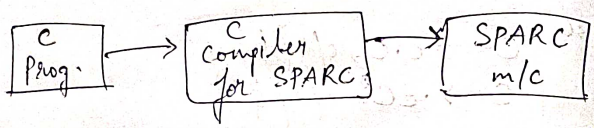
Unit - 4

6/5/22

Intermediate code Generation:-

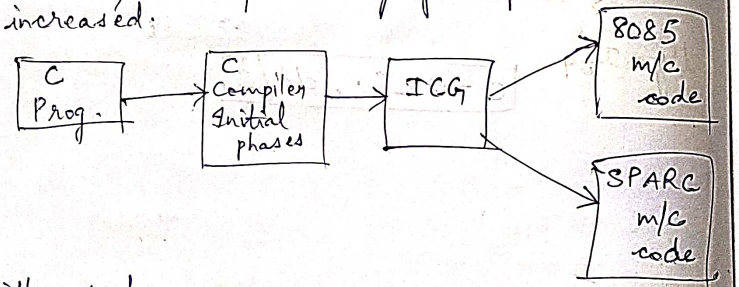
For any lang., a compiler is designed and if it is to be run on diff. machines then the entire compiler has to be reconstructed for the other machine.

ex:



If ICG, is introduced then there is no need of reconstruction of the compiler with all the phases for a new machine.

The code generated in IC is suitable for any machine, so the portability of compiler is increased.



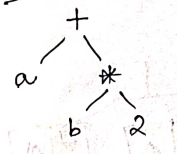
Different form of ICG:-

1) Abstract Syntax Tree
2) Reverse Polish Notation (Postfix) — ?
3) Three Address Code

1) Abstract Syntax Tree:

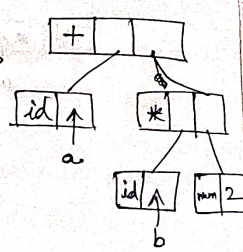
→ a+b*2

Parse Tree



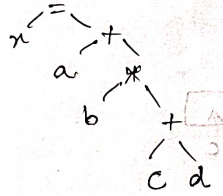
mknode(op, left, child)
mkleaf(id, entry)
mkleaf(const, value)
num

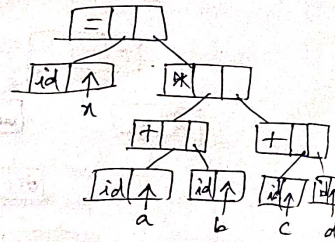
⇒



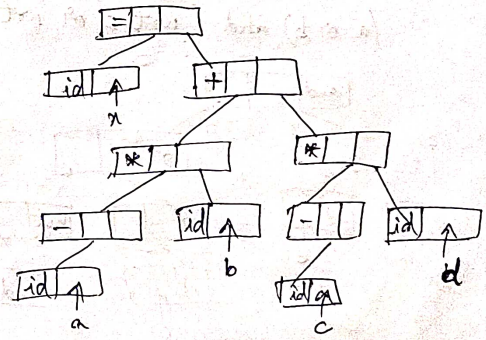
→ x = a+b*(c+d)

Parse Tree

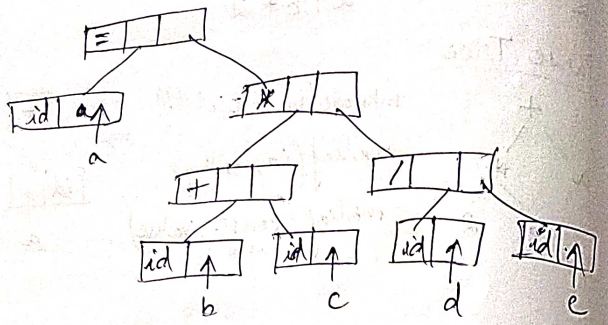




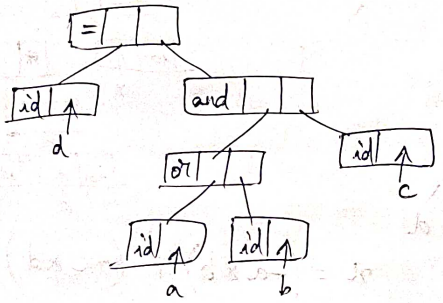
→ x = (-a*b) + (-c*d)



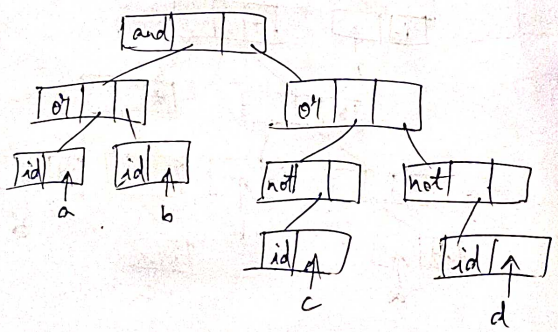
→ $a = (b+c) * (d/e)$



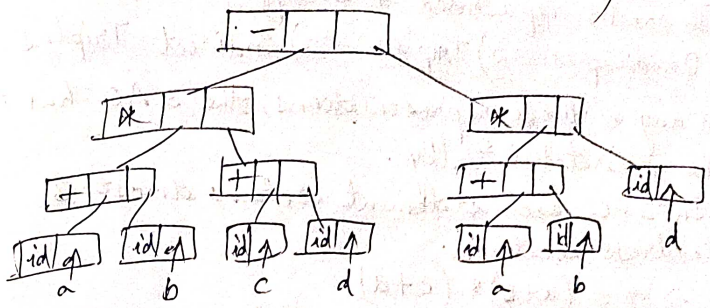
→ $d = (a \text{ or } b) \text{ and } c$



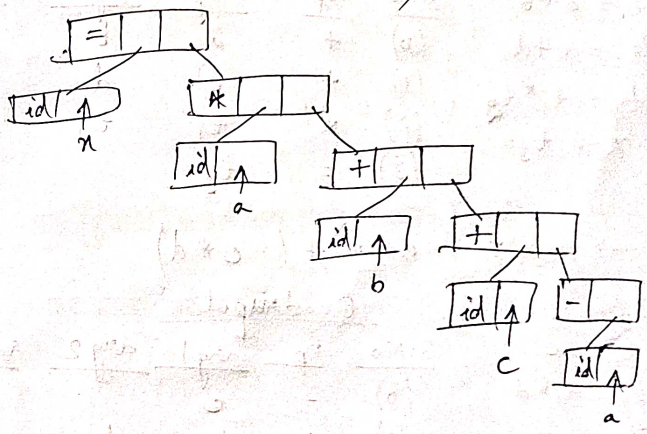
→ $(a \text{ or } b) \text{ and } (\text{not } c \text{ or not } d)$



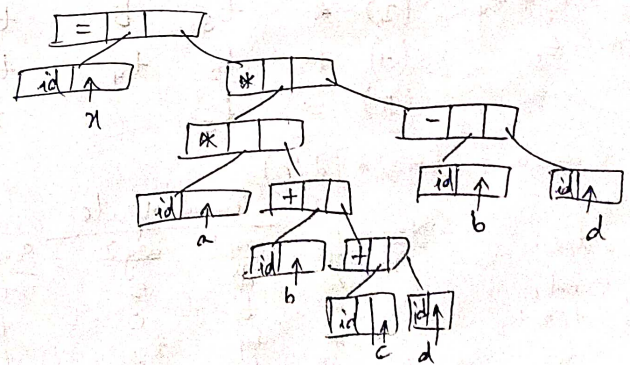
→ $((a+b) * (c+d)) - ((a+b) * d)$



→ $x = (a * (b+c+(-a)))$



→ $x = (a * b+c+d) * b-d$



Three Address Code :-

It can be represented in 3 diff. forms ;
1) Quadruples 2) Triples 3) Indirect Triples.

For any of these representations, the 3 AC has to be generated initially.

* In 3 AC, each statement contains atmost 3 address locations.

ex: → $x = (a+b) * (c+d)$

3 AC

$t_1 = a+b$
$t_2 = c+d$
$t_3 = t_1 * t_2$
$x = t_3$

Quadruple (4)

S.No	Op	arg1	arg2	result
(0)	+	a	b	t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	=	t_3		x

→ $x = (-a * b) + (-c * d)$

3 AC

$t_1 = -c$
$t_2 = t_1 * d$
$t_3 = -a$
$t_4 = t_3 * b$
$t_5 = t_2 + t_4$
$x = t_5$

Quadruple

S.No	Op	arg1	arg2	result
(0)	-	c		t_1
(1)	*	t_1	d	t_2
(2)	-	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		x

→ $a = (b+c) * (d/e)$

3 AC

$t_1 = d/e$
$t_2 = b+c$
$t_3 = t_1 * t_2$
$a = t_3$

Quadruple

S.No	Op	arg1	arg2	result
(0)	/	d	e	t_1
(1)	+	b	c	t_2
(2)	*	t_1	t_2	t_3
(3)	=	t_3		a

→ $d = (a \text{ or } b) \text{ and } c$

3 AC

$t_1 = a \text{ or } b$
$t_2 = t_1 \text{ and } c$
$d = t_2$

Quadruple

S.No	Op	arg1	arg2	result
(0)	or	a	b	t_1
(1)	and	t_1	c	t_2
(2)	=	t_2		d

→ $x = (a \text{ or } b) \text{ and } (\text{not } c \text{ or not } d)$

3 AC

$t_1 = \text{not } c$
$t_2 = \text{not } d$
$t_3 = t_1 \text{ or } t_2$
$t_4 = a \text{ or } b$
$t_5 = t_4 \text{ and } t_3$
$x = t_5$

Quadruple

S.No	Op	arg1	arg2	result
(0)	not	c		t_1
(1)	not	d		t_2
(2)	or	t_1	t_2	t_3
(3)	or	a	b	t_4
(4)	and	t_4	t_3	t_5
(5)	=	t_5		x

→ $x = ((a+b) * (c+d)) - ((a+b) * d)$

3 AC.

$t_1 = a+b$
$t_2 = c+d$
$t_3 = t_1 * t_2$
$t_4 = t_1 * d$
$t_5 = t_3 - t_4$
$x = t_5$

Quadruple

S.No	Op	arg1	arg2	result
(0)	+	a	b	t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	*	t_1	d	t_4
(4)	-	t_3	t_4	t_5
(5)	=	t_5		x

→ $x = (a * (b+c+(-a)))$

3 AC

$t_1 = -a$
$t_2 = b+c$
$t_3 = t_2 + t_1$
$t_4 = a * t_3$
$x = t_4$

Quadruple

S.No	Op	arg1	arg2	result
(0)	-	a		t_1
(1)	+	b	c	t_2
(2)	+	t_2	t_1	t_3
(3)	*	a	t_3	t_4
(4)	=	t_4		x

Triples:-

→ $x = (a+b) * (c+d)$

S.No	Op	arg1	arg2
(0)	+	a	b
(1)	+	c	d
(2)	*	(0)	(1)
(3)	=	x	(2)

Note: [Write 3AC first then triples & indirect triples]

Indirect Triples:-

S.No:	Op	arg1	arg2
(0)	+	a	b
(1)	+	c	d
(2)	*	(10)	(11)
(3)	=	x	(12)

Indirect Table:

(0) → (10)
(1) → (11)
(2) → (12)

→ $x = (-a*b) + (-c*d)$

Triples:

S.No	Op	arg1	arg2
(0)	−	a	
(1)	*	(0)	b
(2)	−	c	
(3)	*	(2)	d
(4)	+	(1)	(3)
(5)	=	x	(4)

Indirect Triples:

S.No	Op	arg1	arg2
(0)	−	a	
(1)	*	(10)	b
(2)	−	c	
(3)	*	(12)	d
(4)	+	(11)	(13)
(5)	=	x	(14)

Indirect Table:

(0) → (10)
(1) → (11)
(2) → (12)
(3) → (13)
(4) → (14)

Syntax Directed Tree:- (Unit-3 part)

The value of an attribute at any node of parse tree is defined by a semantic rule associated with the production used as the node.
There are 2 diff. attributes:
(i) Synthesized att. (ii) Inherited attr.

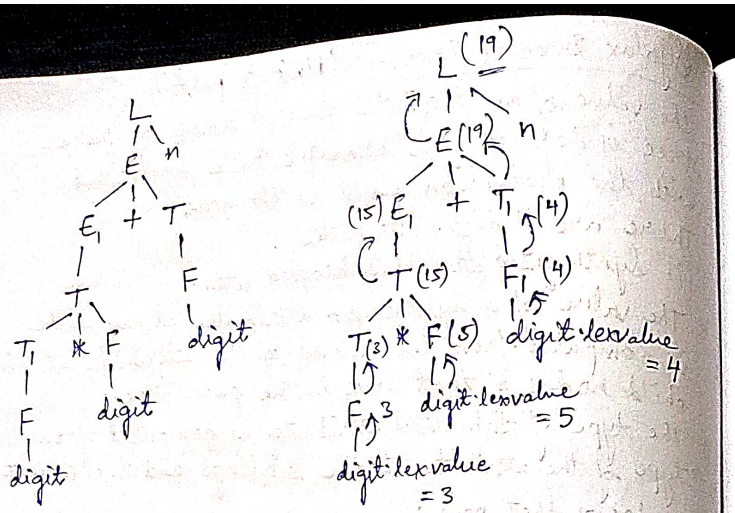
* The value of a synthesized attribute at any node is computed from the values of the attribute at the child nodes of that node in the parse tree.
* The type of inherited attribute is computed from the Ex. → type of the attribute at the siblings and the parent nodes of that node.

note: A syntax directed definition that uses only synthesized attributes is said to be 's'-attributed definition.

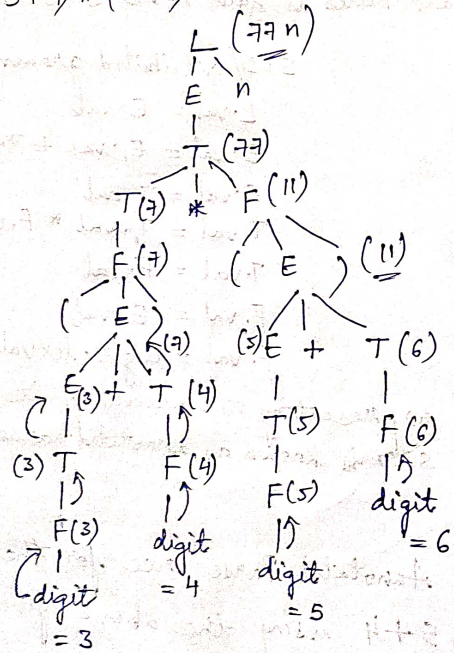
	SDD / Attributed grammar
$L \rightarrow E.n$	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexvalue$

A parse tree representing the values of its attribute using SDD is known as annotated parse tree

Q. (1) Construct Annotated Parse Tree for the string 3*5+4 using the above given grammar.



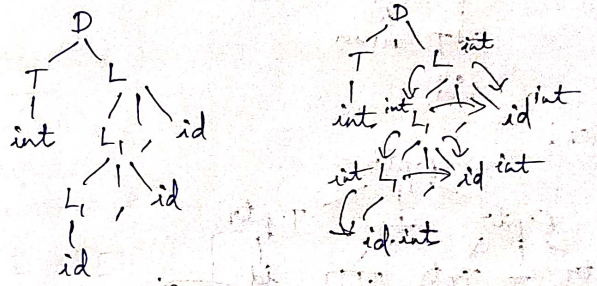
② $(3+4)*(5+6)\,n$



Inherited attributes :-

$D \to TL$	Att. Gr.	$L.in = T.type$
$T \to int$		$T.type = int$
$T \to real$		$T.type = real$
$L \to L_1, id;$		$L_1.in = L.in$ addtype(id.entry, L.in)
$L \to id$		addtype(id.entry, L.in)

Q: ① int a_1, a_2, a_3 ;



Dependency Graph :-

The inter dependencies among the inherited & synthesized attributes can be represented in a parse tree with the help of a directed graph known as "Dependency graph".

Q.

$E \to E+T$	Construct SDD for the given CFG and also construct the syntax tree for the expression a - 4 + c.
$E \to E-T$	
$E \to T$	
$T \to (E)$	
$T \to id$	
$T \to num$	

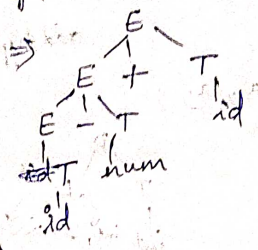
Sol^n:

SDD
$E.val \to E.val + T.val$

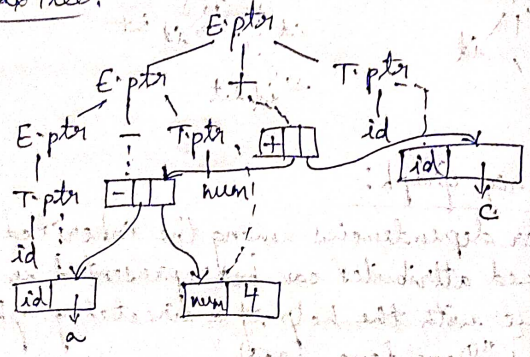
E.ptr = mknode('+', E.ptr, T.ptr)
E.ptr = mknode('-', E.ptr, T.ptr)
E.ptr = T.ptr
T.ptr = E.ptr
T.ptr = mkleaf(id, id.entry)
T.ptr = mkleaf(num, value)

a - 4 + c.

Parse tree ⇒



Syntax Tree:



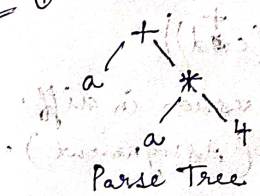
Bottom-up evaluation of S-attributed definition using a Parser ~~stack~~ Stack :-

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	—
+4	T*F	35	F → digit
+4	T	15	T → T*F
4	E+	15	E → T
—	E+4	15	—
	E+F	154	F → digit
	E+T	154	T → F
	E	(19)	E → E+T

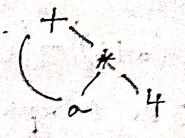
Directed Acyclic Graph (DAG):-

A DAG is a special type of tree used for representing expressions where any node representing a common sub expression has more than 1 parent.

ex: ① a+a*4

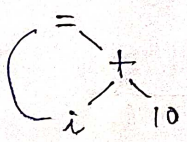


Parse Tree

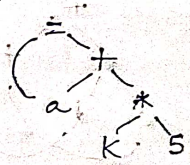


DAG

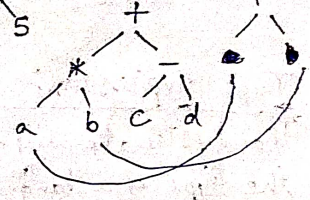
② i = i + 10



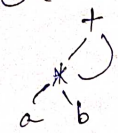
③ a = a + k * 5



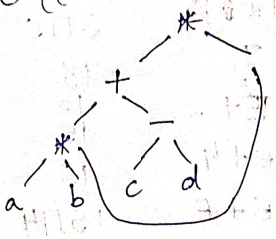
④ (a*b) + (c-d) * (a+b)



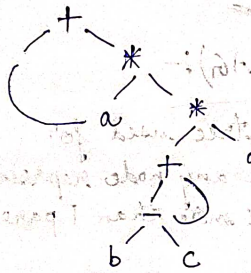
⑤ $(a*b)+(a*b)$



⑥ $((a*b)+(c-d))*(a*b)$



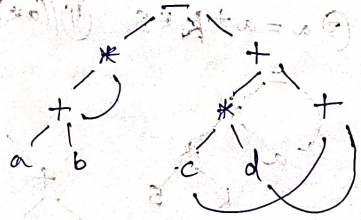
⑦ $a+(a*((b-c)+(b-c))*d)$



⑧ $((a+b)*(a+b))-((c*d)+(c+d))$

Generate 3 AC & represent expression in diff. forms of ICG & also in DAG. (Assignment).

DAG:



Type Expression:-

The systematic way of expressing ~~the~~ / representing the type of a lang. construct is called a type expression.

ex: Lang. construct	Type Expression
→ Array	Array (I, T), where I is the index no. & T is the type.
ex: int a[100]	TE of a is array(100, int).
float sal[20]	TE of sal is array(20, float).
→ Structure	struct((member1 name type) x (m2 name type) x ...)
ex: struct student {char name[20], float avg} student details[100];	TE of student is, struct((name array(20, char)) x (avg float))
	TE of student details is, array(100, student).
→ Pointer int *p	TE of pointer is, pointer(int)
→ Functions	TE of functions is, Type of Parameters → Return type
ex: int sum(int a, int b)	TE for "sum" is, int, int → int
float max(float sal[100])	TE of max is, array(100, float) → float

Type Conversion:-

* Implicit TC or coersion is done by the compiler.
* Explicit TC is done by the user.
* If there is an expression of the form, $E \rightarrow E_1$ op E_2, then the type of E is obtained from the types of E_1 & E_2.

- if E_1.type = int and E_2.type = int
  then E.type = int
- if E_1.type = float and E_2.type = int
  then E.type = float
- if E_1.type = int and E_2.type = float
  then E.type = float
- if E_1.type = float and E_2.type = float
  then E.type = float.

Type Checker :-

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

Type checking of Expressions :-

$E \rightarrow$ literal { E.type = char }
$E \rightarrow$ digit { E.type = int }
$E \rightarrow$ id { E.type = lookup(id.entry) }

$E \rightarrow E_1$ mod 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 }

Symbol Table :-

* It stores the information about the identifiers.
* The info. will be about the name, type, scope, value, binding etc.

There are 2 diff types of ST :
(i) Ordered ST (ii) Unordered ST

Ordered ST :

In this ST, the identifiers are entered in alphabetical order. The main adv. of this ST is that searching is easy. Dis.adv. is that, insertion is difficult.

Unordered ST :

The identifier names are inserted into the ST without following any order.
Adv : insertion is easy.
Dis adv. : Searching is difficult.

Name representation in ST :-

1) Fixed representation 2) Variable length rep.
↓
length.

PC

LC

100

101

Name	Type
age	2 int
salary	4 float

FR

a	g	e	$	$	$	$	$
s	a	l	a	r	y	$	$
				–			

VR

a	g	e	$
s	a	l	a	r	y	$

(or)

a	g	e	$	s	a	l	a	r	y	$
0	1	2	3	4	5	6	7	8	9	10

Start index	length
0	4
4	7

ST management :-

The data structures that are used to represent a ST are linear lists, binary trees, hash tables.

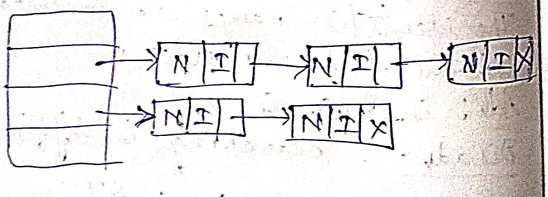
Linear lists :

Name	Info
N2	J2
:	:

Binary Tree :

Left child	Symbol/ Name	Info	Right child

Hash Tables :



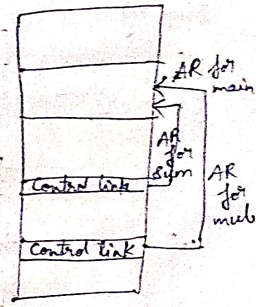
Activation Record :-

It consists of the info. about a single procedure.

Return value
Actual parameters
Control link
Access link
Saved Machine Status
Local Variables
Temporaries

- Return value is the value returned by the procedure.
- Actual parameters consists of the values of actual parameters
- control link points to the activation record of the calling procedure.
- Access link refers to the non-local data in other activation records.
- Saved Machine Status - before the procedure is called some control info. will be stored in the registers.
- Local Variables holds all the local variables of a procedure.
- Temporaries - all temporary variables generated by the compiler in order to execute the procedure are stored in this field.

```
main()                    float sum(x,y)
{ ==                      { ==
  a = sum(b,c);           }
  ==
  b = mul(c,f);           int mul(p,q)
  ==                      { ==
}                         }
```



MM

Access to non-local names:-

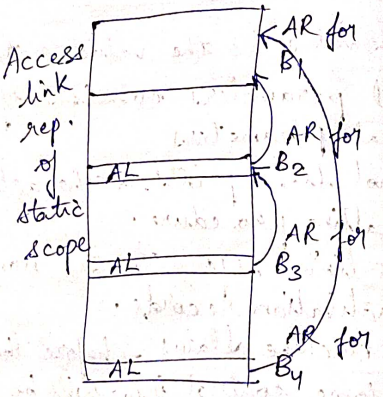
1) Static scope / Lexical scope.
2) Dynamic Scope
   - Deep Access
   - Shallow Access

Static / Lexical scope:-

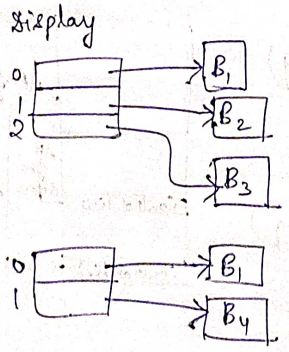
Access link Display

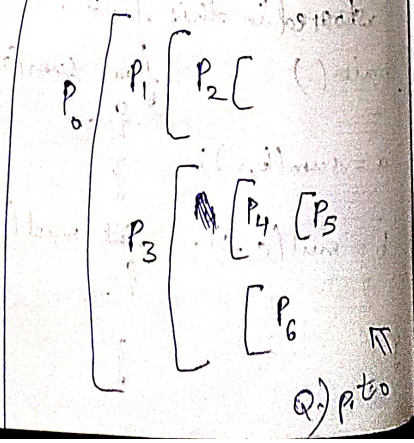
- In access link the activation record is represented using the access link.

ex:
```
B1()
{ ==
   B2()
   { ==
      B3()
      { ==
      }
   }
   ==
   B4()
   { ==
   }
}
```



- Display : An auxiliary array is used.



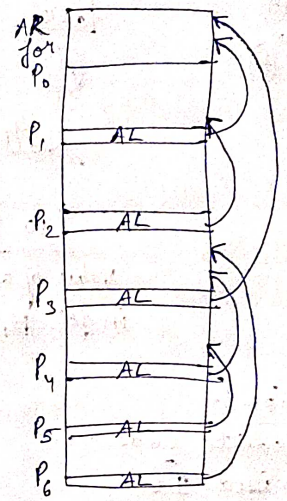


Q) P1 to

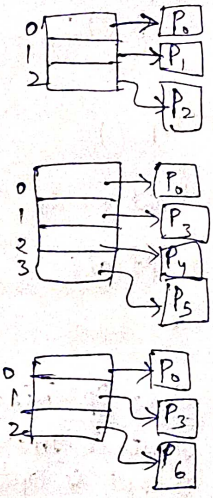
Q. Represent the access link & display representations for the given procedures.

sol^n:
```
P0()
{ ==
   P1()
   { ==
      { P2()
        ==
      }
   }
   ==
   P3()
   { ==
      P4()
      { ==
         P5()
         { ==
         }
      }
      P6()
      { ==
      }
   }
}
```

Access link:



Display:

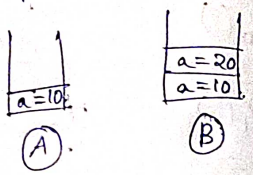


Deep Access:-
A stack is maintained in order to retrieve the most recent value for that symbol.

Shallow Access:-
A central storage is kept with one slot for every variable.

```
B1()
{
  int a = 10;
  pf(a);  ---> (A)
  B2();   ---> (B)
}
B2()
{
  int a = 20;
  pf(a);
}
```

Deep Access



Shallow Access

Value of a in B_1 — Value of a in B_2.

Code Optimization :-

Input is Intermediate Code.
Output is Optimised IC.

* The main tasks of CO are ------
  (i) The semantic equivalence of the program should not be changed.
  (ii) It should produce efficient object code.
* The main challenges for CO :
  (i) The optimizing algorithm must be efficient enough in such a way that target code should not become time and space in-efficient.
  (ii) The algo. should preserve its semantic equivalence.

Types of CO :-

(i) Machine Dependent Optimization :-

It depends on architecture of machine like instruction mode, registers etc.

(ii) Machine Independent Optimization :-

It depends on the programming language.

Types of Optimization techniques :-

* **Folding :-**
  The computation of a constant can be performed at compile time rather than at execution time.
  ex: a = (15%4) * p
  ↳ executed at compile time.

* **Constant propagation :-**
  The value of a variable is replaced and the expression is evaluated at compilation time.
  ex: r = 5
  a = b * r * r
  ↳ a = b * 25

* **Common sub expression elimination :-**
  An expression which is computed previously can be eliminated if it appears further in the program.
  ex: $T_1 = 4 * i$
  $T_2 = a[T_1]$
  $T_3 = 4 * i$
  $T_4 = b[T_3]$ ⟶ $T_4 = b[T_1]$
  ⋮

* **Variable propagation :-**
  There is no need of using 2 diff. variables for 1 single value.
  ex:
  x = Pi
  Pi = 3.14
  a = x * r * r
  }
  x = 3.14
  a = x * r * r

* Code Movement :-

Some portion of the code can be moved out of the loop inorder to reduce time and space.

ex:
```
for(i=0; i<10; i++)
{
    x = y*10
    z = y*10 + w
}
```
⟶ optimised code
```
j = y*10
for(i=0; i<10; i++)
{
    x = j
    z = j+w
}
```

note: Code movement is also known as loop invariant method.

ex:
```
while(i<=max-1)
{
  ==
}
```
⟶
```
n = max-1
while(i<=n)
{
  ==
}
```

* Strength reduction / Induction variable elimination :-

Higher strength operators like * and / can be replaced by + and −.

ex:
```
for(i=1; i<10; i++)
{
    c = i*5
}
```
⟶
```
p = 5
for(i=1; i<10; i++)
{
    c = t;
    t = t+5;
}
```

here t is known as induction variable.

ex: b = 2*a ⟶ b = a+a.

* Dead code elimination :-

A variable is said to be dead if its value is never used in the program.

ex:
```
i = 2
j = 5   ⟶ not used
x = i+K
```

ex:
```
i = 1
if(i==0)
{
  ==
}
```
} dead.

* Loop Unrolling :-

The no. of tests and jumps can be reduced by writing the code in the loop for 2 or more times.

ex:
```
while(i<=200)
{
  a[i] = b[i]+c[i]
  i++
}
```
⟶
```
int i=1
while(i<=100)
{
  a[i] = b[i]+c[i]
  i++
  a[i] = b[i]+c[i]
  i++
}
```

* Loop Fusion :-

Several loops can be merged to a single loop.

ex:
```
for(i=1 to n)
  for(j=1 to m)
    a[i,j] = 20
```
⟶
```
for(i=1 to n*m)
  a[i] = 20
```