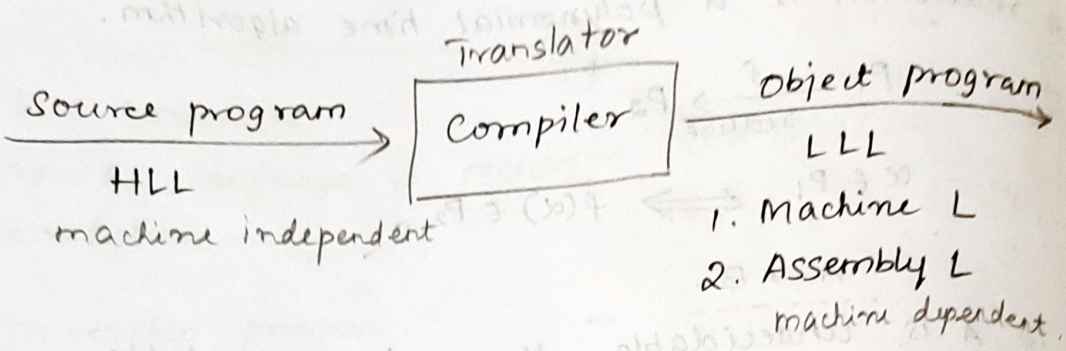
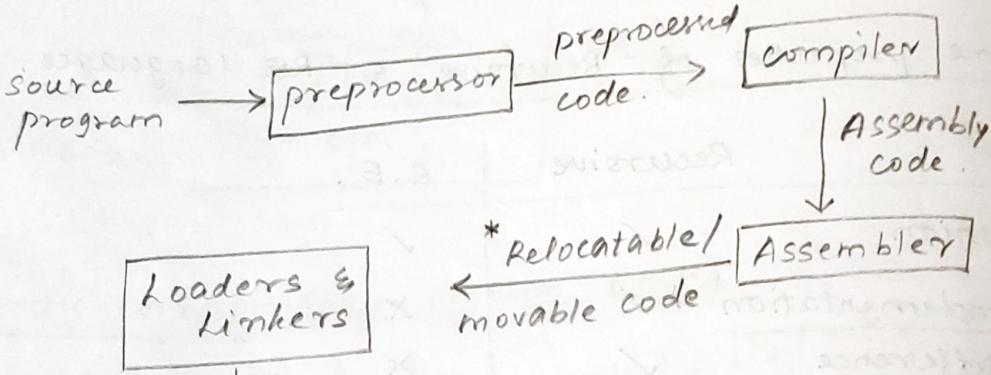
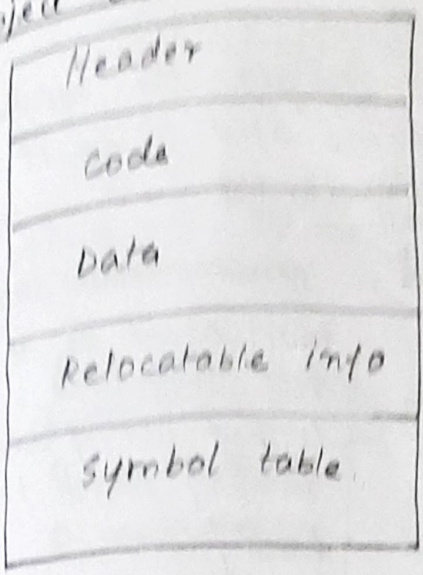
Principles of Compiler design.

Text : Compilers by Aho Ullman
System Programming by Dhamdhere





Object code:



Modules of a compiler.

1. Lexical Analyser
   - ↓ source program
   - ↓ stream of tokens / lexemes

Tokens:

i. keywords
ii. identifiers
iii. operators
iv. constants
v. delimiters

Token → (type, value).

e.g.

Token	Representation
if	(1, -)
for	(2, -)
while	(3, -)
+	(4, 1)
x ← identifier	(7, ptr to symbol table)
(	(8, 1)
)	(8, 2)
{	(8, 3)
}	(8, 4)
;	(8, 5)
<=	(6, 2)
=	(6, 3)

Symbol table

	name	type	address
0	i	int var	
1	10	int const	
2	a	int var	
3	b	" var	
4	1	" const	

```
while ( i <= 10)
{
    a = a+b;
    i = i+1;
}
```

↓

Stream of tokens

(3, -),
(8, 1),
(7, 0),
(6, 2),
(9, 1),
(7, 2),
(6, 3), (7, 2)
(4, 1), (7, 3)
(8, 5),
(7, 0),
(6, 3), (7, 0),
(4, 1), (7, 4),
(8, 5),
(8, 4)

lexical analyser maintains the symbol table and to do it efficiently...

symbol table is maintained as a self organised linked list / balanced binary search tree / hash / trie.

Two tasks / problems of the lexical analyser.

i. description of a token (regular expression)
ii. recognition of a token. (automaton)

2. ↓ stream of tokens

Parser / Syntax analyser

↓ Syntax tree / parse tree

* combines tokens to form a valid syntax.
uses grammar $G(N, T, P, S)$.

$E \to (E)$
1. $E \to E + E$
2. $E \to E - E$
3. $E \to E * E$
4. $E \to E / E$
5. $E \to id$

$N = \{E\}$

$T = \{+, -, *, /, id\}$

$S = \{E\}$

grammar for an infix expression (arithmetic)

ex. $a + b * c$.

$id + id * id$.

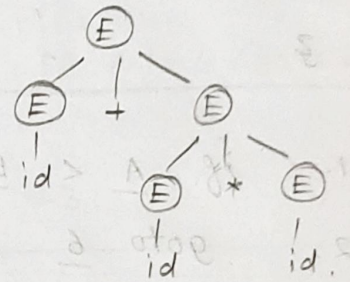
$E \overset{1}{\Rightarrow} \underline{E} + E$

$\overset{5}{\Rightarrow} id + \underline{E}$

$\overset{3}{\Rightarrow} id + \underline{E} * E$

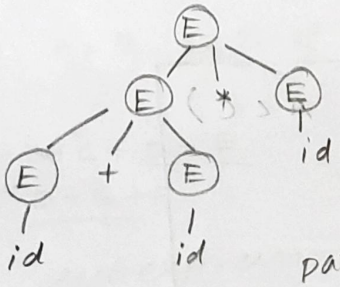
$\overset{5}{\Rightarrow} id + id * \underline{E}$

$\overset{5}{\Rightarrow} id + id * id$.



since $E \overset{*}{\Rightarrow} id + id * id$, $id + id * id$ is a valid infix expression.

(or)



parse tree

Derivation can be top-down / bottom-up.

3. Syntax tree

Intermediate code Generator

b/w HLL & LLL

an intermediate code is Three Address Code (a max. of 3 references)

i. A = B op C
ii. A = op B
iii. goto C
iv. if A rel_op B goto L1

```
while ( A < B)
{
    A = A+B
}
```

1. if A < B goto 3
2. goto 6
3. T1 = A + B
4. A = T1
5. goto 1
6.

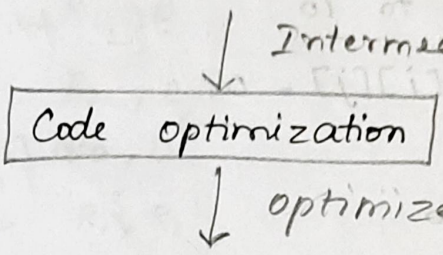
```
while ( A < B and A < C)
{
    A = A+B
}
```

1. if A < B goto 3
2. goto 8
3. if A < C goto 5
4. goto 8
5. T1 = A+B
6. A = T1
7. goto 1
8.

90-10 rule

90% of execution time is spent on 10% of code.

4.



eg. ① if A = B goto L2
goto L3
L2 :

1	CMP
1	JZ L2
0.5	JMP L3
* 2.5	

→

if A != B goto L3
L3 :

1	CMP
1	JNZ L3
2	∴ optimized.

* assuming A = B 50 % of time.

② A = B+C+D+E
B = D+E+F

→

T1 = D+E
A = B+C+T1
B = T1+F

common subexpression

Optimize:

```
I = 1 ;
while   I <= 100   do
{
        a[I] = 0;
        I = I+1;
}
```

→

Technique:
Loop unrolling.

```
while  I <= 50 do
{  a[I] = 0;
   a[I] = 0;
   I++;
   I++;
}
```

Loop jamming

```
for i=1 to 10
    for j=1 to 10.
        a[i][j] = 0
```

→

```
for i=1 to 10
{   for j=1 to 10
        a[i][j]=0
    a[i][i] = 1.
}
```

```
for i=1 to 10
    a[i][i] = 1
```

other techniques:
index variable elimination.
using cheap operations (+ cause adder is directly available as hardware)

↓ optimized code

Code generation

5.

$X = (A+B) * (C-D)$

Basic instruction format

opcode	operation

```
ADD  R1 , A , B
SUB  R2 , C , D
MUL  X , R1 , R2
```

3 address instruction (General Register organisation)

```
MOV  R1 , A
ADD  R1 , B
MOV  R2 , C
SUB  R2 , D
MUL  R1 , R2
MOV  X , R1
```

2 address instruction.

LoaD Accumulator

```
LDA  A
ADD  B
```

STore Accumulator.

```
STA  T
LDA  C
SUB  D
MUL  T
STA  X
```

1 address instruction. (using accumulator). AC is one of the operands and holds the result after. (single register organisation)

```
PUSH  A
PUSH  B
ADD
PUSH  C
PUSH  D
SUB
MUL
POP   X
```

0 address instruction. (stack organisation)

Three address code ⇒ 1 address instruction.

A = B OP C ⟹

```
LDA  B
OP   C
STA  A
```

A = OP B ⟹

```
LDA  B
OP
STA  A
```

if A >= B goto L1 ⟹

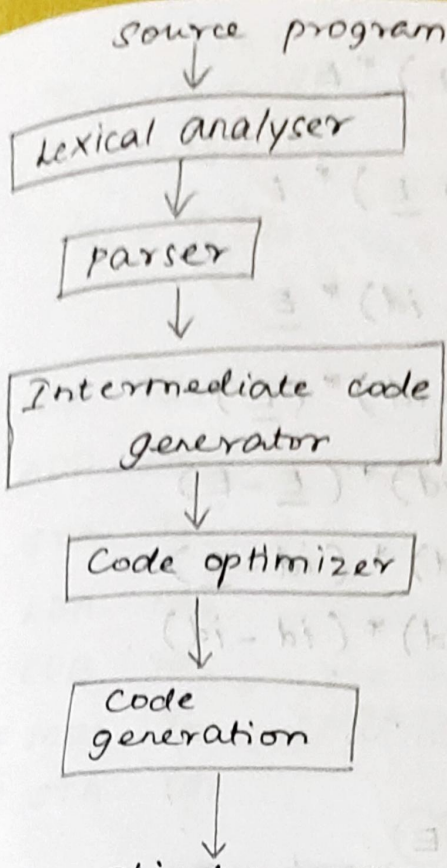
```
LDA  A
CMP  B
JNC  L1
jump no carry
```

goto L1 ⟹

```
JMP  L1
```

Syntax Directed Translation Scheme (SDTS)

Translated to code during Syntax analyzation.



$X = (A + B) * (C - D)$

G1

grammer for assignment operation.

start symbol

A → id = E

E → (E)

E → E+E

E → E-E

E → E*E

E → E/E

E → id.

i. 'id1', '=', '(', 'id2', '+', 'id3', ')', '*', '(', 'id4', '-', 'id5', ')'

ii. $A \Rightarrow id = \underline{E}$

$\Rightarrow id = \underline{E} * E$

$\Rightarrow id = (\underline{E}) * E$

$\Rightarrow$ id = (E + E) * E

$\Rightarrow$ id = (id + E) * E

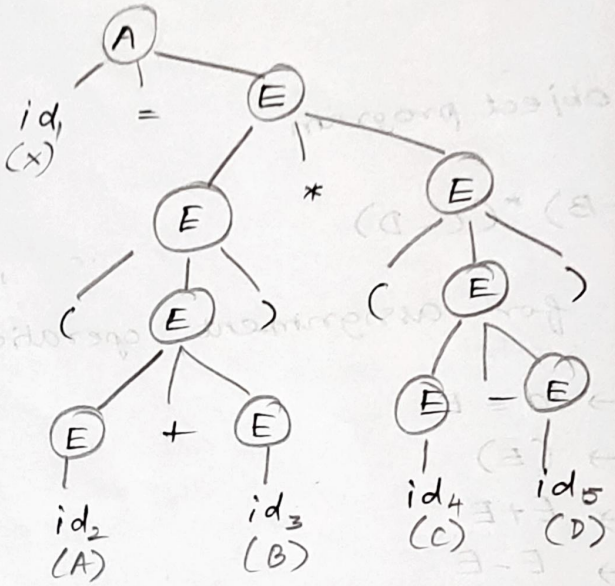
$\Rightarrow$ id = (id + id) * E

$\Rightarrow$ id = (id + id) * (E)

$\Rightarrow$ id = (id + id) * (E - E)

$\Rightarrow$ id = (id + id) * (id - E)

$\Rightarrow$ id = (id + id) * (id - id)



iii.

$T_1 = id_2 + id_3$

$T_2 = id_4 - id_5$

$T_3 = T_1 * T_2$

$id_1 = T_3$

iv. $T_1 = id_2 + id_3$ → instruction 1

$T_2 = id_4 - id_5$

$id_1 = T_1 * T_2$

v.

```
1 LDA  id2
2 ADD  id3
3 STA  T1
4 LDA  id4
5 SUB  id5
6 STA  T2
7 LDA  T1
8 MUL  T2
9 STA  id1
```

instruction 1

redundant LDA & STA

$S = P * N * R / 100$

To G, add

i. 'id1', '=', 'id2', '*', 'id3', '*', 'id4', '/', '1', '0', '0'

$E \rightarrow I$

$I \rightarrow I.digit \mid digit$

$digit \rightarrow 0|1|2|3 \cdots |9$

Grammar for integer const, I

ii. $A \Rightarrow id_1 = \underline{E}$

$\Rightarrow id_1 = \underline{E} * E$

$\Rightarrow id_1 = id_2 * \underline{E}$

$\Rightarrow id_1 = id_2 * \underline{E} * E$

$\Rightarrow id_1 = id_2 * id_3 * \underline{E}$

$\Rightarrow id_1 = id_2 * id_3 * \underline{E} / E$

$\Rightarrow id_1 = id_2 * id_3 * id_4 / \underline{E}$

$\Rightarrow id_1 = id_2 * id_3 * id_4 / \underline{I}$ ← digit

$\Rightarrow id_1 = id_2 * id_3 * id_4 / \underline{I}.d$

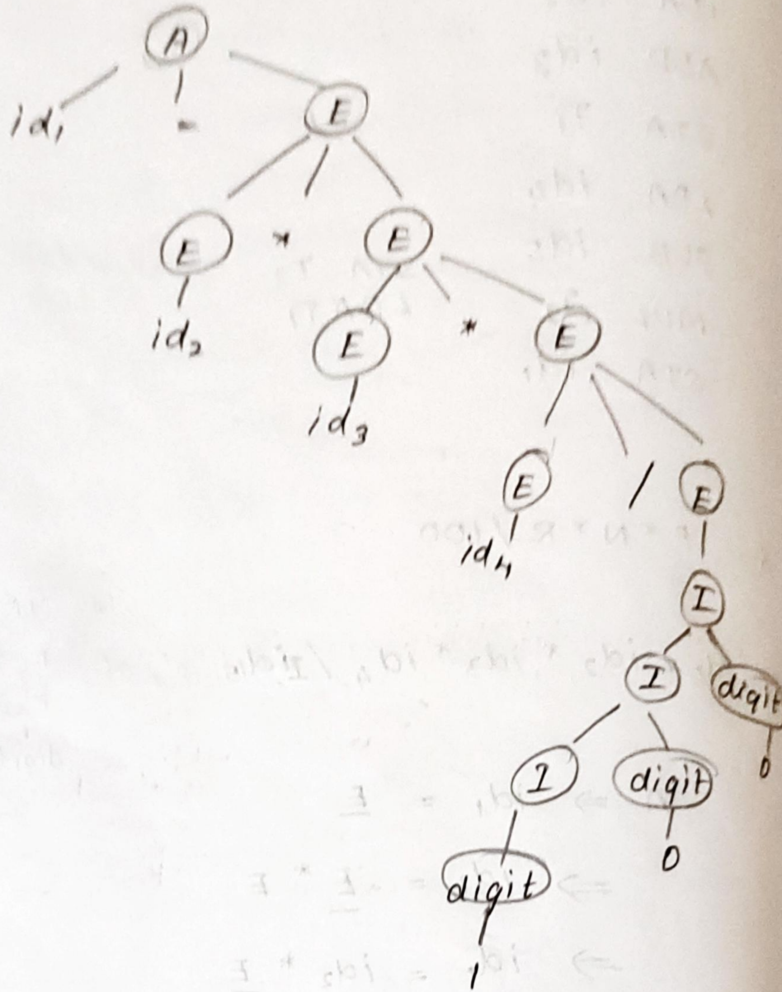
$\underline{I}.dd$

$\Rightarrow \underline{d}dd$

$\Rightarrow id_1 = id_2 * id_3 * id_4 / I\underline{d}d$

$\Rightarrow$ " $/ 10\underline{d}$

$\Rightarrow id_1 = id_2 * id_3 * id_4 / 100$



iii.

$T_1 = P * N$

$T_2 = T_1 * R$

$T_3 = T_2 / 100$

$S = T_3$

if then else grammar.

condition, statement, statement

$S \rightarrow i\ c\ t\ S\ e\ S$

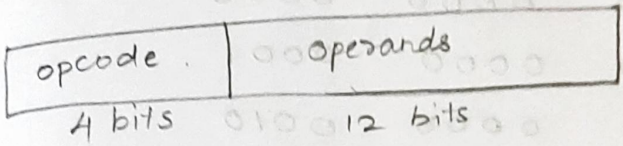
if then else

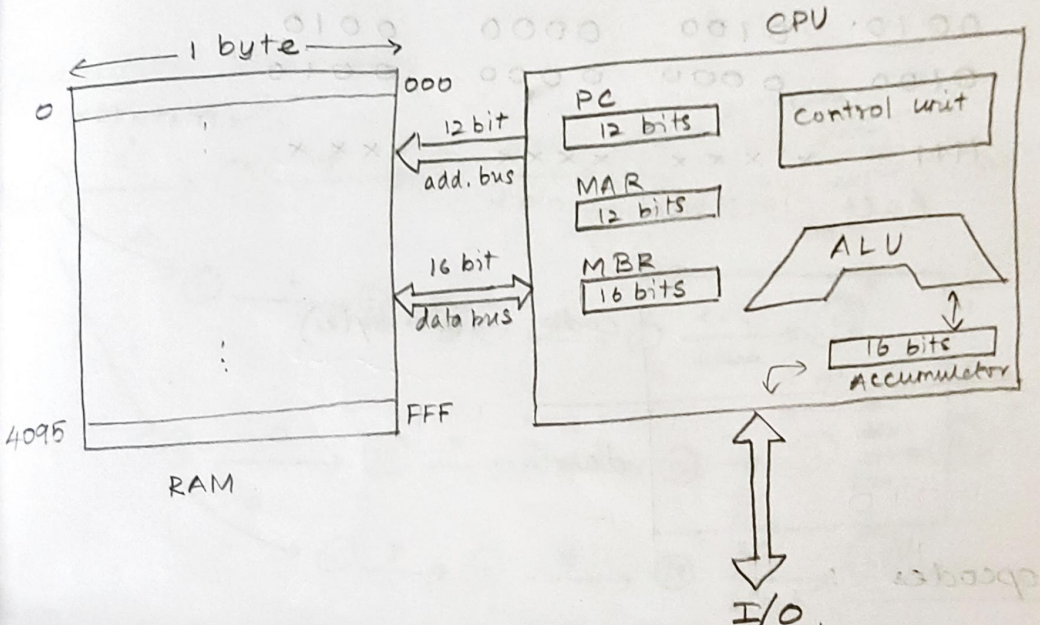
$e \rightarrow id\ relop\ id$.

$S \rightarrow A$

if $a > b$ then $a = a+2$ else $b = b*2$. H.W.

system with 16 bit instruction






```
int a, b, c ; // make entries in S.T.
Read a, b ; // read from keyboard.
c = a+b ;
print c ;
```

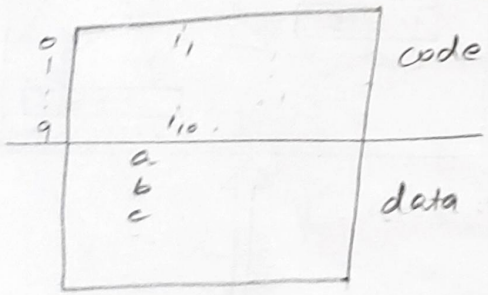
Symbol Table

name	type	address (in hexadec)
a	int	400
b	int	401
c	int	402

Keyboard

1	0000	0000	0000	0001
2	0001	0100	0000	0000
3	0000	0000	0000	0001
4	0001	0100	0000	0001
5	0010	0100	0000	0000
6	0011	0100	0000	0001
7	0001	0100	0000	0010
8	0010	0100	0000	0010
9	0100	0000	0000	0010
				monitor.
10	1111	xxxx	xxxx	xxxx

halt instruction



(20 bytes)

opcodes :

0000 - Fetch a word from I/P device specified

0001 - Transfer AC to memory address specified.

0010 - Load the AC from the " " ".

0011 - Add AC to memory specified
0100 - transfer AC to O/P device.

Lexical Analyzer :

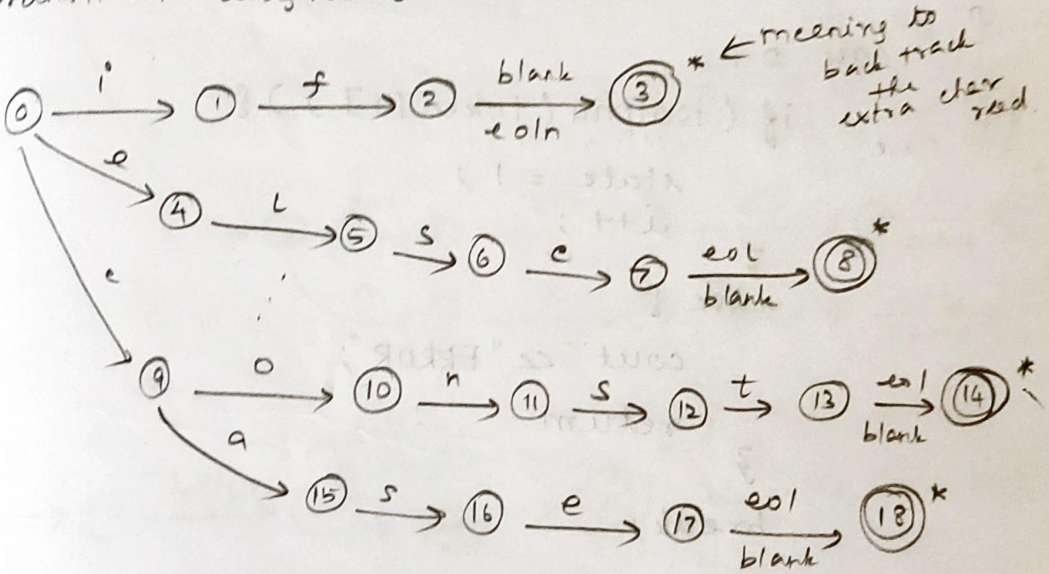
an effort

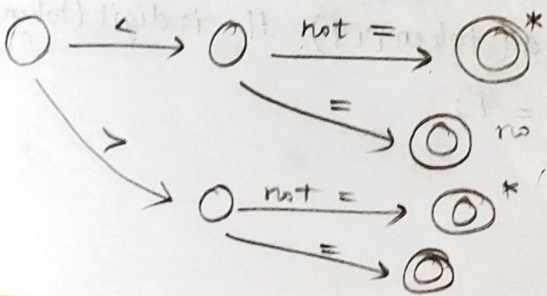
Assume its following.

i. keywords : if, else, break, volatile, void, const, case.

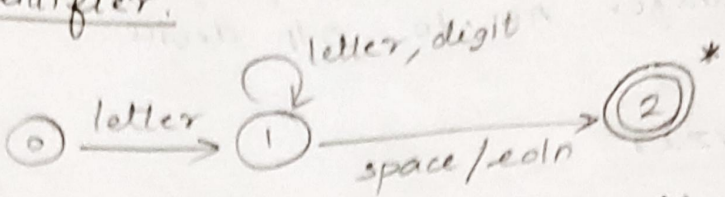
ii. relop : <
<=
>
>=
!=
==

Transition diagrams.





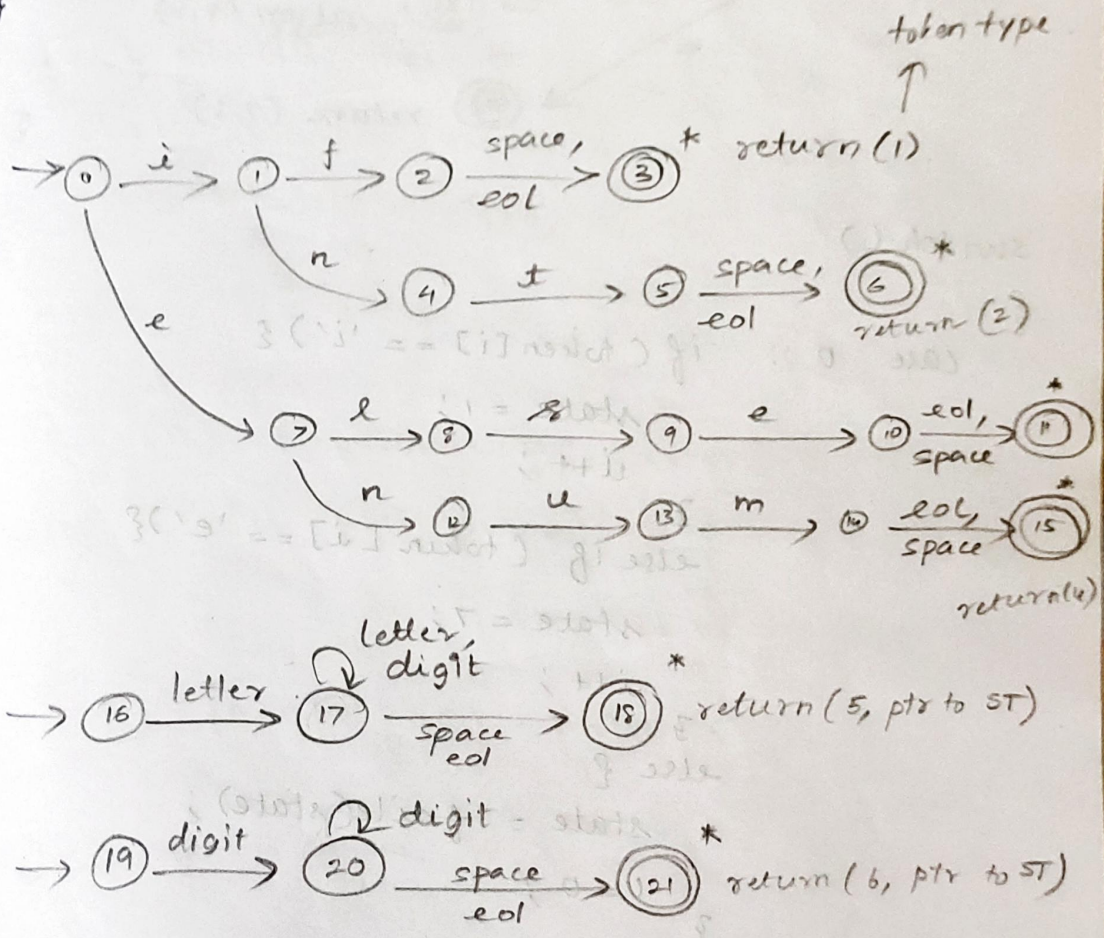
Identifier.

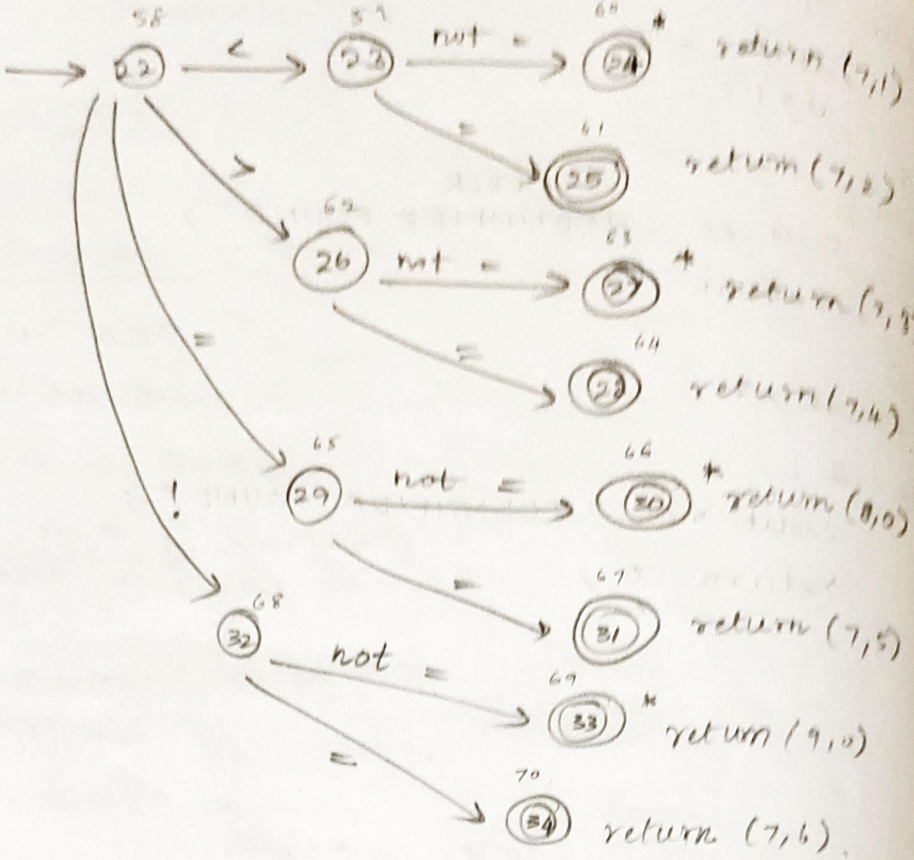


Program to read an identifier

```
int state, i;
char token[50];
cin >> token;

state = 0;
i = 0.

switch (state)

while(1) {
switch ( state)
{
    case 0:
        if ( isalpha ( token[i] ) ) {
            state = 1;
            i++;
        }
        else {
            cout << "ERROR";
            return ;
        }
        break;

    case 1:
        if ( isalpha ( token[i] ) || isdigit (token[i]))
            state = 1;
            i++;
    }
```

```
else if ( token[i] == ' ' || token[i] == '\0' ){
        state = 2 ;
        i++ ;
    }
    else {
        cout << " IDENTIFIER FOUND " ;   (ERROR)
        return ;
    }
    break ;
case 2 :
    cout << " IDENTIFIER FOUND " ;
    return 0 ;                    // retract if there are more tokens
}
}
```






```
switch ()
{
   case 0 :  if (token[i] == 'i') {
                state = 1;
                i++;
             }
             else if (token[i] == 'e'){
                state = 7;
                i++;
             }
             else {
                state = fail (state);
                i = 0;
             }
             break;
case 1 :
   ...
```

```
int fail ( int s)
{  if (s >= 22) {
        cout << "NOT a TOKEN ";
        return ;
        exit()
   }
   else if ( s >= 19) {
        return 22 ;
   }
   else if (s >= 16) {
        return 19 ;
   }
   else {
        return 16;
   }
}
```

Regular expression over a language L.

Regular expression is a finite tool that can represent a finite/infinite language.

i. ϵ is a Regular expression representing $\{\epsilon\}$.

ii. for $a \in L$, 'a' is a regular expression representing $\{a\}$.

iii. if R and S are regular expressions representing L_R and L_S respectively.

a) $R+S$ is a regex, denotes $L_R \cup L_S$

b) $R.S$ is a regex, denotes $L_R . L_S$

c) R^* is a regex, denotes L_R^*.

0 or more 'a's

Find language of a^*.

→ 'a' is a reg ex. representing $L_1 = \{a\}$.

→ a^* is a reg ex. representing L_1^*

$$L_1^* = \bigcup_{i=0}^{\infty} L_1$$

$$= \{\epsilon\} \cup \{a\} \cup \{aa\} \cup \{aaa\} \cup \ldots$$

$$= \{\epsilon, a, aa, aaa, aaaa, \ldots\}$$

Priority note: (* , . , /).

$(a/b)^*$

→ a is a regex representing $L_1 = \{a\}$

→ b is " " $L_2 = \{b\}$

→ a/b is " " " $L_3 = L_1 \cup L_2$
$= \{a, b\}$.

→ $(a/b)^*$ is " " " L_3^*

$$L_3^* = \bigcup_{k=0}^{\infty} L_3^k$$

$= \{\epsilon\} \cup \{a, b\} \cup \{aa, ab, ba, bb\} \cup \ldots$

$= \{\epsilon, a, b, aa, ab, ba, bb, \ldots\}$.

$a(a/b)^*$?

$L = \{a\}.\{\epsilon, a, b, aa, ab, ba, bb, \ldots\}$

$= \{a, aa, ab, aaa, aab, aba, abb, \ldots\}$.

Letter = a | b | c | ... | z

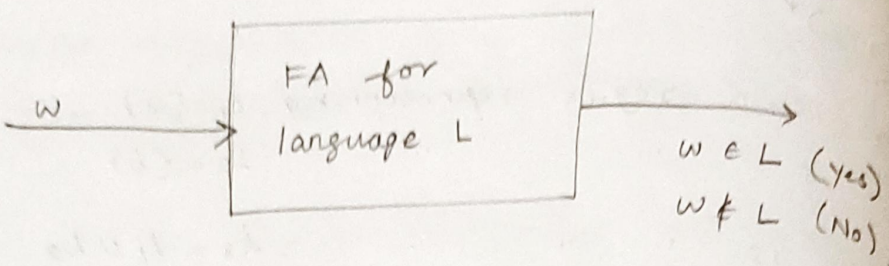
digit = 0 | 1 | 2 | ... | 9

id = letter . (letter | digit)*

const = digit . digit*

keywords = if | else | switch | case .

Finite Automaton



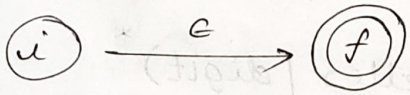
i. Non Deterministic Finite-state Automaton
   → allow ϵ transitions.
   → allow many transition from a state on taking the same input symbol.

ii. Deterministic Finite-state Automaton.
   → one and only one path for a string

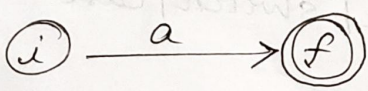
Regular expression to NFA :

Input : RegEx representing Language L over Σ
Output : NDFA for the language L.

1. For RegEx, ϵ.

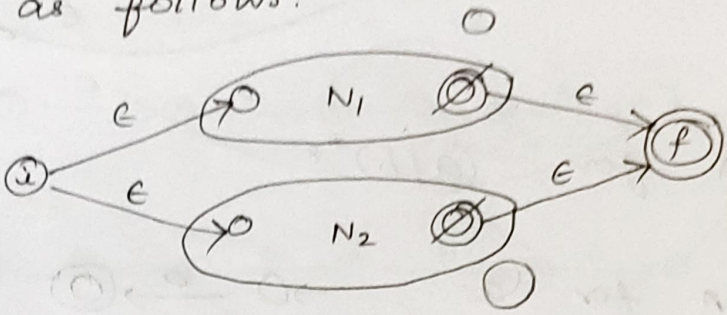


2. For RegEx 'a' over Σ.

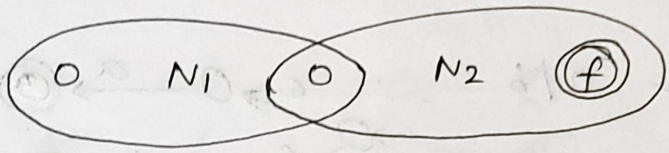


3. if there are regular expressions R_1 and R_2 representing automata N_1 and N_2 respectively.

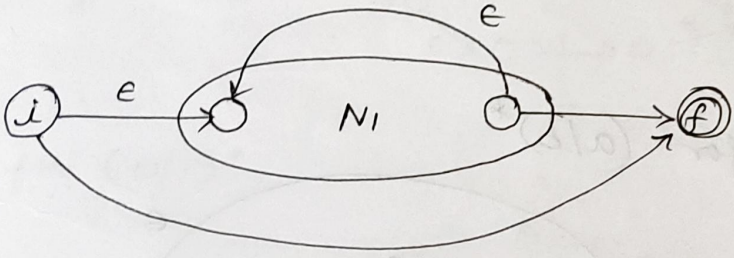
a) For $R_1 | R_2$, NFA is constructed as follows.



b) For $R_1 . R_2$

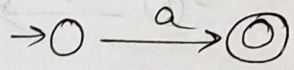


c) For R_1^*

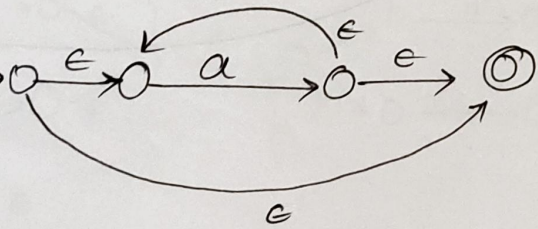


Consider aa^*

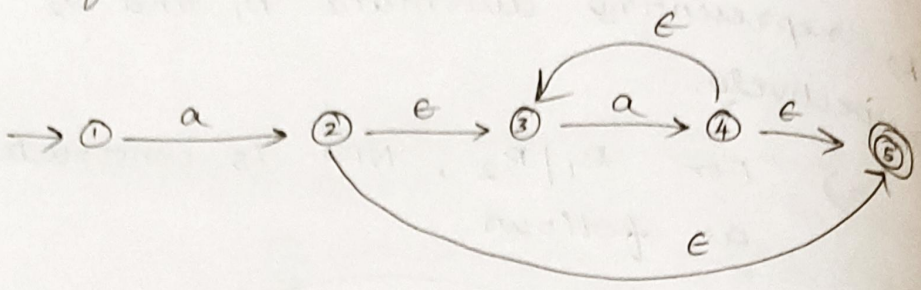
NFA for a



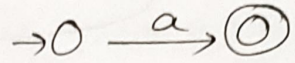
NFA for a^*

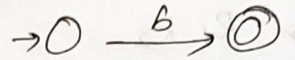


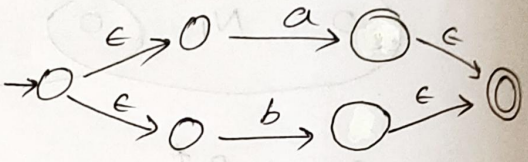
NFA for aa^*



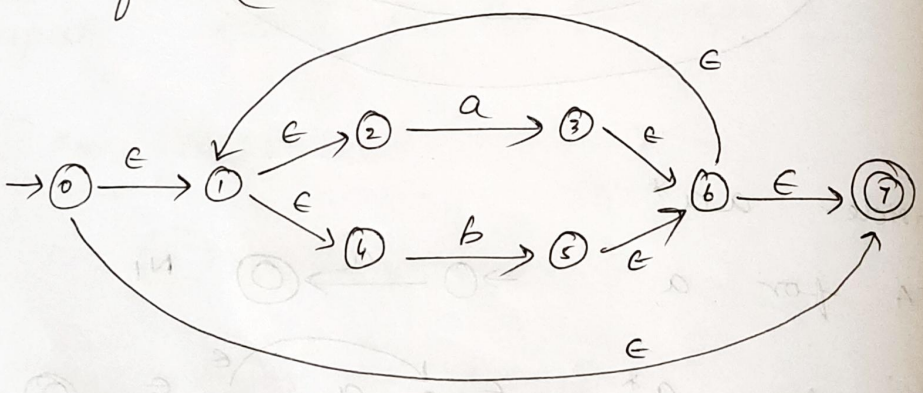
NFA for $(a/b)^*$?

NFA for a 

" for b 

for a/b 

NFA for $(a/b)^*$

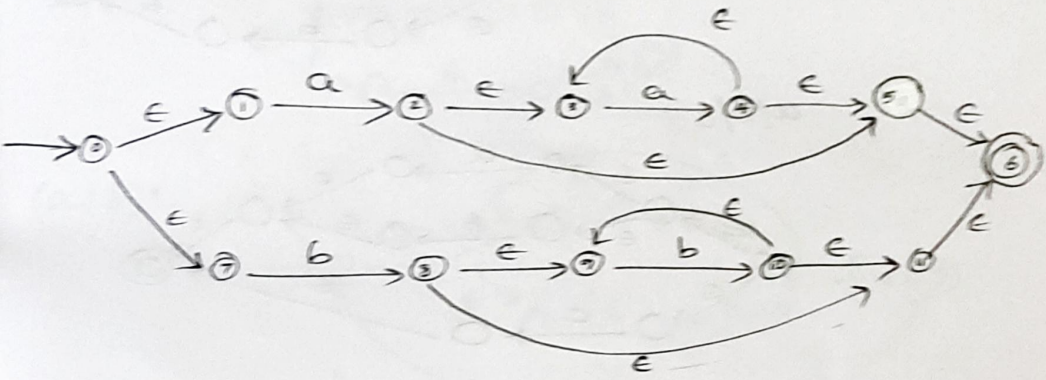


i. aa^*/bb^*

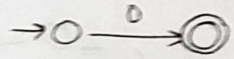
ii. $(0/1)^*(0/11)$

iii. $(a/b/c/d)^*$

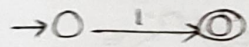
i.



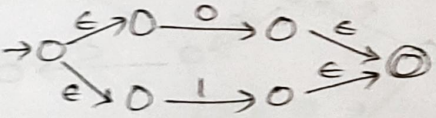
ii. NFA for 0



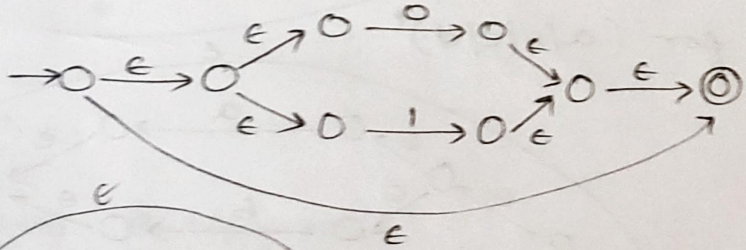
" for 1

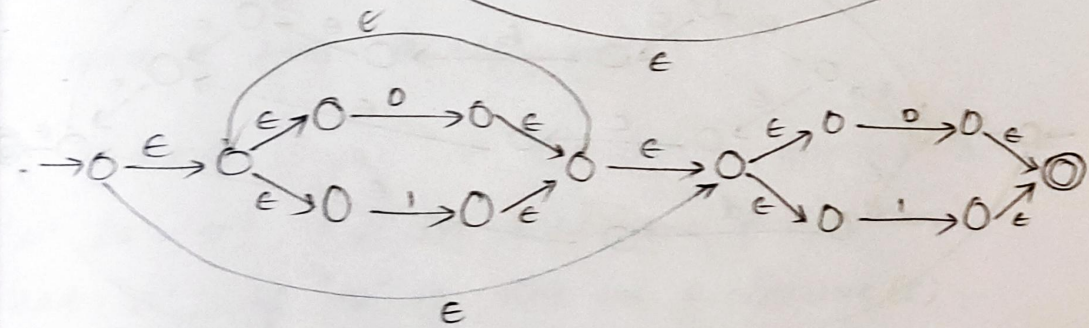


" for 0/1



NFA for $(0/1)^*$

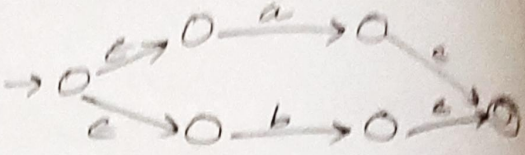




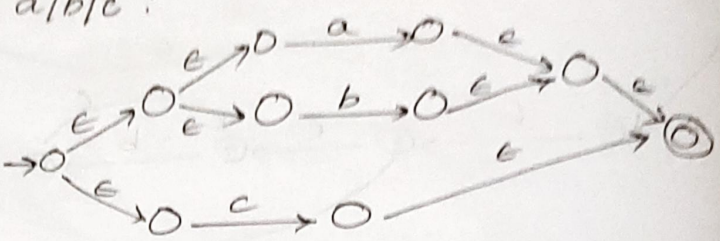
iii. NFA for a :

b :

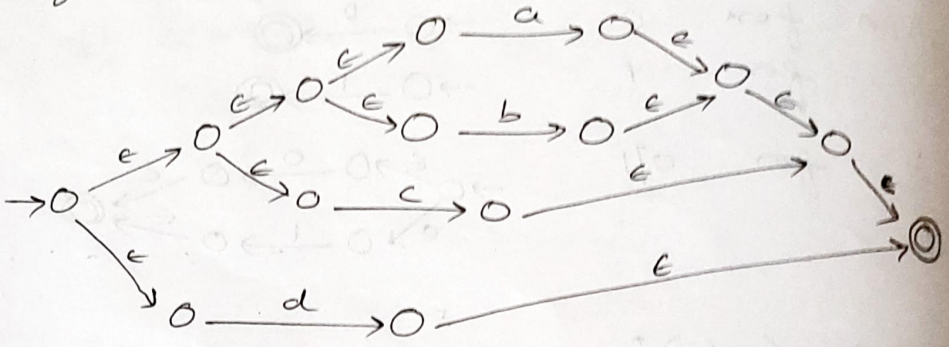
a/b :



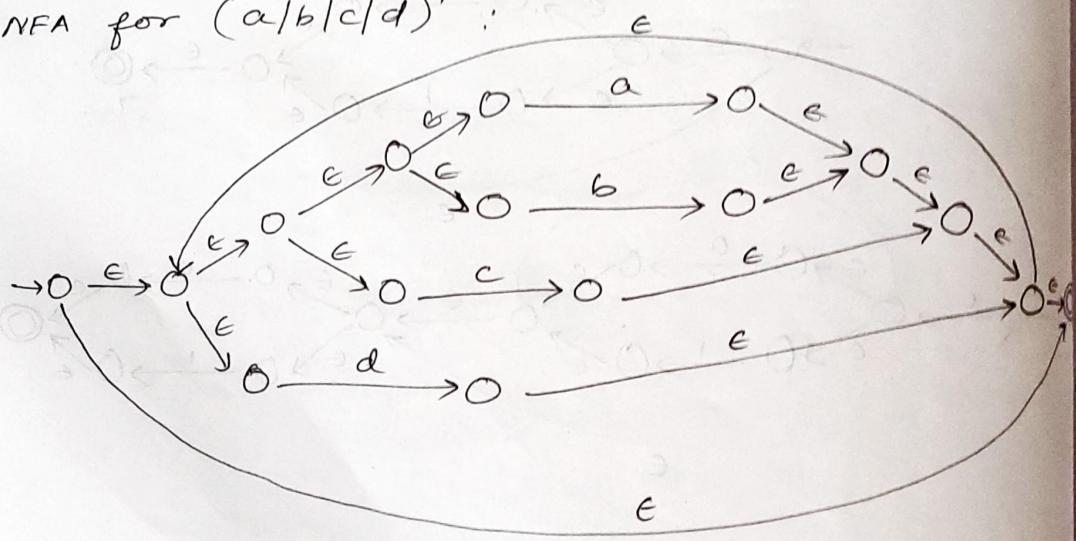
a/b/c :



NFA for a/b/c/d :

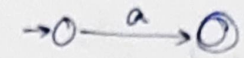


NFA for (a/b/c/d)* :

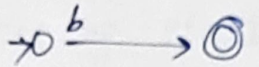


$(a|b)^*abb$

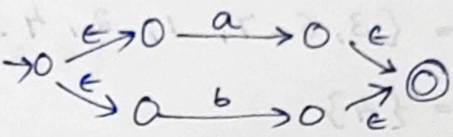
a :



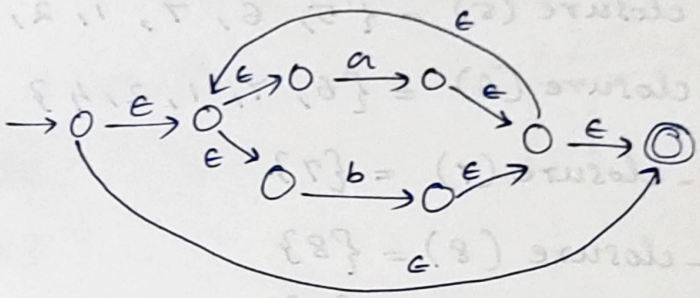
b :



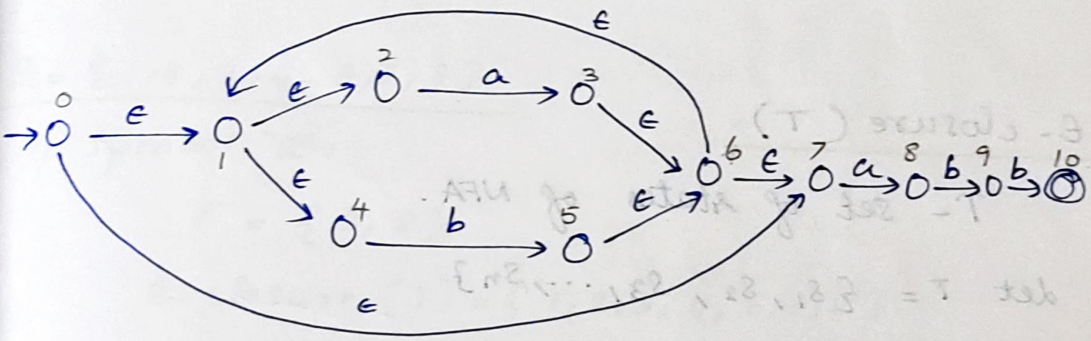
a|b :



$(a|b)^*$:



$(a|b)^*abb$:



<u>ϵ closure (s)</u> :

1. add s into ϵ closure (s).
2. If there is an ϵ transition ~~from 's' to 't'~~ from 'u' to 'v', ('u' in ϵ closure of s) then add 'v', if 'v' is not in ϵ-closure (s).
3. repeat step 2 till no more states can be added.

ϵ-closure (0) = {0, 1, 7, 2, 4}

ϵ-closure (1) = {1, 2, 4}

ϵ-closure (2) = {2}

ϵ-closure (3) = {3, 6, 7, 1, 2, 4}

ϵ-closure (4) = {4}

ϵ-closure (5) = {5, 6, 7, 1, 2, 4}

ϵ-closure (6) = {6, 7, 1, 2, 4}

ϵ-closure (7) = {7}

ϵ-closure (8) = {8}

ϵ-closure (9) = {9}

ϵ-closure (10) = {10}

ϵ-closure (T)

T- set of states of NFA.

let $T = \{s_1, s_2, s_3, \cdots, s_n\}$

ϵ-closure (T) = ϵ-closure (s_1) $\cup$
ϵ-closure (s_2) $\cup$
$\vdots$
ϵ-closure (s_n).

ϵ-closure ({1, 9}) = {1, 2, 4, 9}

Consider A = {0,1,2,4,7} = ε-closure(0). ← The initial of DFA

(Initial of NFA)

i/p symbol 'a'.

T = set of states that can be reached from A through 'a'.

= {3,8}

ε-closure(T) = {1,2,4,3,6,7,8} new state

= B.

'a' transitions	'b'
2→3	4→5
7→8	8→9
	9→10

i/p symbol 'b':

T = {5}

ε-closure(T) = {1,2,4,5,6,7}

= C.

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

i/p symbol 'a':

T = {3,8}

ε-closure(T) = {1,2,3,4,6,7,8}

= B.

i/p symbol 'b':

T = {5,9}

ε-closure(T) = {1,2,4,5,6,7,9}

= D.

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

i/p symbol 'a':

T = {3,8}

ε(T) = {1,2,3,4,6,7,8} = B

i/p symbol 'b':

$T = \{5\}$

$\bar{e}(T) = \{1, 2, 4, 5, 6, 7\}$

$= C.$

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

i/p symbol 'a':

$T = \{3, 8\}$

$\bar{e}(T) = \{1, 2, 3, 4, 6, 7, 8\}$

$= B.$

i/p symbol 'b':

$T = \{5, 10\}$

$\bar{e}(T) = \{1, 2, 4, 5, 6, 7, 10\}$

$= E.$

$\therefore$ final.

$E = \{1, 2, 4, 5, 6, 7, (10)\}$ final state from NFA

i/p symbol 'a':

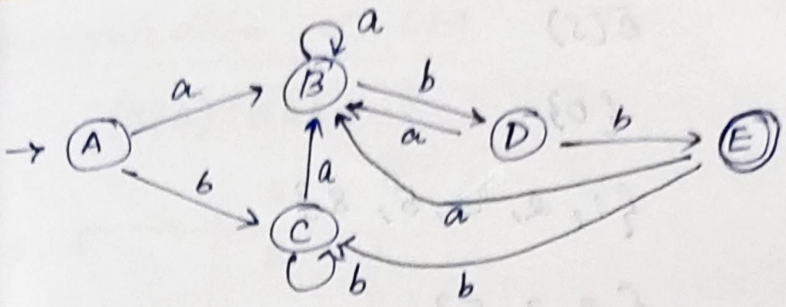
$T = \{3, 8\}$

$\bar{e}(T) = B.$

i/p symbol 'b':

$T = \{5\}$

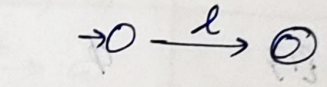
$\bar{e}(T) = C.$



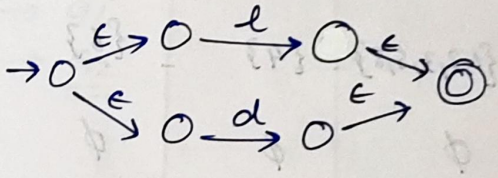
		a	b
→	A	B	C
	B	B	D
	C	B	C
	D	B	E
*	E	B	C

Construct DFA for $l.(l/d)^*$

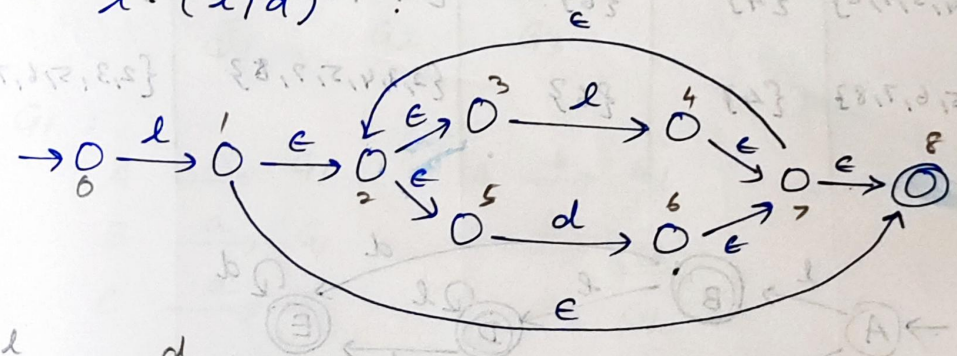
NFA for l :



l/d :



$l.(l/d)^*$:



l

0 → 1

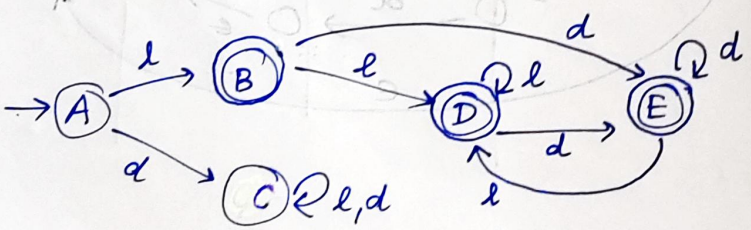
3 → 4

d

5 → 6.

s	$\bar{e}(s)$
0	{0}
1	{1, 2, 3, 5, 8}
2	{2, 3, 5}
3	{3}
4	{4, 2, 3, 5, 7, 8}
5	{5}
6	{6, 2, 3, 5, 7, 8}
7	{2, 3, 5, 7, 8}
8	{8}

	T		$\bar{e}(T)$	
	l	d	l	d
A = {0}	{1}	ϕ	{1,2,3,5,8} B	ϕ
*B = {1,2,3,5,8}	{4}	{6}	{2,3,4,5,7,8} D	{2,3,5,6,7,
C = ϕ	ϕ	ϕ	ϕ	ϕ
*D = {2,3,4,5,7,8}	{4}	{6}	{2,3,4,5,7,8} D	{2,3,5,6,7,8
*E = {2,3,5,6,7,8}	{4}	{6}	{2,3,4,5,7,8} D	{2,3,5,6,7,



Minimization of DFA.

$(a/b)^*abb$.

		a	b
→	A	B	C
	B	B	D
	C	B	C
	D	B	E
*	E	B	C

$\pi_0 = \{ (\text{Non final}) \ (\text{Final}) \}$

$= \{ (ABCD) \ (E) \}$
$\quad\quad G_1 \quad\quad G_2$

G_1 :

$A \xrightarrow{a} G_1$ $\quad$ $A \xrightarrow{b} G_1$

$B \xrightarrow{a} G_1$ $\quad$ $B \xrightarrow{b} G_1$

$C \xrightarrow{a} G_1$ $\quad$ $C \xrightarrow{b} G_1$

$D \xrightarrow{a} G_1$ $\quad$ $D \xrightarrow{b} G_2$

$\pi_1 = \{ (ABC) \ (D) \ (E) \}$
$\quad\quad G_1 \quad G_2 \quad G_3$

G_1 :

$A \xrightarrow{a} G_1$ $\quad$ $A \xrightarrow{b} G_1$

$B \xrightarrow{a} G_1$ $\quad$ $B \xrightarrow{b} G_2$

$C \xrightarrow{a} G_1$ $\quad$ $C \xrightarrow{b} G_1$

$\pi_2 = \{ (AC) \ (B) \ (D) \ (E) \}$

G_1 :

$A \xrightarrow{a} G_2$ $\quad$ $A \xrightarrow{b} G_1$

$B \xrightarrow{a} G_2$ $\quad$ $B \xrightarrow{b} G_1$

$\therefore \pi = \{ (AC) \; (B) \; (D) \; (E) \}$.

A & C are indistinguishable

minimized DFA.

	a	b
A	B	& A
B	B	D
& A	B	& A ← redundent
D	B	E
E	B	& A.

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

minimize DFA of identifier. (L.(L/d)*)

	l	d
→ A	B	C
* B	D	E
C	C	C
* D	D	E
* E	D	E

$\pi_0 = \{ \underset{G_1}{(AC)} \quad \underset{G_2}{(BDE)} \}$

G_1 :

$A \xrightarrow{l} G_2$ $\quad$ $A \xrightarrow{d} G_1$

$C \xrightarrow{l} G_1$ $\quad$ $C \xrightarrow{d} G_1$

$\pi_1 = \{ \underset{G_1}{(A)} \quad \underset{G_2}{(C)} \quad \underset{G_3}{(BDE)} \}$

G_3 :

$B \xrightarrow{l} G_3$ $\quad$ $B \xrightarrow{d} G_3$

$D \xrightarrow{l} G_3$ $\quad$ $D \xrightarrow{d} G_3$

$E \xrightarrow{l} G_3$ $\quad$ $E \xrightarrow{d} G_3$

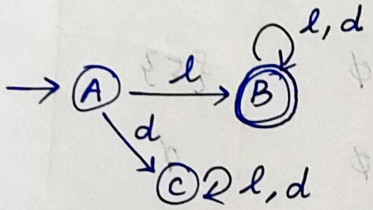
$\therefore \pi = \{ (A) \quad (C) \quad (BDE) \}$.

		l	d
→	A	B	C
*	B	~~D~~ B	~~E~~ B
	C	C	C
*	~~D~~	D	E
*	~~E~~	D	E

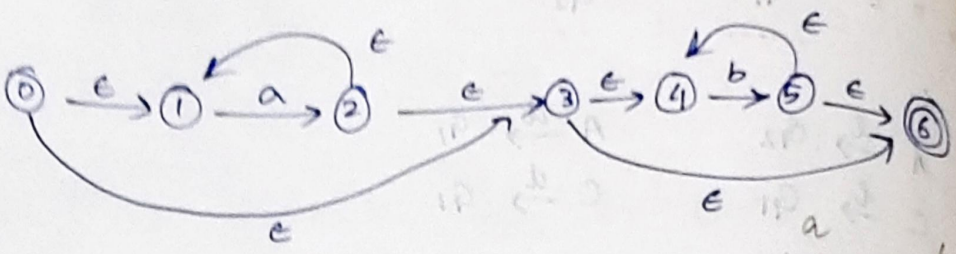
→

		l	d
→	A	B	C
*	B	B	B
	C	C	C

minimized DFA.



Guess DFA for a^*b^*



a
1 → 2

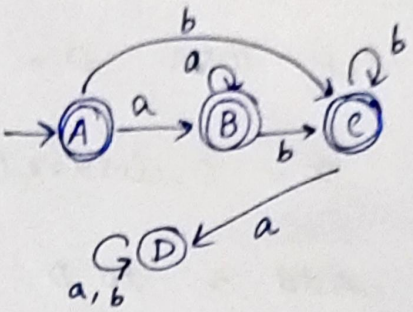
b
4 → 5

s	$\bar{\epsilon}(s)$
0	{0, 1, 3, 4, 6}
1	{1}
2	~~{2, 1}~~ {1, 2, 3, 4, 6}
3	{3, 4, 6}
4	{4}
5	{4, 5, 6}
6	{6}.

	T		$\bar{\epsilon}(T)$	
	a	b	a	b
A = {0, 1, 3, 4, 6}	{2}	{5}	{1, 2, 3, 4, 6} = B	{4, 5, 6} = C
B = {1, 2, 3, 4, 6}	{2}	{5}	B	C
C = {4, 5, 6}	ϕ	{5}	ϕ	C.
D = ϕ	ϕ	ϕ	ϕ	ϕ

DFA

	a	b
→ *A	B	C
*B	B	C
*C	D	C
D	D	D

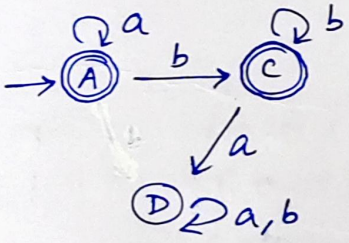


$\pi_0 = \{ (A\ B\ C)\ (D) \}$

$\pi_1 = \{ (AB)\ (C)\ (D) \}$.

minimized DFA.

	a	b
* A	A	C
* C	D	C
D	D	D



Algorithm for converting NFA to DFA.

i. Let DFA $D = \{S_1, S_2, \ldots S_n\}$.

ii. x = ϵ-closure(0) 0 - initial state of NFA.

Add x to D.

unmark x.

iii. While there is an unmarked state 'x' = $\{s_1, s_2, \ldots, s_n\}$ in D. do.

{

for each input symbol 'a' in Σ.

{

T = set of states reacheble from x through 'a'.

y = ϵ-closure (T).

if $y \notin D$ then {

add y to D

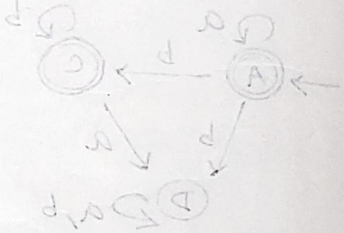
unmark y

}

}

mark x;

}



Algorithm for minimizing DFA.

i. Construct Π = {{(Final states)},{(Non Final states)}}

ii. For each group G of Π do.

{

a) partition the group G such that

two states 's' and 't' are allowed to be in the same group if transition for every i/p symbol should go to the same group of Π.

b) place all the groups in Π_{new}

3 repeat step 2 till $\Pi = \Pi_{new}$.

iii.

lex:

Auxiliary definitions.	i
Translation rules.	ii
Auxilliary function.	iii

i. define regEx.

ii.
```
%%
    pattern    {action}
%%
```

eg: ex1.l

```
letter    [a-z]
digit     [0-9]
id        ({letter}({letter}|{digit})*)

%%
if|then|goto|for    {printf("Keyword"); }
{id}                {printf("Identifier"); }
{digit}+            {printf("Const"); }
<|<=|>|>=           {printf("relop"); }
%%
```

$ lex ex1.l ⟶ lex.yy.c

$ cc lex.yy.c -lfl ⟶ a.out

$ ./a.out

Parser

i. Top Down.
ii. Bottom Up.

Grammar, G = (N, T, P, S)

N → set of non terminals. (A, B, C, ...)
T → set of terminals (a, b, c, d, ...)
P → set of productions (of form $\alpha \to \beta$)
S → start symbol ($S \in N$).

Language of a grammar, G.

$$L(G) = \{ w \mid S \overset{*}{\Rightarrow} w \text{ and } w \in T^* \}$$

Left most derivation.

$S \overset{*}{\underset{lm}{\Rightarrow}} w$ (in every step always replace the left most non terminal).

Right most derivation.

$S \overset{*}{\underset{rm}{\Rightarrow}} w$ (" " " " the right most non terminal).

Left-sentential form.

$S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$

$\alpha \in (NUT)^*$

Regular Grammar.

$\alpha \rightarrow \beta$.

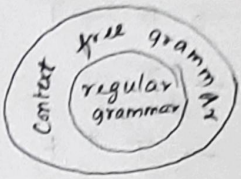
$\alpha \in N$.

β is of the form 'aA' or 'Aa' or 'a'.

$A \rightarrow aA/a$. (a-1).

eg.

Context Free Grammar.



$\alpha \rightarrow \beta$

$\alpha \in N$.

$\beta \in (NUT)^*$

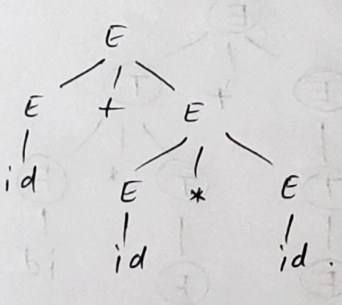
eg. expression grammar.

$E \rightarrow E+E$
$E \rightarrow E*E$
$E \rightarrow E-E$
$E \rightarrow E/E$
$E \rightarrow (E) \mid id$.

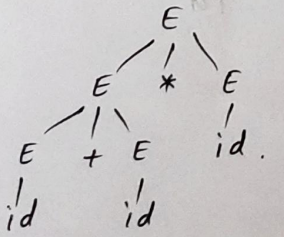
$N = \{E\}$
$T = \{+, -, *, /, (,), id\}$.

derive id + id * id.

$E \Rightarrow E + \underline{E}$
$\Rightarrow \underline{E} + E * E$
$\Rightarrow id + \underline{E} * E$
$\Rightarrow id + id * \underline{E}$
$\Rightarrow id + id * id$



$E \Rightarrow \underline{E} * E$
$\Rightarrow \underline{E} + E * E$
$\Rightarrow id + \underline{E} * E$
$\Rightarrow id + id * \underline{E}$
$\Rightarrow id + id * id$.



Two different derivation trees for same sentence
∴ ambiguous.

introduce priority/precedence

F	id	( )
T	*	/
E	+	-

priority ↑

$E \to E + T$

$E \to E - T$

$E \to T$

$T \to T * F$

$T \to T / F$

$T \to F$

$F \to id$

$F \to (E)$

id + id * id

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

$\underset{lm}{\Rightarrow} \underline{T} + T$

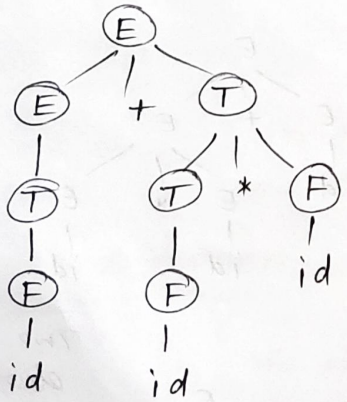
$\underset{lm}{\Rightarrow} \underline{F} + T$

$\underset{lm}{\Rightarrow} id + \underline{T}$

$\underset{lm}{\Rightarrow} id + \underline{T} * F$

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

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



note: right most derivation also results in same tree.

if-then-else grammar.

$S \rightarrow i\ C\ t\ S$

$S \rightarrow i\ C\ t\ S\ e\ S$

$C \rightarrow b$

$S \rightarrow a$.

$N = \{S, C\}$

$T = \{i, t, e, a, b\}$.

ibtibtaea.

$S \Rightarrow i\ \underline{C}\ t\ S\ e\ S$. ①

$\Rightarrow i\ b\ t\ \underline{S}\ e\ S$

$\Rightarrow ibt\ i\underline{C}\ t\ S\ e\ S$

$\Rightarrow ibt\ ibt\ \underline{S}\ e\ S$.

$\Rightarrow ibt\ ibt\ a\ e\ \underline{S}$

$\Rightarrow ibt\ ibt\ a\ e\ a$.

if then.
if then
else.

$S \rightarrow i\ \underline{C}\ t\ S$.

$\Rightarrow ib\ t\ \underline{S}$. ②

$\Rightarrow ib\ t\ i\underline{C}\ t\ S\ e\ S$.

$\Rightarrow ibt\ ibt\ \underline{S}\ eS$

$\Rightarrow ibtib\ t\ a\ e\ \underline{S}$

$\Rightarrow ibtib\ t\ a\ e\ a$.

if then ✓
if then
else.

(else must be paired with the closest unmatched if)

Two different lm derivations
∴ ambiguous.

② is right Now how do we deambiguate the grammar.

$S \rightarrow U \mid M$

$M \rightarrow i\ C\ t\ M\ e\ M \mid a.$

$U \rightarrow i\ C\ t\ M\ e\ U \mid i\ C\ t\ S.$

$C \rightarrow b.$

ibtibtaea

$S \Rightarrow U$

$U \Rightarrow i\ \underline{C}\ t\ M\ e\ U.$

$\Rightarrow i\ b\ t\ \underline{M}\ e\ U$? SVS

$\Rightarrow i\ b\ t\ i\ C\ t\ M\ e\ M\ e\ U.$

all possible paranthesis pairs.

$S \rightarrow (S)S$

$S \rightarrow \epsilon.$

Bottom up parser : (There is a reduction. rhs is replaced with lhs).

$S \rightarrow aAcBe.$

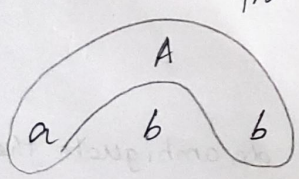
$A \rightarrow Ab \mid b$

$B \rightarrow d.$

abbcde.

The parser thinks, the sentence does not belong to the given grammer.

but there is a derivation actually.



c d e.

Handle:

substring of a given sentence and should be right hand side of a production and if it is reduced to LHS of a production, it should lead to start symbol.

The problem with bottom up parser is the difficuelty to identify the handle.

Handle pruning Technique

Right Most Derivation in reverse.

eg. rmd of $id + id * id$.

$$E \underset{rm}{\Rightarrow} E + \underline{E}.$$

$$\underset{rm}{\Rightarrow} E + E * \underline{E}.$$

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

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

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

right sentential form	Handle	Production used.
$id_1 + id_2 * id_3$.	id_1	$E \to id$.
$E + id_2 * id_3$	id_2	$E \to id$.
$E + E * id_3$	id_3	$E \to id$.
$E + E * E$	$E * E$	$E \to E * E$
$E + E$	$E + E$	$E \to E + E$
E		

Operator Precedence Parser.

Method - 1.

Three relationships defined b/w terminals

$<\cdot$, $\cdot>$, $\doteq$

Terminals are operators.

i. if θ_1 and θ_2 are operators and θ_1 has higher precedence than θ_2 then set.

$\theta_1 \cdot> \theta_2$ and $\theta_2 <\cdot \theta_1$

ii. if θ_1 and θ_2 are equal in precedence,

left association. - set $\theta_1 \cdot> \theta_2$ and $\theta_2 \cdot> \theta_1$.

right association - set $\theta_1 <\cdot \theta_2$ and $\theta_2 <\cdot \theta_1$.

iii. $id \cdot> \theta$ $\quad \theta <\cdot id$. $\quad (\doteq)$.

$\$ <\cdot \theta$ $\quad \theta \cdot> \$$.

$\$$ ← delimiter

$E \to E + E$

$E \to E - E$

$E \to E * E$

$E \to E / E$

$E \to id$

	+	-	*	/	id	$
+	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$
-	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$
*	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$<\cdot$	$\cdot>$
/	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$<\cdot$	$\cdot>$
id	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$		$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	

Stack	relation	input buffer
$ → top most terminal.	<·	id + id * id $
$ <· id	·>	+ id * id $
$ E	<·	+ id * id $
$ <· E +	<·	id * id $
$ <· E + <· id (handle)	·>	* id $
$ <· E + E	<·	* id $
$ <· E + <· E *	<·	id $
$ <· E + <· E * <· id	·>	$
$ <· E + <· E * E	·>	$

<· or = : shift

·> : handle at top of stack.
string between <· and ·> is the handle.

$ <· E + E	·>	$
$ E	accept.	$

This is called shift reduce parser.

Reduce the foll.

i. id - id + id

ii. id * id * id - id.

stack	relation	input buffer
$	<.	id - id + id $
$ <. id	.>	- id + id $
$ E	<.	- id + id $
$ <. E -	<.	id + id $
$ <. E - <. id	.>	+ id $
$ <. E - E	.>	+ id $
$ E	<.	+ id $
$ <. E +	<.	id $
$ <. E + <. id	.>	$
$ <. E + E	.>	$
$ E	accept	$

ii.

stack	relation	input buffer
$	<.	id * id * id - id $
$ <. id	.>	* id * id - id $
$ E	<.	* id * id - id $
$ <. E *	<.	id * id - id $
$ <. E * <. id	.>	* id - id $
$ <. E * E	.>	* id - id $
$ E	<.	* id - id $
$ <. E *	<.	id - id $
$ <. E * <. id	.>	- id $
$ <. E * E	.>	- id $
$ E	<.	- id $

$ <. E -	<.	id $
$ <. E - <. id	.>	$
$ <. E - E	.>	$
$ E	accept.	$

Method - 2

Using operator precedence grammar.

- operator grammar. (No two consecutive non terminals allowed, No ε productions).
- relations <. .> ≐ are disjoint.

Leading (A)

A - Non Terminal.

1. If there is a production of the form

$A \rightarrow \gamma a \delta$, where

γ is a single non terminal or ε

then 'a' is in Leading (A).

''' LEADING (A) is also sometimes referred to as FIRST OF (A) '''

2. If there is a production of the form.

$A \rightarrow B \alpha$,

all LEADING (B) are in LEADING (A).

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id.$

	LEADING
F	(, id
T	*, (, id
E	+, *, (, id

TRAILING (A)

A - non terminal.

1. If there is a production of the form $A \to \gamma a \beta$, where β is a single non terminal or e then 'a' is in TRAILING (A).

2. If there is a production of the form $A \to \alpha B$ then All trailing (B) are in TRAILING (A).

	TRAILING
F	), id.
T	*,), id
E	+, *,), id.

$S \to aAcBe$
$A \to Ab \mid b$
$B \to d$.

	LEADING	TRAILING
S	a	e.
A	b	b.
B	d.	d.

$S \to (L) \mid a$
$L \to L, S \mid S$

	LEADING	TRAILING
S	'(', 'a'	')', 'a'
L	',', '(', 'a'	',', ')', 'a'

"The method"

1. $a \doteq b$, if there is an RHS of a production of the form $\alpha a \beta b \gamma$, where β is a single non terminal or ϵ.

2. $a <\cdot b$, if there is an RHS of a production of the form $\alpha a A \beta$, $\forall b$ in LEADING(A), $a <\cdot$ LEADING(A).

3. $a \cdot> b$, if there is an RHS of a production of the form $\alpha A a \beta$, $\forall b$ in TRAILING(A), TRAILING(A) $\cdot>$ a

4. $\$ <\cdot$ LEADING(S).
   TRAILING(S) $\cdot>$ $\$$
   S - start symbol.

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$.

	LEADING	TRAILING
E	+ * (id	+ *) id
T	* (id	*) id
F	(id	) id

	+	*	(	)	id	$
+	$\cdot>$	$<\cdot$	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$\cdot>$	$\cdot>$	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
(	$<\cdot$	$<\cdot$	$<\cdot$	$\doteq$	$<\cdot$	
)	$\cdot>$	$\cdot>$		$\cdot>$		$\cdot>$
id	$\cdot>$	$\cdot>$		$\cdot>$		$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$		$<\cdot$	

	a	b	c	d	e	$
a		$<\cdot$	$\doteq$			
b		$\cdot>$	$\cdot>$			
c				$<\cdot$	$\doteq$	
d					$\cdot>$	
e						$\cdot>$
$	$<\cdot$					

	(	)	a	,	$
(	$<\cdot$	$\doteq$	$<\cdot$	$<\cdot$	
)		$\cdot>$		$\cdot>$	$\cdot>$
a		$\cdot>$		$\cdot>$	$\cdot>$
,	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$	
$	$<\cdot$		$<\cdot$		

Top - down parser :

Elimination of left recursion :

$$A \rightarrow A\alpha \mid \beta$$

$$\Downarrow$$

$$A \rightarrow \beta A'$$

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

In general,

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$$

$$\Downarrow$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \mid \alpha_m A' \mid \epsilon$$

i. $A \rightarrow Ac \mid Ad \mid e \mid f$.

$$A \rightarrow eA' \mid fA'$$

$$A' \rightarrow cA' \mid dA' \mid \epsilon$$

ii. $S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid e$

$$\Downarrow$$

$$A \rightarrow Ac \mid Aad \mid bd \mid e.$$

$$A \rightarrow eA' \mid bdA'$$

$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

iii. $E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T*F \mid T/F \mid F$

$F \rightarrow (E) \mid id.$

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid -TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid /FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

Left factoring.

$A \rightarrow \alpha\beta \mid \alpha\gamma$

α : the common prefix.

eliminate it using.

$A \rightarrow \alpha A'$

$A' \rightarrow \beta \mid \gamma$

Recursive Descent Parser:

Procedure for every non terminal.

Sample code.

```
char token[50];
int i;
int main() {
    cin >> token;
```

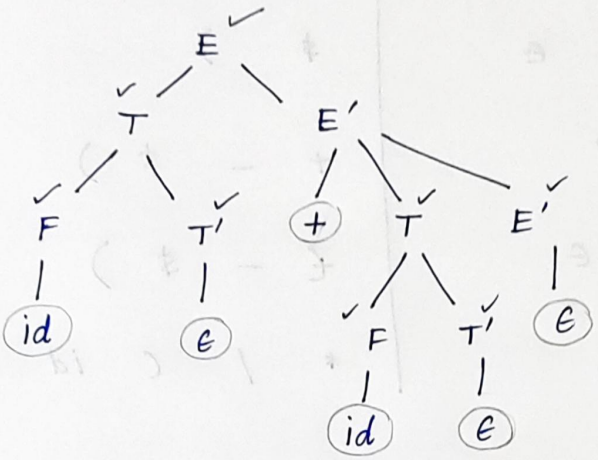
Remember:
grammar should have no left recursion and no common prefix

```
    i = 0;
    E();
    if (i == strlen(token)) cout << "ACCEPTED";
    else cout << "NOT ACCEPTED";
}
void E()
{
    T();
    EPRIME();
}
void EPRIME()
{
    if (token[i] == '+')
    {
        i++;
        T();
        EPRIME();
    }
else if (token[i] == '-')
    {
        i++;
        T();
        EPRIME();
    }
    else
    {
        // E' -> e.
    }
}
void T()
{
    F();
    TPRIME();
}
```

```
void TPRIME()
{
    if (token[i] == '*')
    {
        i++;
        F();
        TPRIME();
    }
    else if (token[i] == '/')
    {
        i++;
        F();
        TPRIME();
    }
    else
    {
        // T' → ∈
    }
}

void F()
{
    if (token[i] == '(')
    {
        i++;
        E();
        if (token[i] == ')')
        {
            i++;
        }
        else
        {
            cout << "ERROR" << i;
            exit(0);
        }
}
```

```
else if ( token[id] == 'i')
{  i++ ;
}
else
{  cout << " ERROR " << i ;
   exit(0);
}
}
```

id + id .



Predictive Parser :

FIRST(X)

1. if X is a terminal then FIRST(X) = {X}
2. if x is a non terminal and there is a production $X \rightarrow a\alpha$, then 'a' is added to FIRST(X).
3. if x is a non terminal and there is a production $X \rightarrow Y_1 Y_2 Y_3 \ldots Y_k$ Y_i – Non terminal FIRST(Y_1) added to FIRST(X). and

if FIRST(Y_1) contains ϵ

FIRST(Y_2) is added to FIRST(X).

if FIRST(Y_2) contains ϵ

FIRST(Y_3) " " "

.

.

.

	FIRST	FOLLOW
E	(id	$)
E'	+ - ϵ	$)
T	(id	+ - $)
T'	* / ϵ	+ - $)
F	(id	* / (id

FOLLOW(A).

A - non terminal

1. $ is added to FOLLOW(s), s — start symbol.

2. if there is a production of the form

$A \rightarrow \alpha B \beta$.

then FIRST(β) (except ϵ) is added to FOLLOW(B).

if FIRST(β) contains ϵ, then add all FOLLOW(A) to FOLLOW(B).

3. If there is a production $A \to \alpha B$, add all FOLLOW(A) to FOLLOW(B).