Compiler design

LR parser is one type of bottom up parsing used to pass large number of grammar.

L stand for left to right scanning of input.

R stand for constructing Right most derivation. in reverse.

LR parser

LR(0) SLR(1) LALR(1) CLR(1)

It requires stack, input output and parsing table.

table contain action and go to part.

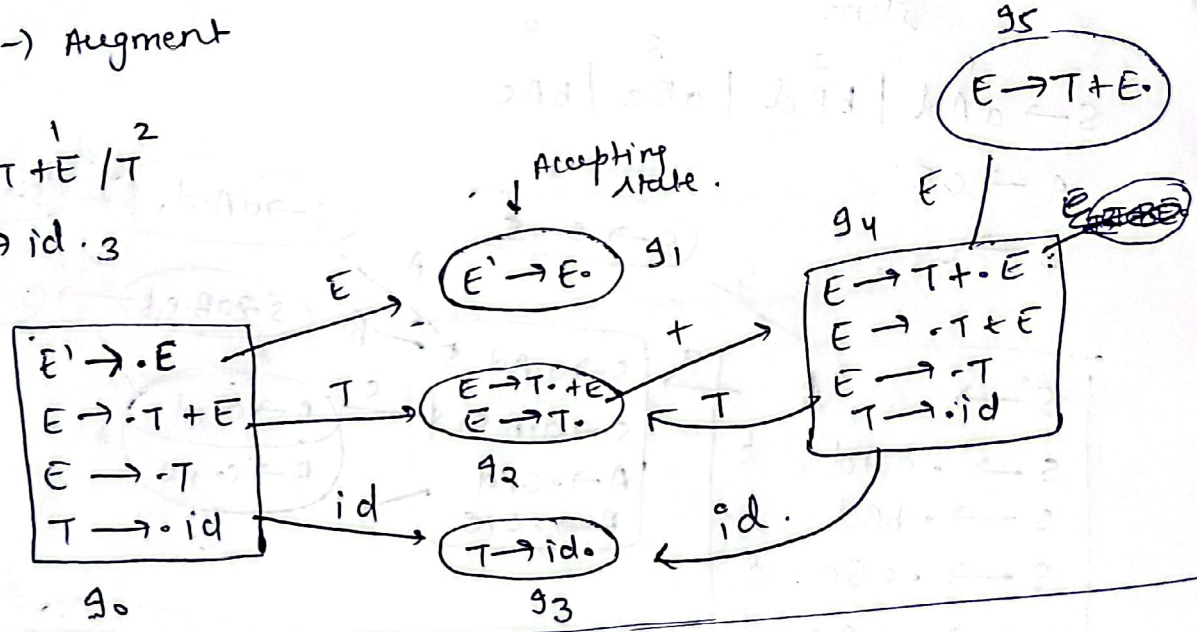
LR(0) parser

LR(0) canonical item

→ Augment

$E \to T + E^{(1)} \mid T^{(2)}$

$T \to id^{(3)}$



Table

State	Action			Goto	
	id	+	$	E	T
0	s_3			1	2
1			Accept		
2	r_2	s_4/r_2	r_2		
3	r_3	r_3	r_3		
4	s_3			5	2
5	r_1	r_1	r_1		

Shift reduce conflict

This grammar is not LR(0)

SLR(1) parsing table

It works on LR(0) canonical items.

State	Action			GO TO	
	id	+	$	E	T
0	s_3			1	2
1					
2		s_4	r_2		
3		r_3	r_3		
4	s_3			5	2
5			r_1		

Reduction is not put inside the whole row but inside follow of left

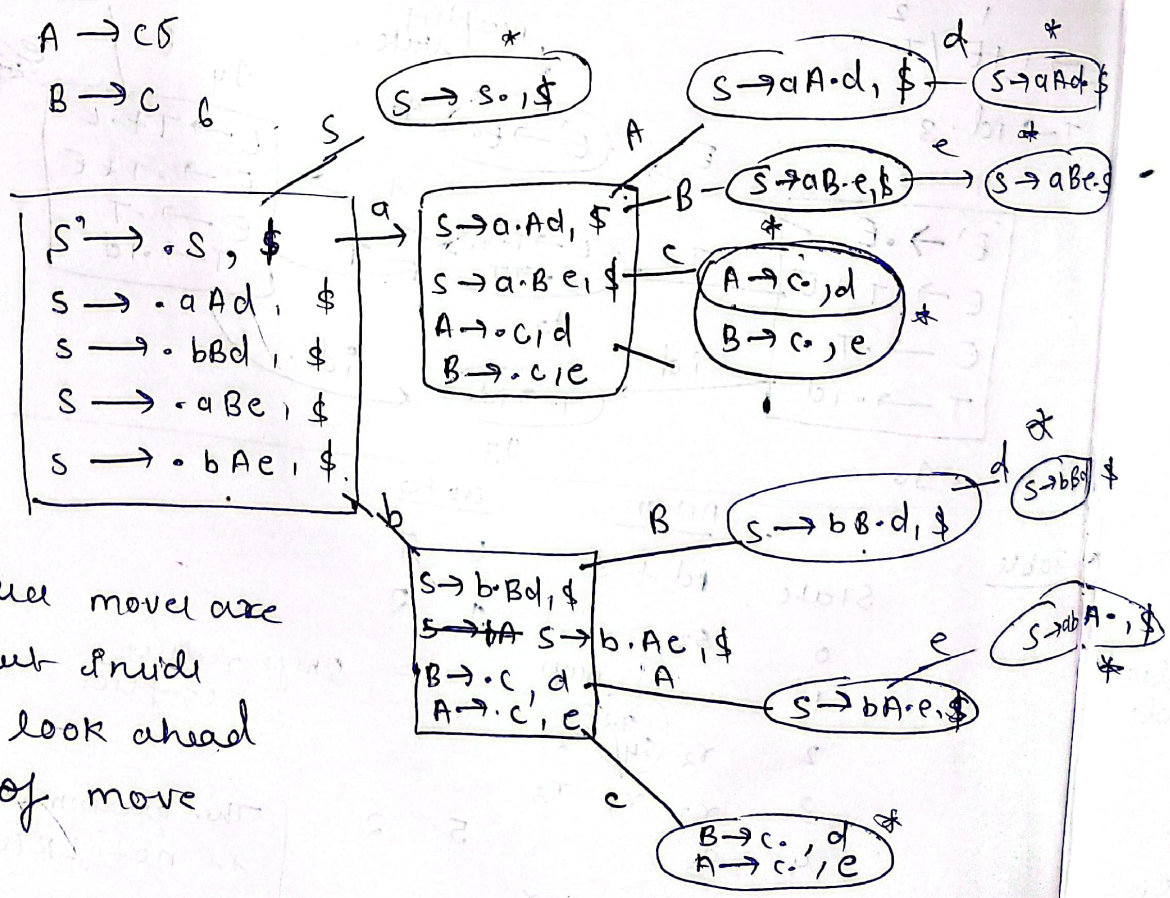
There is no conflict. in this the grammar is SLR.(1), SLR is more ~~ful~~ than LR(0)

CLR - works on LR(1) ~~parsing table~~ canonical item.

$S \to aAd \mid bBd \mid aBe \mid bAe$ (1, 2, 3, 4)

$A \to c$

$B \to c$



reduce moves are put inside look ahead of move

LALR parser

uses LR(1) canonical item.

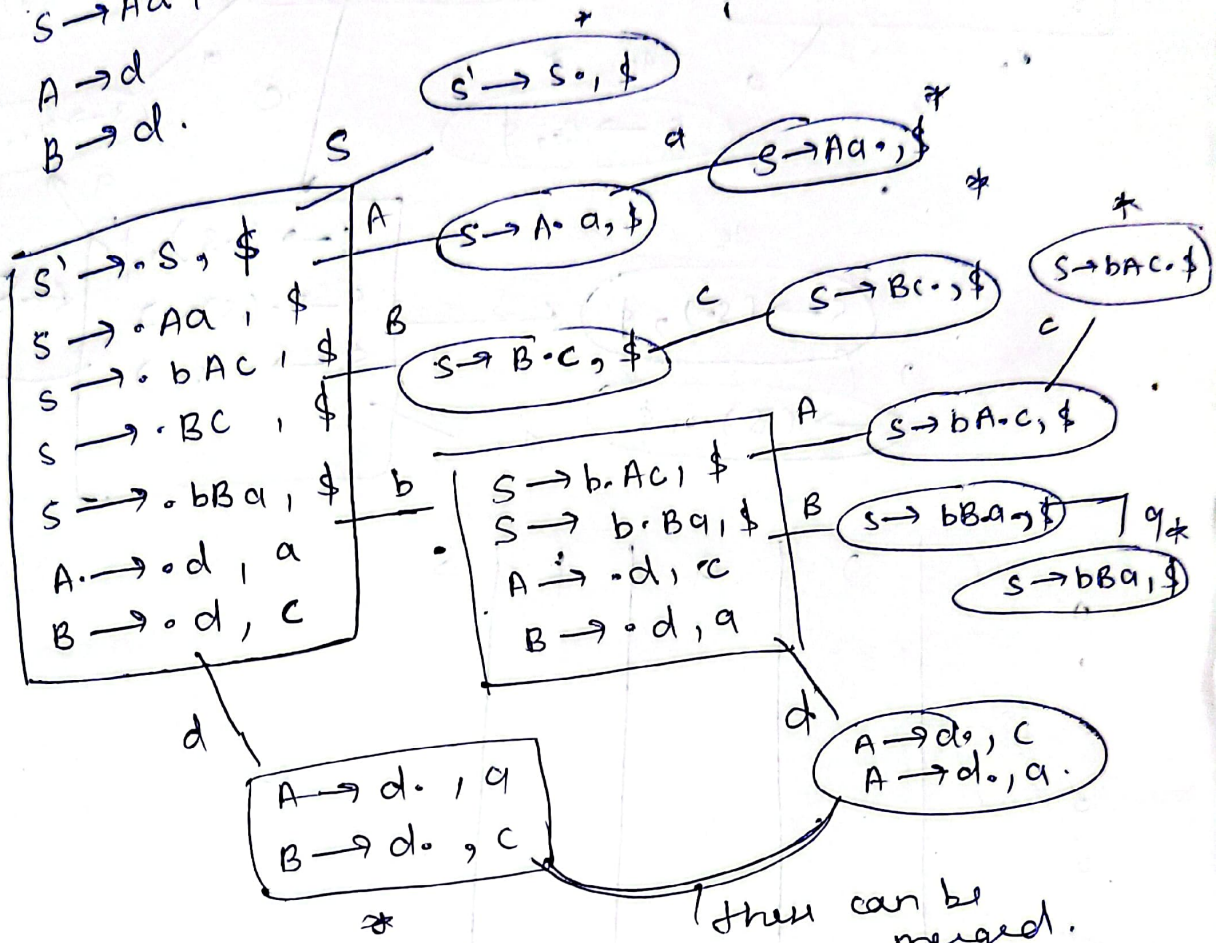
LALR is less powerful than CLR.

some states are merged.

$S \to Aa \mid bAc \mid Bc \mid bBa$

$A \to d$

$B \to d.$



a | c

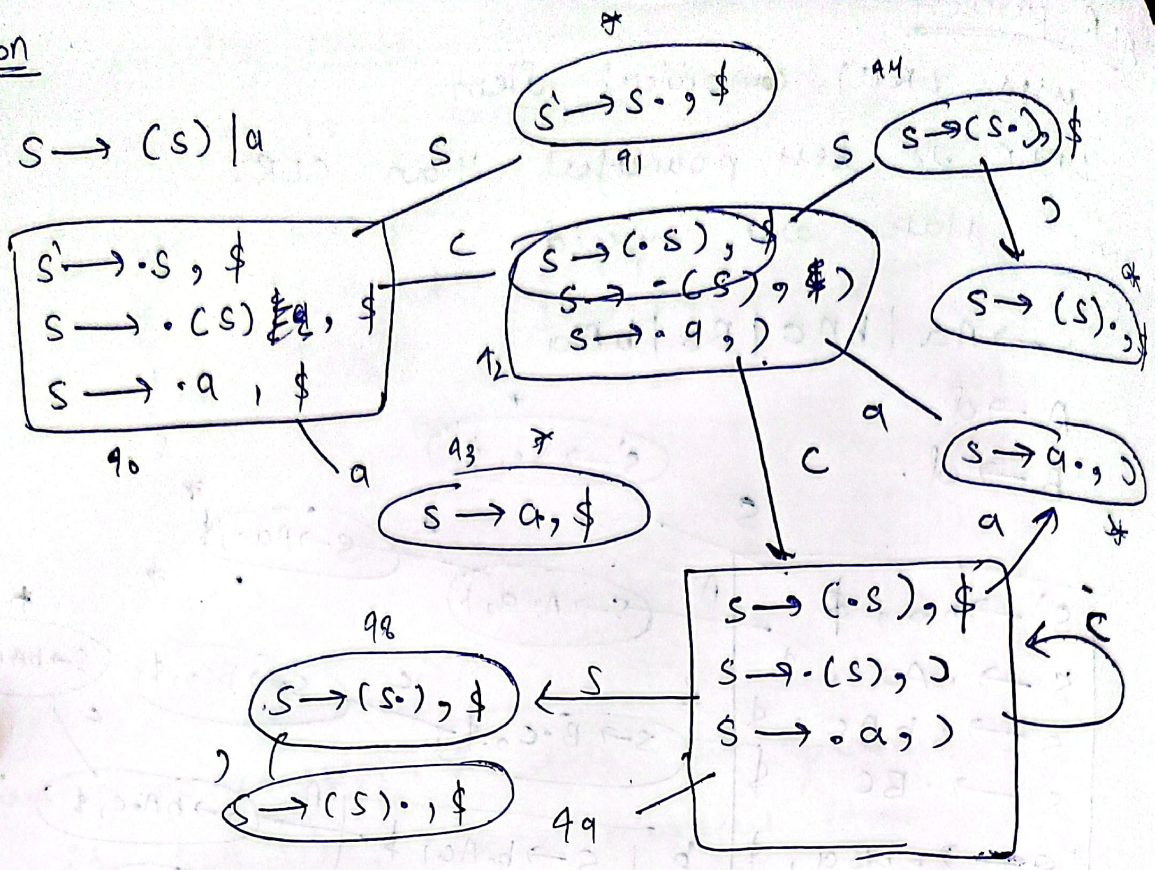
= | =

multiple entry in grammar

not in CALR

Question

$S \rightarrow (S) \mid a$



	(	)	a	$	S
0					
1					
2					
3					
4					

Syntax directed translation :

Grammar + Semantic rules = SDT

Parse tree is evaluated in SDT

& PT → [SDT] → Annoted tree.

Application -

1) Executing arithmetic expression
2) infix to postfix.
3) infix to prefix
4) binary to decimal
5) count no. of reduction
6) Syntax tree
7) generate intermediate code
8) Type checking
9) store type info in symbol table.

$S \rightarrow S \# A / A$ { S.val = S.val * A.val; }

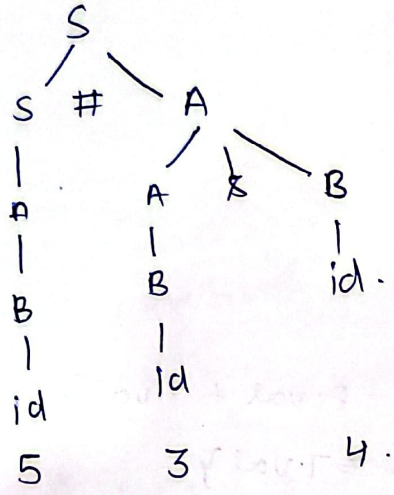
$A \rightarrow A \& B / B$ { A.val = A.val + B.val; }

$B \rightarrow id$ { B.val = id.val; }.

grammar

Rule

find - 5 # 3 & 4



= 5 * 3 + 4

= 15 + 4

= 19

SDT evaluation

Top down.

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

$E \rightarrow E' + T$

$E \rightarrow T$

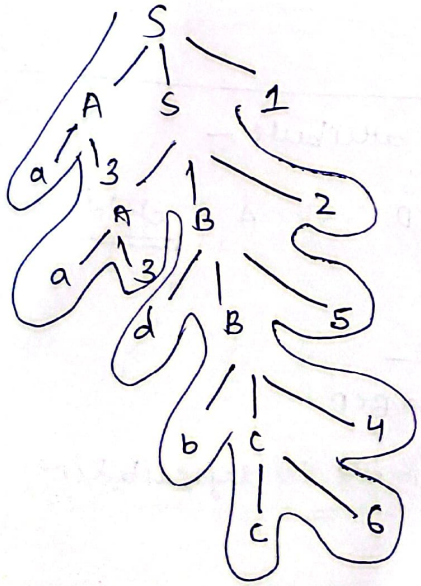
$T \rightarrow T' * F$

$T \rightarrow F$

$F \rightarrow num.$

3 + 5 * 4 = 23

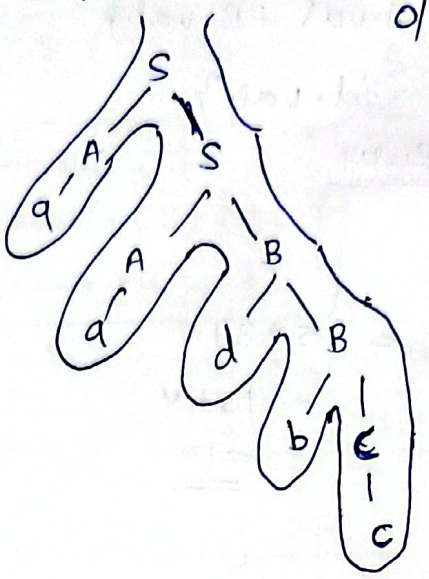
i/p: aadbc



o/p: 3 3 6 4 5 2 1

Bottom up

o/p: 3 3 6 4 5 2 1



SDT

$E \rightarrow E \& T$ { E.val = E.val + T.val }

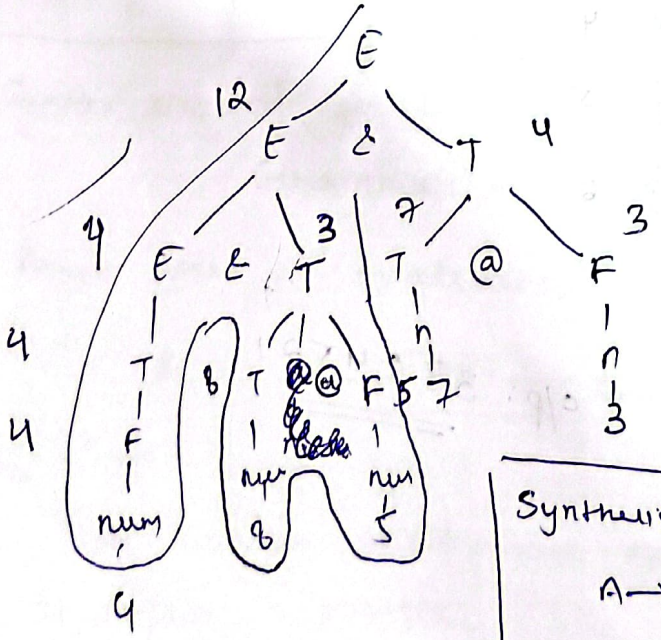
$\quad | T$ { E.val = T.val }

$T \rightarrow T @ F$ { T.val = T.val + F.val }

$\quad | F$ { T.val = F.val }

$F \rightarrow num$ { F.val = num }

i/p: 4 & 8 @ 5 & 7 @ 3



= 48

= 12 x 4

= 48

Synthesised attribute –

A → BCD A = child

Inherited –

A → BCD

Parent + sibling

Types of SDT

S-attributed	L attributed
Based on synthesised	Based on synthesised and inherited attribute
Use bottom up Parsing	Top down parsing
semantic rule always written on rightmost position in RHS.	semantic rule anywhere in RHS

Eg

A → LM { L.i = l(A.i) ; M.i = m(L.s) , A.s = f(M.s) } (In, In, Syn) — L

A → QR { R.i = r(A.i) ; Q.i = q(R.s) ; A.s = f(Q.s) } (In, In, Syn.) — Not L

Intermediate code generation -

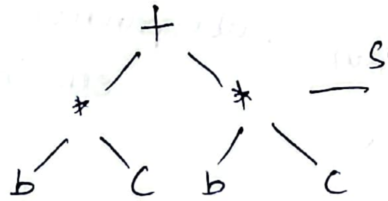
- machine independent
- Abstract syntax tree
- Directed Acyclic graph
- Postfix
- 3- address (Quadruple, Triple)

} methods to generate ICG

Intermediate code can work on different platform.

most famous - 3 address code.

(b * c) + (b * c)



— Syntax tree

DAG representation

postfix → bc* bc* +

3 address

$t_1 = b * c$
$t_2 = b * c$
$t_3 = t_1 + t_2$

} max three address.

Various operation

Assignment - $x = x \text{ op } z$ — $x = x + y$

$x = \text{op } y$ $x = +y \Rightarrow x = x + y$

$x = y$

relation operator

Jump - Conditional - if x relo y goto L if (x>y) goto L

Unconditional - goto L

Array assignment -

$x = y[i]$

$x[i] = y$

Pointer assignment = $x = \&y$

$x = *y$.

Code optimisation

Platform dependent technique

- peephole
- instruction lvl parallelism
- Data level parallelism
- cache optimisation
- Redundant resources.

Platform independent

- loop optimization (area)
  - loop jamming, reduction
  - loop unrolling
- constant folding
- constant propagation
- common subexpression elimination.

22/7 √2 = 1.41

3.14

$a = b * c$

$x = b * c$ calc again

loop optimization

1) code motion (frequency reduction)

```
a=100
while (a>0)
{ x = y+z;        } Not changing can be shifted.
  if (a-x==0)
  print ("y-d", a);
}
```

2) loop fusion

```
int i, a[100], b[100]

for(i=0; i<100; i++)
    a[i]=1;
for (i=0; i<100; i++)
    b[i]=2;
```

3) loop unrolling

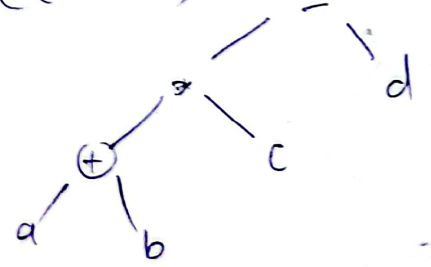
```
for (i=0; i<5; i++)
    printf("variety");
```

C

```
fy()
fy()
fy()
```

5 times

((a+b) * c) - d



infix - a+b*c-d

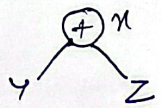
post - ab+c*d-

pre - -*+abcd

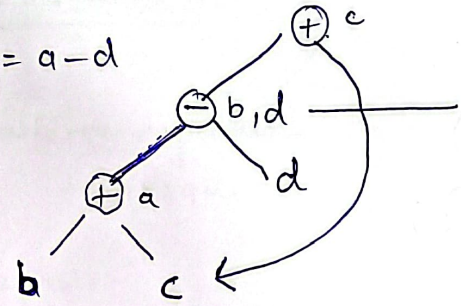
DAG representation of basic blocks

- Directed acyclic graphs
- determine common subexpression
- leaves are labelled by variable name or constant- initial values subscripted with 0
- interior node with 0
- internal node also represent- results of expression.

$x = y + z$

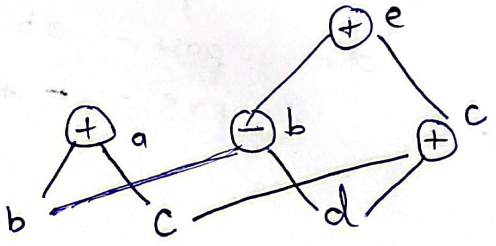


Ex- $a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$

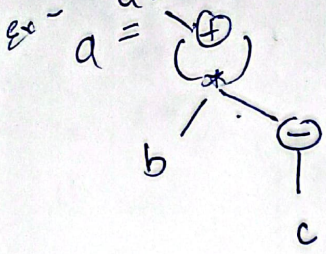


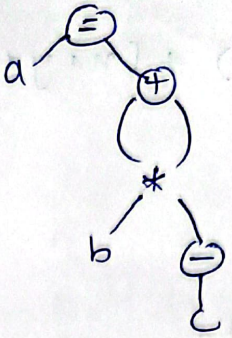
common subexpression.

Ex $a = b + c$
$b = b - d$
$c = c + d$
$e = b + c$



Ex - $a = b * -c + b * -c$





Quadruple ICG - arg1, arg2, op, result

$a = b * -c + b * -c$

0 $t_1 = -c$
1 $t_2 = b * t_1$
2 $t_3 = -c$
3 $t_4 = b * t_3$
4 $t_5 = t_2 + t_4$
5 $a = t_5$

	op	arg1	arg2	result
0	un(-)	c		t1
1	*	b	t1	t2
2	un(-)	c		t3
3	*	b	t3	t4
4	+	t2	t4	t5
5	assign	t5		a.

Triple - op, arg1, arg2.

	op	arg1	arg2
0	un(-)	c	
1	*	b	(0)
2	un(-)	c	
3	*	b	(2)
4	+	(1)	(3)
5	Assign	a	(4)

quad and triple

$(x+y) * (y+z) + (x+y+z)$

$t_1 = x+y$

$t_2 = y+z$

$t_3 = t_1 * t_2$

$t_4 = t_1 + z$

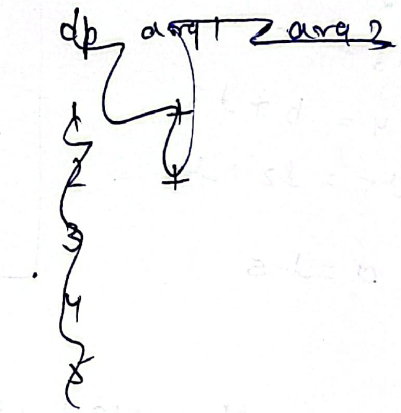
$t_5 = t_3 + t_4$

Quad

	op	arg 1	arg 2	result
0	+	x	y	t_1
1	+	y	z	t_2
2	*	t_1	t_2	t_3
3	+	t_1	z	t_4
4	+	t_3	t_4	t_5

Triple

	op	arg 1	arg 2
0	+	x	y
1	+	y	z
2	*	0	1
3	+	0	z
4	+	2	3



9) int i

j=1

while [i<10 do

if x>y then

i=x+y

else

i=x-y

3ac.

1	i=1
2	if i<10 goto (4)
3	goto(10)
4	if x>y goto (6)
5	goto(8)
6	i=x+y
7	goto(2)

```
8    i = n-4
9    goto (2)
10   stop
11
```

8-
```
c = 0
do
{
    if (a < b)
        n++
    else
        n--
    c++;
} while (c<5);
```

```
1    c = 0
2    if (a<b) goto (4)
3    goto (7)
4    n = n+1
5    c = c+1
6    goto (10)
7    n = n-1
8    c = c+1
9    goto (10)
10   if (c < 5) goto (2)
11   goto (12)
12   stop
```

more translation

1) Array reference in arithmetic
2) procedure call
3) case statement.

1-D
2-D

Address of A[i] = B + w * (i - LB)
(w: width, LB: lower bound)

B = 1100
LB = 0
w = 4

A[4] = 1100 + 4 x (4-0)
= 1116

= B + 4*i

```
int i
int a[10]

i=1;

while (i<10)
{
   a[i]=0
     i=i+1
}
```

```
1   i=1
2   if (i<10) goto (4)
3   goto (8)
4   J1 = 4*i
5   a[J1] = 0
6   i = i+1
7   goto(2)
8   stop
9
10
```

two dimension array

A[3][4]

M N

row major representation

column major representation.

Lr \ Lc	0	1	2	3
0	8	6	3	2
1	4	5	9	1
2	6	3	2	4

8 6 3 2 4 5 9 1 6 3 2 4 row

8 4 6 6 5 3 3 9 2 2 1 4 column...

$A[i][j] = B + w * [N*(i-Lr) + (J-Lc)]$

$= B + w * [(i-Lr) + m(J-Lc)]$

B+4 * (10*(i-0)+i)

BT = 4[11i]

= B+ 44i

8

```
int i
int a[10][10]
j=0
while (i<10)
{a[i][i] = 1;
   i++;
}
```

```
1   i=0
2
2   if (i<10) goto(4)
3   goto (8)
4   J1 = 44*i
5   a[J1] = 1
6   i=i+1
7   goto(2)
8   stop
```

case statement

```
switch (ch)
{ case 1 : c = a+b
           break;
  case 2 : c = a-b
           break;
}
```

```
1   if ch=1 : goto(3)
2   if (ch=2  goto(5)
3   c=a+b
4   goto(7)
5   c=a-b
6   goto(7)
7   stop
8
9
10
```

Procedural call - P(A1, A2, -- An)

Param A1

:

Param An

call P, n

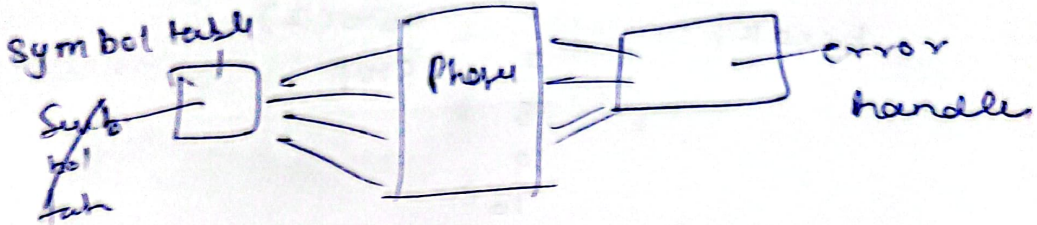
```
void main()
{ int x, y;
  swap(&x, &y)

  void swap(int *a, int *b)
  { int i;
    i = *b
    *b = *a;
    *a = i;
  }
```

```
1   call main
2   Param &x
3   Param &y
4   call swap, 2
5   i = *b
6   *b = *a
7   *a = i
8   stop
9
10
```

Symbol table

It is a data structure created and maintained by compiler in order to store information about variable, function, class, object etc.



Format to store -

1) data type, Name, scope

address, other attributes.

Eg.

SN	Name	Type	attr.
1	a	int	static
2	b	float	

static int a
float b.

Representation - fixed length
variable length.

Operation

insert - insert(a, int)
lookup - lookup(a)
delete - delete(a)

Scope management - local and global.

```
int value;
void one()
{
    int a;
    int b;
    {
      int c;
      int d;
    }
    int e
      { int f;
        int g;
      }
}
void two()
{ int n;
  int y;
    { int p;
      int q;
    }
    int r;
}
```

Symbol table

Val	var	type
one	procid	void.
two	Proc	void.

a	var	int
b	var	int
e	var	int

n	var	int
7	var	int
r	var	int

c	var	int
d	var	int

f	var	int
g	var	int

p	var	int
q	var	int

implementation

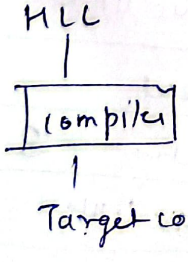
1) linear list -- array -

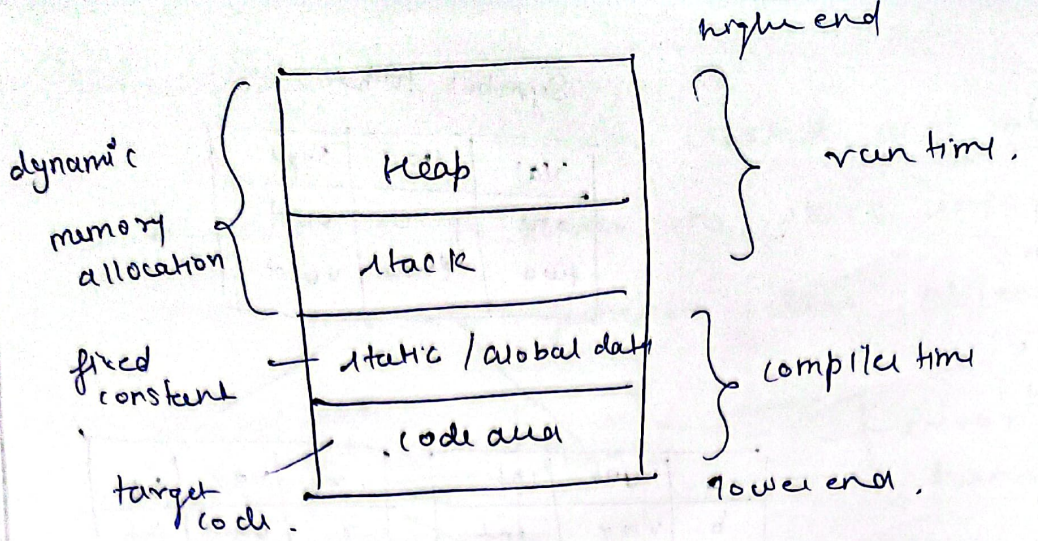
2) self organising list - linked list.

3) Binary search tree.

4) Hash map / table

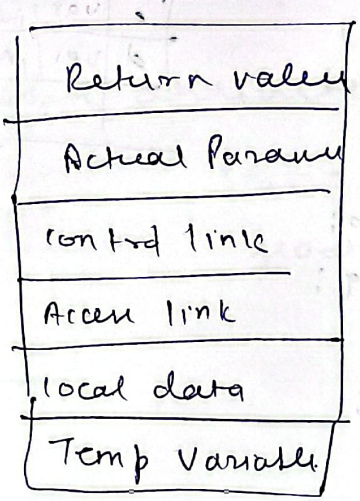
Runtime storage Administration



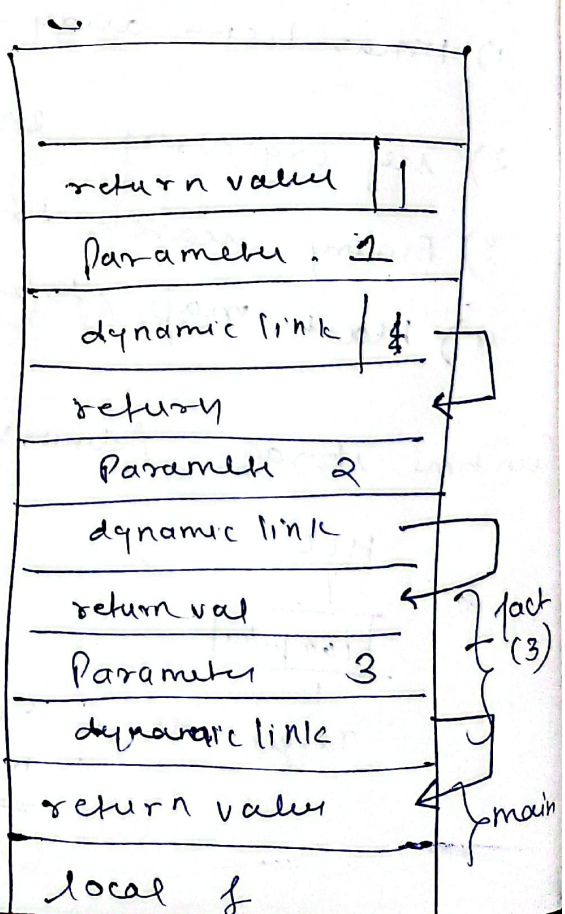


simple stack allocation Administration.

Activation record for each function.



Q - main()
{ int f
f = fact(3);
}
int fact (int n)
{
if (n==1)
return 1
else
return n * fact(n-1)
}.



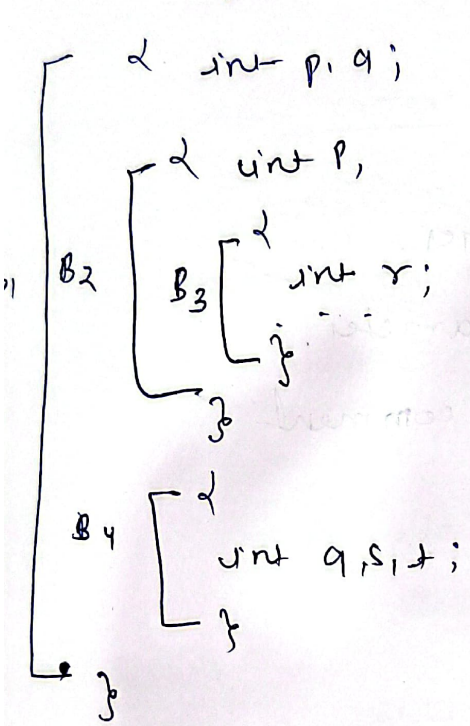
Block structured

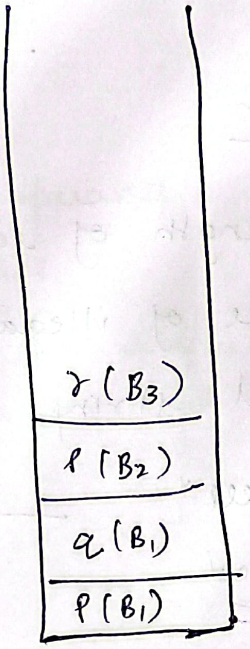
local data	Non local
A( ) { int a; }	test( ) { int a, b; { int x, y } { int c, d } }

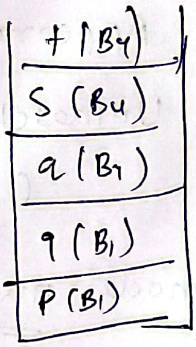
Storage

- local → AR
- Non-local → Static → Block structured storage
- Non-local → dynamic → Non-Block dto seg.

eg. compiler( )



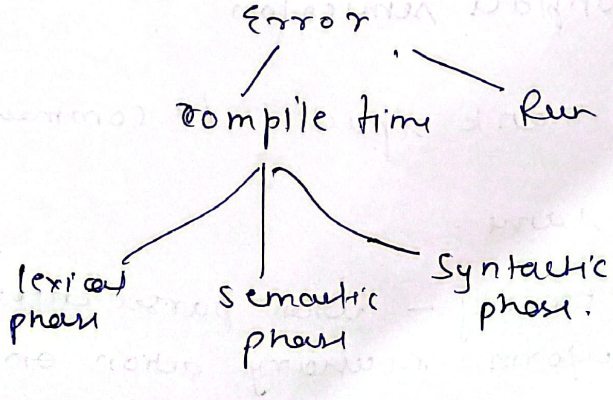




Error detection and recovery

(connected with all phases –

function – error detection
Error Reporting
Error recovery.



Recovery method	lexical	Syntactic	Semantic
Panic mode	✓	✓	✗
Phase level	✗	✓	✗
Error production	✗	✓	✗
Global ~~production~~ correction	✗	✓	✗
with symbol table	✗	✗	✓

lexical phase error

i) Exceeding length of identifier.
ii) Appearance of illegal character.
iii) Unmatched string or comment

C

(here "

____ * /

Panic mode method

$$;

Syntactic phase error –

i) missing paranthesis.
2) missing operator.
3) misspled keyword.
4) colon inplace semicolon
5) Extra Blank space. /* commen _ /

1) Panic mode – same.
2) Phase level recovery – when parser encounter error it perform necessary action on

Error production - Add extra grammar and make augmented grammar and parse input

global correction - parser examine whole program and tries to find closest match for it which is error free it is not practically implemented.

Semantic phase error -

i) Incompatible type of operand

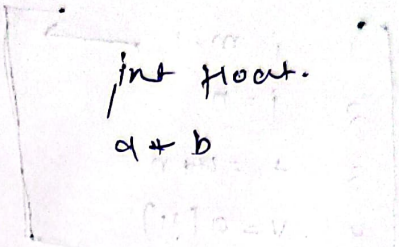
ii) Undeclared variable

iii) Not matching actual argument with formal argument.

```
(
  int a[10] = b;
      a = b;
```

Recovery

Symbol table -



remaining input and parser rest input.

Basic blocks

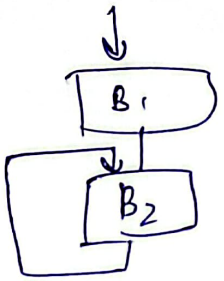
```
begin
  prod=0
  i=1
  do
  begin
     prod = prod + a[i] * b[i]
     i=i+1
  end
  while (i<=20)
end
```

1	prod = 0
2	i = 1
3	$t_1 = 4 * i$
4	$t_2 = a[t_1]$
5	$t_3 = 4 * i$
6	$t_4 = b[t_3]$
7	$t_5 = t_2 * t_4$
8	prod = prod + t5
9	i = i+1
10	if i<=20 goto (3)

first statement

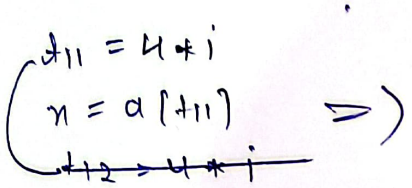
goto (1)

condition



flow graph.

common subexp.

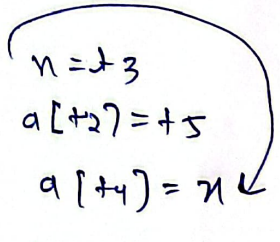


$t_{11} = 4 * i$

$n = a[t_{11}]$;

$a[t_{11}] = y$

Copy propagation



dead code elimin

$a[t_2] = t_5$

$a[t_4] = n$.

constant folding - $\pi r^2 = 3.14 * r * r$.