(3.) Indirect Triples

				(0)	(11)
S.No.	op	arg1	arg2	(1)	(12)
(0)	*	a	b	(2)	(13)
(1)	*	c	d	(3)	(14)
(2)	+	(11)	(12)		
(3)	=	x	(13)		

Syntax Directed Definitions

The value of an attribute at a parse tree node is defined by a semantic rule associated with the production used at that node.

The value of a synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree.

The value (type) of an inherited attribute is computed from the values (types) of attributes at the siblings and parent of that node.

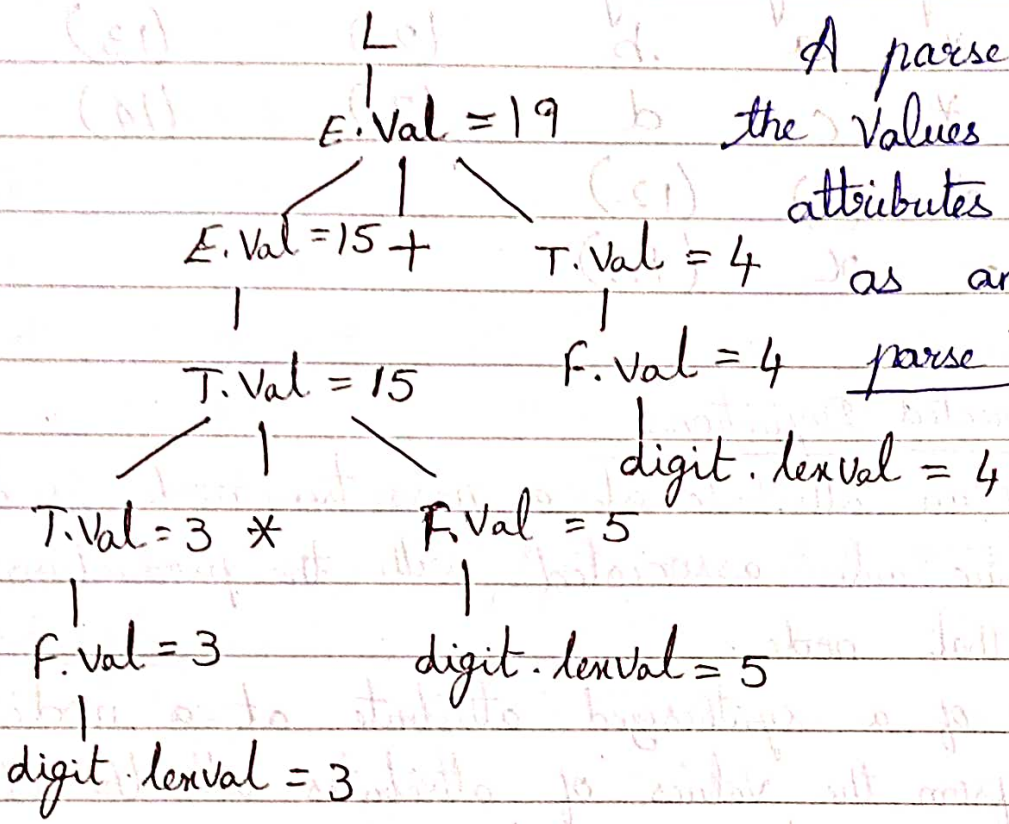
$L \rightarrow E$	print (E.val)
$E \rightarrow E_1 + T$	E.val $E.Val \rightarrow E_1.Val + T.Val$
$E \rightarrow T$	$E.Val \rightarrow T.Val$
$T \rightarrow T_1 * F$	$T.Val \rightarrow T_1.Val * F.Val$
$T \rightarrow F$	$T.Val \rightarrow F.Val$
$F \rightarrow (E)$	F.val $\rightarrow E.Val$
$F \rightarrow digit$	$F.Val \rightarrow digit.lexVal$

Syntax Directed Definition or Attributed grammar

NOTE

NOTE A syntax directed definition that uses only synthesized attributes is said to be S- attributed definition

3 * 5 + 4

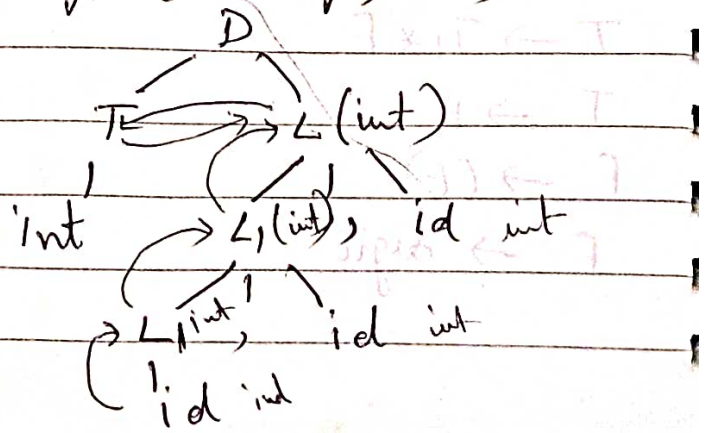


A parse tree showing the values of its attributes is known as an annotated parse tree.

Inherited attributes

$D \rightarrow T\ L$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in$ addtype(id.entry, L.in)
$L \rightarrow id$	addtype(id.entry, L.in)

for int id1, id2, id3



Dependency graph The interdependencies among the inherited & synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called dependency grpl.

Syntax Trees

a-4+c p1 = mkleaf(id, entry a)

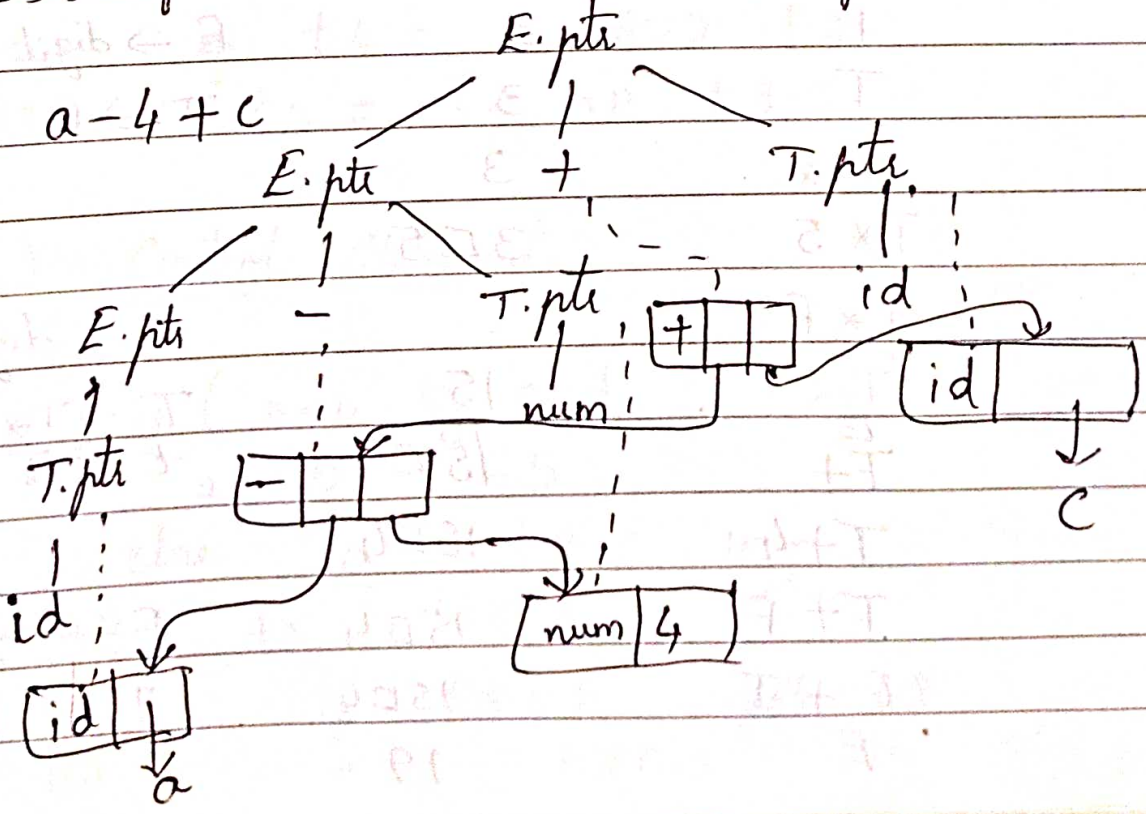
p2 = mkleaf(num, 4), p3 = mknode('-', p1, p2)

p4 = mkleaf(id, entry c), p5 = mknode(+, p3, p4)

Ex

$E \to E_1 + T$	E.ptr = mknode('+', E_1.ptr, T.ptr)
$E \to E_1 - T$	E_1.ptr = mknode('-', E_1.ptr, T.ptr)
$E \to T$	E.ptr = ~~Tnknode~~ T.ptr
$T \to (E)$	T.ptr = E.ptr
$T \to id$	T.ptr = mkleaf(id, id.entry)
$T \to num$	T.ptr = mkleaf(num, Value)

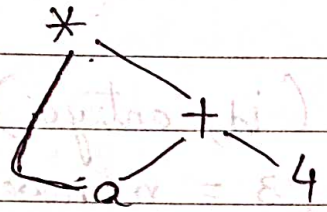
SDD for CFG to Construct syntax tree.



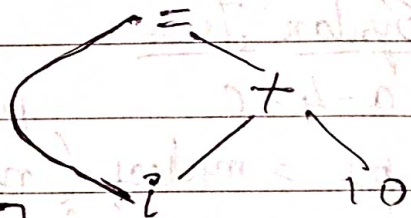
Directed Acyclic Graphs (DAG)

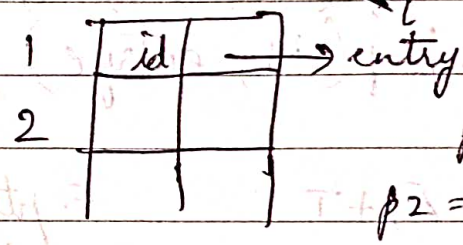
Any node in a DAG representing a common subexpression has more than one parent.

a * a + 4



i = i + 10





p_1 = mkleaf(id, i)

p_2 = mkleaf(num, 10)

p_3 = mknode(+, p_1, p_2)

p_4 = mknode(=, p_1, p_3)

Bottom up Evaluation of S-attributed Definitions on Parser stack

I/p	State	val	Prod. used
3*5+4			
*5+4	3	3	
*5+4	F	3	F → digit
*5+4	T	3	T → F
5+4	T*	3	
+4	T* 5	3□5	
+4	T*F		F → digit
+4	T	15	T → T*F
+4	E	15	E → T
+4	E+	15□	
	E+4	15□4	
	E+F	15□4	F → digit
	E+T	15□4	T → F
	E	19	

$= 100 + 5040 * 4 = 100 + 20160$

$= 20260.$

Type Checking

Type Expressions

The systematic way of expressing type of language construct is called type expression.

Some type expressions

(1.) Array array(I, T)

I – index, T – type

Eg. int a(100) array(100, int)

(2.) structure

struct ((membername type) X (m2name type) X ...)

Eg. struct stud

{ char name[10];

float avg;

} student[10];

type exp. for struct stud: struct ((name X array(10, char)) X (avg float))

TE for student is array(10, stud)

(3.) Pointer

type
Pointer (float)

Ex. float * p; TE for p is pointer (float)

(4) Functions

Domain → Range is the TE for functions.

(or) Parameters → Return Type

Ex. int sum (int a, int b)

TE for sum is int x int → int

Ex. int max (int a, int b, float * c)

TE for max is int x int x pointer (float) → int

Type conversions

Implicit type conversion or coercion, no loss of inf.
i.e., from int to float (done by compiler)
Explicit type conversion, should be done by users.

$E \rightarrow E_1$ op E_2

$E.type$ = if $E_1.type$ = int & $E_2.type$ = int
then int
= if $E_1.type$ = int float
then float
= if float int
then float
= if float float
then float

Type checker Type checker is a translation scheme in which the type of each expression from the types of its subexpressions.

Type checking of Expressions

$E \rightarrow$ literal $\{E.type = char\}$

$E \rightarrow$ digit $\{E.type = int\}$

$E \rightarrow$ id $\{E.type = lookup(id.entry)\}$

$E \rightarrow E_1 \bmod E_2$ $\{E.type =$ if $E_1.type = int$ & $E_2.type = int$ then int else type-error $\}$

$E \rightarrow E_1$ op E_2 $\{E.type =$ if $E_1.type = int$ & $E_2.type = int$ then int else type error $\}$

In place of op we can use $+, -, *, /, \%$

Array reference

$E \rightarrow E_1[E_2]$ $\{E.type =$ if $E_2.type = int$ & $E_1.type = array(range, t)$ then t else type-error $\}$

Pointer

$E \rightarrow *E_1$ $\{E.type =$ if $E_1.type = pointer(t)$ then t else type error $\}$

Function call

$E \rightarrow E_1(E_2)$ { if E_1.type = $s \rightarrow t$ then t
else type error }

Equivalence of Type Expressions

Any two expr.s are said to be equivalent if they have same basic type or formed by applying the same constructor.

Ex: Char S_1 — S_1 & S_2 are equivalent as both have same basic type.
Char S_2

Ex: char * S_1 — pointer (char) — S_1 & S_2 are equivalent as same constructor is applied
char * S_2 — pointer (char)

Alg for Equivalence

```
struct-eq (x, y)
{ if x & y have same basic type
  then return true
  else if (x = array(x1, x2) and y = array(y1, y2))
       then return (struct-eq(x1, y1) and
            struct-eq.(x2, y2))
  else if x = pointer(x1) & y = pointer(y1)
       then return struct-eq (x1, y1)
  else if x = x1 → x2 & y = y1 → y2
  then return (struct eq (x1, y1) & struct eq(x2, y2))
  else return false }
```

Unit - V

Symbol Table It stores inf. about scope, binding, type, value etc of about names (identifiers)

Attributes of Symbol Table

Variable names, constants, Data Types, compiler generated temporaries, Function names, Parameter names, Scope inf.

Ordered Symbol Table

Here, entries of variables is made in alphabetical order. Searching of particular variable is efficient.

Insertion – Difficult, Searching – easy

Unordered Symbol Table

whenever a variable is encountered, then its entry is made in symbol table without any ordered manner.

Insertion – easy, Searching – Difficult

Name Representation

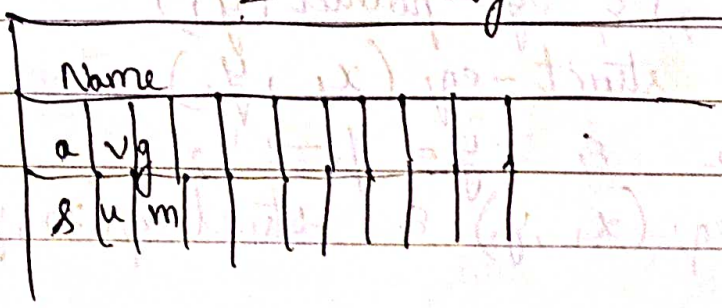
(1.) Fixed length Name

A fixed space for each name is allocated in symbol table.

Disadv: wastage of space.

LC	Type	Value
100	2	sum
101	2	avg

Value is nothing but name



(2.) Variable Length Name

The amount of space required to store the names is stored i.e., starting index & length of each name.

starting index	Length
0	4
4	4
⋮	⋮

a	v	g	$	s	u	m	$	...
0	1	2	3	4	5	6	7	...

Data Structures used in Symbol Table (SymbolTable Mgmt)

1) Linear List

(i) New names can be added as they enter/arrive.
(ii) "Available" pointer specifies that there is an empty slot.
(iii) To search for an item we start from beginning.
(iv) while inserting we should ensure that it should not be already th..

Name & associated inf.

Name1	Info1
Name2	Info2
⋮	⋮

available

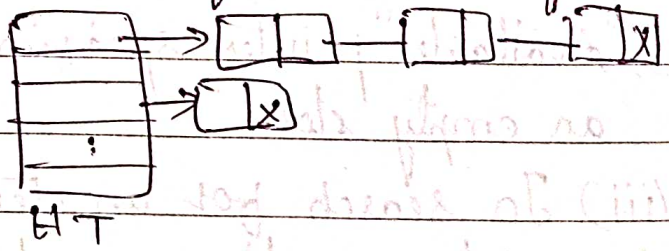
3) Binary Trees

Left child	Symbol	Info.	Right child

Symbol field is used to store the name of the symbol.
Inf. field is used " " inf. about " "
Left child field stores the addr. of prev. symbol.
Right child " " " " " next " .
Efficient but requires more space.

4) Hash Tables

(i) Two tables (hash & symbol) are used.
(ii) position = h (name) Now, "position" gives the exact position of the name in symbol table.
(iii) Very quick search, but more space is required & hash func. calculation is also complicated.



Storage Allocation Strategies

(3.) Data Structures cannot be created. used.	(3.) DS & Data objects can be created dynamically.	→ "
(4.) Simple & Fast	(4.) complex & slow	(4.) complex & fast, Efficient
(5.) Recursive procedures are not supported.	(5.) Supports recursive procedure calls.	(5.) → "

Activation Record

It is the inf. stored in mem. for execution of a single procedure.

Temporaries : All the temp. variables generated by the compiler.

Local variables : All the variables that are local to the procedure.

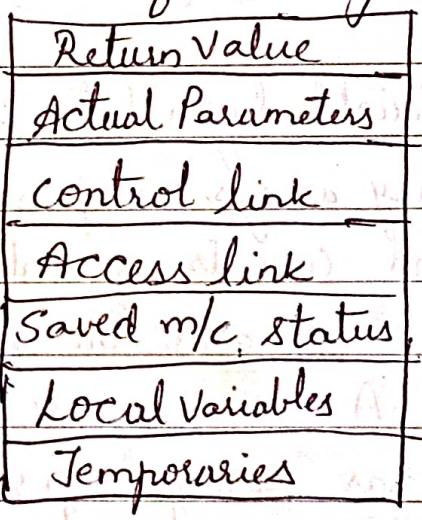
Saved m/c status : Before the procedure is called some control inf. should be stored in the reg.

Access link (optional) : It refers to non local data in other activation records.

Control link (optional) : It points to the activation rec. of the calling procedure.

Actual Parameters : It holds the values of actual parameters

Return Value : It stores the result of the procedure call.



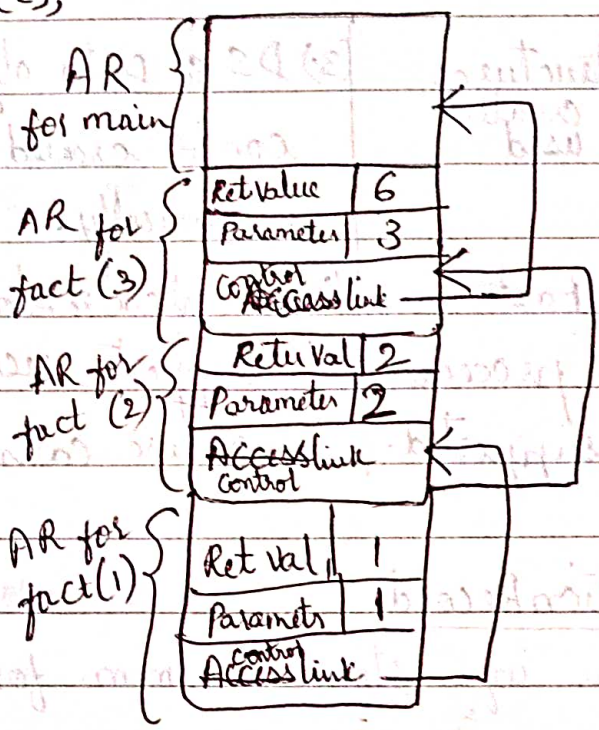
Ex! main()
{
=
x = sum(2,3);
=
}
sum(a,b)
{ c = a+b;
ret c;
}

main()
{
=
sum(2,3);
=
}
sum(a,b)
{ c = a+b;
display(c);
}

display(c)
{ Pf(c);
}

Ex. Factorial

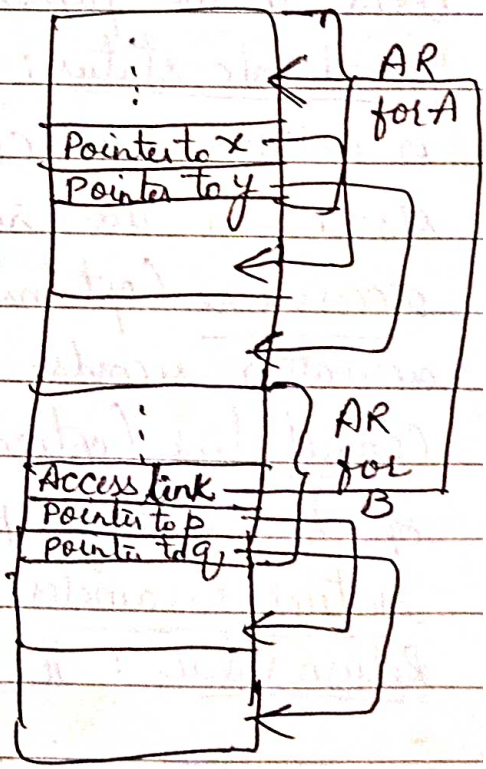
```
main()
{ int f;
  f = fact(3);
}
int fact (int n)
{ if (n == 1)
  return 1;
else return (n*fact(n-1));
}
```



Variable length data :

If any arrays are present in procedures, then the activation record consists of only pointers to these arrays. The arrays are stored after the AR.

Ex. A has arrays x & y & A calls B.
B has 2 arrays p & q.



Access to Non Local Names

(1.) Static scope or lexical scope — Access link, Display

(2.) Dynamic scope — Deep access, Shallow access

Static scope

Set of instr. enclosed within { } are known as block.

```
Ex()
{
    int p,q;
    {
        int p;
        {
            int r;
            =
        }
        =
        =
    }
    {
        int q, s, t;
        =
    }
    =
}
```

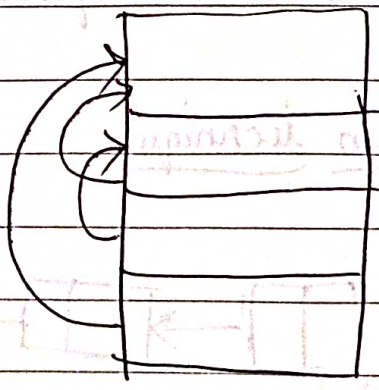
B1 = Ex(), B2 = outer block, B3 = inner block, B4 = second block

Nesting Depth: (ND)

whenever "{" is encountered $ND = ND + 1$ & where "}" is encountered $ND = ND - 1$.

①	②	③	④	⑤	⑥	⑦
p(B1)	p(B1)	p(B1)	p(B1)	p(B1)	p(B1)	p(B1)
q(B1)	q(B1)	q(B1)	q(B1)	q(B1)	q(B1)	q(B1)
	p(B2)	p(B2)	p(B2)		q(B4)	
		r(B3)			s(B4)	
					t(B4)	

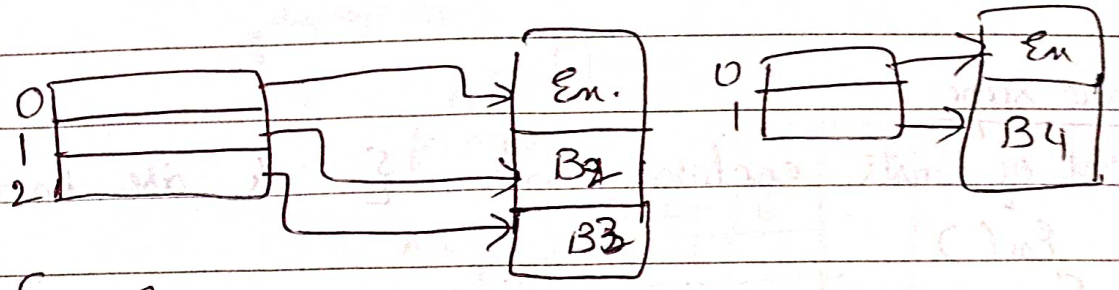
(1) Access Link



Disadv: If ND is large, we have to follow long chains of links to reach the data we need.

An auxiliary array "d" is used in Display.

2. Display

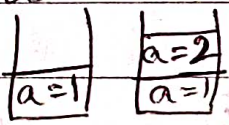


Ex. P0 [P1 [P2 [P3 [P4 [

Write both access links & displays.

Dynamic Scoping

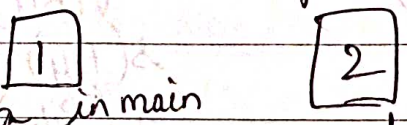
Deep Access stack is maintained & by popping the top most element of stack we get the most recent value for that Symbol.




```
main()
{ int a=1;
  pf(a);
  first();
}
first()
{ int a=2;
  pf(a);
}
```

Shallow Access

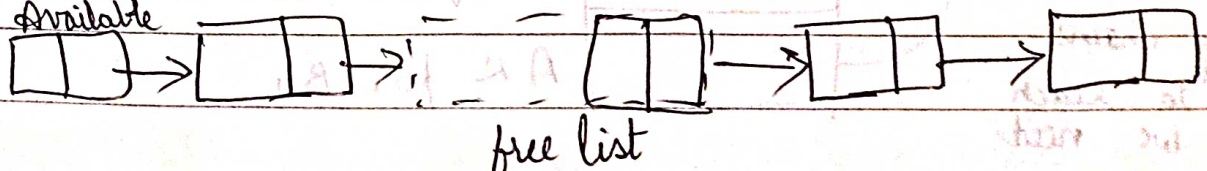
Central storage is kept with one slot for every variable.



Again the previous value of "a" is restored when control leaves "first" func.

Dynamic storage Allocation techniques

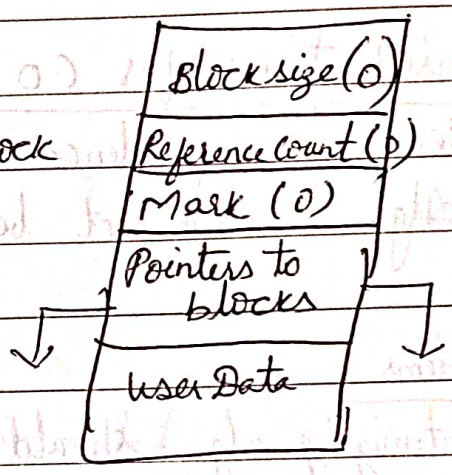
(1.) Explicit Allocation



DeAllocating mem. means free list gets appended.
Allocating mem. means we use free list.

Implicit Allocation

Block size — size of the block
Reference count — How many times that block is referred.
Mark — Presently if that block is used (called) by any other block then mark it.
Pointers to all other blocks that this block is referring.



UNIT- VI CODE OPTIMIZATION

Code optimization Advantages (CO)

(1.) To produce efficient object code.
(2.) Faster execution of source prog. (by making minor modifications)

Considerations for CO

(1.) Semantic equivalence should ~~be~~ not be changed.
(2.) Alg. should not be changed.

Problems

(1.) optimizing alg. should be efficient enough such that the target code should not become time & space inefficient
(2.) Design of intermediate code must be simple enough to optimize the code.

CO
- (1) M/c Dependent opt.
- (2) M/c Independent opt.

(1) Based on char. of target m/c for instr. set & addressing modes.
(2) Based on char. of prog. lang.

Folding The computation of constant is done at compile time instead of execution time.

Ex $a = (17/4) * x$

Evaluated at compile time rather than execution time.

Constant Propagation The value of the variable is replaced & computation of expr. is done at compilation time.

Eg `a = 5; pi = 3.14; area = pi * a * a;`

Common Sub-Expression Elimination

An expr. which is computed previously can be eliminated.

Eg
```
t1 = 4 * i
t2 = a[t1]
t3 = 4 * j
t4 = 4 * i   — can be eliminated.
t5 = n;
t6 = b[t4] + t5
```

Variable Propagation No need of using 2 variables for one value.

`x = pi; pi = 3.14; a = x * r * r;`

It can be written as `a = pi * r * r;`

Code Movement

We can move some code out of loop so as to reduce time & space.

Eg.
```
for (i = 0; i < 10; i++)
{
    x = y * 5;
    K = (y * 5) + 10;
}
```
⟹
```
j = y * 5;
for (i = 0; i < 10; i++)
{
    x = j;
    K = j + 10;
}
```

Loop Invariant

```
for(i = 1; i<100; i++)
{ a[i] = b[i] + i + t*5/x;
}
```

⇒

```
t = t*5/x;
for (i=1; ——>)
{ a[i] = b[i] + i + t;
}
```

Code movement

Loop Invariant Method

Ex.

```
while (i <= max-1)
{ =
}
```

⇒

```
n = max-1;
while (i<=n)
{ =
}
```

Loop Invariant & Code Movement are same known as Loop Invariant Code Motion

Strength Reduction & Induction variable elimination

Higher strength operators can be replaced by lower strength operators. * by + & / by −.

Ex

```
for(i=1;i<10;i++)
{ c = i*7;
}
```

⇒

```
t=7;
for (i=1; i<10; i++)
{c=t;
 t = t+7;
}
```

Here "t" is induction variable. # = V ± constant

Dead code Elimination

A variable is said to be dead if the value in it is never used. we can eliminate such code.

Ex:

```
i = 2;
j = i;
=
x = i+10;
=
```

Ex

```
i = 1;
if (i==0)
{ =
}
```

Loop optimization

If no. of instr. are less in inner loop then the running time of the prog. will get decreased drastically.

Loop optimization techniques:

Code Motion, Strength Reduction, Loop invariant, Induction variables, Loop unrolling & Loop fusion.

Loop unrolling

No. of tests & jumps can be reduced by writing the code two times.

Ex.

```
int i=1;
while (i<=200)
{ a[i]= b[i];
  i++;
}
```

⇒

```
int i=1;
while (i<=100)
{ a[i] = b[i];
  i++;
  a[i] = b[i];
  i++;
}
```

Loop Fusion

Several Loops are merged to one loop.

Ex.

```
for i = 1 to n do
  for j = 1 to m do
    a[i, j] = 10
```

⇒

```
for (i=1; i<n*m; i++)
  a[i] = 10;
```

Leader (Rules to find Leader)

(1.) The 1st stmt is a leader.

(2.) Any conditional or unconditional goto's target stmt is a leader.

(3.) Any stmt. that immediately follows a goto condition is a leader.

Rules for partitioning into Blocks

(1.) Identify leaders.

(2.) Basic block is formed starting at the leader stmt. & ending just before the next ladder stmt.

Ex.
```
s = 0;
i = 1;
do
{ s = s + a[i] * b[i];
  i = i + 1;
} while (i <= 10);
```

TAC is

1. $s = 0$
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. $s = s + t_5$
9. $i = i + 1$
10. if $i <= 10$ goto 3.

B_1

$s = 0$
$i = 1$

B_2

$t_1 = 4 * i$
$t_2 = a[t_1]$
⋮
if $i <= 10$ goto B2

Induction Variables

```
i=1 for ( )
{ i = i+1
  t1 = 4*i
```

i & t_1 are in lock state.

So, they are called induction variables.

```
i=1;
t1= 4*i.
for ( )
{ i = i+1
  t1 = t1+4
}
```

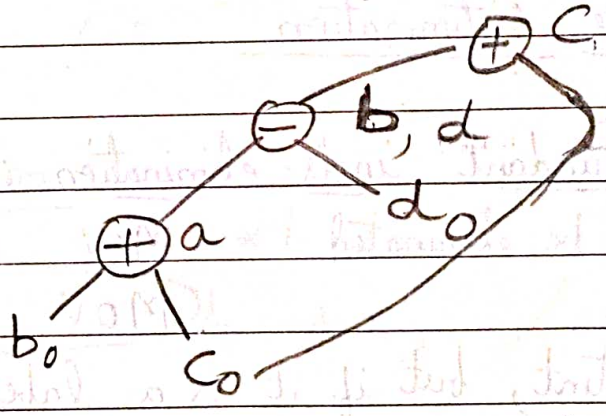
Ex doubt

Optimization of Basic Blocks

(1.) Structure Preserving transformation
(2.) use of algebraic identities

(1.) Structure Preserving transformations can be carried out by applying common sub expr. elimination, dead code elimination, variable & constant propagation, code mvmt.

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



Here, b+c & a-d are common sub expr. but we cannot eliminate a=b+c or c = b+c bcoz initially a=b+c then b is changed as b = a-d then c = b+c are computed. So, they cant be eliminated, whereas b = a-d is computed and a or d is not changed later, so we can eliminate d = a-d

(2) Use of algebraic identities

(i) Some algebraic identities can be used as

$a+0 = a$, $a*1 = a$, $a/1 = a$

(ii) strength reduction tech. can be applied to obtain algebraic transformation.

$2*a \Rightarrow a+a$

$a/2 \Rightarrow a*0.5$

(iii) constant folding tech, " "

$a = 2*5.4 \Rightarrow a = 10.8$

(iv) SW-exps. elimination " " "

$x = y*z$
$t = z*x*y$
$\Rightarrow$
$x = y*z$
$t = x*x$

Peephole optimization

(1.) Redundant instr. elimination : Redundant loads & stores can be eliminated.

MOV R0, x
MOV x, R0 we can eliminate this stmt, but if it is a label stmt. then we cannot eliminate it. Unreachable code :

Ex.
```
s = 0
if(s)
{
=
}
```

Ex.
```
int fun (int a, int b)
{ c = a+b;
  return c;
  pf("%d", c);
}
```

(2.) Flow of control optimization unnecessary jumps on jumps can be eliminated.

```
goto f1                    goto L1
  ≡                          ≡
f1: goto L1        ⇒       f1: goto L1
  ≡                          ≡
L1: =                      L1: =
```

(or)

```
if (a<b) goto L1           if (a<b) goto L2
  ≡                          ≡
L1: goto L2        ⇒       L1: goto L2
  ≡                          ≡
L2: =                      L2: =
```

(3.) Algebraic simplification

$x = x+0$ or $x = x*1$ can be eliminated

(4.) Strength Reduction * by +, / by −

(5.) Machine idioms

Increment or Decrement instr. to implement certain operations.

$i = i+1;$ ⟹ INC i

Unit - VII

Data Flow Analysis

Flowgraph It is a directed graph in which

(i) the nodes to the flow graph are basic blocks.

(ii) The block whose leader is the 1st stmt is initial block.

(iii) There is edge from B_1 to B_2 if B_2 immediately follows B_1 in the given sequence.

Dominator

In a flow graph, d is said to dominate n if every path to node n from initial node goes through d only.

(d dom n)

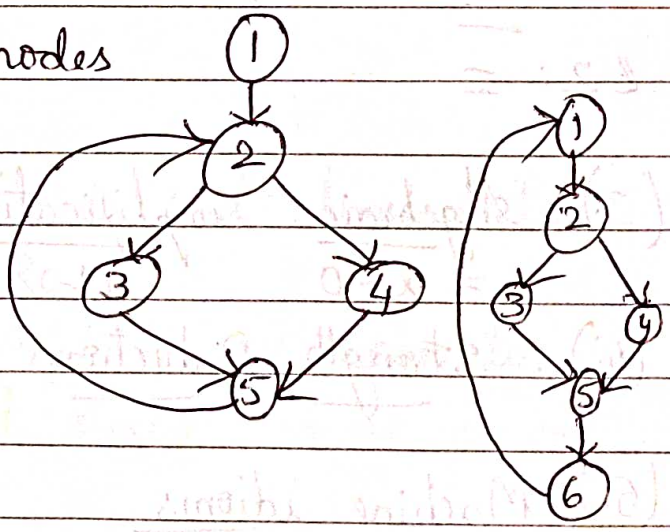
1 dominates all the other nodes

2 dominates 3, 4, 5.

3 dominates 5

4 dominates 5

5 dominates none.



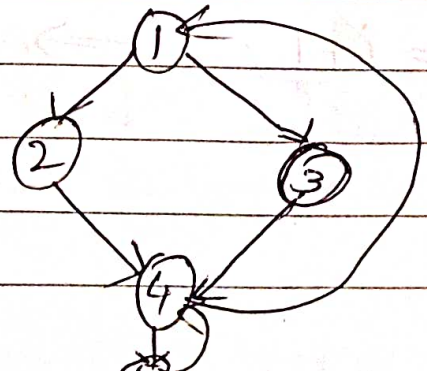
Natural Loop

Loop in a flow graph can be denoted by $n \rightarrow d$ such that d dom n.

$\{4 \rightarrow 1$, 1 dom 4

$\{5 \rightarrow 4$, 4 dom 5

$\rightarrow$ Back Edges



Code Generation Alg.

Ex $x = y \text{ op } z$

1. if (Addr. Descriptor (y) != R)
generate (MOV y, R_1) else R_1 = getReg(y)

2. if (Addr. Descriptor(z) != R)
generate (MOV z, R_2) else R_2 = getReg(z)

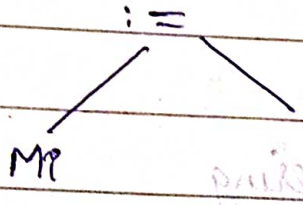
3. generate (OP R_1, R_2)

4. generate (MOV R_2, x)

Generic Code generation Alg.

I/P to the code generator will be sequence of trees. It uses tree rewriting rules (tree translation scheme).

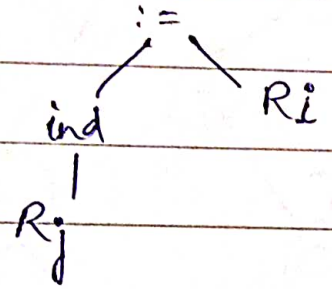
(1.)



op
dest src

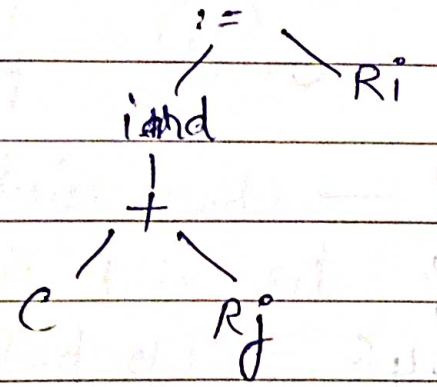
MOV Ri, Mi

(2.)



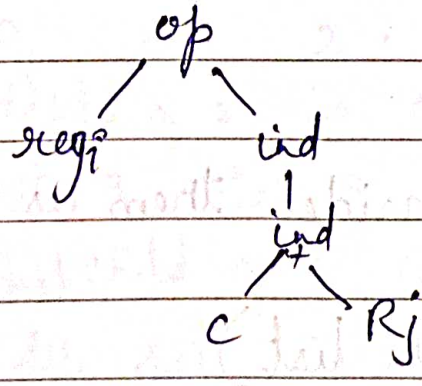
MOV Ri, *Rj

(3.)



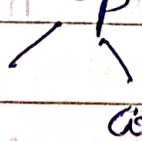
MOV Ri, c(Rj)

(4.)



MOV *c(Rj), reg i

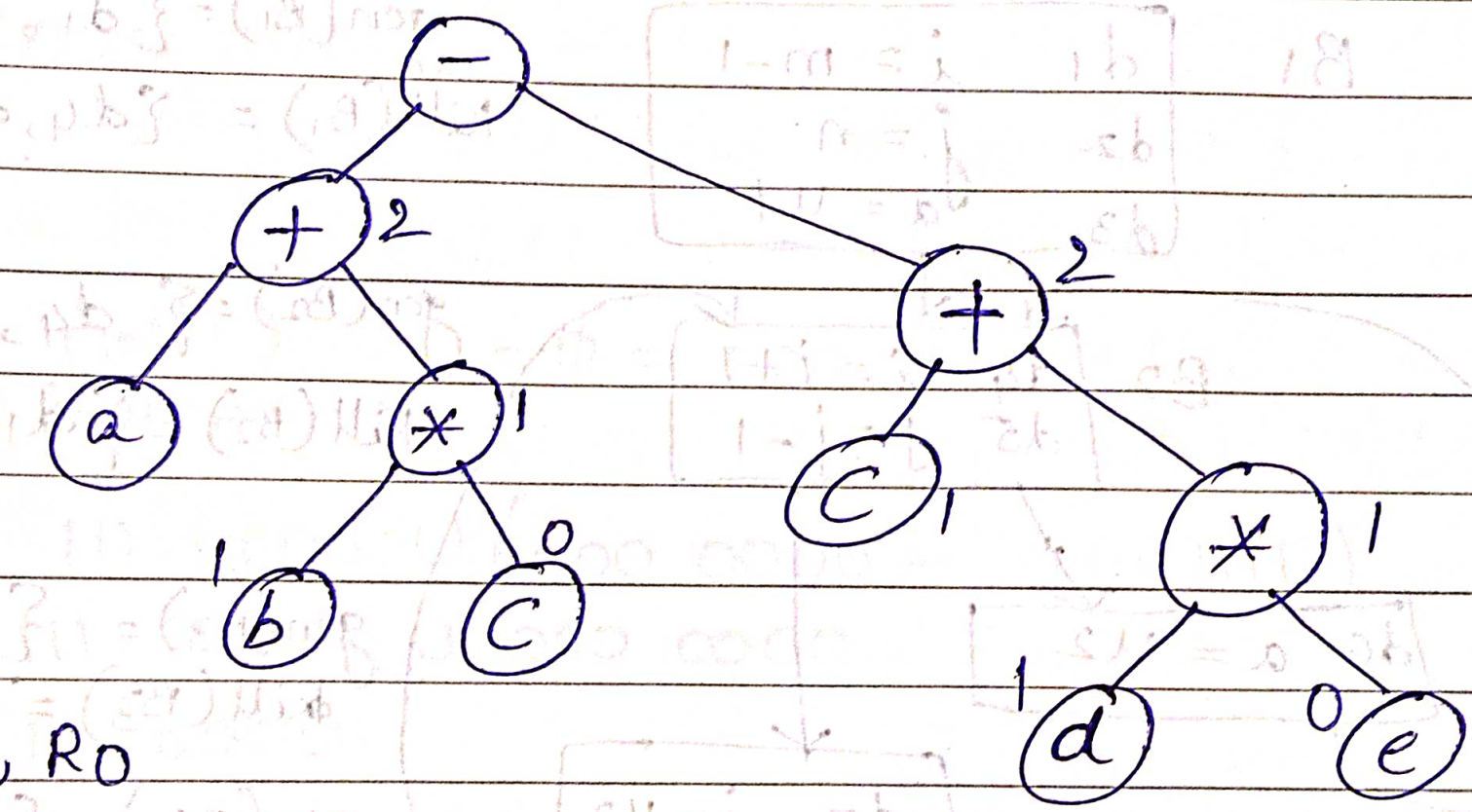
(5)



MOV const, Rj

op Rj, Ri

Ex.




```
MOV  b, R0
MUL  C, R0
MOV  a, R1
ADD  R1, R0
MOV  R0, T0
MOV  d, R1
MUL  e, R1
```

```
MOV  C, R0
ADD  R0, R1
SUB  R1, T0
```