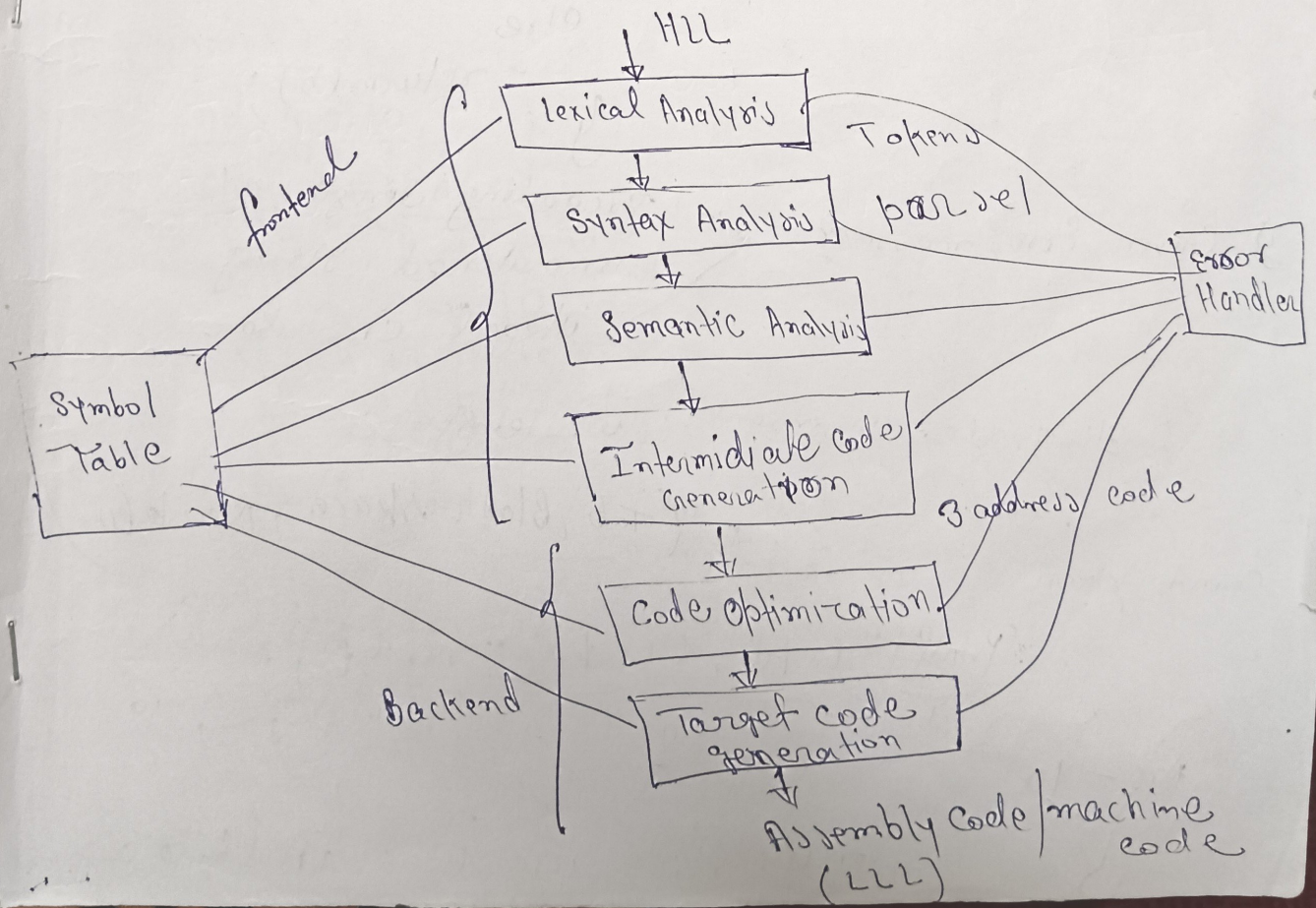
Compiler Design

gate topic

- Lexical analysis
- parsing
- Syntax-directed translation
- Runtime environments.
- Intermediate code generation
- local optimisation
- Data flow analyses : constant propagation
- liveness analysis
- common subexpression elimination

⇒ Phases of Compilier with examples.

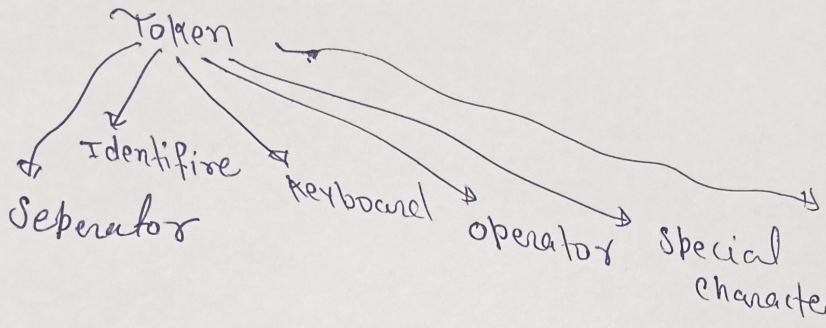
Compiler :- It is the is the convert of language from Higher level language to low level language



Lexical Analysis

Lexical Analysis [Lexer, Tokeniser, Scanner]

1.) Tokenization




```
int main ( )
{
    /* find max of a and b
int a=10, b=20
 of (a>b)
    return (a);
  else
      return (b);
}
```

2) Given Error messages ← Exceeding length
unmatched string
illegal character

Eliminate Comments, white space
(Tab, Blank Space, New line)

Count token

```
Printf("i=%d, &i=%x", i, &i);
```

1 2 — 3 — 4,5 6 7 8 9 10

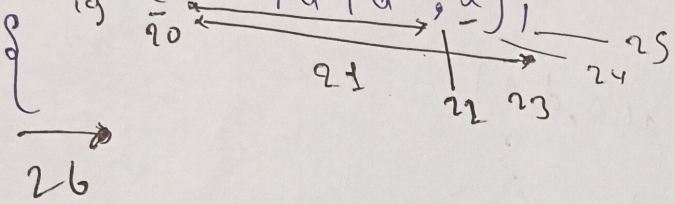
Ans - 10

→ Token के लिए हम use करते है FA (finite autom

→ Token is define by finite automater.

→ Token Count using lexical Analysis

Q
```
int main() {
int x = y + z;
int x, y, z;
Printf ("sum = %d %d", x);
}
```



total = 26

Q
```
main() {
a = b+++---++==;
printf ("%d %d", a, b);
}
```

total = 18

Q
```
main() {
int a = 10;
char b = "abc";
int c = 30;
char d = "xyz";
in /* Comment */ t m = 40.5;
}
```

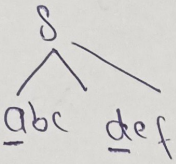
find first and follow ( )

first (A) contains all terminals present in first place of every string derived by A.

1.) $S \rightarrow abC \mid def \mid ghi$

2.) first (terminal) = terminal

3.) first (ε) = ε



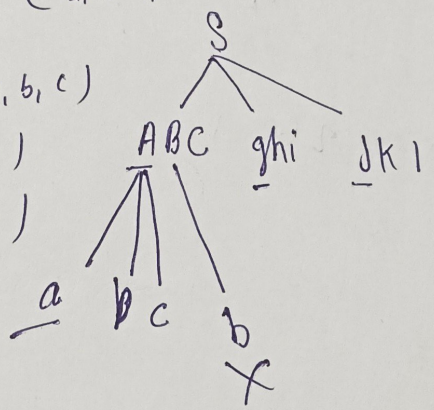
Q

$S \rightarrow ABC \mid ghi \mid JKl$ (a, b, c, d, g)

$A \rightarrow a \mid b \mid c$ (a, b, c)

$B \rightarrow b$ (b)

$D \rightarrow d$ (d)



f(S)

f(ABC)

f(A)

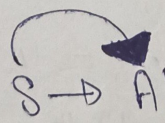
{a, b, c}

Q $S \rightarrow ABC$ (a, b, ε) ✗

$A \rightarrow a \mid b \mid \varepsilon$ (a, b, ε)

$B \rightarrow c \mid d \mid \varepsilon$ (c, d, ε)

$C \rightarrow e \mid f \mid \varepsilon$ (e, f, ε)



$S \rightarrow ABC$ (

Q $f(S) \rightarrow f(ABC) \rightarrow f(A) \rightarrow (a, b$

Q $E \to TE'$

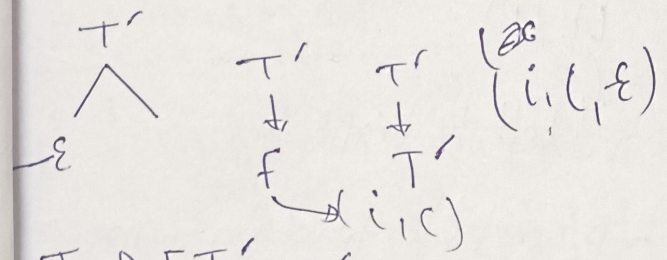
$E' \to *TE' | \varepsilon$

$T \to FT'$

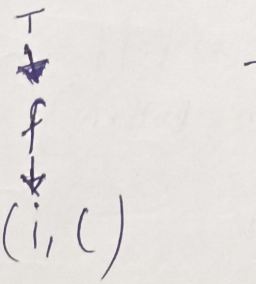
$T' \to \varepsilon | +FT'$

$F \to id | (E$ $\quad (i, ($

$T' \to \varepsilon | +FT$ $\quad (\varepsilon, F)$



$T \to FT'$ $\to (F$



$T \to f + FT'$

$E' \to *TE' | \varepsilon$

$(\varepsilon, i, ($

Ans → $F \to id | (E$ $\quad id, ($

$T' \to \varepsilon | +FT'$ $\quad \varepsilon, +$

$T \to FT'$ $\quad id, ($

$E' \to *TE' | \varepsilon$ $\quad *, \varepsilon$

$E \to TE'$ $\quad id, ($

Q find follow ( ) in Compiler Design.

→ parsing table को बनाने में first and follow help करता है।

⇒ follow(A) contains set of all terminals present immediate in right of 'A'. Rules

1.) $Fo(A) = \{\$\}$

2.) $S \to ACD$

$C \to a | b$

$fo(A) = first(C) = \{a, b\}$

$fo(D) = follow(S) = \{\$\}$

3.) $S \to aSbS | bSaS | \varepsilon$

follow never contains ε

→ follow of start symbol is always $\$$.

Q $S \to AaAb | BbBa$

$A \to \varepsilon$

$B \to \varepsilon$

$\therefore fo(A) = \{a, b, \$\}$

$fo(B) = \{b, a\}$

≠ Q.

$S \to ABC$
$A \to DEF$
$B \to \varepsilon$
$C \to \varepsilon$
$D \to \varepsilon$
$E \to \varepsilon$
$F \to \varepsilon$

$f_o(A) = f(B) \to \varepsilon$ ये never write
to put it

$S \to AC$

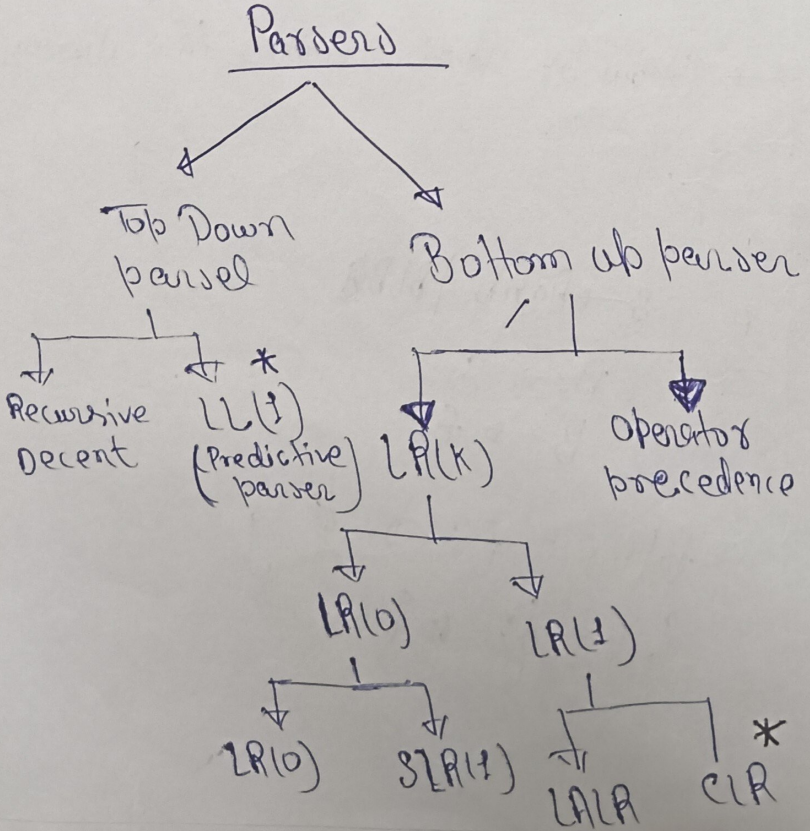
$f_o(A) = f(C) \to \varepsilon$

$f_o(A) = f_o(S) = \{\$\}$

$f_o(B) = f_o(S) = \{\$\}$

$f_o(C) = f_o(S) = \{\$\}$

What is parsing & Types of Parsers



Parsing:- It is a process of deriving string from given grammer.

→ यहाँ tokens syntacally check किया जाता है।

→ Parser tree use होता है।

LL(1) → LMD
↓
Left to Right

→ context free Grammer use.

LR → Right to left (from Bottom)
↓
left to Righ

powerwise see.

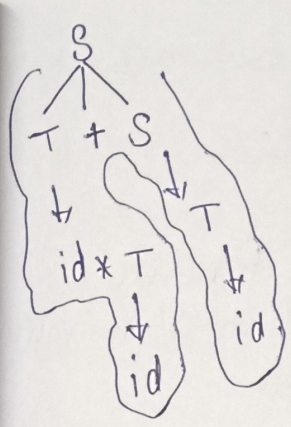
LR(0) < SLR(1) < LALR

Q. $S \to \underset{1}{T} + S \mid \underset{2}{T}$

$T \to \underset{3}{id} * T \mid \underset{4}{id} \mid \underset{5}{S}$

$w = id * id + id$

4 →



id * id + id

this @ is only start symbol.

$S \to a\$bS \mid bSaS \mid \varepsilon$ (1, 2, 3)

	a	b	$
S	1 3	2 3	3

$f(S) = \{a, b, \varepsilon\}$

$f_o(S) = \{\$, b, a\}$

LL(1) Parsing table

	first	follow
$S \to (L) \mid a$ (1, 2)	{(, a}	{$,), ,}
$L \to SL'$ (3)	{(, a}	{)}
$L' \to \varepsilon \mid , SL'$ (4, 5)	{ε, ,}	

on this table more than one entry in box. so this is Not called LL(1)

follow

$f_o(S) = \{\$,), ,\}$

$f_o(L') = \{)\}$

$f_o(L) = \{), \$\}$

*LL(1) due to no more than 1 entry in cell.

	(	)	a	,	$
S	S	1		2		
L	L	3		3		
L'	L'		4		5	

$L' \to \varepsilon$ (4) $f_o(L') =)$

$f(L') = \varepsilon$

यदि किसी का first ε आता है तो उसके follow में जाते हैं।

How to check a Grammer is LL(1) or Not

$S \to aSa \mid bS \mid c$

$S \to i(t\,SS_1 \mid a$

$S_1 \to eS \mid \varepsilon$

$C \to b$

Trick

$S \to \alpha_1 \mid \alpha_2 \mid \alpha_3$ No Null (ε)

$f(\alpha_1) \cap f(\alpha_2) \cap f(\alpha_3) = \phi$ LL(1)

$\neq \phi$ Not (1)

≠ ⇒ यदि Null production (means ε) हो तो.

$S \longrightarrow \alpha_1 | \alpha_2 | \varepsilon$

$f(\alpha_1) \cap f(\alpha_2) \cap f_o(S)$

$= \phi$ LL(1)

$\neq \phi$ Not LL(1)

∴ this is not equal to LL(1).

#What is LL(1) Parser

LL(1) parser

- LMD (Left most derivation)
- left to right Scanning
- look ahead Sym.

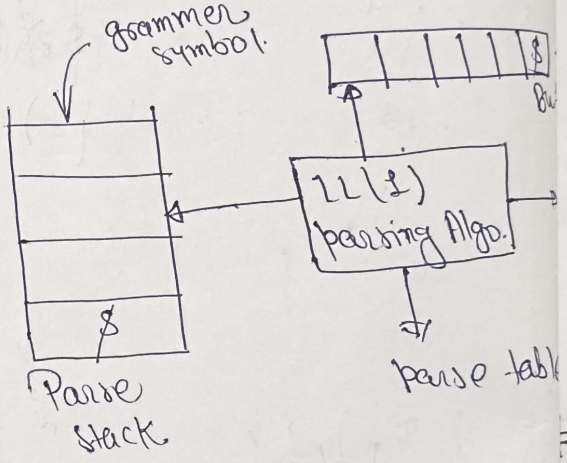
Φ $S \longrightarrow \underset{\alpha_1}{aSa} | \underset{\alpha_2}{bS} | \underset{\alpha_3}{c}$

To check LL(1) or not.

$f(S) = \{a, b, c\}$

$a \cap b \cap c = \phi$

∴ this is LL(1).



$f(aSa) \cap f(bS) \cap f(c)$

Φ $S \longrightarrow iCtSS_1 | a$

$S_1 \longrightarrow eS | \varepsilon$

$C \longrightarrow b$

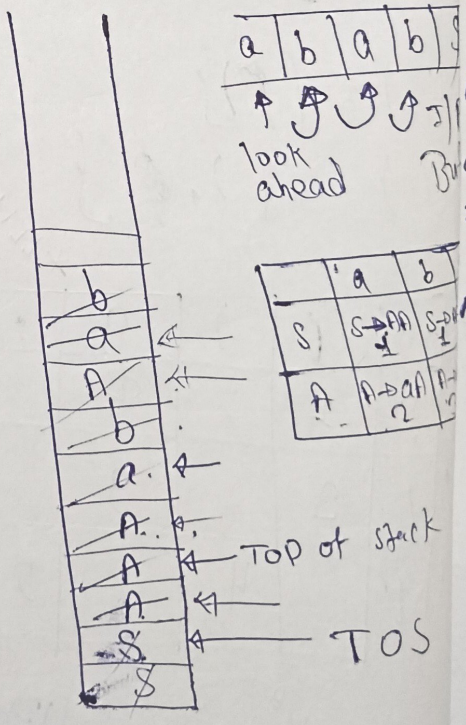
$f(iCtSS_1) \cap f(a)$ —— ①

$i \cap a = \phi$

$f(eS) \cap f_o(S_1)$ → $f(S) \rightarrow f(S_1)$ e, $

$e \cap \$, \neq \phi$

f(



Parse stack :- this is help to decide which grammer is enter to parse stack.

Parse table :-

	a	b	$
S	S→AA (1)	S→AA (1)	
A	A→aA (2)	S→b (3)	

S → AA 1
A → aA 2
A → b 3

LR(0) parsing table

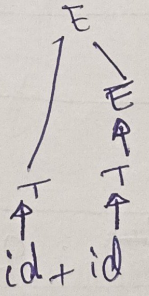
LR(0) parsing table को बनाने के Canonical Item को use करते है

LR parser is a bottom-up Syntax analysis technique that can use for large class of context free grammer.

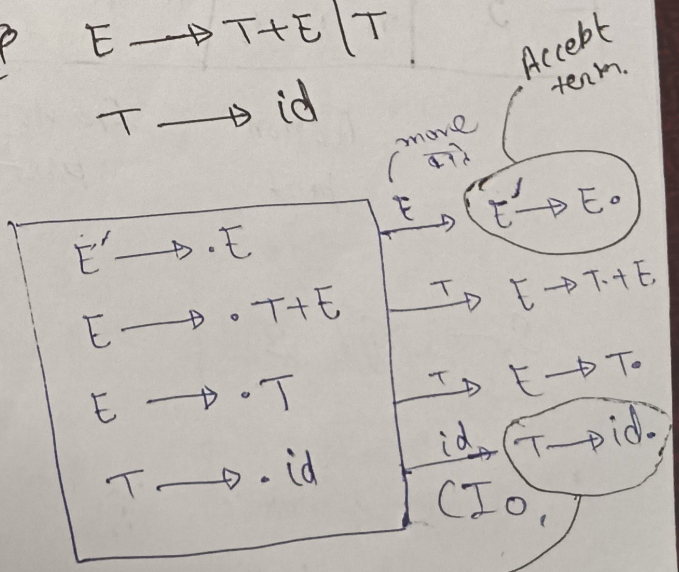
LR(0) → AND (Right most derivative)
→ no. of input symbols of look-ahead
Left to right scanning

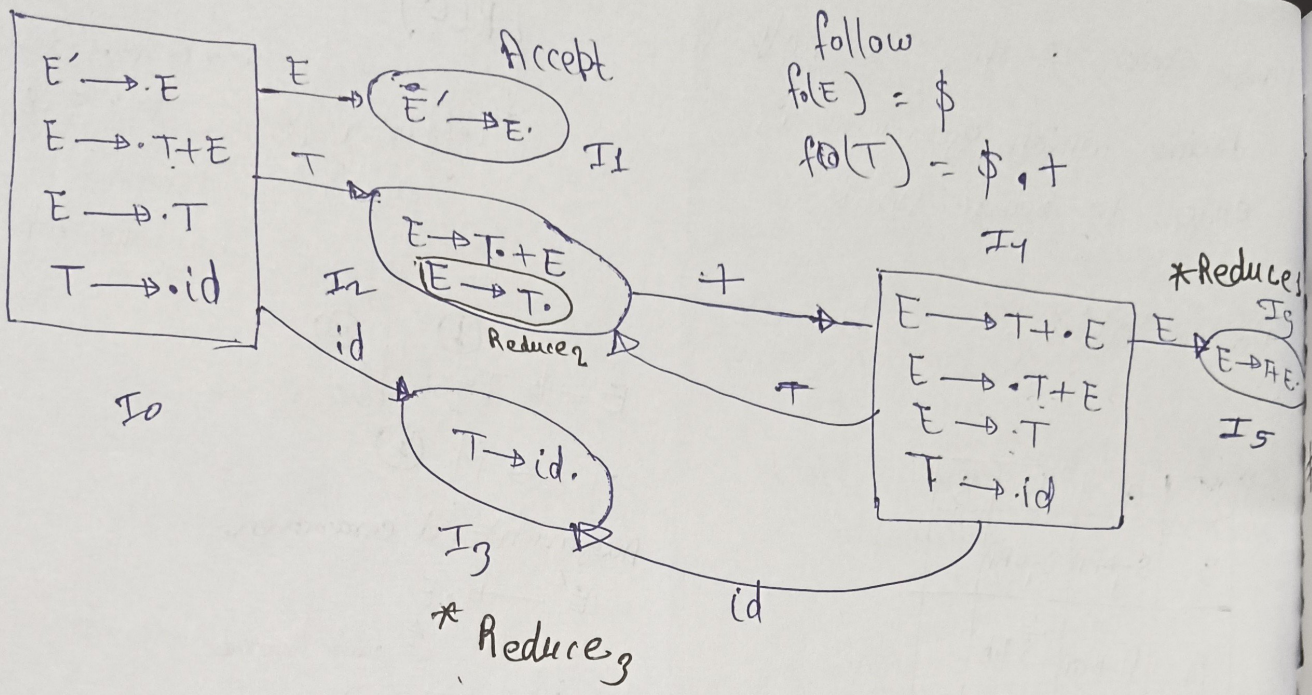
E → T+E (1) | T (2)
T → id (3)

Augmented grammer
E' → .E
.id ← not seen
id.+ ← not seen
id +.id ← not seen
(id + seen)



E → T+E | T
T → id





follow
fo(E) = $
fo(T) = $, +

State	Action			Goto	
	id	+	$	E	T
0	S_3			1	2
1			Accept		
2	r_2	S_4/Reduce2	r_2		
3	r_3	r_3	r_3		
4	S_3		Reduce 5		2
5	r_1	r_1	r_1		

Action part

Go to part

S_3 → shift 3

∴ Given grammer is not LR(0)

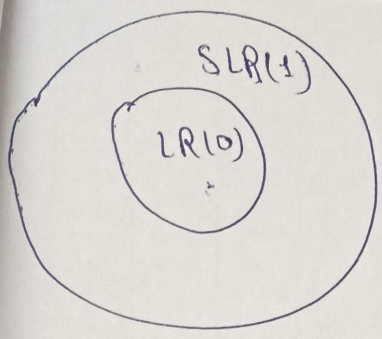
Check LR(0) or Not

Action part में यदि shift and reduce या फिर reduce and reduce दोनों ही state में आ जाता है तो कहते है Not LR(0).

SLR(1) parsing table

State	id	+	$
0	S_3		
1			Accept
2		S_4	r_2
3		r_3	r_3
4	S_3		
5			r_1

4 item.



CLR Parsing Table

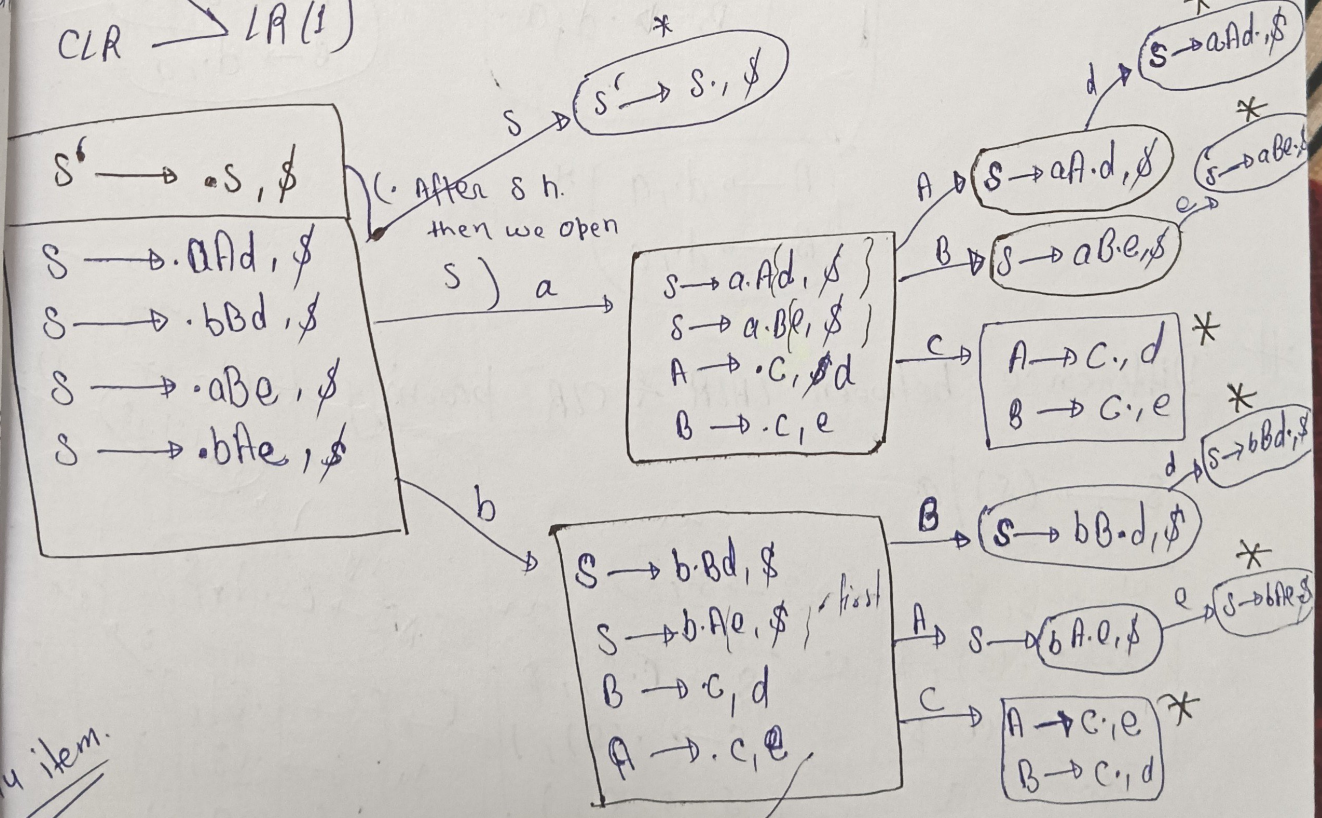
$ → this use LR1 (Canno-nical items)

S → aAd | bBd | aBe | bAe

A → c

B → c

LALR
CLR → LR(1)

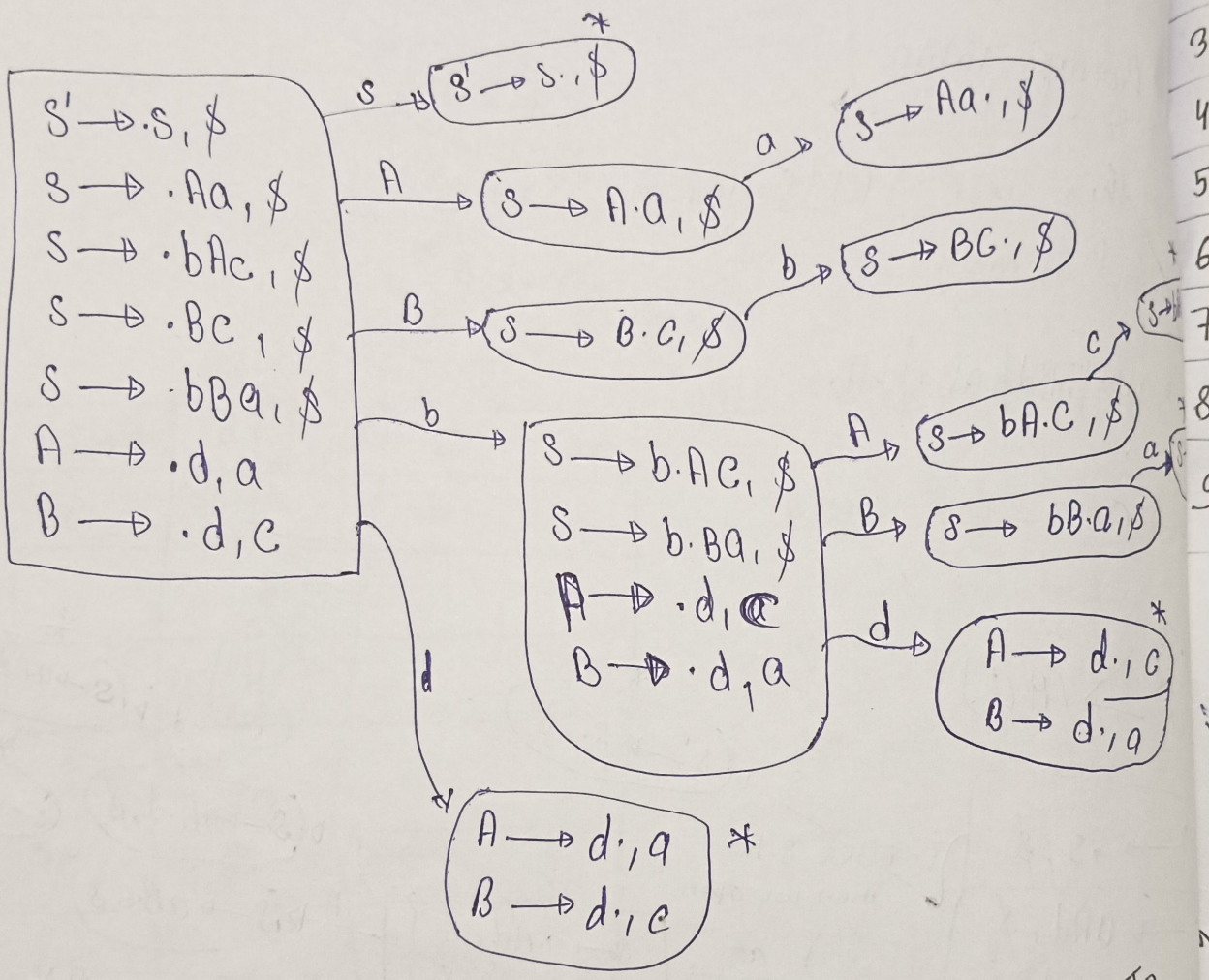


LALR Parsing table

$S \to A'a \;(1) \mid bAc \;(2) \mid Bc \;(3) \mid bBa \;(4)$

$A \to d \;(5)$

$B \to d \;(6)$

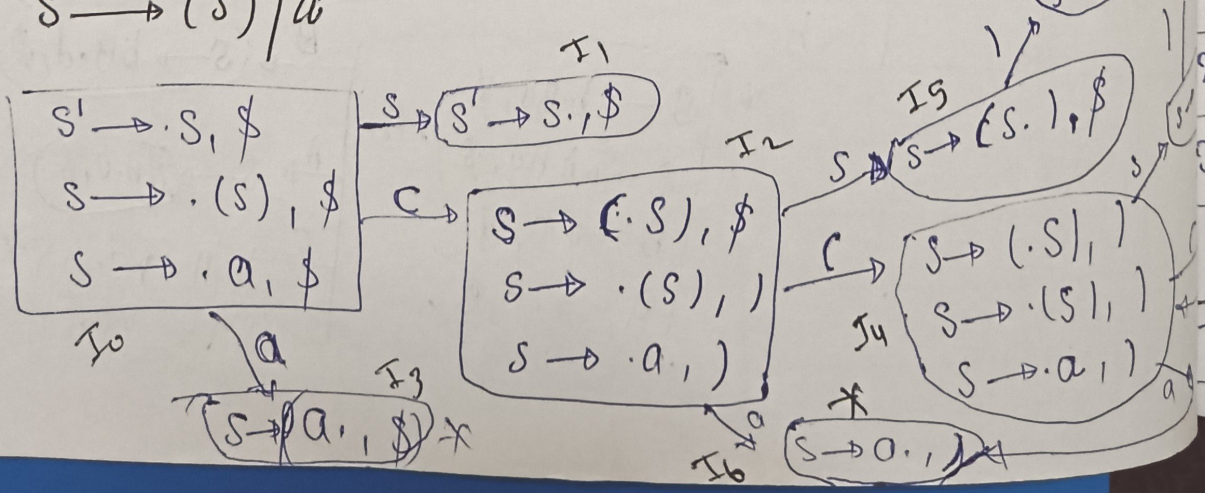


	c	)
0	S2	
1		
2	S4	
3		
4	S4	
5		S7
6		r2
7		
8		S9
9		r1

यहाँ
in which
one
CAL
Now conve

Difference between LALR & CLR parsing table

$S \to (S) \mid a$



	(	)
0	S2	
1		
24	S4	
36		
58		
79		

	(	)	a	$	S
0	S_2		S_3		1
1				Accept	
2	S_4		S_6		5
3				r_2	
4	S_4		S_6		8
5		S_7			
6		r_2		r_2	
7				r_1	
8		S_9			
9		r_1			

यहाँ

in which in shell only one entery. so this CRL

Now convert into LALR

	(	)	a	$	S
0	S_2		S_3		1
1				Accept	
24	S_4		S_6		58
36		r_2		r_2	
58		S_79			
79		r_1		r_1	

Note → यदि CRL not comes SR conflict. then also not comes SR conflict in LALR.

- CRL LALR
  SRX SRX
  RRX RRL

→ यदि यदि CRL RR conflict not comes, then in LALR, RR ~~may~~ conflict may comes.

SDT (Syntax Directed Translation) & its applications.

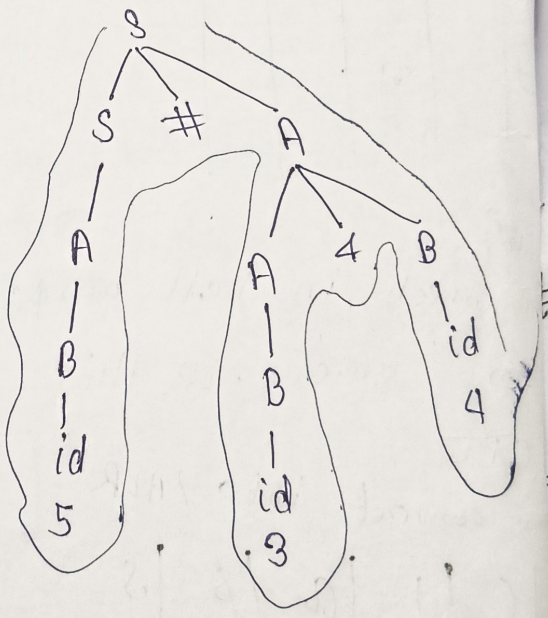
Grammer + Semantic Rules = SDT

Applications:

1) Executing Arithmetic Expression.
2) Conversion from infix to postfix.
3) Conversion from infix to prefix.
4) Conversion from binary to decimal.
5) Counting numbers of reductions.
6) Creating syntax tree
7) ~~Generator~~ Generating intermidiate code
8) Type checking
9) Storing type information ~~and~~ into symbol table.

G	Rules.
$S \to S\#A \mid A$	{ S.val = S.val * A.val; }
$A \to A \mid B \mid B$	{ A.val = A.val + B.val }
$B \to id$	{ B.val = ~~id.lVal~~ id.lval ; }

given string
5#3&4



= 5x3+4
= 15+4 = 19 X

= (5*(3+4))
= 5*7 = 35

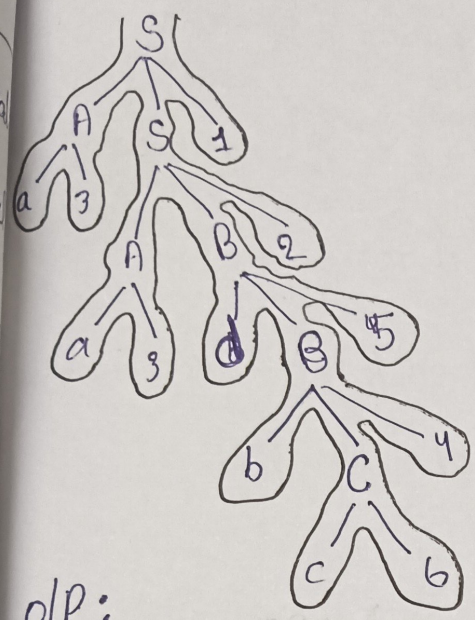
Top Down Vs Bottom Down parsing (parse SDT)

$S \to AS$	{ printf(1) }
$S \to AB$	{ printf(2) }
$A \to a$	{ Pf(3) }
$B \to bC$	{ Pf(4) }
$B \to dB$	{ Pf(5) }
$C \to c$	{ Pf(6) }

I/P = aadbc

Top down parsing

I/P = aadbc

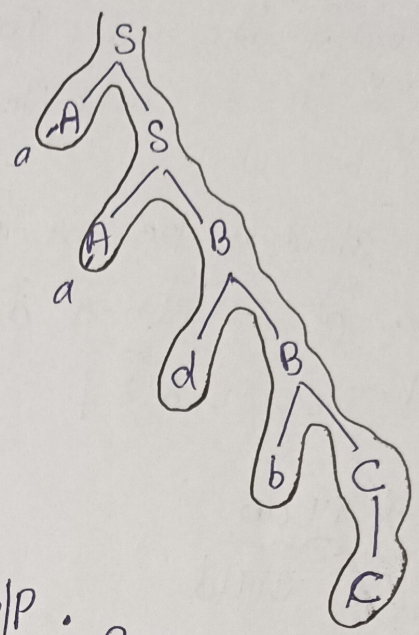


O/P:

3 3 6 4 5 2 1

⇒ ~~a a c b d~~

Bottom Down parsing



O/P : 3 3 6 4 5 2 1

How to Evaluate Arithmetic Expression using SDT

P E → E + T { E.val = E.val * T.val }
| T { E.val = T.val }

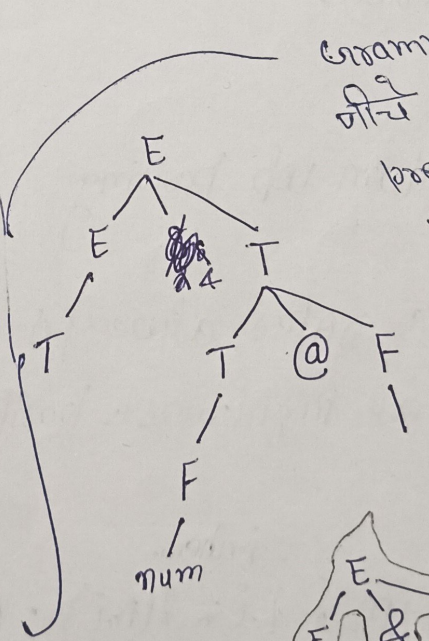
T → T @ F | F { T.val = T.val - F.val }
{ T.val = T.val }

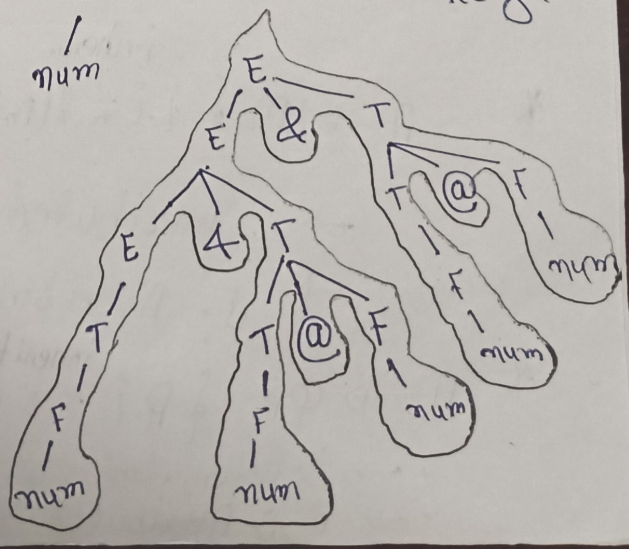
F → num { F.val = num }

I/p: 4 8 @ 5 4 7 @ 3

output: {4 * {8 - 5}} * {7 - 3}

= 12 * 4 = 48





Type of SDT

जब parent से कोई value (child) लिया जाता है तो उसे inherited कहते है

वही जब child से parent कोई value ले रहा होता है तो उसे Synthesise कहते है।

SDT/SDD = use both synthesis and inherit

A → XYZw

parent Child

1) S-Attributed SDT

→ Based on synthesized attribute

→ use Bottom up parsing

→ Semantic rules always written at Rightmost position in RHS

2) L-Attributed SDT

→ Based on both synthe and inherited attribut

* (parent, Left sibling)

→ Top down parsing

→ Semantic rules anywhere in RHS

* $A \to LM \{ L.i = l(A.i) ; M.i = m(L.S) ; A.S = f(M.S) \}$ (inher. inherited Synthesize)

→ L-Attributed SDT ✓

→ S-Attributed SDT ✗ (not only synthesize)

* $A \to QR \{ R.i = r(A.i) ; Q.i = q(R.S) ; A.S = f(Q.S) \}$ (inherited inherited Synthesize)

→ L-Attributed SDT ✗ (due to not Q not take value Right)

→ S-Attributed SDT ✗

Q $A \rightarrow PQ$ and $A \rightarrow XY$

{ Rule 1: $P.i = A.i + 2$, $Q.i = P.i + A.i$ (inherited, inhe.)

and $A.s = P.s + Q.s$ } (synthesize.)

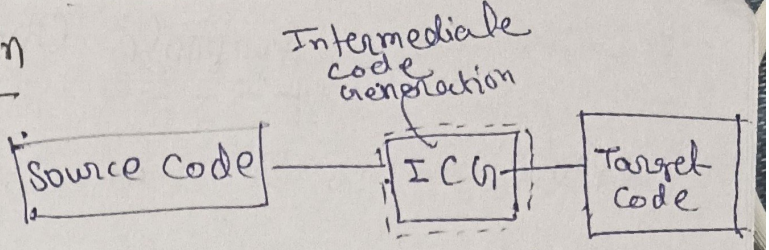
{ Rule 2: $X.i = A.i + Y.s$ and (inher. (Right sibling))

$Y.i = X.s + A.i$ } (inher.)

Ans- only ~~left~~ Rule 1

is left attributed.

Intermediate Code Generation



1.) machine independent.

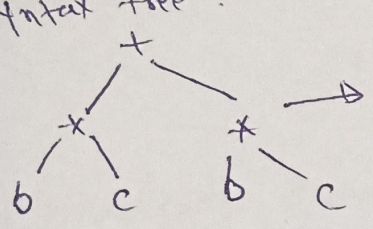
LA	LA
SA	SA
IC	IC
P1	P2

ICG, Source code को general purpose code में convert करता है जो machine independent होता है.

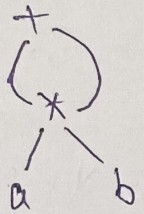
2.) Abstract Syntax tree.

e.g (b*c)+(b*c)

Syntax tree



AST



3-Address code (x = y op z)

x = (a*b)+(b*c)+(d*c)

t1 = (a*b)+(b*c)

t2 = d*c

x = t1+t2

3.) Direct Acyclic Graph.

⇒ Variable Statements in 3-Addr-ess Code.

1.) Assignment — 'op' ⇒ operand

x = y op z	// x = y+z
x = op y	// x = -y
x = y	// x = y

4.) Postfix

bc*bc*+

2.) Jump → Conditional

if x rel op y goto L

→ unconditional goto L

** 5.) 3-Address Code

⇒ in which we have use 3 address code.

e.g t1 = b*c

t2 = b*c

t3 = t1+t2

3.) Array Assignment

x = y[i]

x[i] = y

4.) Addressess assig.

x = &y

x = *y

Peephole Optimization

'peephole' means small window.

Peephole optimization is a type of code optimization. which is perform on small part of code.

→ It is perform very small part of instruction in a segment code.

e.g. (i) initial code

```
x = 2x3;
```

optimal code

```
x = 6;
```

(ii)

```
x = 2*y ;
```

```
x = y+y ;
```

→ peephole optimization is code machine dependend.

objective

(i) To improve performance

(ii) reduce memory size

(iii) To reduce code size.

1) Redundant load and Store

```
a = b+c
d = a+e
```

when we have

```
d = b+c+e
```

```
mov b, R0
mov c, R0
Add c, b
```

```
mov b, 50
mov c, 45
Add b
mov d, c
Add c
```

remove

```
1. LOAD R0, b
2. ADD R0, C
3. Store a, R0
4. LOAD R0, a
5. ADD R0, e
6. STORE d, R0
```

2.) Strength Reduce.

$x^2 \rightarrow x * x$

$x * 2 \rightarrow x + x$

$y = x^2$

initial code	optimal code
$y = x^2$	$y = x * x$
$y = 2 * x$	$y = x + x$
$y = \frac{x^2}{2}$	$y = (\frac{x}{2}) * x$

5.) Deadcode Elimination

```
int dead(void){
  int b = 30
  int a = 10
  int c;
    c = a+b
  return c
  { b = 10*5;   dead
    d = b+10;   code.
  return 0;
```

3.) Simplify ~~Alge~~ Algebric Exp.

initial code	optimal code
$b = a + 0$	$b = a$
$b = \frac{a}{1}$	$b = a$
$b = a - 0$	$b = a$
$b = a * 1$	$b = a$

4.) Replace shaon instruction with faster.

SUB 01, A

Add B, 01 ⇒ ~~Add~~ INC B

SUB B, 01 ⇒ DCR B

ino c++

initial code	optimal code
$b = b + 1$	$b++;$
$b = b - 1$	$b--;$
$b = b * i$	$b *= i;$

code optimization

platform dependent optimization

→ peephole optimization

platform independent optimization

→ ~~Platform~~ loop optimization

decreasing the test cases.

- loop unrolling
- code movement
- frequency reduction
- loop jumping

→ instruction level parallelism
→ Data level parallelism
→ Cache optimization
→ Redundant resource.

loop optimization

loop optimization is a method for increasing the execution ~~time~~ speed and decrease the overheads associated with loop.

* → most execution time ~~Spen~~ of scientific program spent on loops.

→ this optimization is machine independent.

1) Code motion (frequency Reduction)

```
int x=5, y=6;
int a= 100;   int p=x+y;
for( int i=0; i<a; i++)
{ int p = x+y
  if (p*i==0)
    { cout<< i <<endl;
    }
}
return 0;
```

ii) → Constant folding

$P_i = \frac{22}{7}$, $P_i = 3.14$

$a = \sqrt{2}$, $a = 1.414$

→ Constant propagation (copy.)

$A = \pi r^2$, $A = 3.14 \times 2$

$r = 2$ $= 4 \times 3.14$

$A = 12.56$

→ Common subexpression elimination. (Redundant motion)

```
int a = b*c
int d = b*c
int p = a+d
int p = 2*a
```

2.) loop fusion (try to less loop)

```
int a=100;   int arr1[a];   int arr2[a];
for (int i=0; i<a; i++)
{ cin>> arr1[i]
}
for (int i=0; i<a; i++)
{ cin>> arr2[i];
}
```

so, here one loop remove and enter

cin>> arr1[i] >> arr2[i]

3.) loop

initial

for (in
{
}

optimiza

Cou
Cou
Cou

Cont

Basic

* first
loo

How
blo

Step 1

Step 2

3) loop unrolling

initial code

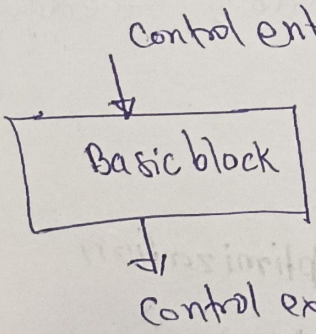
```
for(int i=0; i<3; i++)
{   cout<<"rohit"<<endl;
}
```

optimial code

```
cout<<"rohit"<<endl;
cout<<"rohit"<<endl;
cout<<"rohit"<<endl;
```

Control flow graph.

Basic block



* first line of block is called leaders.

How to find leader in basic block.

Step 1 → first statement is always leader

Step 2 → Address of conditional, unconditional goto are

```
for(i=0; i<n; i++)
{   for(j=0; j<n; j++)
    {   if(i+2==0)
        {   x+=(4*j+5*i)
            y+=(7+4*j)
        }
    }
}
```

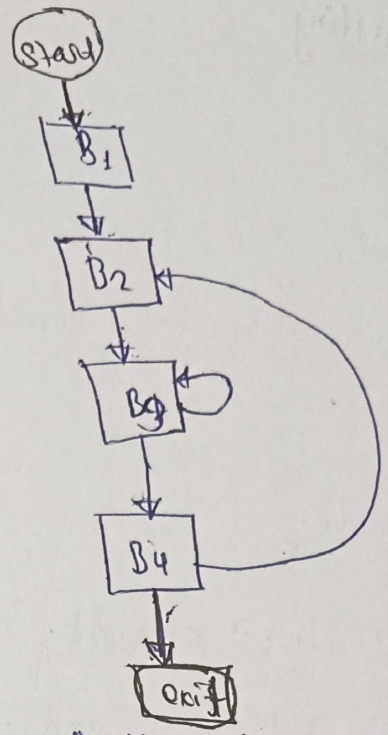
optimial code

```
for(i=0; i<n; i++)
{   if(i+2==0)
    {   x+= 9*i
    }
    y+= 7+4*j
}
```

step 3 → Next line of conditional and unconditional goto are leaders.

Consider the intermediate code;

1. i=1 B_1
2. j=1 B_2
3. $t_1 = 5 * i$
4. $t_2 = t_1 + j$
5. $t_3 = 4 * t_2$
6. $t_4 = t_3$
7. $a[t_4] = -1$
8. $j = j + 1$
9. if $j <= 5$ goto (3) B_3
10. $i = i + 1$
11. if $i < 5$ goto (2) B_4



→ 4 nodes, 5 edges

Q Find no. of nodes and edges in control flow graph?

If no in option then add start block and exit block

→ Ans: 6 nodes, 7 edges.

Scope of optimization

Local optimization

→ within basic block

e.g.
→ local common subexpression elimination

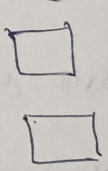
initial code	optimal code
A = x*y + k	$t_1 = x*y$
B = x*y - P	A = t_1 + k
	B = t_1 - P

→ Deadcode Elimination within in BB

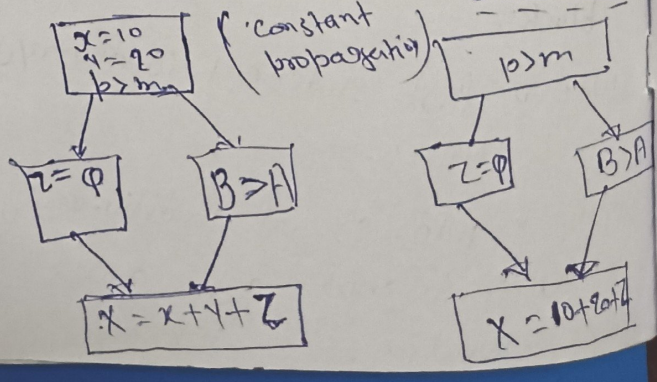
```
a = b*c
return a
b = 5
return a;  // dead code
```

Global optimization

→ Between basic blocks



→ Extended to entire control flow graph.

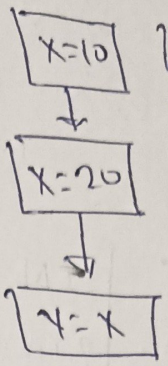


Constant propagation

```
x = 10;
y = x + p * q
```

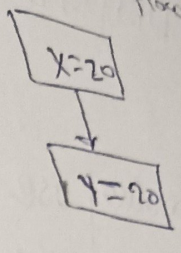
optimal code

```
y = 10 + p * q
```



dead code.

Optimal flow



Algebric law

```
y = a * b
z = b * a
```

optimal code

```
y = a * b
z = y
```

Reordering statement

Liveliness Analysis

- liveliness analysis is a technique to remove dead code

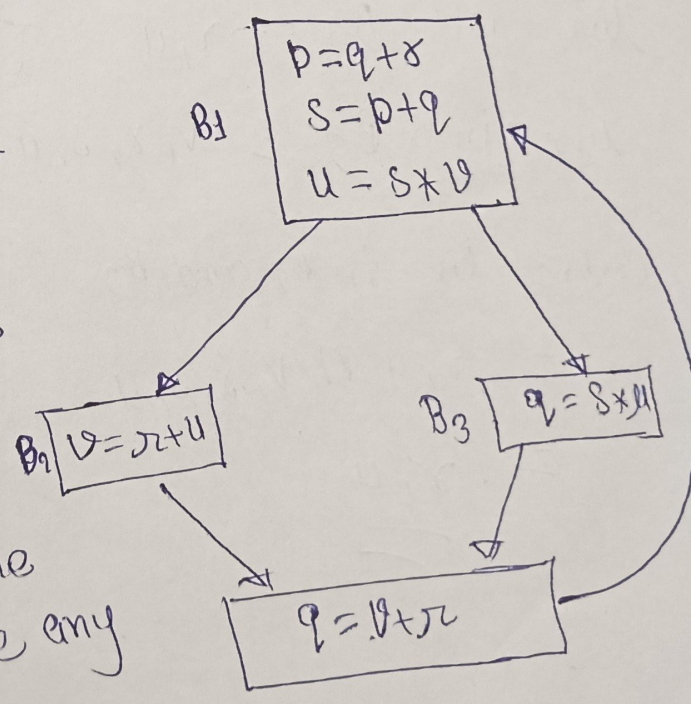
$$In[n] = use[n] \cup (OUT[n] - DEF[n])$$

$$OUT[n] = \bigcup_{S \in Succ[n]} In[S]$$

IN[B] = set of variable live at begining of B.

OUT[B] = set of variable live just after B.

USE/GEN[B] = variables that are used in B before any assignment
(variables in R.H.S but not in L.H.S of prior statement in B)



* DEF/KILL[B] = variables that are assigned a value in B. (variable in L.H.S)

node	use	DEF	IN	OUT	IN	OUT
			1st go		OUT - DEF	
1	q, r, v	p, s, u	q, r, v	r, u, s	r, q, v	
2	r, u	v	r, u	v, r	r, u	v, r, s, u
3	s, u	q	s, u	v, r	v, r, s, u	r, v
4	v, r	q	v, r	q, r, v	r, v, a	r, q, v

* A - B
means A but not in B.

live = on final In variable

* live in B_2 = r, u [means In variable]

live in B_3 = v, r, s, u

Combined live in B_2 and B_3

= r, u ∩ v, r, s, u

= r, u

* $E \to E+id / id$

Same left recursion.

$E+id+\epsilon / id$

right recursion,

$A \to A\alpha / \beta$

~~$L = \{\beta, \beta\alpha, \beta\alpha^2, \ldots\}$~~

$= \beta\alpha^*$

$L = \{\beta, \beta\alpha, \beta\alpha^2, \ldots\}$

$= \beta\alpha^*$

$A \to \alpha A / \beta$

~~$L = \{\alpha\beta, \alpha\beta, \alpha\beta^2, \ldots\}$~~

$L = \{\alpha\beta, \alpha\beta, \alpha^2\beta, \ldots\}$

$\alpha^*\beta$

→ Top down parser : left recursion को Accept नहीं करता है।

Remove left recursion

i) $E \to E+T / T$

left recursion

$A \to A\alpha / \beta$

$\alpha = +T, \beta = T$

$E = A$

$E \to \beta\alpha^*$

$E \to TE'$

$E' \to +TE' / \epsilon$

ii) $S \to SOSOS / OI$

$A \quad \alpha \quad \beta$

$S \to OIS'$

$S' \to \epsilon / OSOSS'$

iii) $L \to L,S / S$

A

$L \to SL'$

$L' \to \epsilon / ,SL'$

Q $7+3*4\uparrow 3+2$

$+ < *$

L-R

R-L

$(7+(3*(4\uparrow 3)))+2$

Q2.20

$S \to ictSS_1 | a$

$S_1 \to eS | \varepsilon$

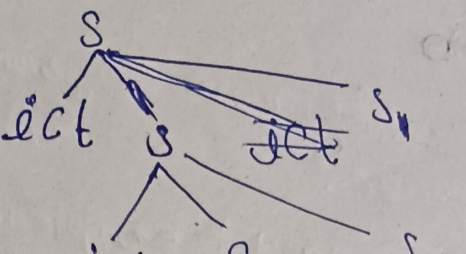
$C \to b$

$S \to ictSeS | a | \varepsilon$

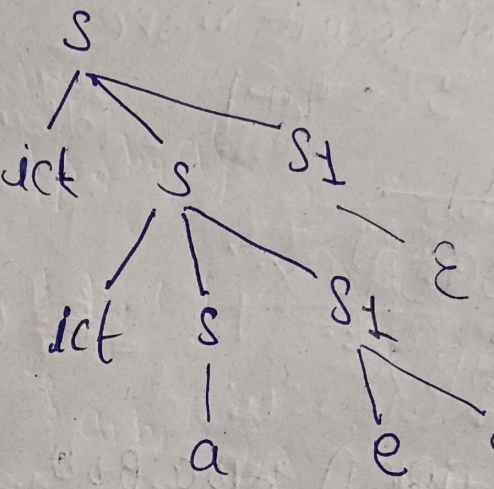
$S \to ictS | a$

$S_1 \to eS | \varepsilon$

ibtaibtea



ibtibtaeea



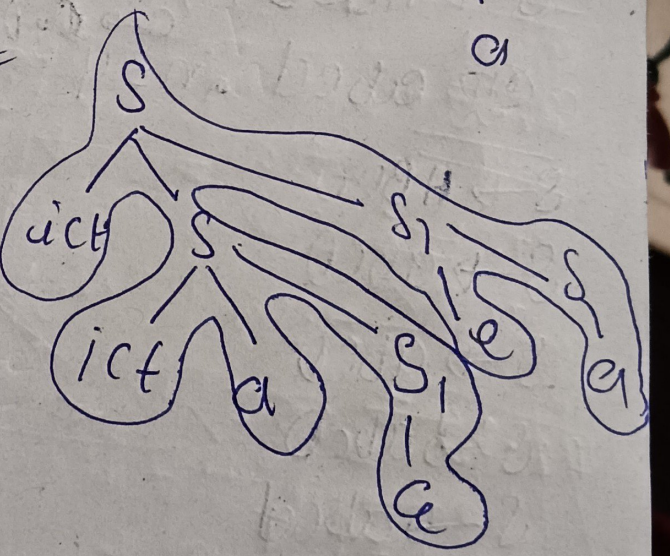
$S \to ibtS | a$

$S_1 \to eS | \varepsilon$

$C \to b$

Ambigious ⇒ when we draw more then one parse tree then called Ambigious grammer.

II



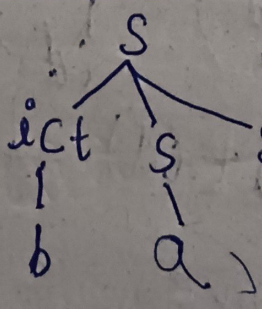
$S \to ictSS_1 | a$

$S_1 \to eS | \varepsilon$

$C \to b$

ibbtaaeaic

ibbtaeea

① 

** Q2.29 not solved.

$A \to B$

$A \to a$

$B \to a$

n=2

AB

ab = Ab = AB = S

2n-1

→ production में operator (*, +) जहाँ use हुआ
→ यदि वहाँ left recursion हुई तो left associative होगा।
→ यदि वहाँ right recursion हुई तो right associative होगा।

$S \to T * P$

$T \to U \mid T * U$ left recursion

* left Associative

$P \to Q + P \mid P$ right recursion

+ right associative.

SLR parsing

Stack	Input	Action
$	id*id	Shift
$id	*id	reduce
$F	*id	reduce
$T	*id	Shift
$T*	id	Shift
$T*id	$	reduce
$T*F	$	reduce
$T	$	reduce
$E	$	Accept

Q2.33 Viable prefix

→ The prefixes of right sentential forms that can appear on a stack of shift reduce parser

Q $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id$

$w = id*id$

Q2.34

$D \to 3$

$B \to 4$

$A \to 2$

$C \to 1$