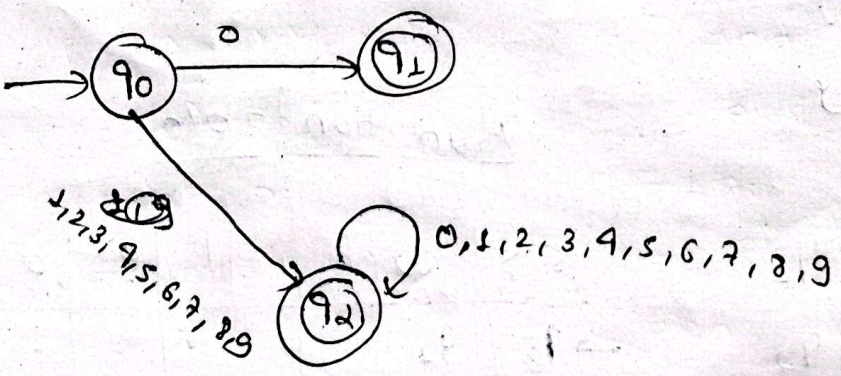
↳ The symbols of any strings are passed through starting state of f.A individually.
If at the end of string, it reaches accepting state then is accepted otherwise, rejected.

↳ if x is $x_1 x_2 x_3 \ldots x_n$ then FA accepts x if and only if $\delta(\delta(\ldots\delta(\delta(\delta(q_0, x_1), x_2), x_3), \ldots x_{n-1}), x_n) \in q_f$.

082-01-202

*Recognizing more complex words:

↳ for unsigned integer (either 0 or it is a series of one or more digits where first digit is from 1 to 9 and subsequent digits are from 0 to 9).



* Closure properties of regular expression:

↳ Regular expressions are closed under

i) union
ii) concatenation
iii) kleene closure

i) Union

↳ if r and s are two regular expressions, then r ∪ s is also regular expression.

ii) Concatenation

↳ if r and s and are two regular expressions then r.s is also regular expression.

iii) Kleene Closure

↳ if r is regular expression, then r^* is also regular expression.

→ There are two types of Finite Automata.

i) Deterministic Finite Automata (DFA):

↳ FA is deterministic, if there is exactly one transition for each (state, input) pair.

↳ DFA is five tuple $(Q, \Sigma, \delta, q_0, F)$ where,

$Q \rightarrow$ finite set of states

$\Sigma \rightarrow$ finite set of inputs

$q_0 \rightarrow$ a start state

$F \rightarrow$ set of final states, $F \subseteq Q$.

$\delta \rightarrow$ transition fxn

$$\delta : Q \times \Sigma \rightarrow Q.$$

Implementing DFA:

↳ Following is the algorithm for simulating DFA to recognizing given string.

↳ for a given string w, in DFA D, with start state q_0, the o/p is 'Yes' if D accepts w, else 'No'.

```
DFAsim(D,q0)
{
   q=q0;
   c=getchar();
   while(c!=eof){
      q=move(q,c);
      c=getchar();
   }
   if(q is in F)
      return 'Yes';
   else
      return 'No';
}
```

ii) Non-Deterministic Finite Automata (NFA):

↳ FA is non-deterministic if there is more than one transition for each (state, input) pair.

↳ NFA is five tuple $(Q, \Sigma, \delta, q_0, f)$ where,

- $Q \rightarrow$ finite set of states
- $\Sigma \rightarrow$ finite set of inputs
- $q_0 \rightarrow$ a start state
- $f \rightarrow$ set of final states; $f \subseteq Q$.
- $\delta \rightarrow$ transition fxn

$$\delta : Q \times \Sigma \longrightarrow 2^Q$$

Algo:

```
S = ε-closure({S0})
c = getchar();
```

```
while( c != eof){
    s = ε-closure(move(s,c));
    c = getChar();
}
if (s ∩ f ≠ ϕ) then
    return 'Yes';
else
    return 'No';
```

Q) for alphabet $\Sigma = \{0,1\}$, write regular expression for strings having 100 or 010 as substrings.

~~R.E = $(0+1)^*100\,1^*0^* + 0^*1^*010\,1^*0^*$~~

$$R.E = (0+1)^*100(0+1)^* + (0+1)^*010(0+1)^*$$

or,

$$= (0+1)^*(100+010)(0+1)^*$$

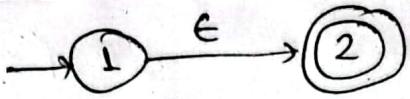
* Regular Expression to NFA (Thompson's Construction)

↳ Here,

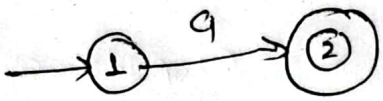
- → i/p is regular expression, r over alphabet A.
- → o/p is ε-NFA accepting L(r).
- → procedure is process in bottom up manner by creating ε-NFA for each symbol in A including ε.
  - then, recursively create for other operations.
- → we will use the rules which define a regular expression as a basis for the construction.

01-22
CD

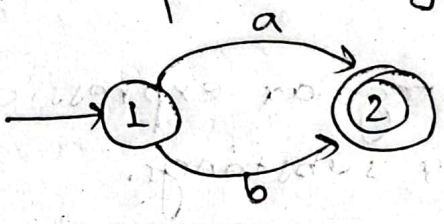
a) the NFA representing empty string.



b) the NFA representing a ϵ A.

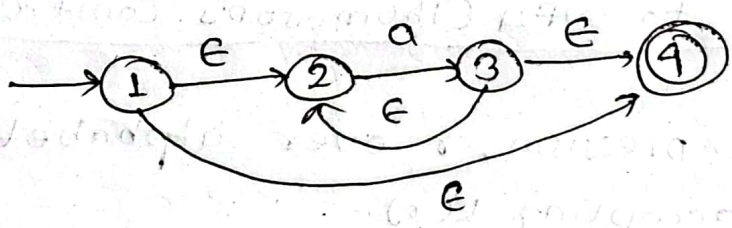


c) the NFA representing a+b or a|b or a∪b.



d) the NFA representing a.b

e) the NFA representing a*

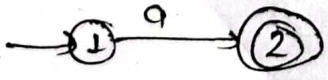


(q) Construct NFA of given R.E using thompson's construction.

$((a.b)|c)^*$

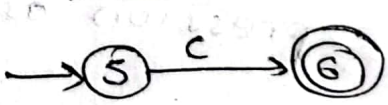
↳

for a



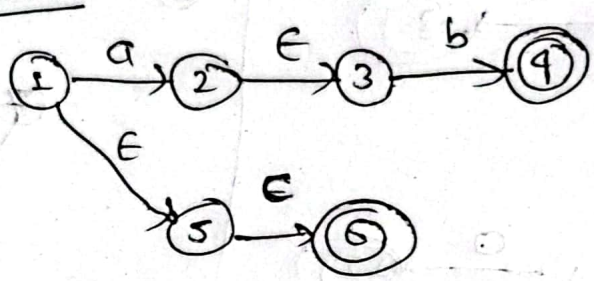
for b

for c

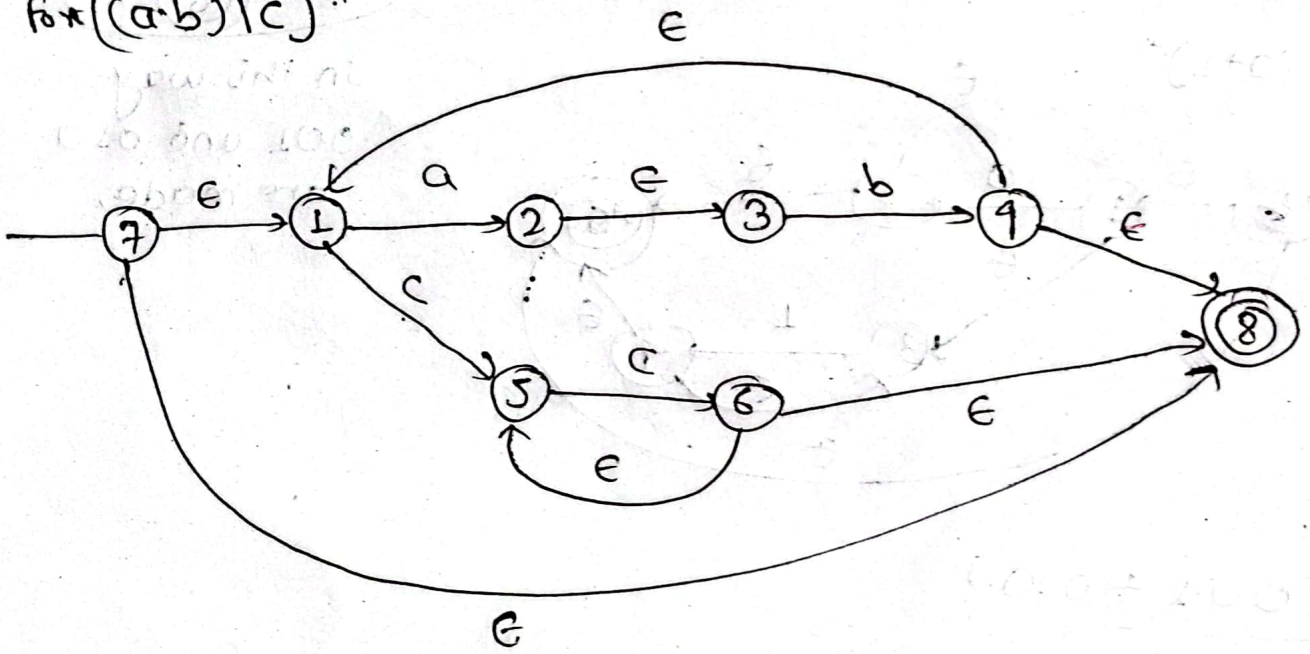


for (a.b);

for (a.b)|c

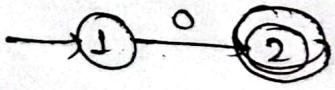


for (a.b)|c)*

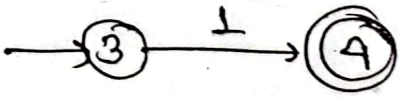


(9) $(0+1)^*(001+010)(0+1)^*$

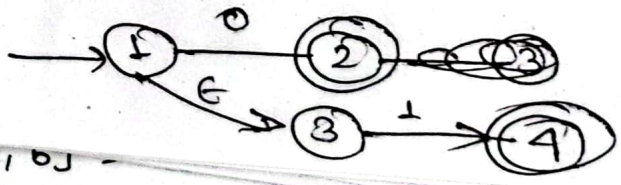
for 0,



for 1,



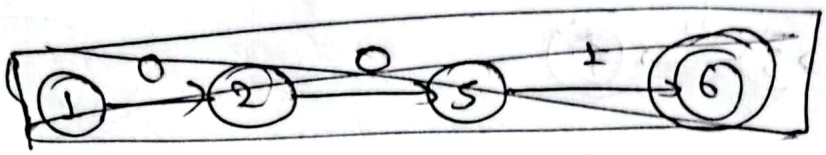
for 0+1,



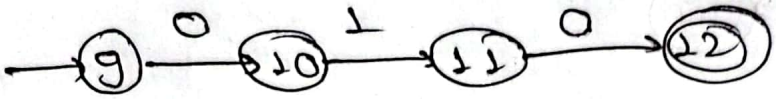
Dtran[C, 0] =

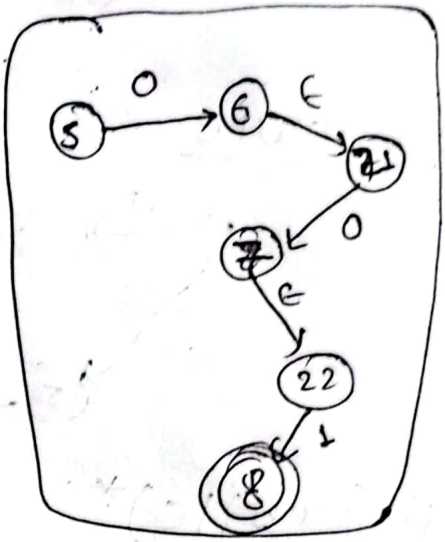
01-22
CD

for 001



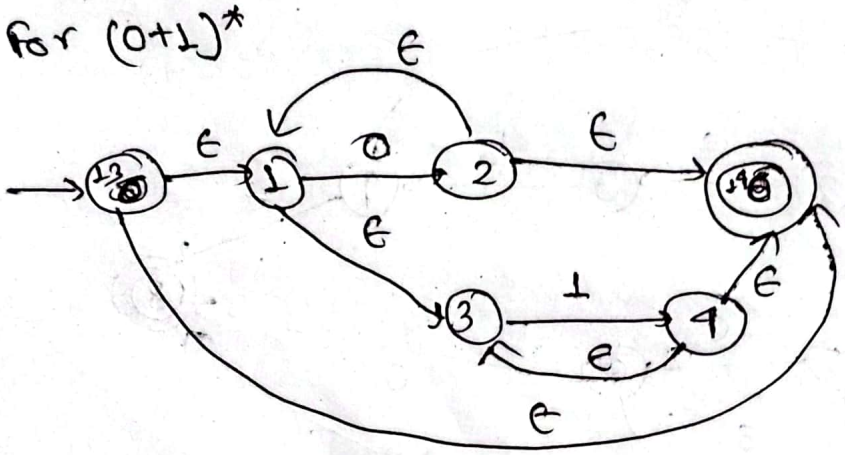
for 010



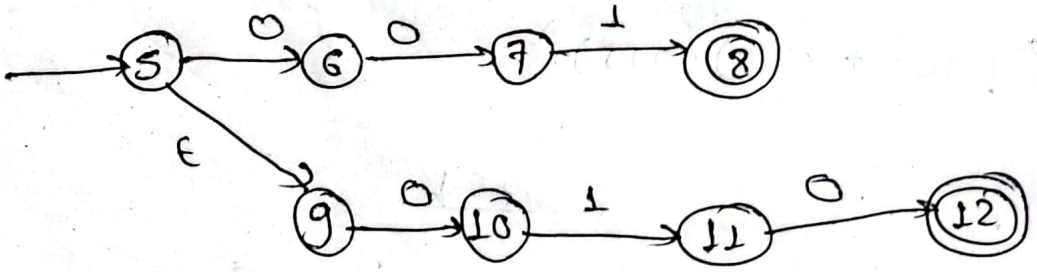


In this way, 001 and 010 are made,

for (0+1)*

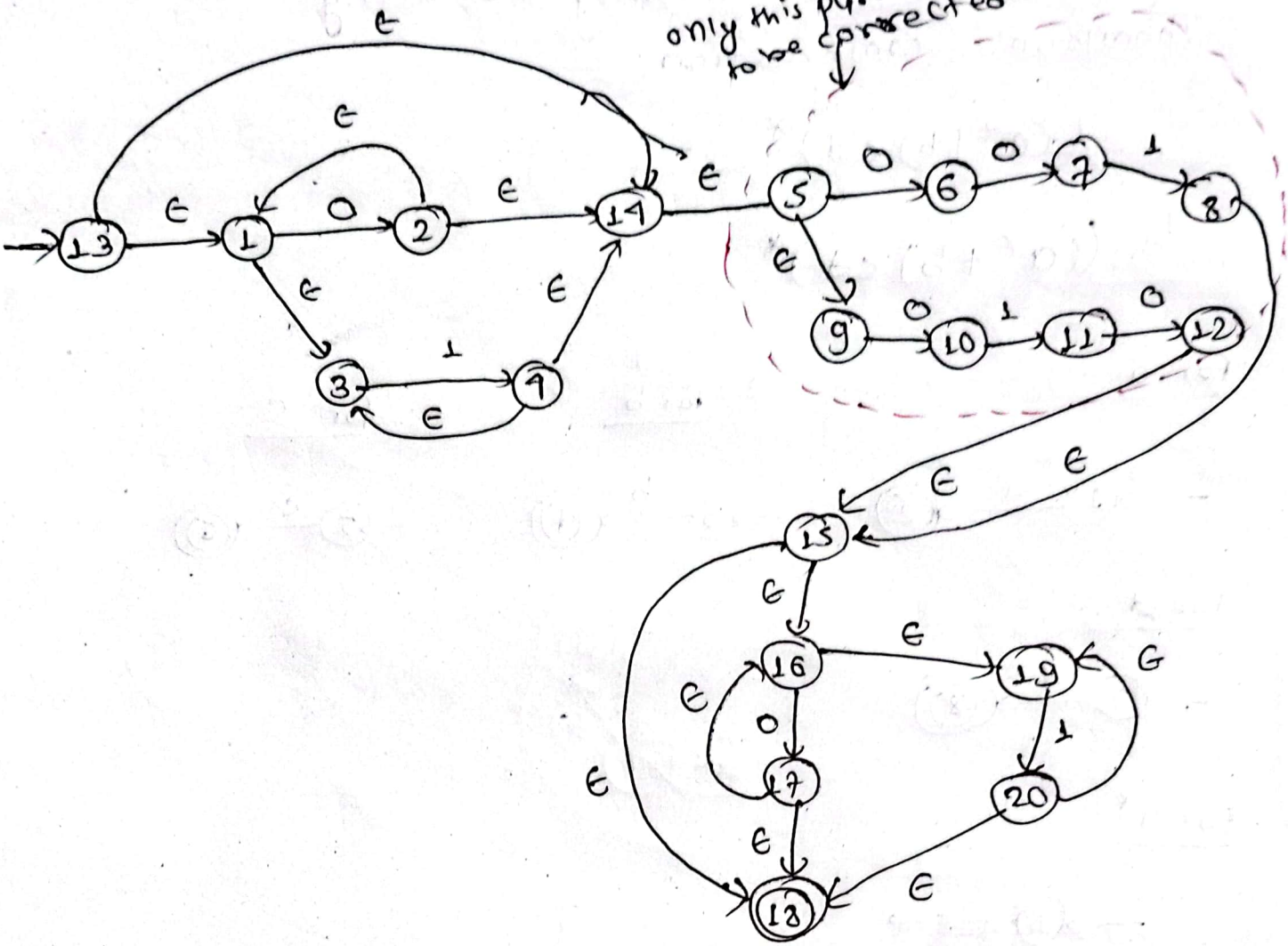


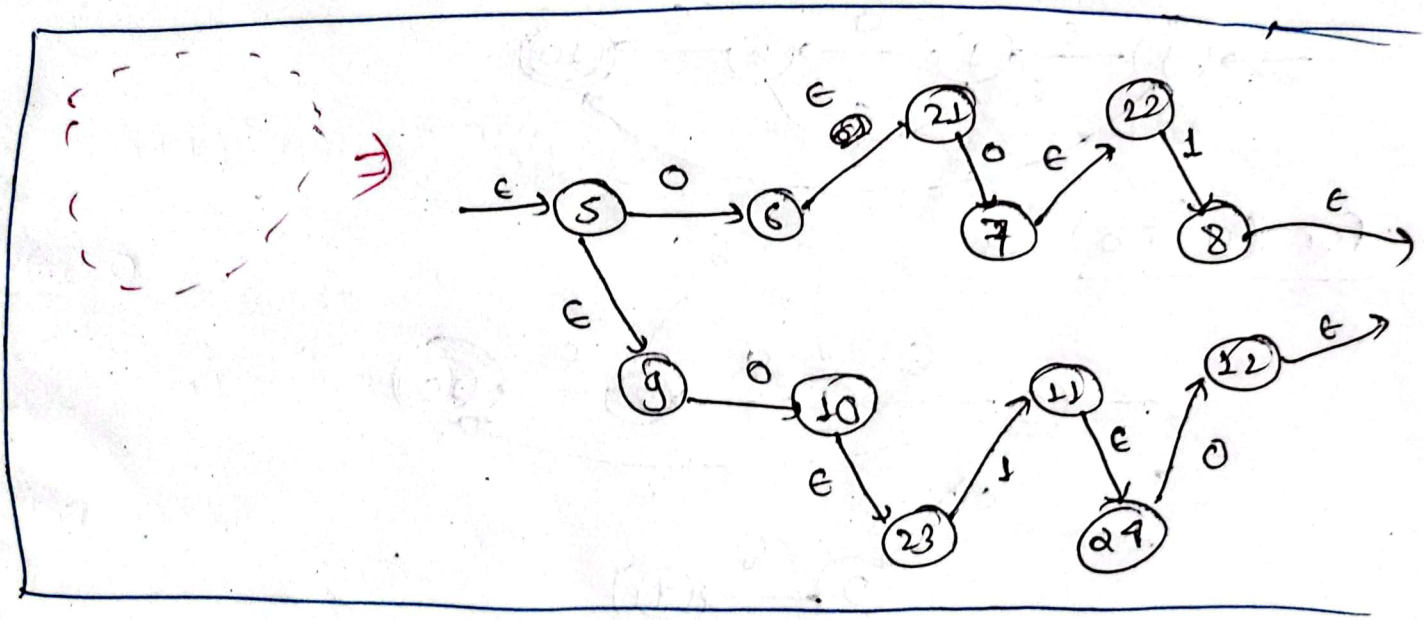
for (001+010)



01-22
CO

for $(0+1)^*(001+010)(0+1)^*$



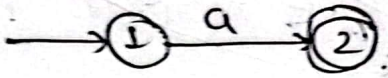


Q) Construct NFA of following RE using Thompson's construction:

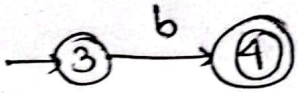
$((a^*|b)cd)^*$

↳ $((a^*+b)cd)^*$

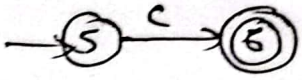
for a



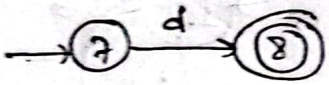
for b



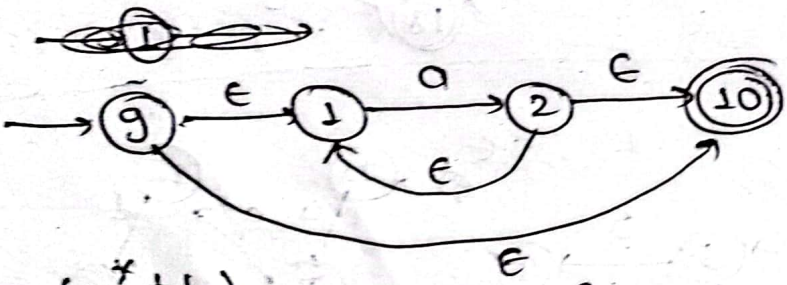
for c



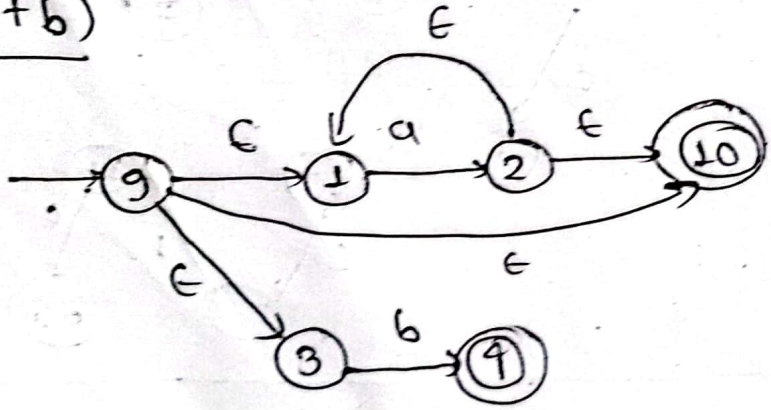
for d



for a^*

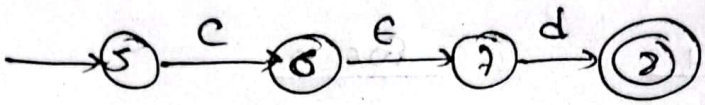


for (a^*+b)

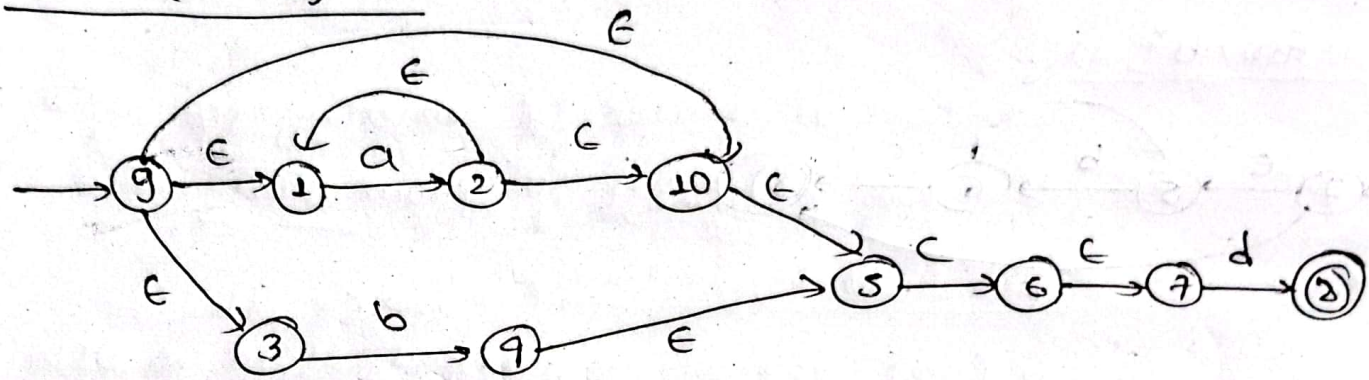


01-23
CD

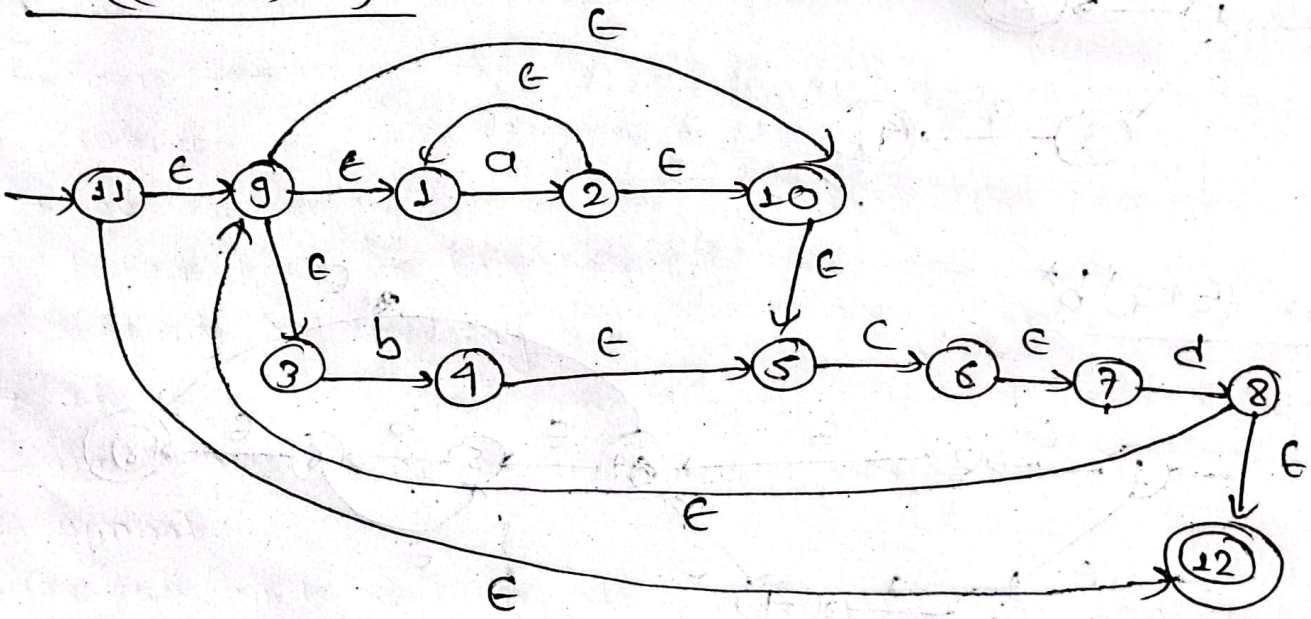
for cd



for (a*+b)cd



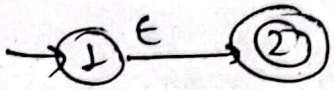
for((a*+b)cd)*



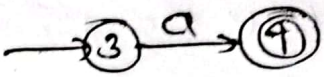
DTran[C, a]

Q : RE = $((\epsilon + a)\, b^*)^*$

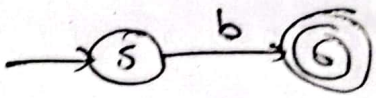
for ϵ



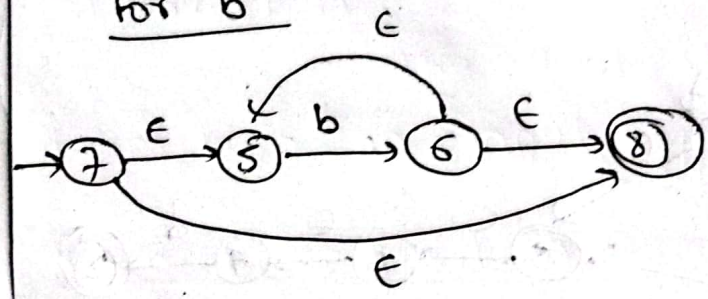
for a



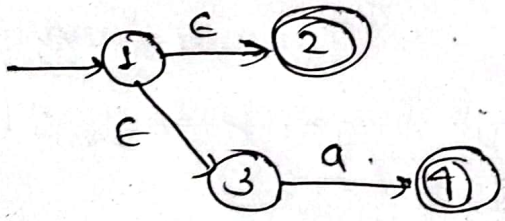
for b



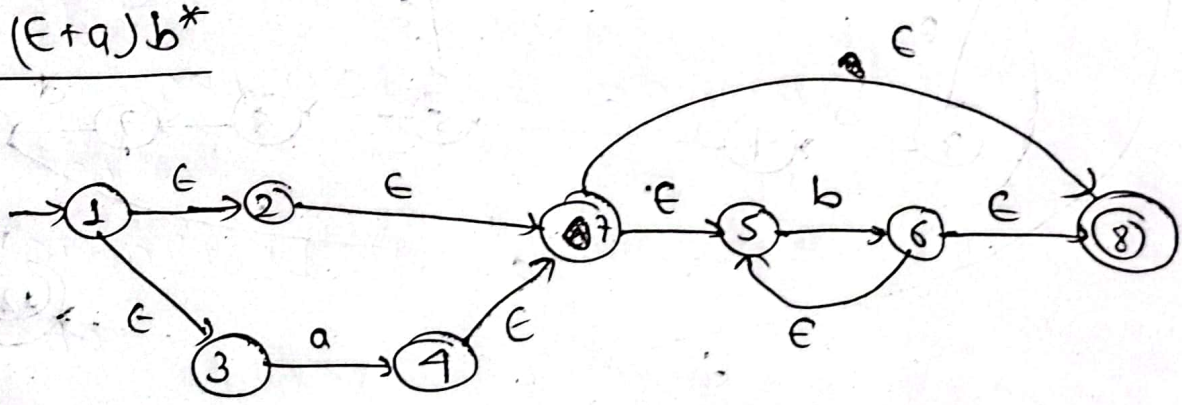
for b^*



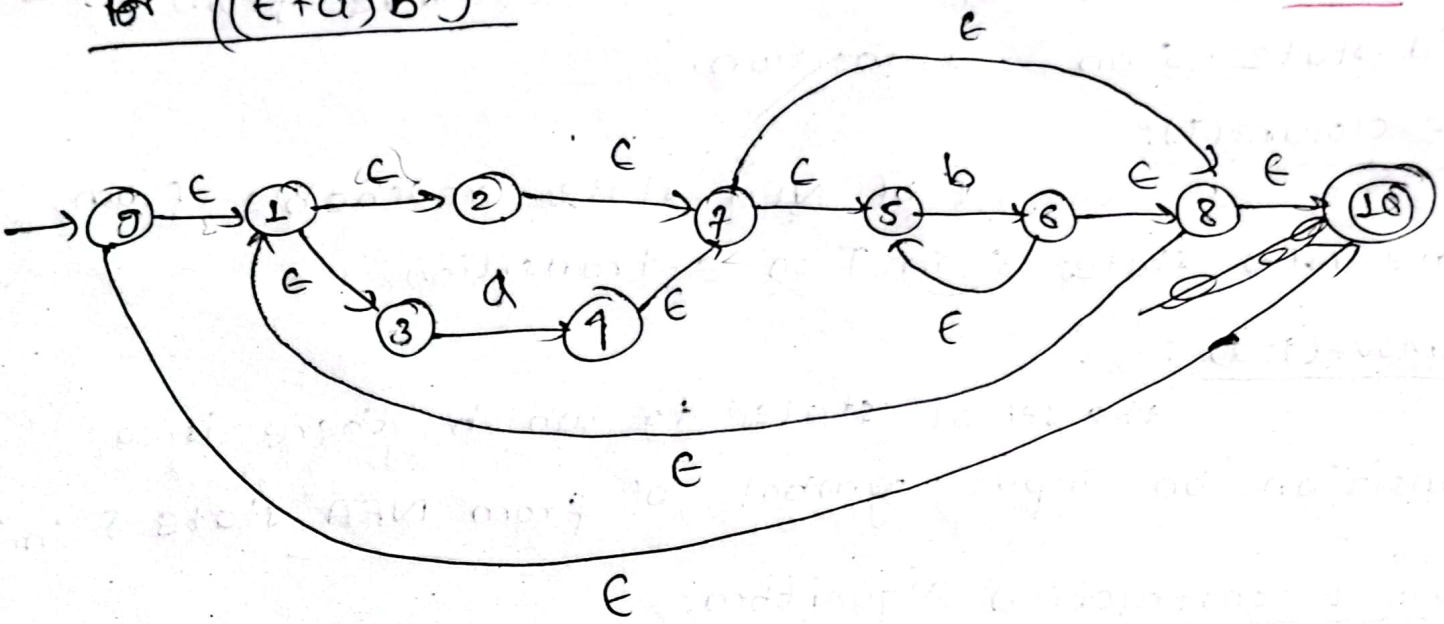
for $(\epsilon + a)$



for $(\epsilon + a)b^*$



for $((\epsilon + a)b^*)^*$



*Conversion from NFA to DFA (subset construction):

↳ Subset construction is an approach that constructs DFA from NFA, that recognize the same language.

↳ There ~~are~~ may be several accepting states in a given subset of Non deterministic states.

↳ The accepting state corresponding to the pattern listed first in the lexical analyzer generator specification has priority.

↳ state transitions are made until a state is reached which has no next state for the current input symbol.

↳ The last input position at which the DFA entered on accepting state gives the lexemes.

↳ following actions are needed:

01-23
CD

1) ε-closure(s):
The set of NFA states reachable from NFA state s on ε-transition.

2) ε-closure(T):
the set of NFA states reachable from some NFA states s in T on ε-transition.

3) Move(T, a):
the set of states to which there is a transition on input symbol 'a' from NFA state s in T.

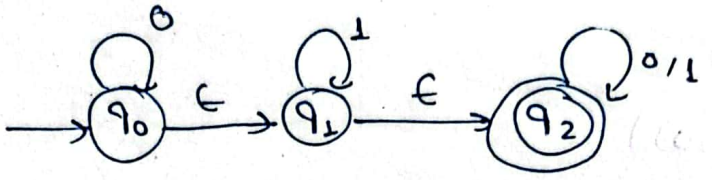
*Subset Construction Algorithm:

Put ε-closure (So) as an unmarked state in Dstates.
while there is an unmarked state T in D states do
    Mark T.
    for each input symbol a∈A do
        U := ε-closure(move (T, a)).
        if U is not in Dstates then
            Add U as an unmarked state to Dstates.
        end if;
        DTran[T, a] = U;
    end do;
end do;

↳Here,
- Dstates is the set of states of the new DFA consisting the set of states of NFA.

- DTran is the transition table of the new DFA. 01-23 CD
- A set of Dstates is an accepting state of DFA if it is a set of NFA states containing at least one accepting state of NFA.
- The start state of DFA is ϵ-closure(s_0) where, s_0 is start state of NFA.

(9).



Soln;

Initially,

ϵclosure (q_0) = $\{q_0, q_1, q_2\}$ = A as unmarked state in Dstate

Mark A.

for i/p symbol '0',

U = ϵ-closure (move (A, 0))

= ϵ-closure ($\{q_0, q_2\}$)

= ϵ-closure($\{q_0\}$) $\cup$ ϵ-closure ($\{q_2\}$)

= $\{q_0, q_1, q_2\} \cup \{q_2\}$

= $\{q_0, q_1, q_2\}$ = A, already in Dstates.

for i/p symbol '1',

U = ϵ-closure(move (A, 1))

= ϵ-closure ($\{q_1, q_2\}$)

= ϵ-closure ($\{q_1\}$) $\cup$ ϵ-closure ($\{q_2\}$).

= $\{q_1, q_2\} \cup \{q_2\}$

DTran [C, 0]

$= \{q_1, q_2\} = B$.

add B as unmarked state in DStates.

DTran[A,0] = A

DTran[A,1] = B

Mark B.

for i/p symbol '0'

U = ϵ-closure (move(B,0))

= ϵ-closure ({q_2})

= {q_2} = C.

add C as unmarked state in DStates.

for i/p symbol '1'

U = ϵ-closure (move(B,1))

= ϵ-closure({q_1, q_2})

= ϵ-closure({q_2}) ∪ ϵ-closure({q_2})

= {q_1, q_2} ∪ {q_2}

= {q_1, q_2} = B, already in DStates.

DTran[B,0] = C

DTran[B,1] = B.

Mark C

for i/p symbol '0'

U = ϵ-closure(move(C,0))

= ϵ-closure({q_2})

$= \{q_2\} = C$, already in Dstates.

for i/p symbol '1',

$U = \epsilon\text{-closure}(\text{move}(C, 1))$

$= \epsilon\text{-closure}(\{q_2\})$

$= \{q_2\} = C$, already in Dstates.

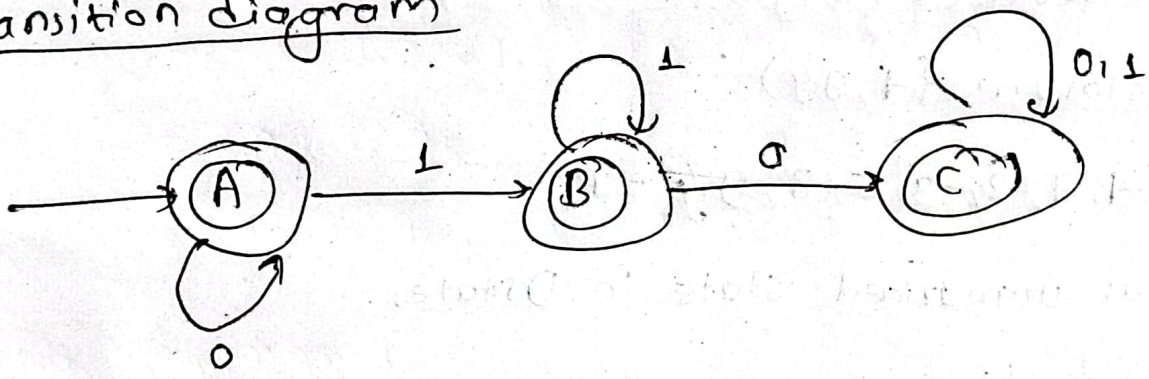
DTran [C, 0] = C

DTran [C, 1] = C

finally, Transition table of DFA

Q/Σ	0	1
→*A	A	B
* B	C	B
* C	C	C

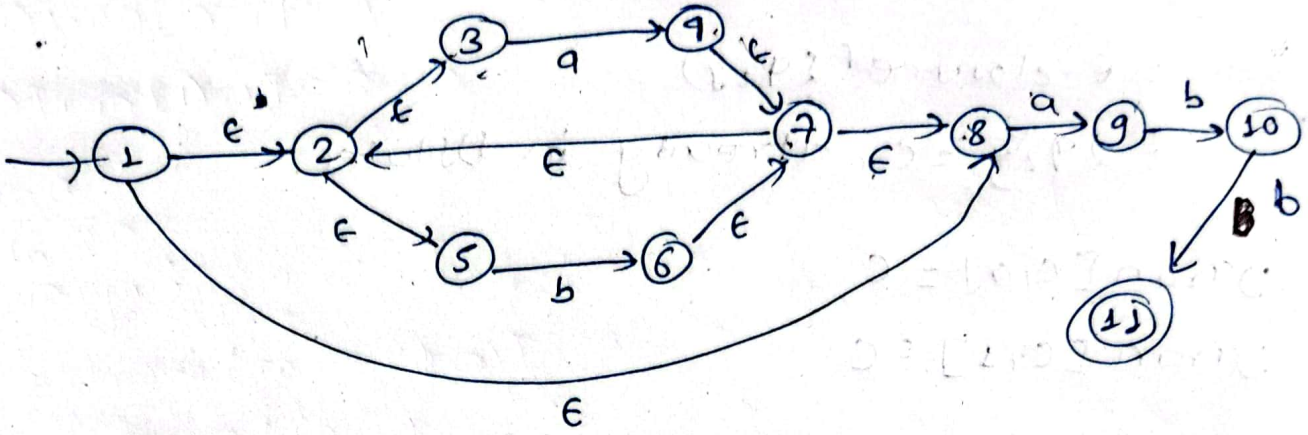
Transition diagram



08-01-24

01-24
CD

Q) find equivalent DFA of following NFA using subset construction algorithm.



Soln;

Initially,

ε-closure (1) = {1, 2, 3, 5, 8} = A as unmarked state in Dstates.

Mark A

For i/p symbol 'a'

U = ε-closure (move(A, a))

= ε-closure ({4, 9})

= {4, 7, 2, 3, 5, 8, 9} = B,

add B as unmarked state in Dstates.

For i/p symbol 'b'

U = ε-closure (move (A, b))

= ε-closure ({6})

= {6, 7, 2, 3, 5, 8} = C,

add C as unmarked state in Dstates.

Dtran [A, a] = B

Dtran [A, b] = C

05-24
CD

Mark B

for i/p symbol 'a'

U = ε-closure (move(B, a))

= ε-closure({4, 9})

= {4, 7, 2, 3, 5, 8, 9} = B, already in Dstates.

for i/p symbol 'b'

U = ε-closure (move(B, b))

= ε-closure({6, 10})

= {6, 7, 2, 3, 5, 8, 10} = D

add D as unmarked states in Dstates.

DTran[B, a] = B

DTran[B, b] = D

Mark C

for i/p symbol 'a'.

U = ε-closure(move(C, a))

= ε-closure({4, 9})

= {4, 7, 2, 3, 5, 8, 9} = B, already in DStates

for i/p symbol 'b'.

U = ε-closure(move(C, b))

= ε-closure({6})

= {6, 7, 2, 3, 5, 8} = C, already in DStates

DTran[C, a] = B.

DTran[C, b] = C

01-24
CD

Mark D.

for i/p symbol 'a'

U = ε-closure(move(D, a))

= ε-closure({4, 9})

= {9, 7, 2, 3, 5, 8, 9} = B, already in Dstates.

for i/p symbol 'b'.

U = ε-closure(move(D, b)).

= ε-closure({6, 11})

= {6, 7, 2, 3, 5, 8, 11} = E.

add E as unmarked state in Dstates.

Dtran[D, a] = B

Dtran[D, b] = E

Mark E

for i/p symbol 'a'

U = ε-closure(move(E, a))

= ε-closure({4, 9})

= {4, 7, 2, 3, 5, 8, 9} = B, already in Dstates

for i/p symbol 'b'.

U = ε-closure(move(E, b))

= ε-closure({6})

= {6, 7, 2, 3, 5, 8} = C, already in Dstates.

01-24

01-24

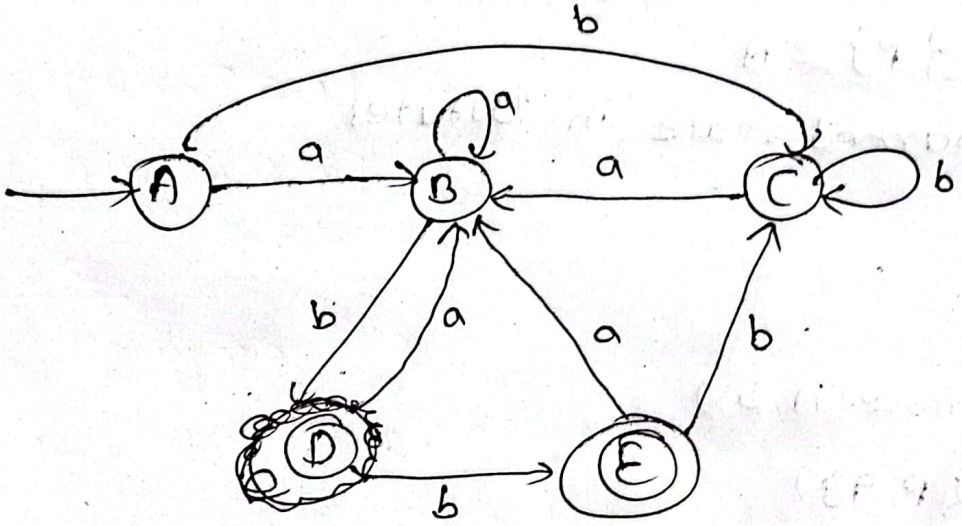
CD

Dtran[E, a] = B

Dtran [E, b] = C

Transition table

	Q/Σ	a	b
→	A	B	C
	B	B	D
	C	B	C
*	D	B	E
*	E	B	C



01-24
CD

Q) Consider the following transition table

q/Σ	0	1
→ p	{p, q}	{p}
q	{r}	{r}
r	{s}	—
*s	{s}	{s}

Now, find equivalent DFA using subset construction algorithm.

Soln;

Initially,

ε-closure(p) = {p} = A

add A as unmarked state in Dstates.

Mark A

for i/p symbol 0,

B = ε-closure(move(A, 0))

= ε-closure({p, q})

= {p, q} = B

add B as unmarked state in Dstates.

for i/p symbol 1,

= ε-closure(move(A, 1))

= ε-closure({p})

= {p} = A, already in Dstates.

DTran[A,0] = B

DTran[A,1] = A

Mark B

for i/p symbol 0

U = ε-closure(move(B,0))

= ε-closure({p, q, r, s})

= {p, q, r, s} = C

add C as unmarked state in DStates.

for i/p symbol 1

U = ε-closure(move(B,1))

= ε-closure({p, r})

= {p, r} = D

add D as unmarked state in DStates.

DTran[B,0] = C

DTran[B,1] = D

Mark C

for i/p symbol 0

U = ε-closure(move(C,0))

= ε-closure({p, q, r, s})

= {p, q, r, s} = E

add E as unmarked state in DStates.

for i/p symbol 1

U = ε-closure(move(C,1))

= ε-closure({p, r})

$= \{p, r\} = D$, already in Dstates.

Dtran [C, 0] = E

Dtran [C, 1] = D

Mark D

for i/p symbol '0'

U = ε-closure (move (D, 0))

= ε-closure ({p, q, s})

= {p, q, s} = F.

add F as unmarked state in Dstates.

for i/p symbol '1'

U = ε-closure (move (D, 1))

= ε-closure (∅{p})

= {p} = A, already in Dstates

Dtran [D, 0] = F

Dtran [D, 1] = A

Mark E

for i/p symbol 0

U = ε-closure (move (E, 0))

= ε-closure ({p, q, r, s})

= {p, q, r, s} = E, already in Dstates.

for i/p symbol 1

U = ε-closure (move (E, 1))

= ε-closure ({p, r, s})

01-24
CD

$= \{p, r, s\} = G$

add G as unmarked state in Dstates.

DTran [E, 0] = E

DTran [E, 1] = G

Mark F

For i/p symbol '0'

U = ε-closure (move (F, 0))

= ε-closure ({p, q, r, s}) already

= {p, q, r, s} = E, already in Dstates.

For i/p symbol '1'

U = ε-closure (move (F, 1))

= ε-closure ({p, r, s})

= {p, r, s} = G, already in Dstates.

DTran [F, 0] = E

DTran [F, 1] = G

Mark G

For i/p symbol '0'

U = ε-closure (move (G, 0))

= ε-closure ({p, q, s})

= {p, q, s} = F, already in Dstates.

For i/p symbol '1'

U = ε-closure (move (G, 1))

= ε-closure ({p, s})

$= \{p, s\} = H$

add H as unmarked state in Dstates,

Dtran [h, 0] = f

Dtran [h, 1] = H

Mark H

for i/p symbol 0.

U = ε-closure (move (H, 0))

= ε-closure ({p, q, s})

= {p, q, s} = F, already in Dstates.

for i/p symbol 1

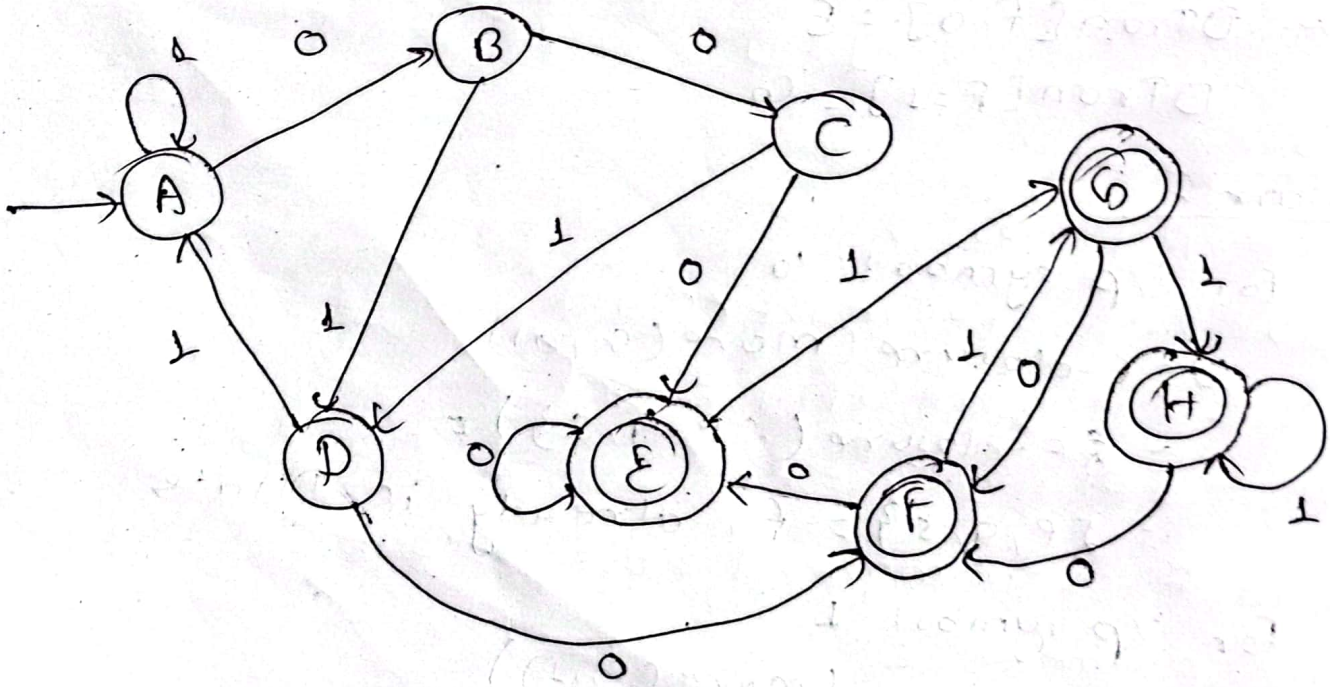
U = ε-closure (move (H, 1))

= ε-closure ({p, s})

= {p, s} = H, already in Dstates.

Dtran [H, 0] = f

Dtran [H, 1] = H



082-02-07

02-07
CD

* State Minimization in DFA:

↳ It refers to process of transforming a given DFA into an equivalent DFA with minimum no. of states.

↳ Two states p and q are called equivalence, if for all i/p string w, $\delta(p,w)$ is an accepting state iff $\delta(q,w)$ is an accepting state

– otherwise distinguishable states.

↳ we say that, string w distinguishes state p from state q if by starting with DFA M in state p and feeding it input w, we end up in an accepting state.

↳ But starting in state q and feeding it with same i/p w, we end up in a non accepting state.

OR, vice-versa.

↳ Each group of states that can't be distinguished is then merged into single state.

* Procedure:

1) Partition the set of states into two partitions.
   a) set of accepting states.
   b) set of non-accepting states.
2) split the partition on the basis of distinguishable state and ~~non~~ put equivalent states in group.
3) To split, we process the transition from state in a group with all i/p symbols.
   – If the transitions on any i/p from the state, in a group is on different group of states, then ~~they~~ they are distinguishable. So, remove those states from

the current partition and create groups.

4) Process until all the partitions contain equivalent states only or have single states.

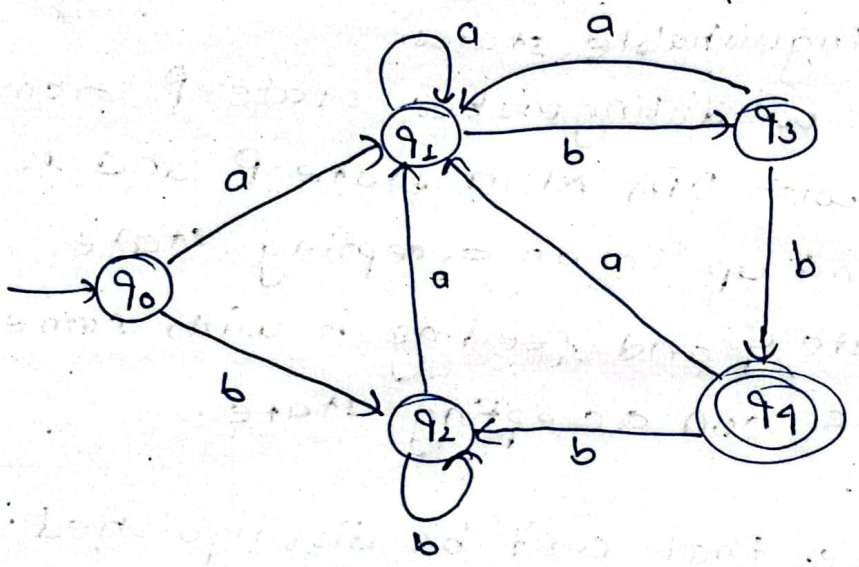
→ State p and q are equivalent if

$\delta(p, w) \in F$ & $\delta(q, w) \in F$

or

$\delta(p, w) \notin F$ or & $\delta(q, w) \notin F$.

Eg:



Transition table:

Q/Σ	a	b
→q0	q1	q2
q1	q1	q3
q2	q1	q2
q3	q1	q4
*q4	q1	q2

Now,

0 equivalent sets = [q0, q1, q2, q3] [q4]

02-04
CD

For 1 equivalent,

For q_0, q_1 : they are equivalent

∵ for a and b, they are in same group.

For q_0, q_2 : they are equivalent.

For q_0, q_3 : they are not equivalent.

∵ for a they are in same group but for b they are in diff^n. group.

∴ 1 equivalent sets = $[q_0, q_1, q_2]\ [q_3]\ [q_4]$

For 2 equivalent,

For q_0, q_1 : they are not equivalent.

∵ for a they are in same group but for b they are not, in same group.

For q_0, q_2 : they are equivalent.

∵ for both a and b they are in same group.

∴ 2 equivalent sets = $[q_0, q_2]\ [q_1]\ [q_3]\ [q_4]$

For 3 equivalent,

For q_0, q_2 : they are equivalent.

∴ 3 equivalent sets = $[q_0, q_2]\ [q_1]\ [q_3]\ [q_4]$

↳ Here, 2 and 3 equivalent sets are equal and further on we will get same result.

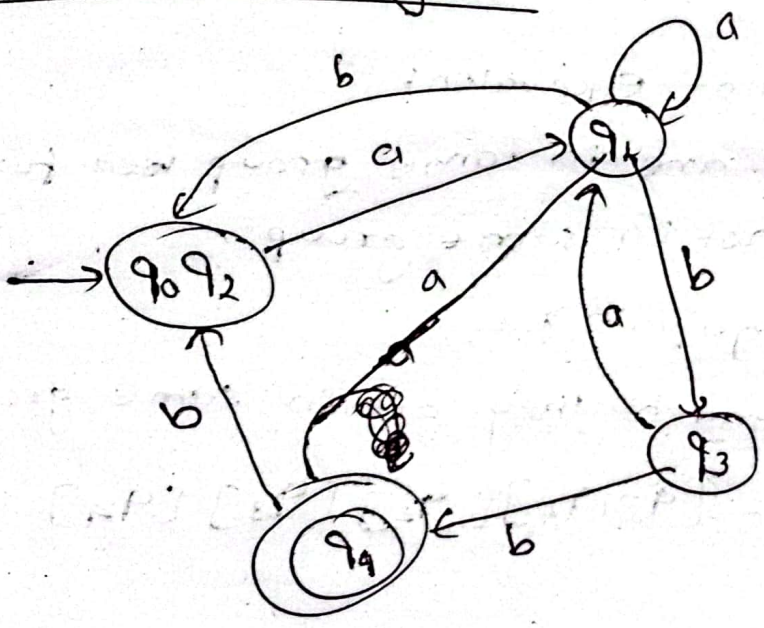
∴ we have four states $\{q_0, q_2\}, \{q_1\}, \{q_3\}, \{q_4\}$

02-07
CD

Transition table:

Q/Σ	a	b
→ q_0q_2	q_1	q_0q_2
q_1	q_1	q_3
q_3	q_1	q_4
*q_4	q_1	q_0q_2

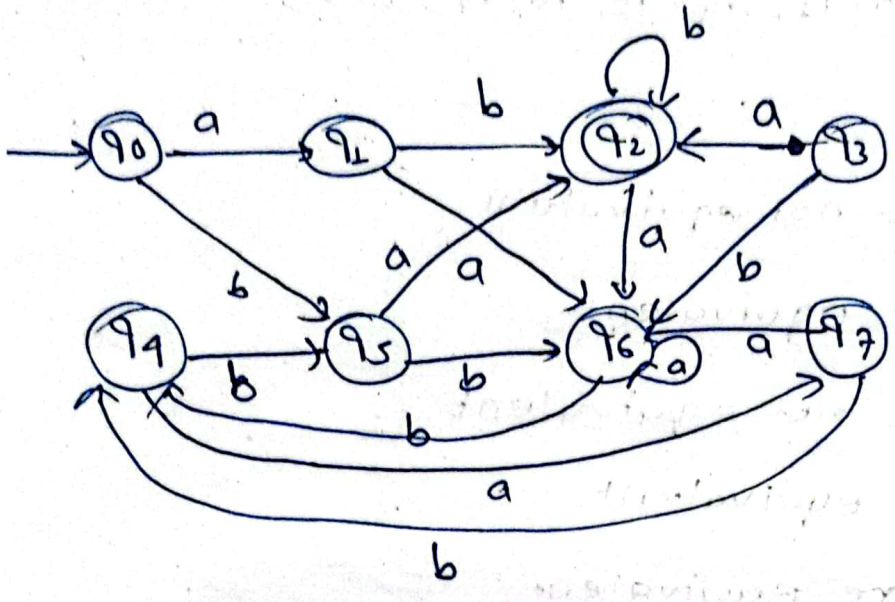
Minimized state diagram:



Ctd →

(8) Minimize the following DFA with equivalence 02-07 .. methods.

CD



Transition table:

Q/Σ	a	b
→ q_0	q_1	q_5
q_1	q_6	q_2
* q_2	q_6	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_4

Now, Remove q_3

02-07
CD

0 equivalent sets = $[q_0, q_1, q_4, q_5, q_6, q_7]$ $[q_2]$

For 1 equivalent

For q_0, q_1 : they are not equivalent

For q_0, q_4 : they are equivalent

For q_0, q_5 : they are not equivalent.

For q_0, q_6 : they are equivalent.

For q_0, q_7 : they are equivalent.

∴ 1 equivalent sets = $[q_0, q_4, q_6, q_7]$ $[q_1]$ $[q_5]$ $[q_2]$

For 2 equivalent

For q_0, q_4 : they are not equivalent

For q_0, q_6 : they are not equivalent

For q_0, q_7 : they are not equivalent

For q_4, q_6 : they are not equivalent

For q_4, q_7 : they are not equivalent.

For q_6, q_7 : they are equivalent

∴ 2 equivalent sets = $[q_6, q_7]$ $[q_0]$ $[q_4]$ $[q_1]$ $[q_5]$ $[q_2]$

For 3 equivalent

For q_6, q_7 : they are equivalent.

∴ 3 equivalent sets = $[q_6, q_7]$ $[q_0]$ $[q_4]$ $[q_1]$ $[q_5]$ $[q_2]$

02-07
CD

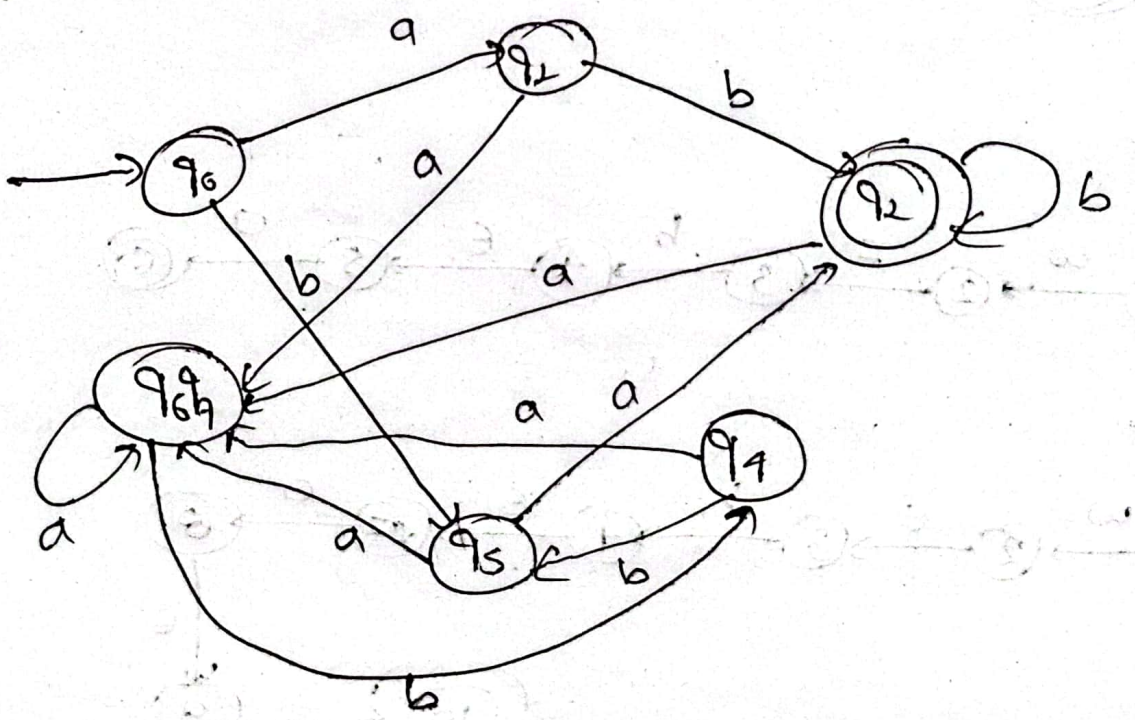
∴ The states we got are:

$\{q_6, q_7\}, \{q_0\}, \{q_1\}, \{q_2\}, \{q_4\}, \{q_5\}$

Transition table

Q/Σ	a	b
→ q_0	q_1	q_5
q_1	q_6q_7	q_2
* q_2	q_6q_7	q_2
~~q_3~~	~~q_2~~	
q_4	q_6q_7	q_5
q_5	q_2	q_6q_7
q_6q_7	q_6q_7	q_4

Minimized state diagram



02-07
CD

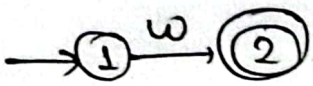
9) Construct the regular expression

who | what | where . Use thompson's construction to build NFA from RE . Use subset construction to build DFA from NFA . Then, minimize DFA.

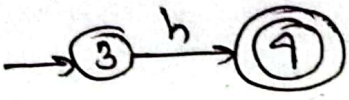
Soln:

R.E : who + what + where

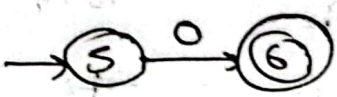
for w,



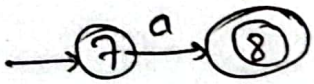
for h.



for o.

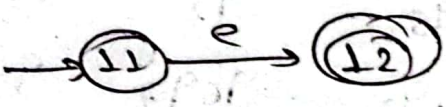


for a,

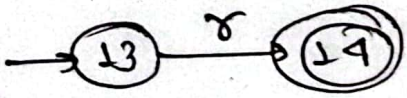


for t.

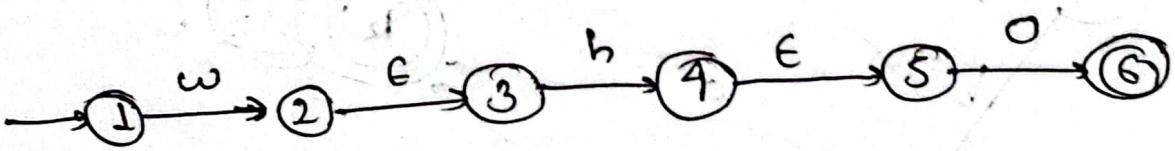
for e.



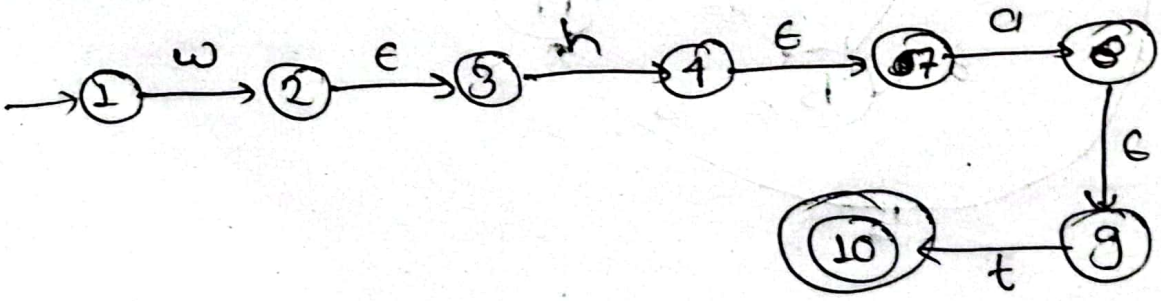
for r,



for who,

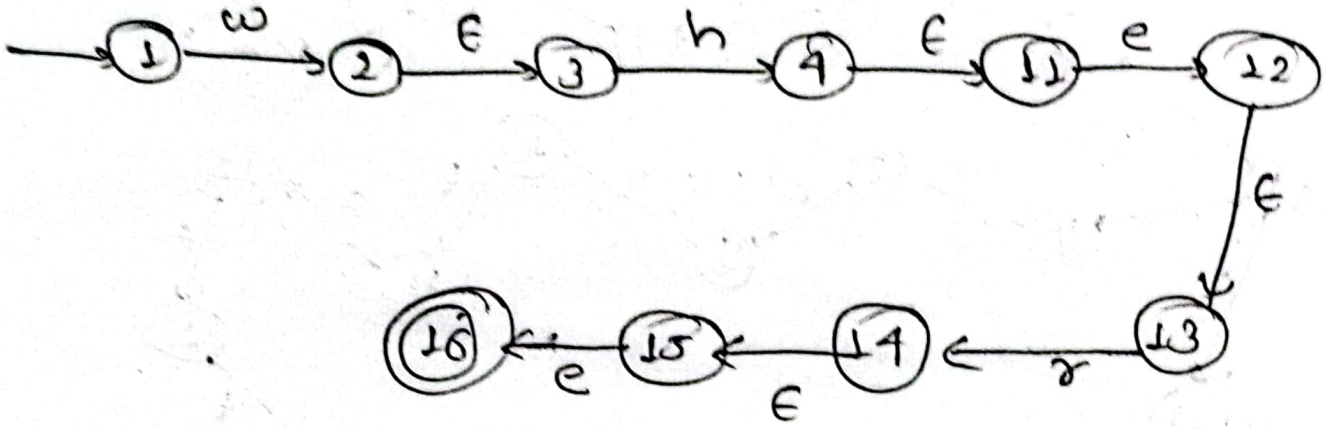


for what,

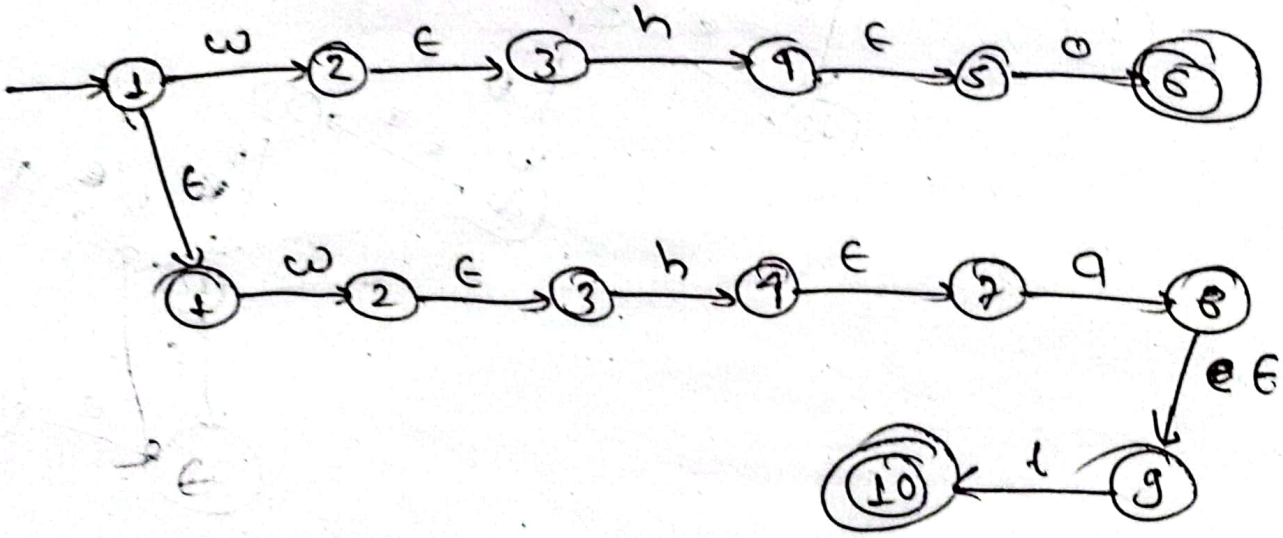


02-07
CD

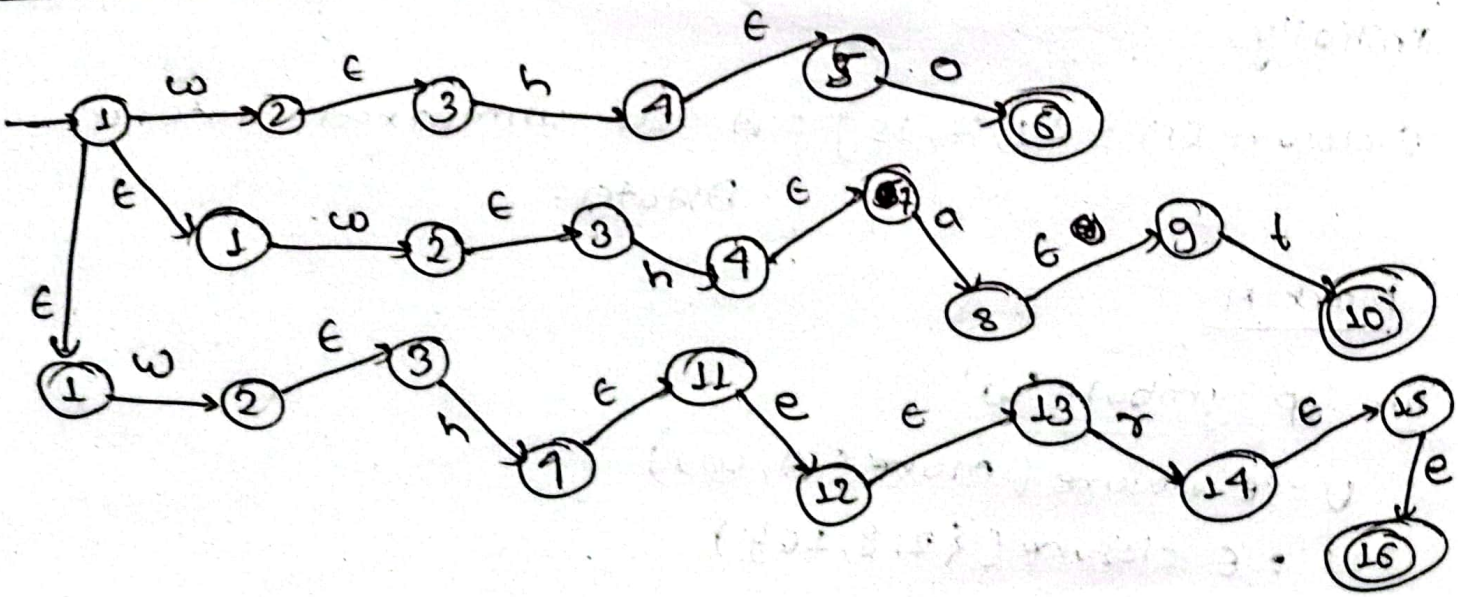
for where



for who + what

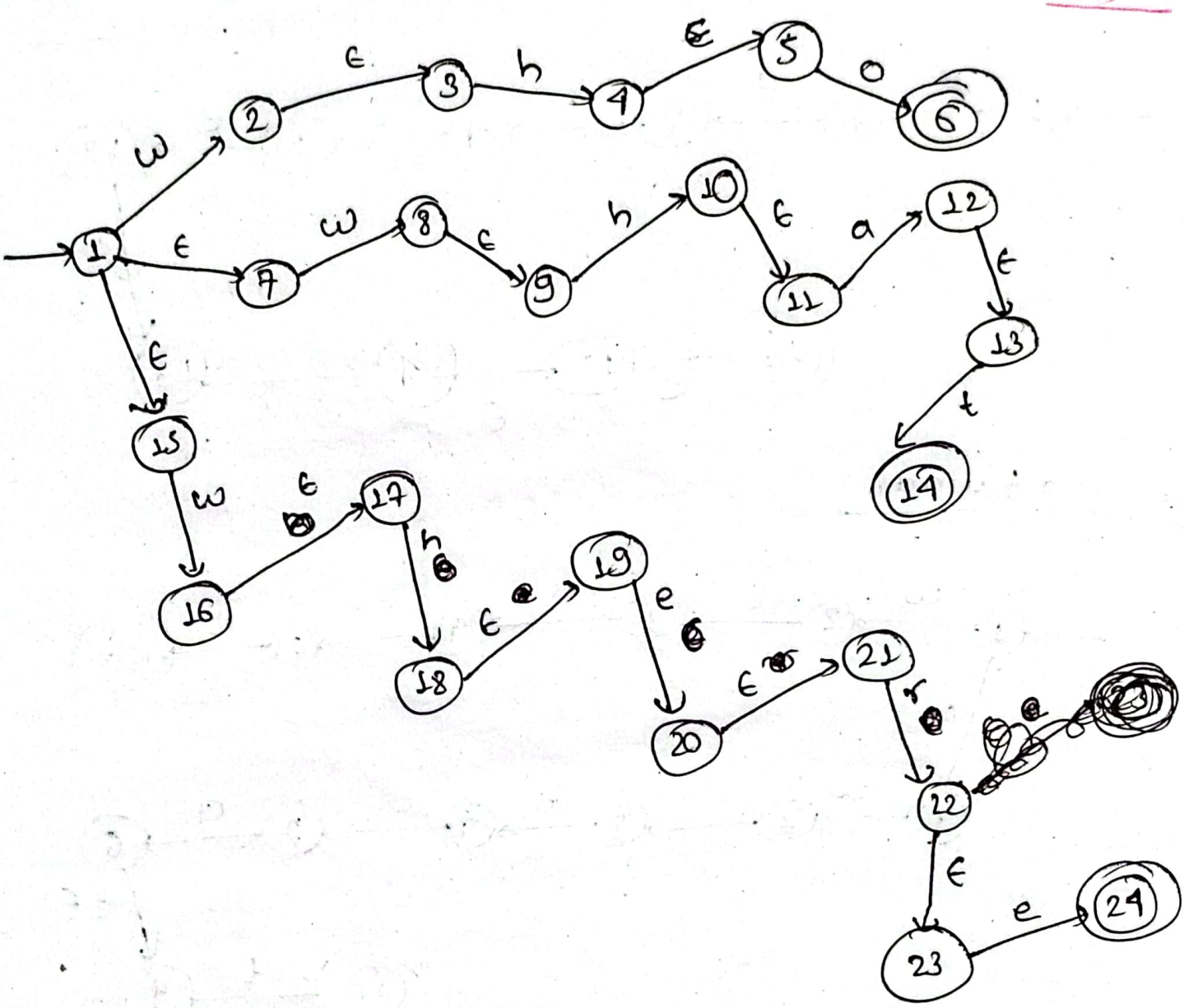


for what what + where



02-07
CD

finally, changing states name.



Conversion of NFA to DFA using subset construction:

Initially,

ϵ-closure(1) = {1, 7, 15} = A as unmarked state in Dstates.

Mark A

for i/p symbol 'w'

U = ϵ-closure(move(A, w))

= ϵ-closure({2, 8, 16})

02-07
CD

$= \{2, 3, 8, 9, 16, 17\} = B$

add B as unmarked state in Dstates.

for i/p symbol 'h'

$U = \epsilon\text{-closure}(move(A, h))$

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

$= \phi$

for i/p symbol 'o'

$U = \phi$

for i/p symbol 'a'

$U = \phi$

for i/p symbol 't'

$U = \phi$

for i/p symbol 'e'

$U = \phi$

for i/p symbol 'r'

$U = \phi$

Dtran [A, w] = B

Dtran [A, h] = Dead

Dtran [A, o] = Dead

Dtran [A, a] = Dead

Dtran [A, t] = Dead

Dtran [A, e] = Dead

Dtran [A, r] = Dead

Mark B.

02-07
CD

for i/p symbol 'w'

U = ε-closure(move(B, w))

= ϕ

for i/p symbol 'h'

U = ε-closure(move(B, h))

= ε-closure{9, 10, 18}

= {9, 5, 10, 11, 18, 19} = C

add C as unmarked state in DStates.

for i/p symbol 'o'

U = ε-closure(move(B, o))

= ϕ

for i/p symbol 'a'

U = ε-closure(move(B, a))

= ϕ

for i/p symbol 't'

U = ε-closure(move(B, t))

= ϕ

for i/p symbol 'e'

U = ε-closure(move(B, e))

= ϕ

for i/p symbol 'r'

U = ε-closure(move(B, r))

= ϕ

02-07
CD

DTran [B, w] = Dead

DTran [B, h] = C

DTran [B, o] = Dead

DTran [B, a] = Dead

DTran [B, t] = Dead

DTran [B, e] = Dead

DTran [B, r] = Dead

Mark C,

for i/p symbol 'w'

U = ε-closure (move (C, w))

= ϕ

for i/p symbol 'h'

U = ε-closure (move (C, h))

= ϕ

for i/p symbol 'o'

U = ε-closure (move (C, o))

= ε-closure ({6, 9})

= {6} = D

add D as unmarked state in Dstates.

for i/p symbol 'a'

U = ε-closure (move (C, a))

= ε-closure ({12})

= {12, 13} = E

add E as unmarked state in Dstates.

ctd →

02-07
CD

for i/p symbol 't',

U = ε-closure (move (C, t))

= ~~ε-closure~~ ϕ

for i/p symbol 'e'.

U = ε-closure (move (C, e))

= ε-closure ({20})

= {20, 21} = F

add F as unmarked state in Dstates.

for i/p symbol 'a'

U = ε-closure (move (C, a))

= ϕ

DTran [C, w] = Dead

DTran [C, h] = Dead

DTran [C, o] = D

DTran [C, a] = E

DTran [C, t] = Dead

DTran [C, e] = F

DTran [C, r] = Dead

Mark D,

for i/p symbol 'w'

U = ε-closure (move (D, w))

= ϕ

CS

02-07
CD

for i/p symbol 'h'

U = ε-closure (move (D, h))

= ϕ

for i/p symbol 'o'

U = ε-closure (move (D, o))

= ϕ

for i/p symbol 'a'

U = ε-closure (move (D, a))

= ϕ

for i/p symbol 't'

U = ε-closure (move (D, t))

= ϕ

for i/p symbol 'e'

U = ε-closure (move (D, e))

= ϕ

for i/p symbol 'r'

U = ε-closure (move (D, r))

= ϕ

DTran [D, w] = Dead

DTran [D, h] = Dead

DTran [D, o] = Dead

DTran [D, a] = Dead

DTran [D, t] = Dead

DTran [D, e] = Dead

DTran [D, r] = Dead.

Mark E,

02-07
CD

for i/p symbol 'w'

U = ε-closure (move (E, w))

= ϕ

for i/p symbol 'h'

U = ε-closure (move (E, h))

= ϕ

for i/p symbol 'o'

U = ε-closure (move (E, o))

= ϕ

for i/p symbol 'a'

U = ε-closure [move (E, a))

= ϕ

for i/p symbol 't'

U = ε-closure (move (E, t))

= ε-closure ({14})

= {14} = G1

add G1 as unmarked state in Dstate

for i/p symbol 'e'

U = ε-closure (move (E, e))

= ϕ

for i/p symbol 'r'

U = ε-closure [move (E, r))

= ϕ

Dtran [E, w] = Dead

Dtran [E, h] = Dead ..... Do It Yourself

Unit-3

Parser

*Syntax Analysis:

↳ is the second phase of compiler process, which checks the syntatical structure of the given i/p i.e, whether the given i/p is in correct syntax or not.

↳ also called parsing.

↳ It does so by building a data structure called a parse tree or syntax tree; which is constructed by using pre-defined grammar of language and the i/p string.

↳ Syntaxes are represented using CFG.

↳ Parsing is the act of performing syntax analysis to verify an i/p program's compliance with source language.

↳ it ensures we have valid sequence of tokens.

*Role of Parser:

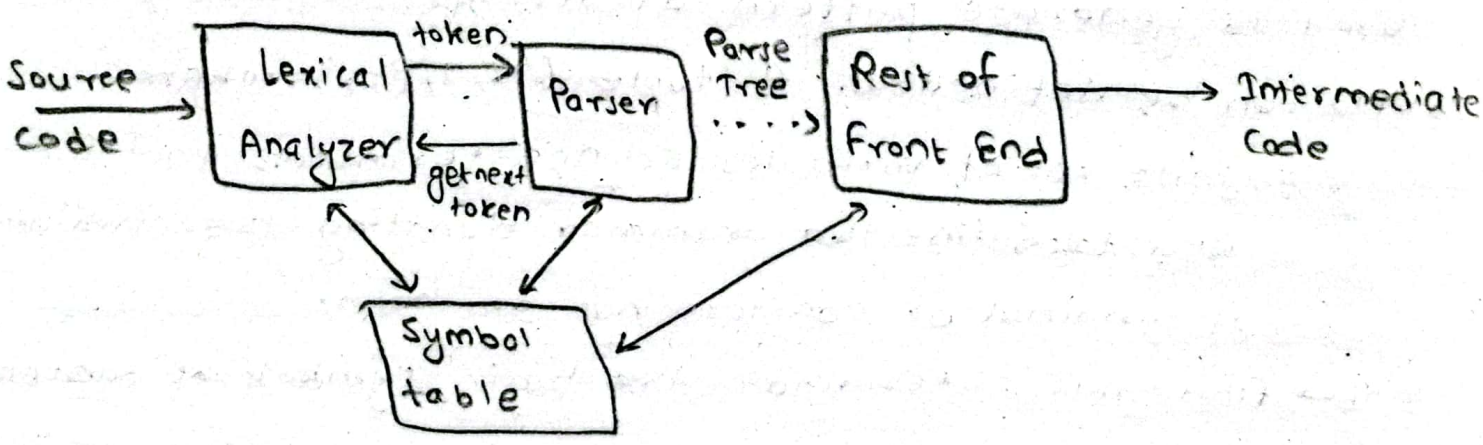


fig:- role of parser in compiler model.

↳ Parser analyze the source program based on the definition of its syntax. 02-13 CD

↳ it works in lock-step with lexical analyzer and is responsible for creating a parse tree of source code.

↳ it ~~used~~ provides the mechanism for ~~checks~~ context sensitive analysis.

↳ it determines error and tries to handle them.

* Types of Parser

1) Top Down Parser

↳ builds parse tree from top to bottom.

↳ easy to build.

↳ works on subset of grammars like LL grammar.

2) Bottom Up Parser

↳ builds parse tree from bottom to top.

↳ difficult to build.

↳ works on subset of grammars like LR grammar.

* Content Free Grammar (CFG):

↳ CFG is a set of recursive rewriting rule or production used to generate pattern of strings.

↳ CFG can be defined as 4 tuple (V, T, P, S) where

$V \rightarrow$ finite set of variables or non-terminals that are used to define the grammar denoting the combination of terminal or non-terminal or both.

$T \rightarrow$ finite set of terminals, the basic symbols of sentence.

$P \rightarrow$ set of productions, which are rules that defines the combination of terminal or non-terminal or both for particular non-terminal.

S → start symbol which is a special non-terminal symbol that appears in the initial string generated by grammar. 02-13 CD

↳ lower case letters and symbols, bold string are used for denoting terminal. eg:- a, b, c, 0, 1 ∈ T

↳ Upper Case letters, italicized strings are used for denoting nonterminals. eg:- A, S ∈ V.

↳ Production rules are of the form $A \to \alpha$

↳ strings comprising of both terminals and non-terminals are denoted by greek letters like $\alpha, \beta, \gamma, \delta$.

* Derivations:

↳ Verification of the sentence defined by the grammar is done using the production rule to obtain the particular sentence by expansion of non terminal is called derivation.

↳ strings of non terminals and terminals is called sentential form, where as strings of terminals is called sentence.

↳ During derivation, if we always expand the left most non terminal first then it is called left-most derivation.

↳ During derivation, if we always expand the right most non terminal first then it is called rightmost derivation.

↳ $\alpha \Rightarrow \beta$ is used to represent that β can be derived from α.

↳ $\alpha \overset{*}{\Rightarrow} \beta$ represents that β can be derived from α in zero or more steps.

↳ $\alpha \overset{+}{\Rightarrow} \beta$ represents that β can derived from α in one or more steps.

↳ L(G) is the language of G which is set of sentences.

↳ A sentence of L(G) is string of terminals of G.

b ie, if S is start symbol of G, then w is a sentence of L(G) if and only if $S \Rightarrow w$, where w is string of terminals of G.

↳ If G is a CFG then, L(G) is a CFL.

↳ Two grammars are equivalent if they produce the same language.

Eg:-

Consider a grammar $G = (V, T, P, S)$

where,

$V = \{E\}$

$T = \{+, *, -, c, >, id\}$

$P = \{ E \to E+E$
$E \to E*E$
$E \to (E)$
$E \to E-E$
$E \to id$
$\}$

$S = E$

For id*id+id

L.M.D

$E \underset{lm}{\Rightarrow} E+E$
$\underset{lm}{\Rightarrow} E*E+E$
$\underset{lm}{\Rightarrow} id*E+E$
$\underset{lm}{\Rightarrow} id*id+E$
$\underset{lm}{\Rightarrow} id*id+id$

R.M.D

$E \underset{rm}{\Rightarrow} E*E$
$\underset{rm}{\Rightarrow} E*E+E$
$\underset{rm}{\Rightarrow} E*E+id$
$\underset{rm}{\Rightarrow} E*id+id$
$\underset{rm}{\Rightarrow} id*id+id$

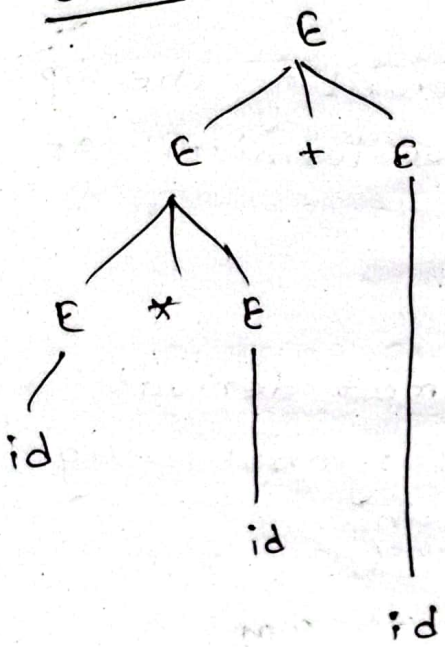
02-13
CD

*Parse Tree:

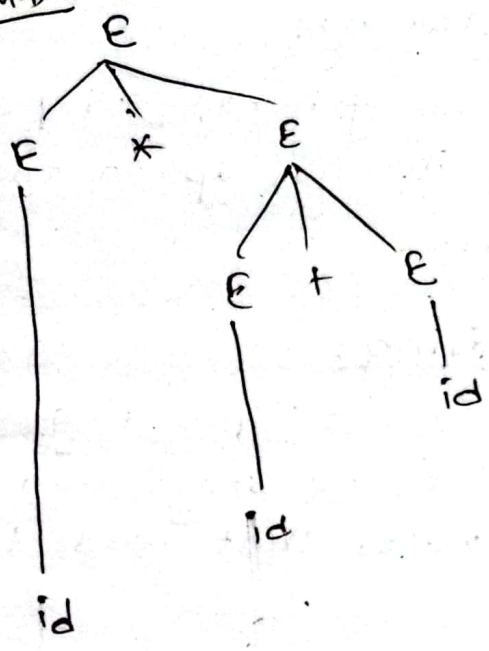
↳ is a graphical representation of CFG derivation.
↳ here, root is the start symbol, internal nodes represent non terminals and leaf nodes represent terminals.

Parse tree for id*id+id

L.M.D



R.M.D



↳ if a grammar produce more than one unique parse tree for a sentence or derivation (left or right) then it is called ambiguous grammar.

Eg:

$E \to E+E$

$E \to E*E$

$E \to id$

for id+id*id ① LMD

$E \underset{lm}{\Rightarrow} E+E$

$\underset{lm}{\Rightarrow} id+E$

$\underset{lm}{\Rightarrow} id+E*E$

$\underset{lm}{\Rightarrow} id+id*E$

$\underset{lm}{\Rightarrow} id+id*id$

② LMD

$E \underset{lm}{\Rightarrow} E*E$

$\underset{lm}{\Rightarrow} E+E*E$

$\underset{lm}{\Rightarrow} id+E*E$

$\underset{lm}{\Rightarrow} id+id*E$

$\underset{lm}{\Rightarrow} id+id*id$

It is ambiguous.

082-02-14

02-14
CD

*Types of Parsing:

1) Top Down Parsing:

↳ Parsing involves generating the string from the first non terminal and repeatedly apply. production rules.

2) Bottom up Parsing:

↳ parsing involves ~~generating~~ repeatedly rewriting the i/p string until it ends up in the first non-terminal of the grammar.

* Top Down Parsing:

↳ Tries to derive the i/p string using leftmost derivation i.e, starting at the start non terminal symbol using production rules that leads to i/p string.

↳ It includes:

a) Recursive Descent Parsing:

↳ Backtracking is needed i.e. if a choice of production rule doesnot work, we have to back track to check for next production rule.

↳ Not widely used, not efficient.

b) Non-recursive Predictive Parsing:

↳ No backtracking.

↳ efficient.

↳ Use LL (left to right scanning, left to right derivation) method.

↳ needs special form of grammar (LL(1) grammar)

↳ Non-recursive (table drive) predictive parse is also known as LL(1) parser.

02-14
CD

* Recursive Descent Parsing :

↳ Start with the starting non-terminal and use the first production, verify with input and if there is no match backtrack and apply another.

↳ **Rules:**

1) Use two pointers : iptr for pointing i/p symbol to be read and optr for pointing the symbol of o/p string that is ~~usue~~ initially start symbol s.
2) if the symbol pointed by optr in non terminal, use the first production rule for expansion.
3) while the symbol pointed by iptr and optr is same increment both pointers.
4) the loop at the above step terminates when
   a) A non terminal at o/p (Case A)
   b) the end of i/p (Case B).
   c) Unmatching terminal symbol pointed by iptr and optr is seen (Case C).
5) If case A is true, expand non terminal using first rule and goto step 2.
6) If case B is true, terminate with success.
7) If case C is true, decrement both pointers to the place of last non terminal expansion and use the next production rule for non terminal.
8) If there is no more production and B is not true, report error.

C↑d →

02-14
CD

Example:

let grammar G where,

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

For i/p cad:

input	output	Rules
(iptr) cad	(optr) S	Rule 1
(iptr) cad	(optr) cAd	Rule 2
c (iptr) ad	c (optr) Ad	Rule 3, match c
c (iptr) ad	c (optr) abd	Rule 5
ca (iptr) d	ca (optr) bd	Rule 3, match a
c (iptr) ad	c (optr) Ad	Rule 7, backtrack
c (iptr) ad	c (optr) ad	Rule 5
ca (iptr) d	ca (optr) d	Rule 3, match a
cad (iptr)	cad (optr)	Rule 3, match d

Now, process terminates with success.

9) Consider a grammar G where

$A \rightarrow 0A1$

$A \rightarrow B$

$B \rightarrow \#$

Perform recursive descent parsing for string 00#11.

Ctd

for i/p 00#11,

input	output	Rule
(iptr) 00#11	(optr) A	Rule 1
(iptr) 00#11	(optr) 0A1	Rule 2
0(iptr) 0#11	0(optr) A1	Rule 3, match 0.
0(iptr) 0#11	0(optr) 0A11	Rule 2.
00(iptr) #11	00(optr) A11	Rule 3, match 0
00(iptr) #11	00(optr) 0A111	Rule 2
00(iptr) #11	00(optr) A11	Rule 7, backtrack
00(iptr) #11	00(optr) B11	Rule 2
00(iptr) #11	00(optr) #11	Rule 2
00(iptr) #11	00#(optr) 11	Rule 3, match #
00#(iptr) 11	00#1(optr) 1	Rule 3, match 1
00#1(iptr) 1	00#11(optr)	Rule 3, match 1
00#11(iptr)		

Now, process terminates with success.

for i/p 0011,

Cld
→

Example:

02-19
CD

input	output	Rules -
(iptr) 0 0 1 1	(optr) A	Rule 1
(iptr) 0 0 1 1	0 (optr) A 1	Rule 2
0 (iptr) 0 1 1	0 (optr) A 1	Rule 3, match 0
0 (iptr) 0 1 1	0 (optr) 0 A 1 1	Rule 2
0 0 (iptr) 1 1	0 0 (optr) A 1 1	Rule 3, match 0
0 0 (iptr) 1 1	0 0 (optr) 0 A 1 1 1	Rule 2
0 0 (iptr) 1 1	0 0 (optr) A 1 1	Rule 7, backtrack
0 0 (iptr) 1 1	0 0 (optr) B 1 1	Rule 2
0 0 (iptr) 1 1	0 0 (optr) # 1 1	Rule 2.
0 0 (iptr) 1 1	0 0 (optr) B 1 1	Rule 7, backtrack

Error! reported. as no production rules are there and i/ps are not ended yet.

* Left Recursion:

↳ the grammar with recursive non terminal at the left of production is called left recursive grammar.

↳ A grammar is left recursive if it has a non-terminal 'x' such that there is derivation

$$X \Rightarrow X\alpha \text{ for some string } \alpha.$$

↳ Left recursion causes recursive descent parser to go to infinite loop.

↳ Top down parsing techniques can't handle left recursive grammar.

↳ Hence, we have to convert left recursive grammar into an equivalent grammar which is not left recursive.

↳ the left recursion may appear in a single step of derivation called immediate left recursion or may appear in more than one step of the derivation.

02-19/21
CD

082-02-21

↳ example of non immediate left recursion

$S \rightarrow Aab \mid c$

$A \rightarrow Sc \mid c$

- this grammar is not immediate left recursive, but is still left recursion i.e,

$S \rightarrow Aab \rightarrow Scab$

or

$A \rightarrow Sc \rightarrow Aabc$

*Removing left recursion

↳ if we have the grammar of the form $A \rightarrow A\alpha \mid \beta$ the rule for removing the left recursion is,

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

↳ Any production of the form

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \beta_n$

we have,

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \alpha_n A' \mid \epsilon$

*Left Factoring:

↳ Backtracking is expensive operation so if we are able to predict the unique prodn rule while parsing then we are saving a lot of time resources.

↳ left factoring is the approach of transforming the grammar that has two or more prodn rules with same

initial substring to the one that has not, so that it will be useful for predictive parsing.

↳ for the grammar of the form,

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$$

left factoring yield,

$$A \to \alpha A'$$

$$A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

* Predictive Parsing:

↳ Recursive descent parsing is inefficient as it causes lot of backtracking and suffers from left recursion problem.

↳ Predictive parsing tries to predict which production produces the least chance of backtracking and infinite loop.

↳ Predictive parser can choose the production to apply solely on the basis of next input symbol and the current non-terminal being processed.

↳ To enable this, the grammar must take a particular form, such a grammar is called LL(1).

↳ First L means scan the i/p from left to right.

↳ second L means create leftmost derivation.

↳ 1 means one i/p symbol of lookahead.

↳ LL(1) has no left recursion prodn and has been left factored.

Eg:-

stmt → if .... |
while .... |
begin .... |
for ....

↳ Here, for non terminal 'stmt', if next token is 'begin', then we use third production rule instead of first production as in recursive descent parsing.

* Non Recursive Predictive Parsing :

↳ it is built by maintaining a stack explicitly rather than implicitly through recursive calls.

↳ is a table driven parsing.

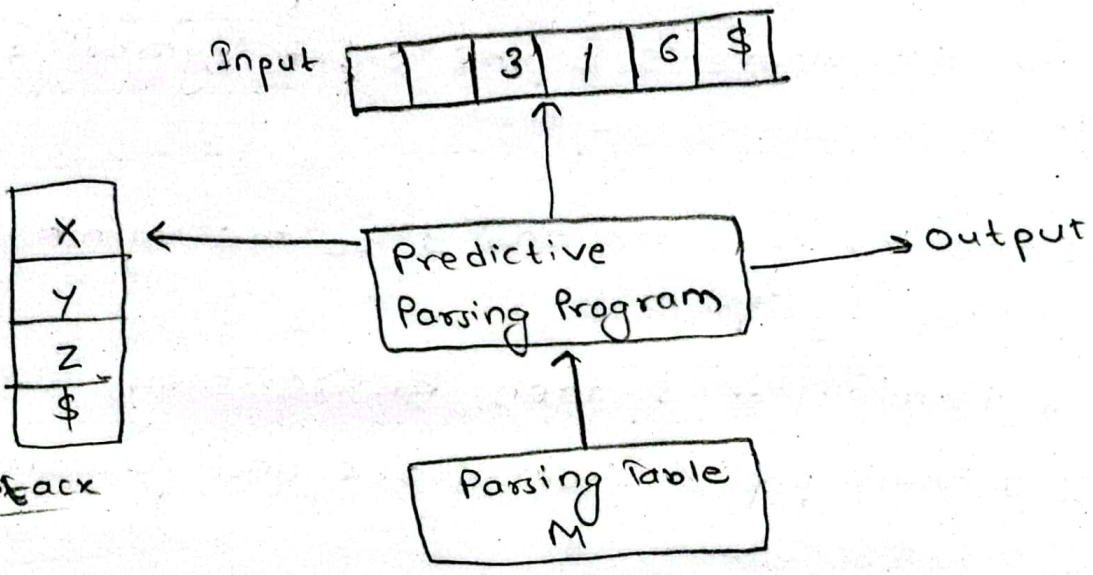


Fig:-Non-Recursive Predictive Parsing

↳ A table driven predictive parser has stack input buffer, parsing table and output stream.

↳ Input buffer contains sentence to be parsed and at the end $ as end marker.

↳ Stack contains symbols of grammar, initially stack contains start symbol on top of $.

↳ When stack is empty (only $ left on stack) parsing is completed.

↳ Parsing table is two dimensional array containing the information about the production to be used on seeing terminal at input and non terminal at stack.

- each entry in parsing table holds prodⁿ rule.
- each row holds non terminal symbol.
- each column holds terminal or $.

ip → input pointer

*Algorithm for Parsing

input : a string w.

output: if w is in L(G), a leftmost derivation of w, else error.

1. set i/p to point the first symbol of i/p stream.
2. set stack to $s where s is the start symbol of grammar.
3. do
   let X be the top of stack and a be the symbol pointed by i/p.
   if X is a terminal or $ then
      if X=a then pop X from stack and forword p.
      else
         error().
   else
      if M[X,a] = $X \rightarrow Y_1 Y_2 \ldots Y_n$, then
         - pop X from stack
         - push $Y_n, Y_{n-1}, \ldots Y_1$ onto stack with Y_1 on top
         - output production $X \rightarrow Y_1, Y_2, \ldots, Y_n$
      else
         error()
4. until X=$

* Constructing LL(1) Parsing Table:

↳ Parsing table requires two functions FIRST and FOLLOW.

↳ for a string of grammar symbols α.

FIRST(α) is the set of terminals that begin all possible strings derived from α.

↳ if $\alpha \overset{*}{\Rightarrow} \epsilon$, then ϵ is also in FIRST(α).

↳ FOLLOW(A) for non terminal A is the set of terminals that can appear immediately to the right A is some sentential form

- if A can be last symbol in a sentential form then $ is also in FOLLOW(A).
- a terminal 'a' is in FOLLOW(A) if $S \overset{*}{\Rightarrow} \alpha A a \beta$.
- $ is in FOLLOW(A) if $S \overset{*}{\Rightarrow} \alpha A$.

Computation of FIRST

↳ FIRST(α) = { set of terminals that begins all strings derived from α }.

→ if 'a' is a terminal symbol then

FIRST(a) = {a}.

→ if 'X' is a non terminal symbol and $X \to \epsilon$ is a prod^n rule then, FIRST(X) = FIRST(X) $\cup \epsilon$

→ if 'X' is a non terminal symbol and $X \to Y_1 Y_2 \dots Y_n$ is a prod^n rule. then

i) if a terminal 'a' is FIRST(Y_1) then

FIRST(X) = FIRST(X) $\cup$ FIRST(Y_1).

cfd

ii) if a terminal 'a' in FIRST(Y_i) and ϵ is in all FIRST(Y_j) for $j = 1 \dots i-1$ then

FIRST(X) = FIRST(X) ∪ a.

iii) if ϵ is in all FIRST(Y_j) for $j = 1 \dots n$ then

FIRST(X) = FIRST(X) ∪ ϵ

↳ if X is ϵ then. FIRST(X) = {ϵ}

↳ if X is $Y_1 Y_2 \dots Y_n$

i) if a terminal 'a' in FIRST(Y_i) and ϵ is in all FIRST(Y_j) for $j = 1 \dots i-1$ then

FIRST(X) = FIRST(X) ∪ a

ii) if ϵ is in all FIRST(Y_j) for $j = 1 \dots n$ then

FIRST(X) = FIRST(X) ∪ ϵ.

Computation of FOLLOW

↳ FOLLOW(A) = { set of terminals that can immediately follow non terminal A }

↳ apply the following rules until nothing can be added to any FOLLOW set.

i) if S is the start symbol, then $ is in FOLLOW(S).

ii) if $A \to \alpha B \beta$ is a prodⁿ rule then everything in FIRST(β) is placed in FOLLOW(B) except ϵ.

iii) if ($A \to \alpha B$ is a prodⁿ rule) or ($A \to \alpha B \beta$ is a rule and ϵ is in FIRST(β)) then everything in FOLLOW(A) is in FOLLOW(B).

02-21/22
CD

Algorithm for Parsing Table

input: LL(1) Grammar

output: Parsing table M

for each prodn rule $A \to \alpha$ in grammar G do

  for each $a \in FIRST(\alpha)$ do

    Add $A \to \alpha$ to $M[A, a]$

  End do

  if $\epsilon \in FIRST(\alpha)$ then

    for each $b \in FOLLOW(A)$ do

      Add $A \to \alpha$ to $M[A, b]$

    End do

  End if

End do

Mark each undefined entry in M error

082-02-22

(1)

Q).

	FIRST	FOLLOW
$S \to aABb$	{a}	{$}
$A \to c \mid \epsilon$	{c, ε}	{d, b}
$B \to d \mid \epsilon$	{d, ε}	{b}

(2).

	FIRST	FOLLOW
$S \to Bb \mid Cd$	{a, b, c, d}	{$}
$B \to aB \mid \epsilon$	{a, ε}	{b}
$C \to cC \mid \epsilon$	{c, ε}	{d}

for FIRST of S

$S \to Bb \mid Cd$

$S \to aBb \mid b \mid cCd \mid d$

02-22
CD

(3).

	FIRST	FOLLOW
S → ACB | CbB | Ba	{d, g, h, b, a, ϵ}	{$}
A → da | BC	{d, g, h, ϵ}	{h, g, b, $}
B → g | ϵ	{g, ϵ}	{$, a, h, g}
C → h | ϵ	{h, ϵ}	{h, g, b, $}

In follow, if non-terminal in right, write the FIRST of that non-terminal, if ϵ is there replace it into main rule

(4).

	FIRST	FOLLOW
S → ABCDE	{a, b, c}	{$}
A → a | ϵ	{a, ϵ}	{b, c}
B → b | ϵ	{b, ϵ}	{c}
C → c	{c}	{d, e, $}
D → d | ϵ	{d, ϵ}	{e, $}
E → e | ϵ	{e, ϵ}	{$}

Q. Consider the grammar

$$S \rightarrow aBa$$

$$B \rightarrow bB \mid \epsilon$$

Now, perform non recursive predictive parsing on string 'abba'

02-22
CD

Soln:

Initially, find FIRST and FOLLOW

	FIRST	FOLLOW
$S \rightarrow aBa$	{a}	{$}
$B \rightarrow bB \mid \epsilon$	{b, ε}	{a}

Now, construct parsing table.

	a	b	$
S	$S \rightarrow aBa$		
B	$B \rightarrow \epsilon$	$B \rightarrow bB$	

Stack	Input	Output
$S	abba$	$S \rightarrow aBa$
$aBa	abba$	
$aB	abba$	$B \rightarrow bB$
$aBb	abba$	
$aB	abba$	$B \rightarrow bB$
$aBb	abba$	
$aB	abba$	$B \rightarrow \epsilon$
$a	abba$	
$	abba$	

Successfully complete i.e. accept.

CS

02-22
CD

Output:

$S \to aBa,\ B \to bB,\ B \to bB,\ B \to \epsilon$

∴ derivation

$S \Rightarrow aBa$

$\Rightarrow abBa$

$\Rightarrow abbBa$

$\Rightarrow abba$

(9)

$S \to aABb$

$A \to c \mid \epsilon$

$B \to d \mid \epsilon$

Perform non-recursive predictive parsing for string 'acdb'

Soln:

Initially - [Take reference from previously done its FIRST & FOLLOW]

Now, Construct paring table -

	a	b	c	d	$
S	$S \to aABb$				
A		$A \to \epsilon$	$A \to c$	$A \to \epsilon$	
B		$B \to \epsilon$		$B \to d$	

02-22
CD

stack	input	o/p.
$S	acdb$	S → aABb
$bBAa	acdb$	A → c
$bBA	acdb$	
$bBc	acdb$	B → d
$bB	acdb$	
$bd	acdb$	
$b	acdb$	
$	acdb$	

Successfully complete i.e, accept

Output:

S → aABb , A → c, B → d

Derivation

S ⇒ aABb

⇒ acBb

⇒ acdb