Marwadi University
Marwadi Chandarana Group
NAAC A+

CD : COMPILER DESIGN

Intermediate Code Generation

Department of CE

Unit no : 4
Intermediate Code Generation and Memory Management
(01CE0714)

Prof. Shilpa Singhal

Outline :

Role of Intermediate Code Generator

Intermediate Languages

Directed Acyclic Graph (DAG)

Quadruple, Triples, Indirect Triples

SDD : Synthesized and Inherited Attribute

Syntax Directed Translation

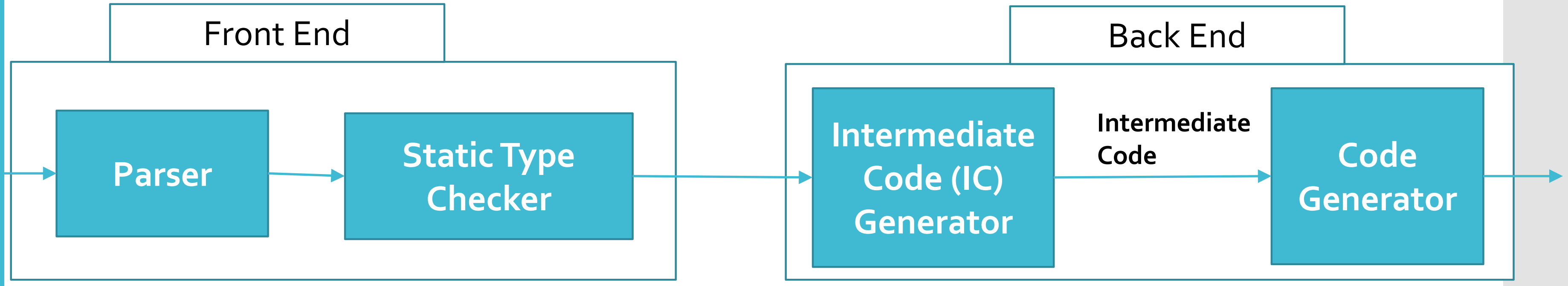
Activation Record

Parameter Passing Methods

Symbol Table

Role of Intermediate Code Generator

- In the analysis – synthesis model of compiler, the front end translates a source program into an intermediate representation from which the back end generates target code.



Source Program can be translated directly into machine language but it is not always possible in one pass so compiler generates intermediate code first and then generates the target code.

Advantages of Intermediate Code Generator

Some of the benefits of using a machine – independent intermediate form are :

1) **Retargeting** is facilitated ; a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2) A compiler for different source language (on same machine) can be created by providing different front ends for corresponding source languages to exiting blackened.
3) A machine independent **code optimizer** can be applied to the intermediate representation.

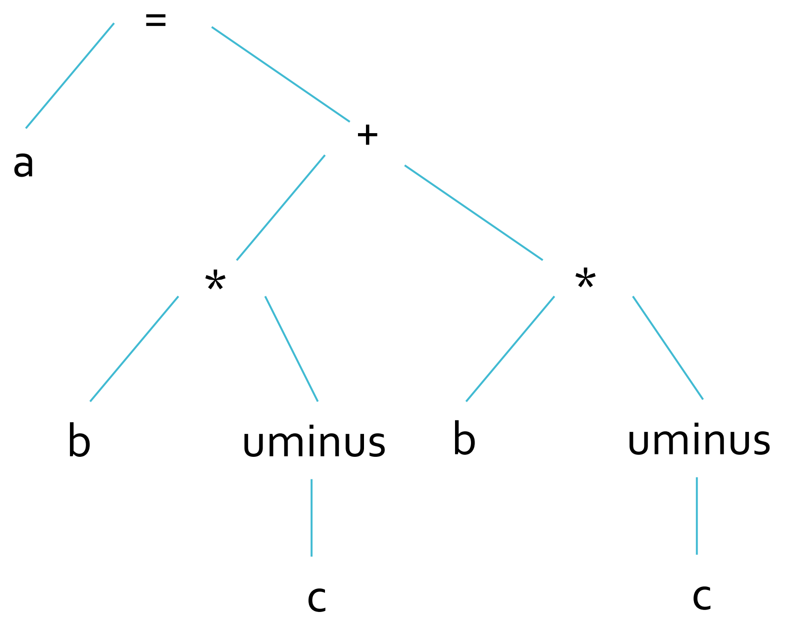
Intermediate Language / Different Intermediate Form

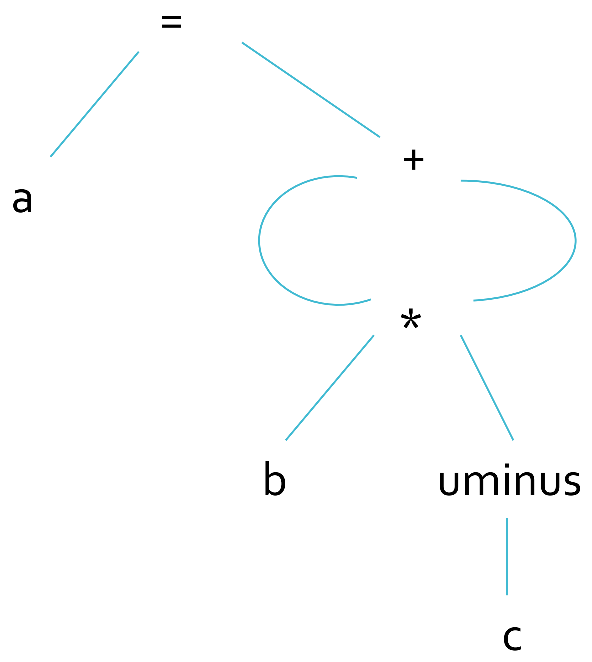
1. Abstract Syntax Tree
2. Postfix notation
3. Three Address Code

Abstract Syntax Tree

Syntax Tree + DAG (Directed Acyclic Graph)

- A syntax tree depicts the natural hierarchical structure of a source program.
- A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common subexpressions are identified.
- Example : a := b * -c + b * -c



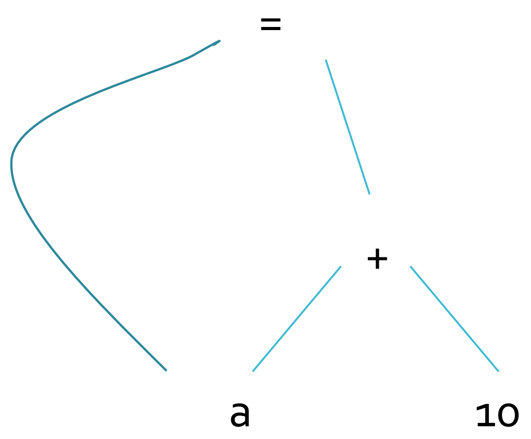


Directed Acyclic Graph

- Each Node of DAG contains a unique value.
- It does not contain any cycles in it, hence called **Acyclic**.

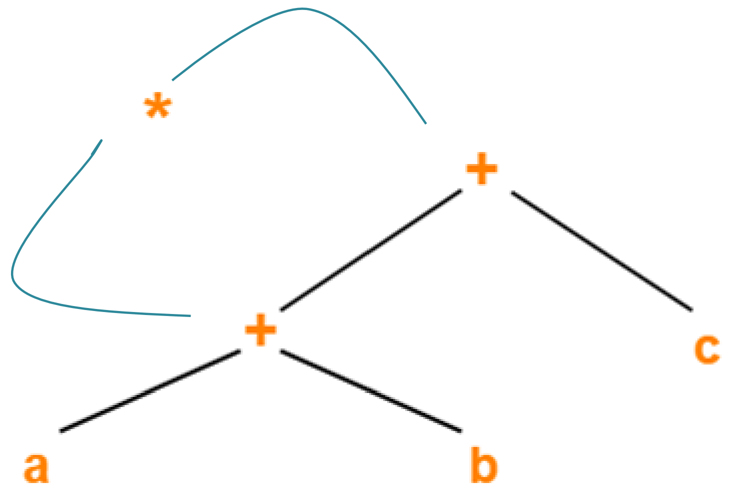
Example : a = a + 10

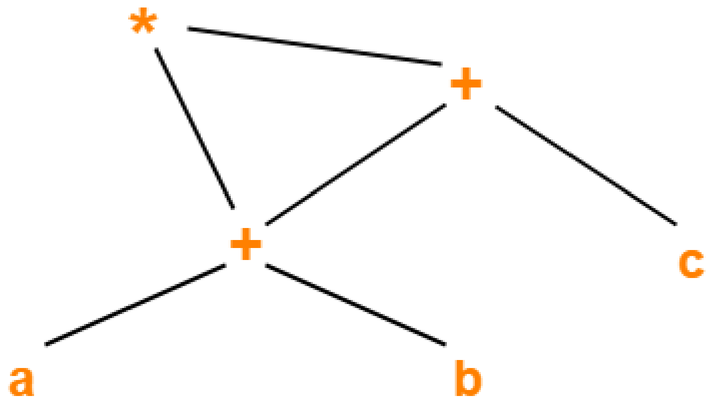
Representation			
1	id	Entry to a	
2	Num	10	
3	+	1	2
4	=	1	3



Directed Acyclic Graph Examples

1) (a + b) * (a + b + c)





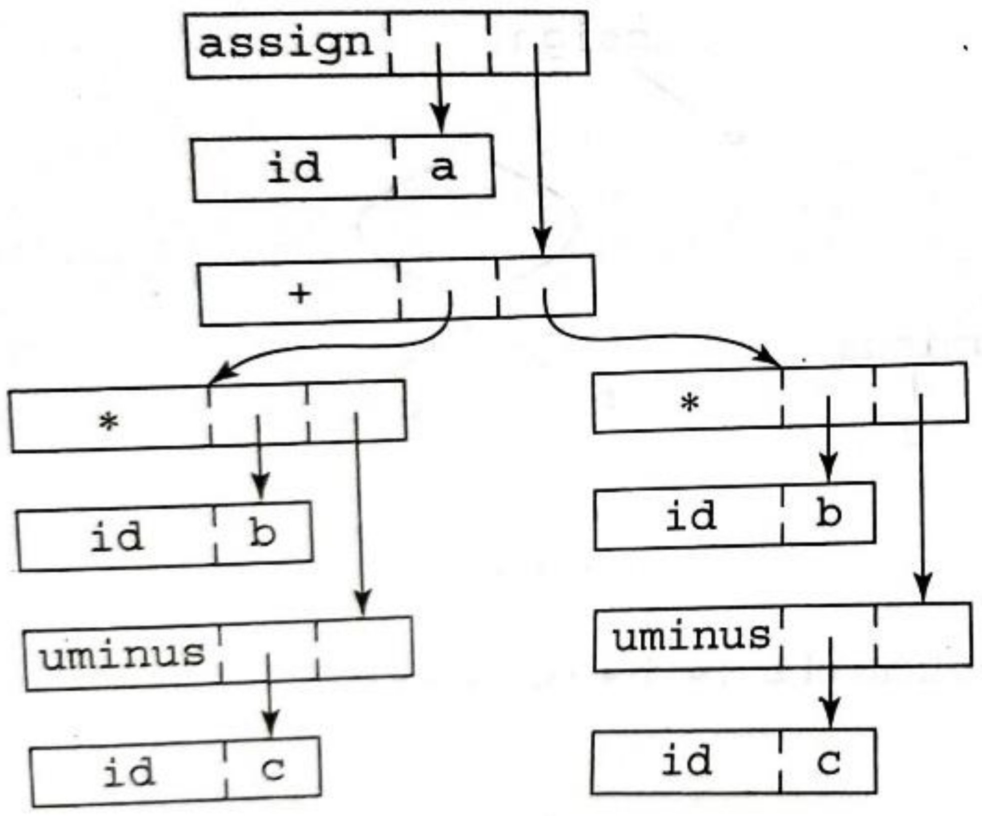
Directed Acyclic Graph Examples (Self Practice)

2) a + a * (b - c) + (b - c) * d

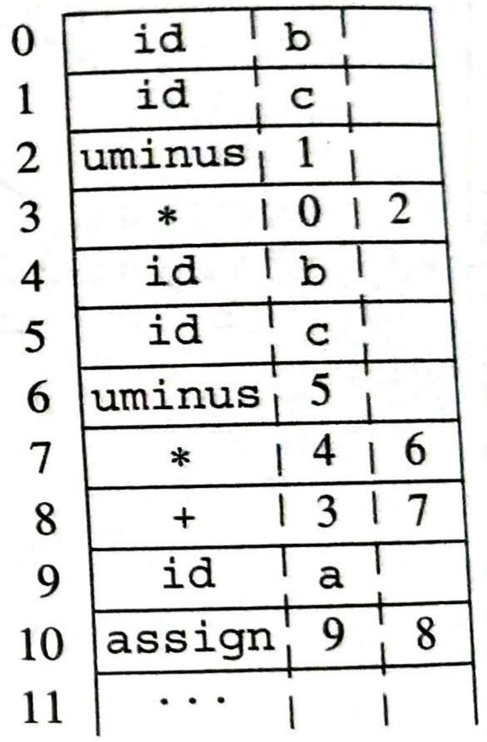
3) i = i + 10

4) (((a + a) + (a + a)) + ((a + a) + (a + a)))

Representation of Syntax Tree
a := b * -c + b * -c



(a)



0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8
11	. . .		

(b)

Postfix Notation

- Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children.
- In postfix notation operands are arranged first then operators are arranged.

Example : a + b * c

Postfix notation : a bc* +

Example : a := b * -c + b * -c

Postfix notation : a b c uminus * b c uminus * + =

Example: a+b*c+d*e^f

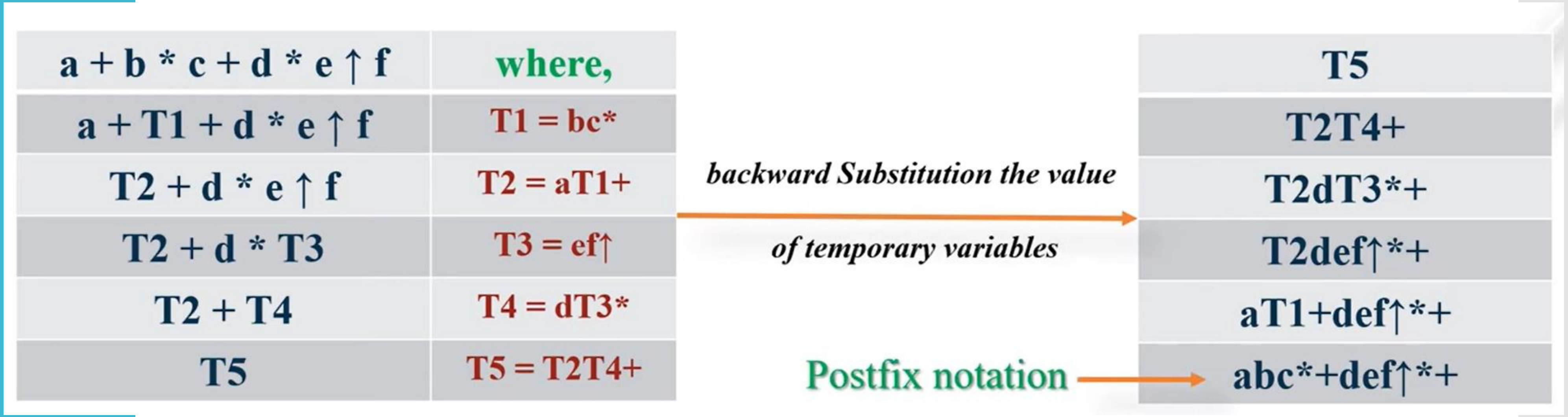
Postfix Notation: abc*+def^*+

Postfix Notation

- To evaluate the postfix notation ,evaluate the expression according to precedence.

Example: a+b*c+d*e^f

Postfix Notation: abc*+def^*+



a + b * c + d * e ↑ f	where,
a + T1 + d * e ↑ f	T1 = bc*
T2 + d * e ↑ f	T2 = aT1+
T2 + d * T3	T3 = ef↑
T2 + T4	T4 = dT3*
T5	T5 = T2T4+

T5
T2T4+
T2dT3*+
T2def↑*+
aT1+def↑*+
abc*+def↑*+

backward Substitution the value
of temporary variables
Postfix notation

Three Address Code

- Three address code is a sequence of statements of the general form

$$x := y \ \mathit{op} \ z$$

Where x, y and z are names, constants, or compiler generated temporaries ; op stands for any operator, such as fixed or floating point arithmetic operator, or a logical operator on a Boolean data value.

- Thus a source language expression like x + y * z might be translated into a sequence

$$t1 := y * z$$

$$t2 := x + t1$$

Where t1 and t2 are compiler generated temporary names

- Three address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.

Three Address Code

- Example : a := b * -c + b * -c

```
t1 := uminus c
t2 := b * t1
t3 := uminus c
t4 := b * t3
t5 = t2 + t4
a = t5
```

- The reason for the term “Three Address Code” is that each statement usually contains three addresses, two for the operands and one for the result.

Three Address Code

(Practise Questions)

1. Write Three Address Code for the following expression-

a = b + c + d

Solution1.

(1) T1 = b + c

(2) T2 = T1 + d

(3) a = T2

2. Write Three Address Code for the following expression-

-(a x b) + (c + d) – (a + b + c + d)

Solution2.

(1) T1 = a x b

(2) T2 = uminus T1

(3) T3 = c + d

(4) T4 = T2 + T3

(5) T5 = a + b

(6) T6 = T3 + T5

(7) T7 = T4 – T6

Quadruple, Triples and Indirect Triples

- A three address statement is an abstract form of intermediate code.

- In a compiler, these statements can be implemented as records with fields for the operator and the operands.

- Three such representations are

1) Quadruples
2) Triples
3) Indirect Triples

Quadruples

Quadruples :

- A Quadruple is a record structure with four fields :
  - ❑ Op
  - ❑ Arg1
  - ❑ Arg2
  - ❑ Result
- The Op field is used to represent the internal node of the operator.
- Arg1,Arg2 represents two operands.
- Result field is used to store the result of the expression.
- Statements with unary operators like x := -y or x := y do not use arg2.
- Conditional and unconditional jumps put the target label in result.

Quadruples

Three Address Code

$x = -a*b + -a*b$

$t_1 = \text{uminus } a$

$t_2 = t_1 * b$

$t_3 = \text{uminus } a$

$t_4 = t_3 * b$

$t_5 = t_2 + t_4$

$x = t_5$

Pointers

Quadruple representation

	Operator	**Arg1**	**Arg2**	**Result**
(0)	uminus	a		t_1
(1)	*	t_1	b	t_2
(2)	uminus	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		x

Quadruple, Triples and Indirect Triples

Example a := b * -c + b * -c

	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

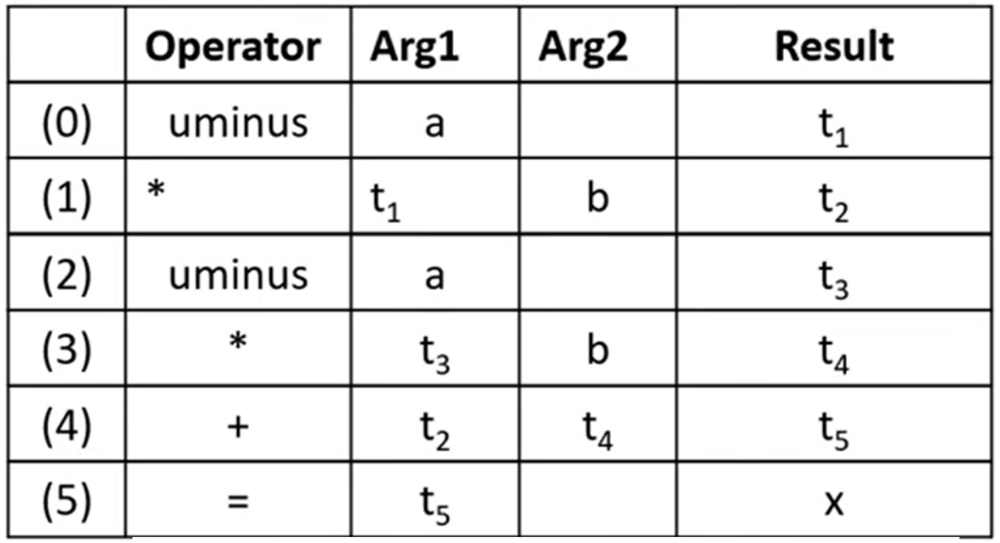
Quadruple, Triples and Indirect Triples

Triples :

- In Triples temporaries are not used instead of that pointers are used directly in the symbol table.
- A Triple is a record structure with three fields :
  - ❑ Op
  - ❑ Arg1
  - ❑ Arg2
- Numbers in the ( ) round brackets are used to represent pointers into triple structure.

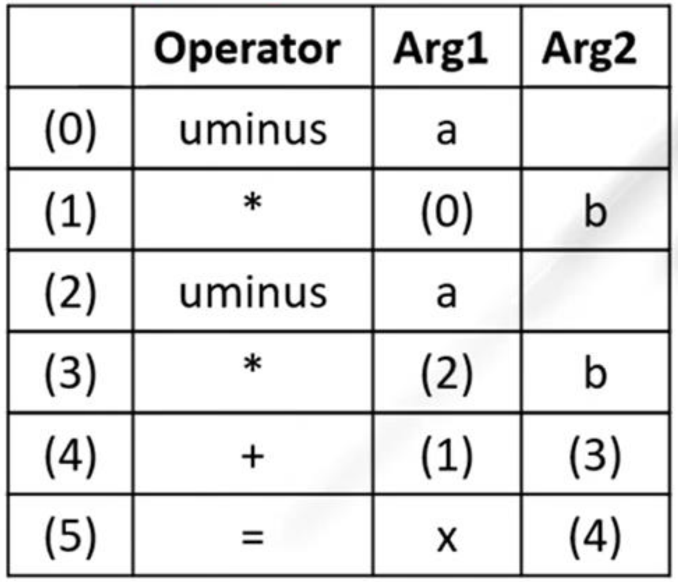
Quadruple, Triples and Indirect Triples

Quadruple representation



	Operator	Arg1	Arg2	Result
(0)	uminus	a		t_1
(1)	*	t_1	b	t_2
(2)	uminus	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		x

Triple representation



	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	x	(4)

Quadruple, Triples and Indirect Triples

Example a := b * -c + b * -c

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Quadruple, Triples and Indirect Triples

Indirect Triples :

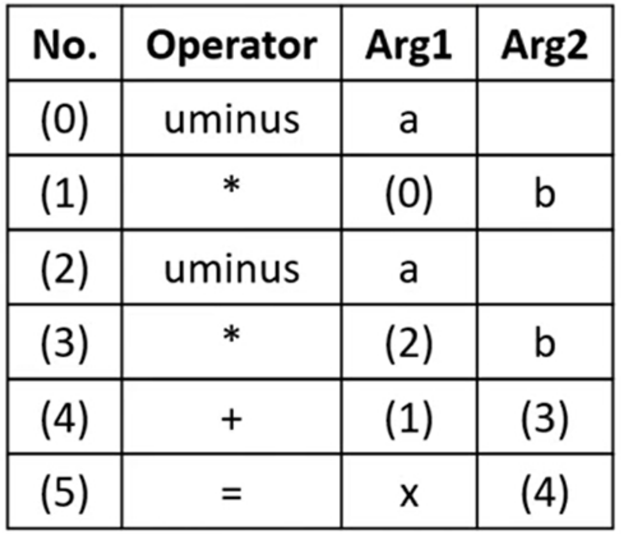
- Another implementation of three address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves.
- This implementation is naturally called indirect triples.

	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg1	arg2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

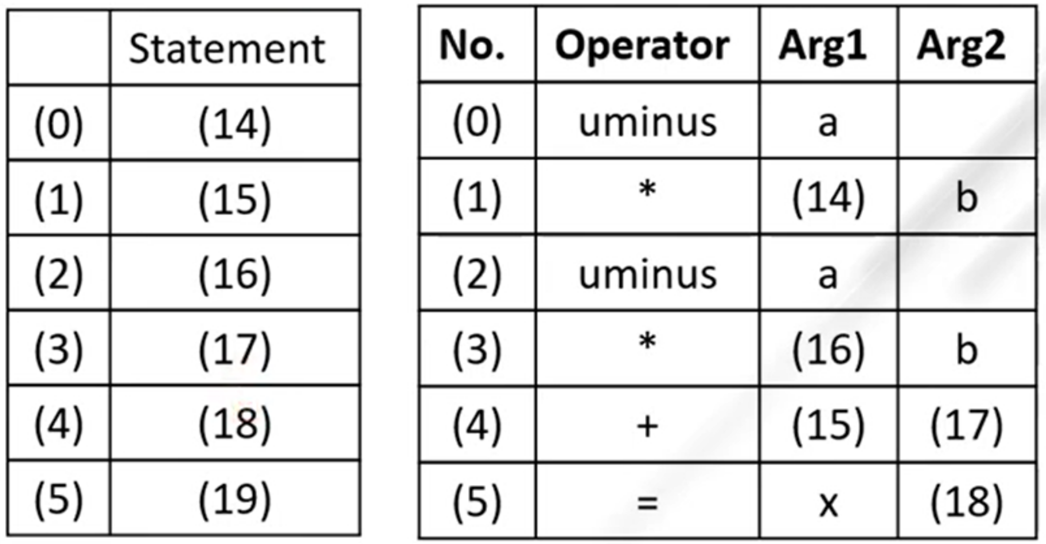
Quadruple, Triples and Indirect Triples

Triple representation



No.	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	x	(4)

Indirect Triple representation



	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

No.	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*	(14)	b
(2)	uminus	a	
(3)	*	(16)	b
(4)	+	(15)	(17)
(5)	=	x	(18)

Quadruple, Triples and Indirect Triples

Translate the following expression to quadruple, triple and indirect triple-

a + b x c / e ↑ f + b x a

<u>Solution-</u>

Three Address Code for the given expression is-

T1 = e ↑ f

T2 = b x c

T3 = T2 / T1

T4 = b x a

T5 = a + T3

T6 = T5 + T4

Quadruple, Triples and Indirect Triples

Quadruple

Location	Op	Arg1	Arg2	Result
(0)	↑	e	f	T1
(1)	x	b	c	T2
(2)	/	T2	T1	T3
(3)	x	b	a	T4
(4)	+	a	T3	T5
(5)	+	T5	T4	T6

Triple

Location	Op	Arg1	Arg2
(0)	↑	e	f
(1)	x	b	c
(2)	/	(1)	(0)
(3)	x	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

Quadruple, Triples and Indirect Triples

Indirect Triples

Statement	
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

Location	Op	Arg1	Arg2
(0)	↑	e	f
(1)	x	b	e
(2)	/	(1)	(0)
(3)	x	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

Syntax Directed Definition

- A syntax directed definition specifies the values of attributes by associating **semantic rules** with the grammar productions.
- Grammar + Semantic Rules
- A syntax-directed translation(SDT)is an executable specification of SDD.
- Syntax directed definition is a generalization of context free grammar in which each grammar symbol has an associated set of attributes.
- Types of attributes are:
  1. Synthesized attribute
  2. Inherited attribute

Synthesized Attribute

- Value of **synthesized attribute at a node** can be **computed** from the **value of attributes at the children** of that node in the parse tree.
- Syntax directed definition that uses synthesized attribute exclusively is said to be **S-attribute definition**.
- A parse tree for an S-attribute definition can always be annotated by evaluating the semantic rules for the attribute at each node bottom up, from the leaves to root.
- An **Annotated parse tree** is a parse tree showing the value of the attributes at each node.
- The process of computing the attribute values at the node is called Annotating or Decorating the parse tree.

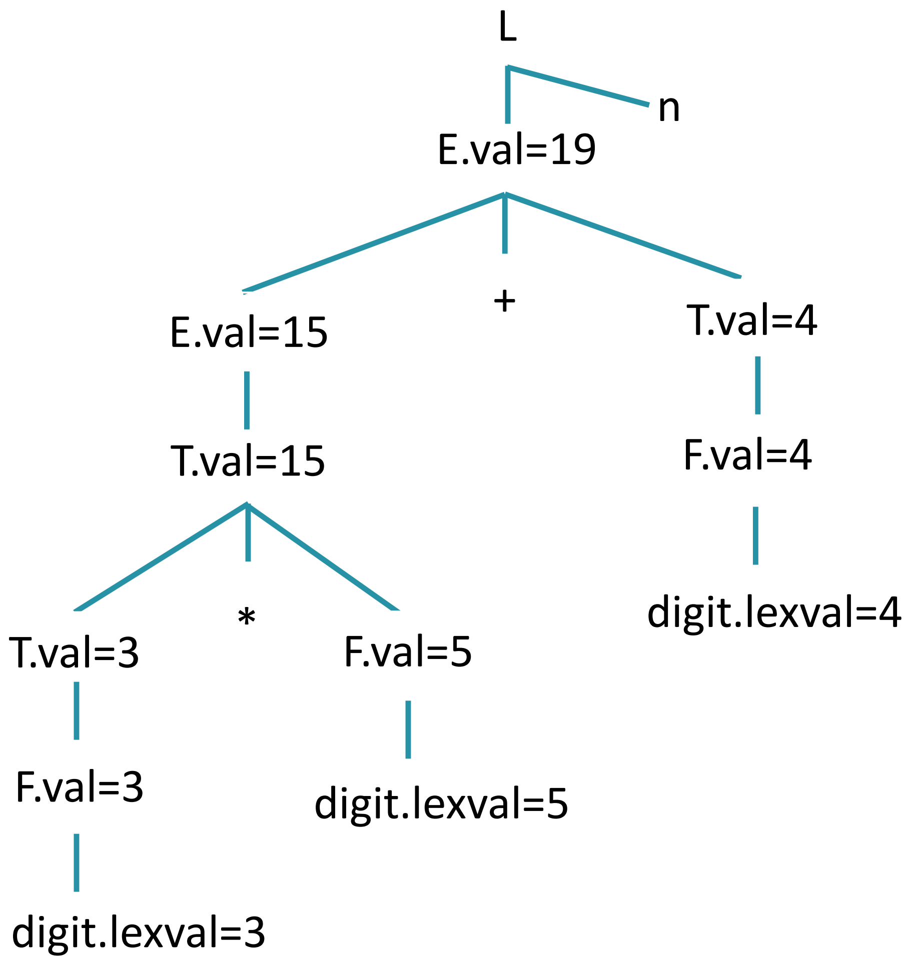
Synthesized Attribute

- Example: Simple desk calculator

Production	Semantic rules
L → E_n	Print (E.val)
E → E_1+T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val=T1.val*F.val
T → F	T.val=F.val
F → (E)	F.val=E.val
F → digit	F.val=digit.lexval

Synthesized Attribute

- Example: Simple desk calculator
- String: 3*5+4n;



Annotated parse tree for 3*5+4n

Production	Semantic rules
L → E_n	Print (E.val)
E → E_1+T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val=T1.val*F.val
T → F	T.val=F.val
F → (E)	F.val=E.val
F → digit	F.val=digit.lexval

Inherited Attribute

- An inherited attribute at a node in a parse tree is defined in terms of attributes at the parent and/or siblings of the node.
- Convenient way of expressing the dependency of a programming language construct on the context in which it appears.
- We can use inherited attributes to keep track of whether an identifier appears on the left or right side of an assignment to decide whether the address or value of the assignment is needed.
- The inherited attribute distributes type information to the various identifiers in a declaration.

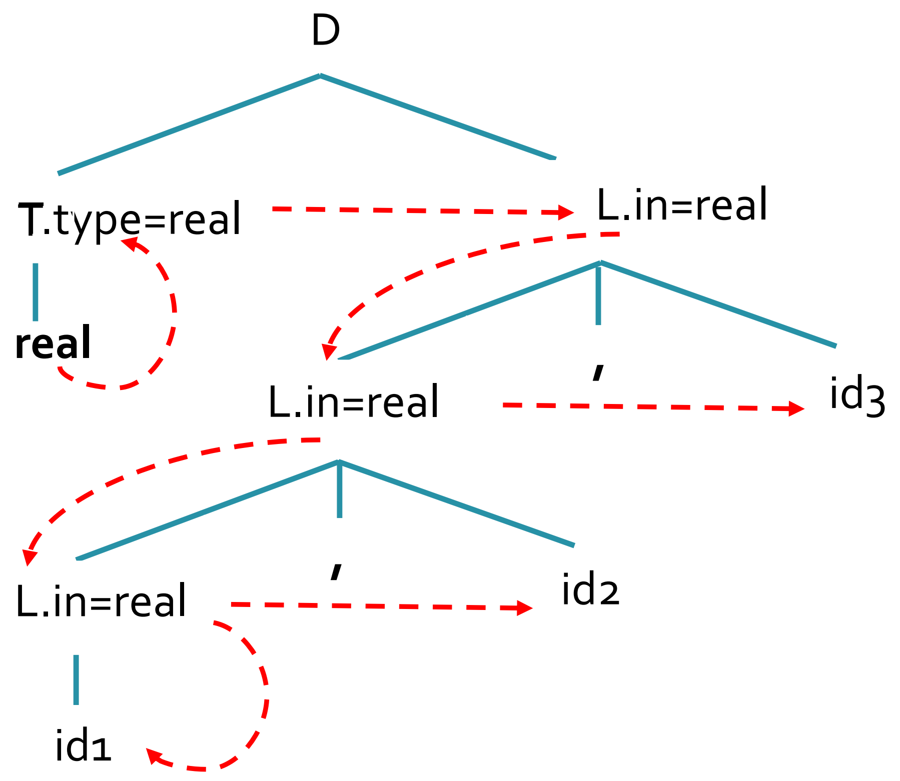
Inherited Attribute

Production	Semantic rules
D→TL	L.in = T.type
T → int	T.Type = integer
T → real	T.type= real
L → L1,id	L1.in = L.in, addtype(id.entry,L.in)
L → id	addtype(id.entry,L.in)

Symbol T is associated with a **synthesized** attribute *type*.
Symbol L is associated with an **inherited** attribute *in.*

Inherited Attribute

real id1,id2,id3



$L \longrightarrow id$
$D \rightarrow TL$
$L \longrightarrow L_1 , id$

Production	Semantic rules
D→TL	L.in = T.type
T → int	T.Type = integer
T → real	T.type= real
L → L1,id	L1.in = L.in, addtype(id.entry,L.in)
L → id	addtype(id.entry,L.in)

Difference

Synthesized attribute	**Inherited attribute**
Value of synthesized attribute at a node can be computed from the value of attributes at the children of that node in parse tree.	Values of inherited attribute at a node can be computed from the value of attribute at the parent and/ or siblings of the node
Pass the information from bottom to top in the parse tree	Pass the information top to bottom in the parse tree or from left siblings to the right siblings.
Synthesized attribute is used by both S-attributed SDT and L-attributed STD.	Inherited attribute is used by only L-attributed SDT.

Run time Environment

- A compiler must accurately implement the abstractions embodied in the source language definition. These abstractions typically include the concepts such as names, scopes, bindings, data types, operators, procedures, parameters, and flow-of-control constructs.
- The compiler must cooperate with the operating system and other systems software to support these abstractions on the target machine.
- To do so, the compiler creates and manages a *run-time environment* in which it assumes its target programs are being executed.

Run time Environment

1. Procedure Call:-

- A Procedure definition is a declaration that associates an identifier with a statement.
- The identifier is the procedure name and the statement is the procedure body.
- For example, the following is the definition of procedure named readarray :

```
Procedure readarray;
var I : integer;
begin
    For i := 1 to 9 read(a[i])
end;
```

- When procedure name appears within an executable statement, the procedure is said to be called at that point.

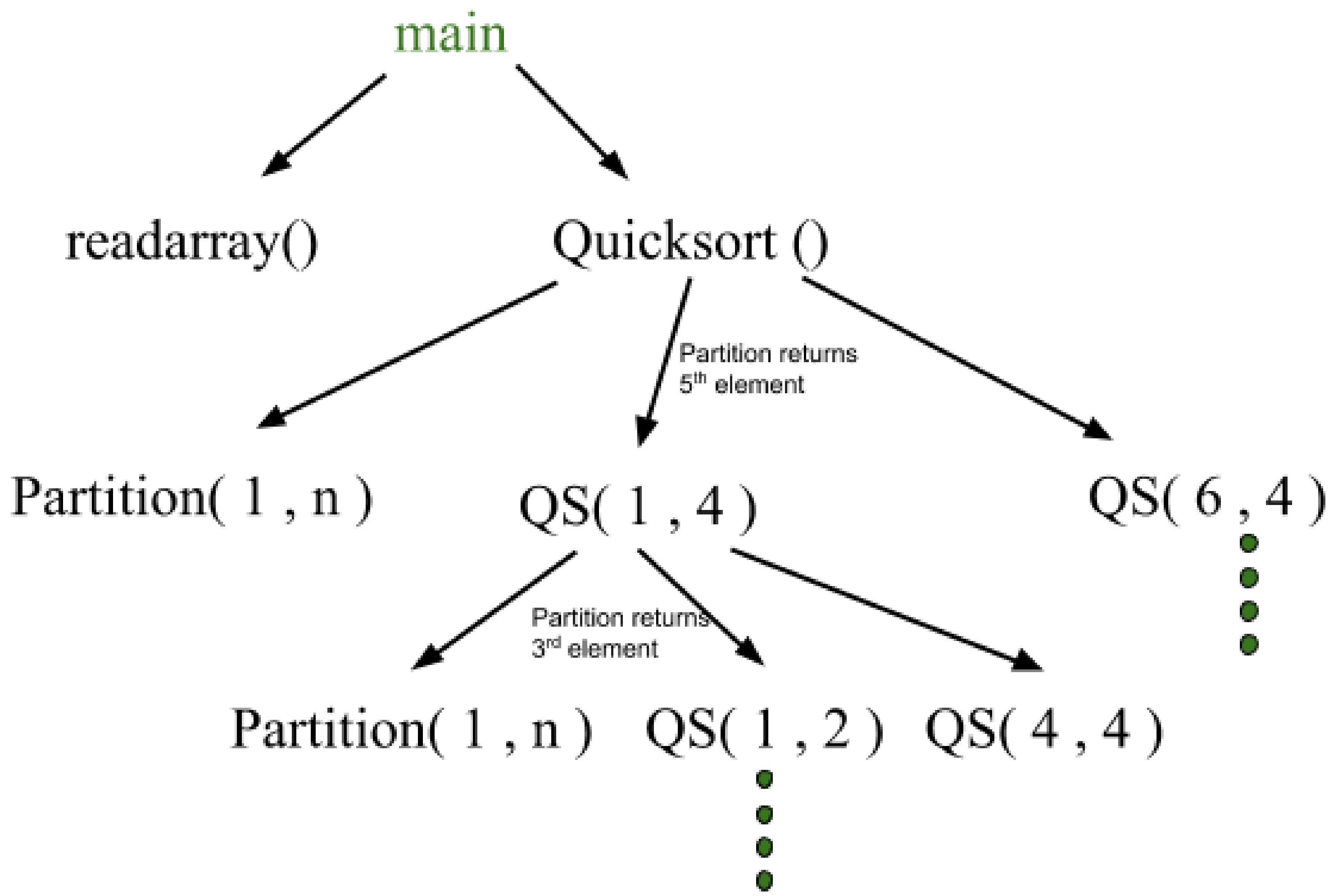
Run time Environment

2. Activation tree

- Each execution of a procedure body is referred to as an activation of the procedure.
- The lifetime of an activation of a procedure p is the sequence of steps between the first and last steps in the execution of the procedure body.
- An activation tree is used to depict the way control enters and leaves activations. In an activation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for a is the parent of the node for b if and only if control flows from activation a to b.
4. The node for a is to the left of the node for b if and only if the lifetime of a occurs before the lifetime of b.

Run time Environment



Run time Environment

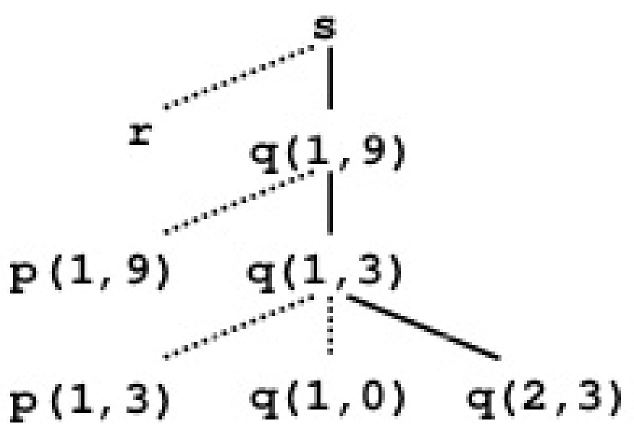
3. Control Stack

- The flow of control in a program corresponds to a depth first traversal of the activation tree that starts at the root, visits a node before its children, and recursively visits children at each node in a left to right order.
- A control stack is used to keep track of live procedure activations.
- The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.
- The contents of the control stack are related to paths to the root of the activation tree.
- When node n is at the top of control stack, the stack contains the nodes along the path from n to the root.

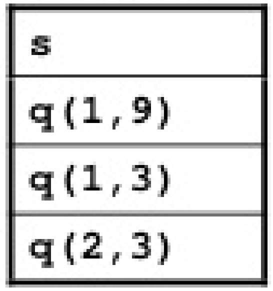
Run time Environment

Control Stack

Activation tree:



Control stack:



Activations:

```
begin sort
enter readarray
leave readarray
enter quicksort(1,9)
enter partition(1,9)
leave partition(1,9)
enter quicksort(1,3)
enter partition(1,3)
leave partition(1,3)
enter quicksort(1,0)
leave quicksort(1,0)
enter quicksort(2,3)
…
```

Run time Environment

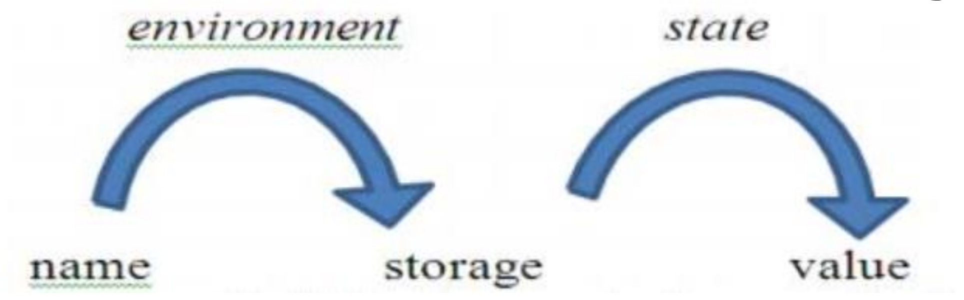
4. The Scope of a Declaration

- A declaration is a syntactic construct that associates information with a n Declarations may be explicit, such as: var i : integer ;
- Or they may be implicit.
- Example, any variable name starting with I is assumed to denote an integer in a Fortran program unless otherwise declared.
- The portion of the program to which a declaration applies is called the scope of that declaration.
- An occurrence of a name in procedure is said to be local to the procedure if it is in the scope of a declaration within the procedure ; otherwise , the occurrence is said to be non local.

Run time Environment

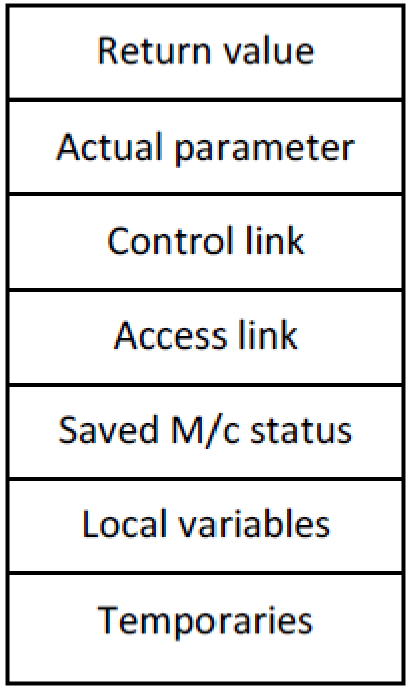
5. Binding of names

- Even if each name is declared once in a program, the same name may denote different data objects at run time.
- "Data object" corresponds to a storage location that holds values.
- The term environment refers to a function that maps a name to a storage location.
- The term state refers to a function that maps a storage location to the value held there.
- When an environment associates storage location s with a name x, we say that x is bound to s.
- This association is referred to as a binding of x.



Activation Record

- The activation record is a block of memory used for managing information needed by a single execution of a procedure.



Activation record

- Fortran uses the static data area to store the activation record where as in pascal and C the activation record is situated in stack area.

Activation Record

1. Temporary values: The temporary variables are needed during the evaluation of expressions. Such variables are stored in the temporary field of activation record.
2. Local variable: The local data is a data that is local to the execution of procedure which is stored in this field of activation record.
3. Saved machine register: This field holds the information regarding the status of machine just before the procedure is called. This field contains the machine registers and program counters.
4. Control link: This field is optional. It points to the activation record of the calling procedure. This link is also called the dynamic link.
5. Access link: This field is also optional. It refers to the non local data in other activation record. This field is also called static link field.

Activation Record

6. Actual parameter: This field holds the information of actual parameter. These actual parameter are passes to the called procedure.

7. Return values: This field is used to store the result of a function call.

Static memory and Dynamic memory Allocation

Static memory allocation	Dynamic memory allocation
Memory is allocated before the execution of program begins.	Memory is allocated during the execution of program.
No memory allocation or de-allocation actions are performed during execution.	Memory bindings are established and destroyed during execution.
Execution speed is faster compared to dynamic allocation.	Execution speed is slower compared to static allocation.
To access the variable pointer is needed.	No need of dynamically allocated pointers.
More memory space is required as we allocate memory before execution.	Less memory space required in dynamic memory allocation.
Variable remains permanently allocated.	Allocated only when program unit is active.
Implemented using stack and heaps.	Implemented using data segment.

Parameter Passing Methods

- All programming languages have a notion of a procedure, but they can differ in how these procedures get their arguments. In this section, we shall consider how the actual parameters are associated with the formal parameters.

1. Call by value
2. Call by reference
3. Copy restore
4. Call by name

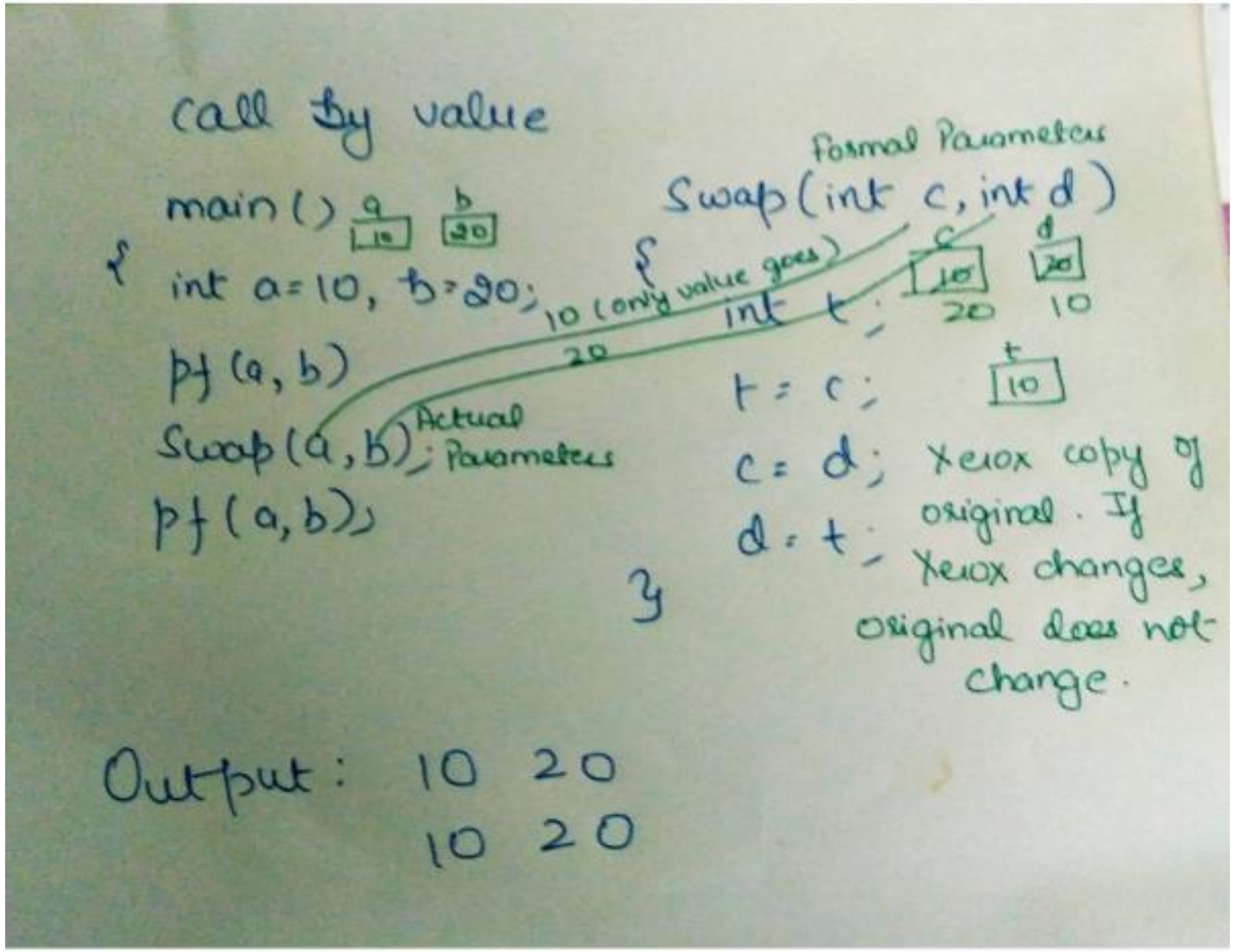
Parameter Passing Methods

1. **Call by Value:**

- In *call-by-value,* the actual parameter is evaluated (if it is an expression) or copied (if it is a variable). The value is placed in the location belonging to the corresponding formal parameter of the called procedure.
- The operation in formal parameter do not change its value in actual parameter.
- This method is used in C, Java and is a common option in C++, as well as other languages.

Parameter Passing Methods

1. **Call by Value:**



call by value

```
main()
{
int a=10, b=20;
pf(a,b)
Swap(a,b);
pf(a,b);
}
```

Actual Parameters

Formal Parameters

```
Swap(int c, int d)
{
int t;
t = c;
c = d;
d = t;
}
```

Xerox copy of original. If Xerox changes, original does not change.

```
Output: 10 20
        10 20
```

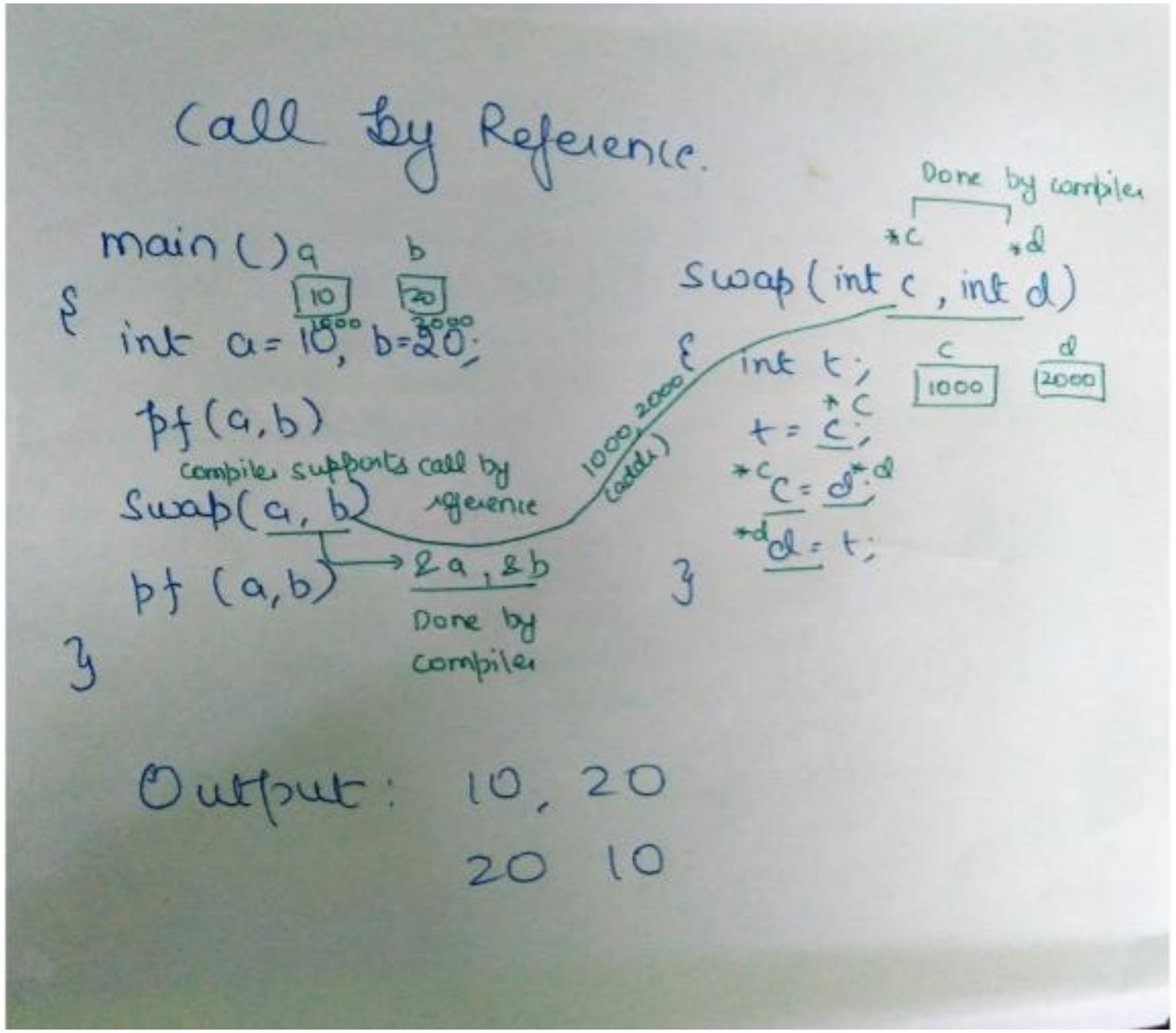
Parameter Passing Methods

2. **Call by Reference:**

- This method is also called call by address or call by location.
- In *call-by-reference,* the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter.
- Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller.
- Changes to the formal parameter thus appear as changes to the actual parameter.

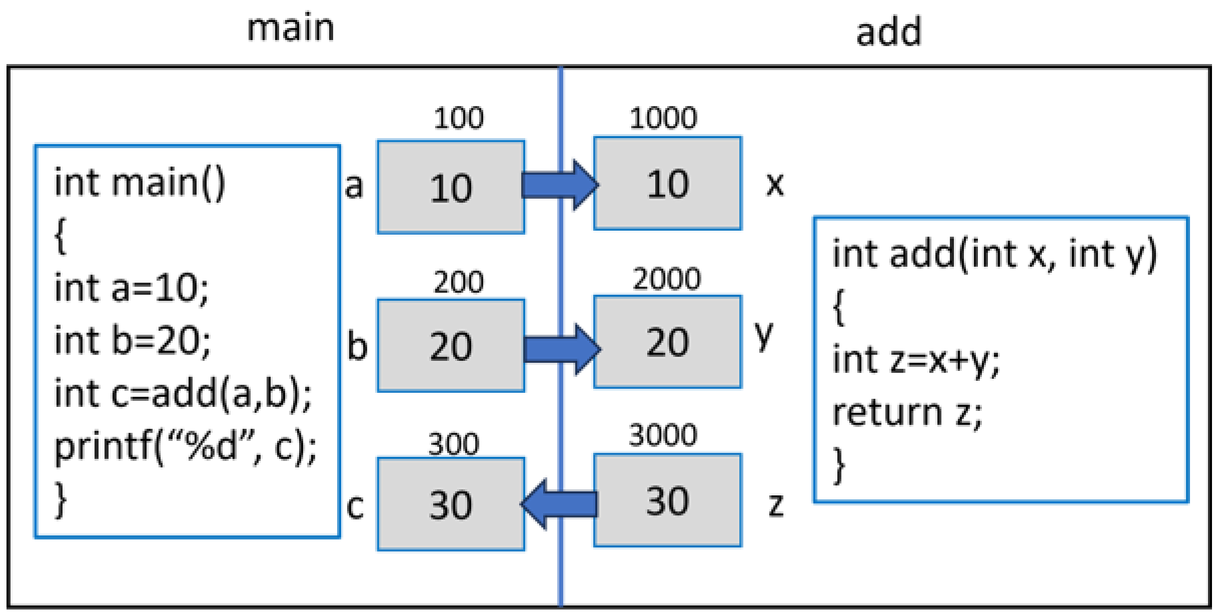
Parameter Passing Methods

2. Call by Reference:



Parameter Passing Methods

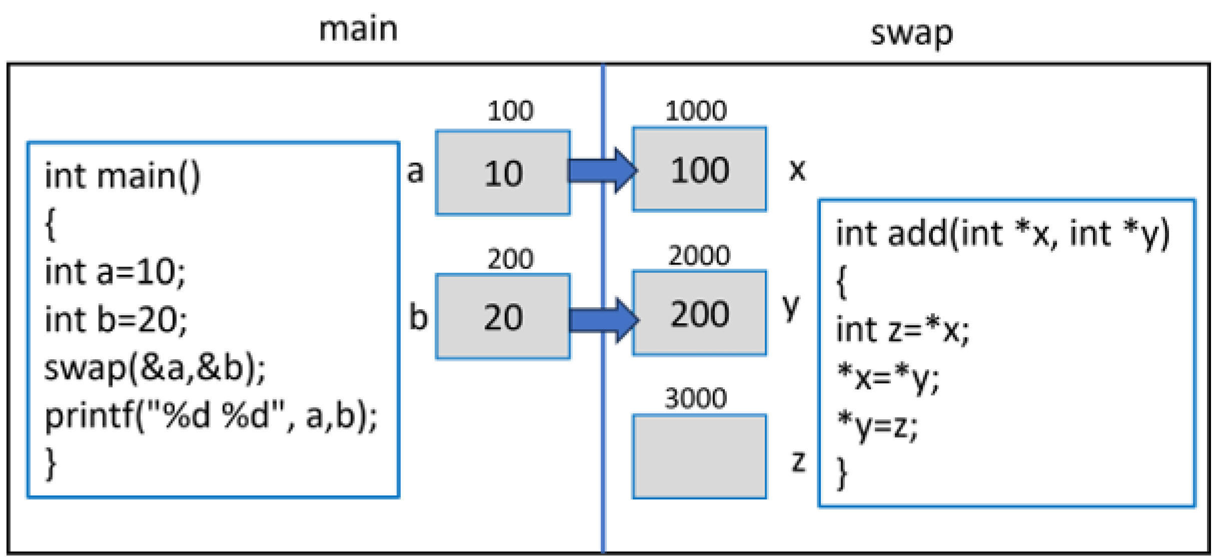
1. Call By Value:



```
int main()
{
int a=10;
int b=20;
int c=add(a,b);
printf("%d", c);
}
```

```
int add(int x, int y)
{
int z=x+y;
return z;
}
```

2. Call By Reference:



```
int main()
{
int a=10;
int b=20;
swap(&a,&b);
printf("%d %d", a,b);
}
```

```
int add(int *x, int *y)
{
int z=*x;
*x=*y;
*y=z;
}
```

Parameter Passing Methods

Parameters	Call by Value	Call by Reference
Basic	A copy of the variable is passed.	A variable itself is passed.
effect	Change in a cop of variable doesn’t modify the original value of variable.	Change in a copy of variable modify the original value of variable.
Syntax	function_name(variable_name1,variable_name2…)	function_name(&variable_name1,&variable_name2…)
Default Calling	Primitive types are passed using “call_by_value”.	Objects are implicitly passed using “call_by_reference”.

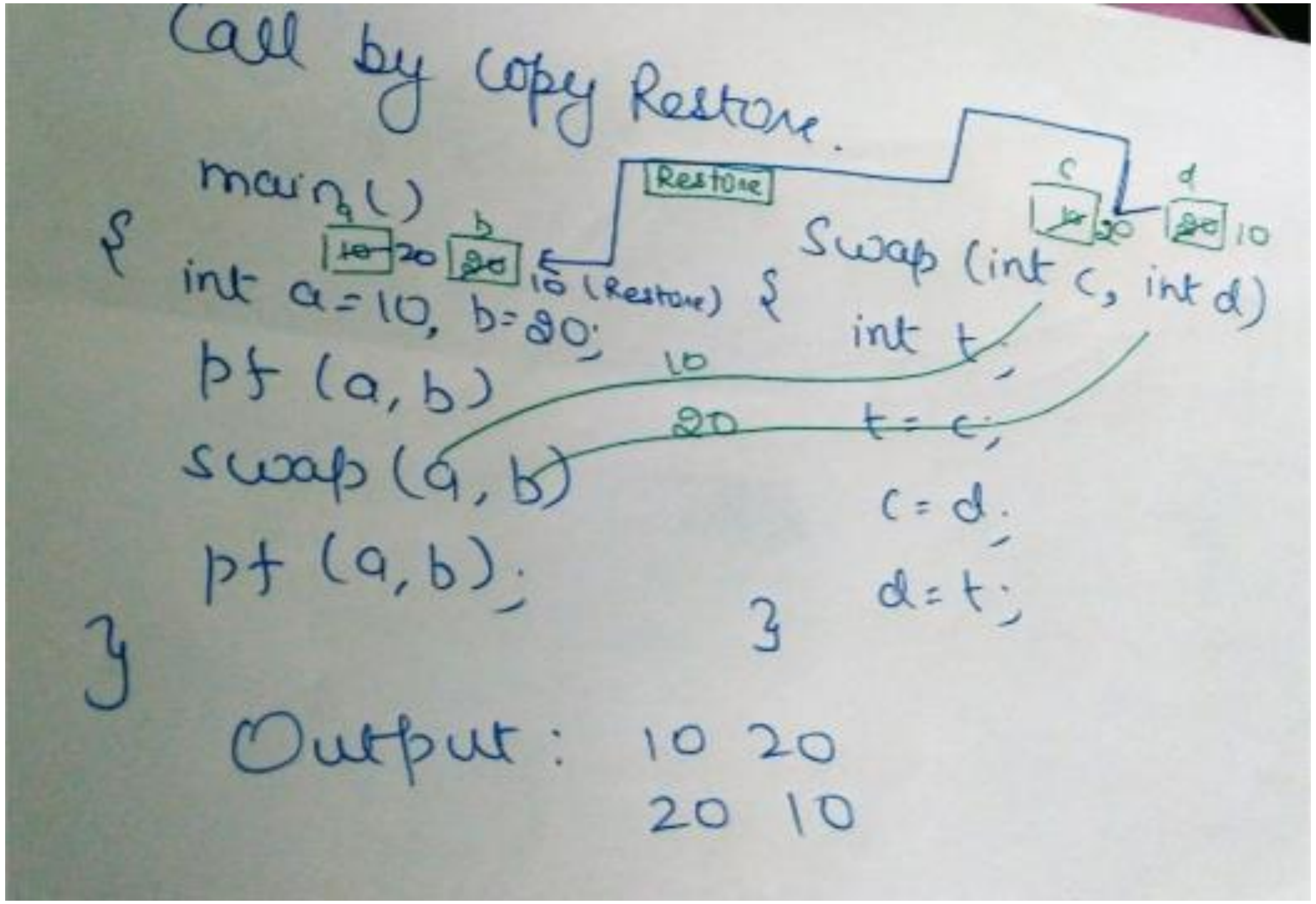
Parameter Passing Methods

3. **Call by Copy - Restore:**

- This method is **hybrid** between call by value and call by reference. This method is also called **copy-in-copy-out** or **values result**.
- The calling procedure calculates the value of actual parameter and it then copies to activation record for the called procedure.
- During execution of called procedure, the actual parameters value is not affected.
- If the actual parameter has X-values then, at return the value of formal parameter is copied to actual parameter.

Parameter Passing Methods

3. **Call by Copy - Restore:**



a
b
~~10~~ 20
~~20~~ 10 (Restore)
Restore
c
d
~~10~~ 20
~~20~~ 10
10
20

Call by Copy Restore.

```
main()
{
 int a=10, b=20;
 pf(a,b)
 swap(a,b)
 pf(a,b);
}
                    swap(int c, int d)
                    {
                      int t;
                      t=c;
                      c=d;
                      d=t;
                    }
```

```
Output: 10 20
        20 10
```

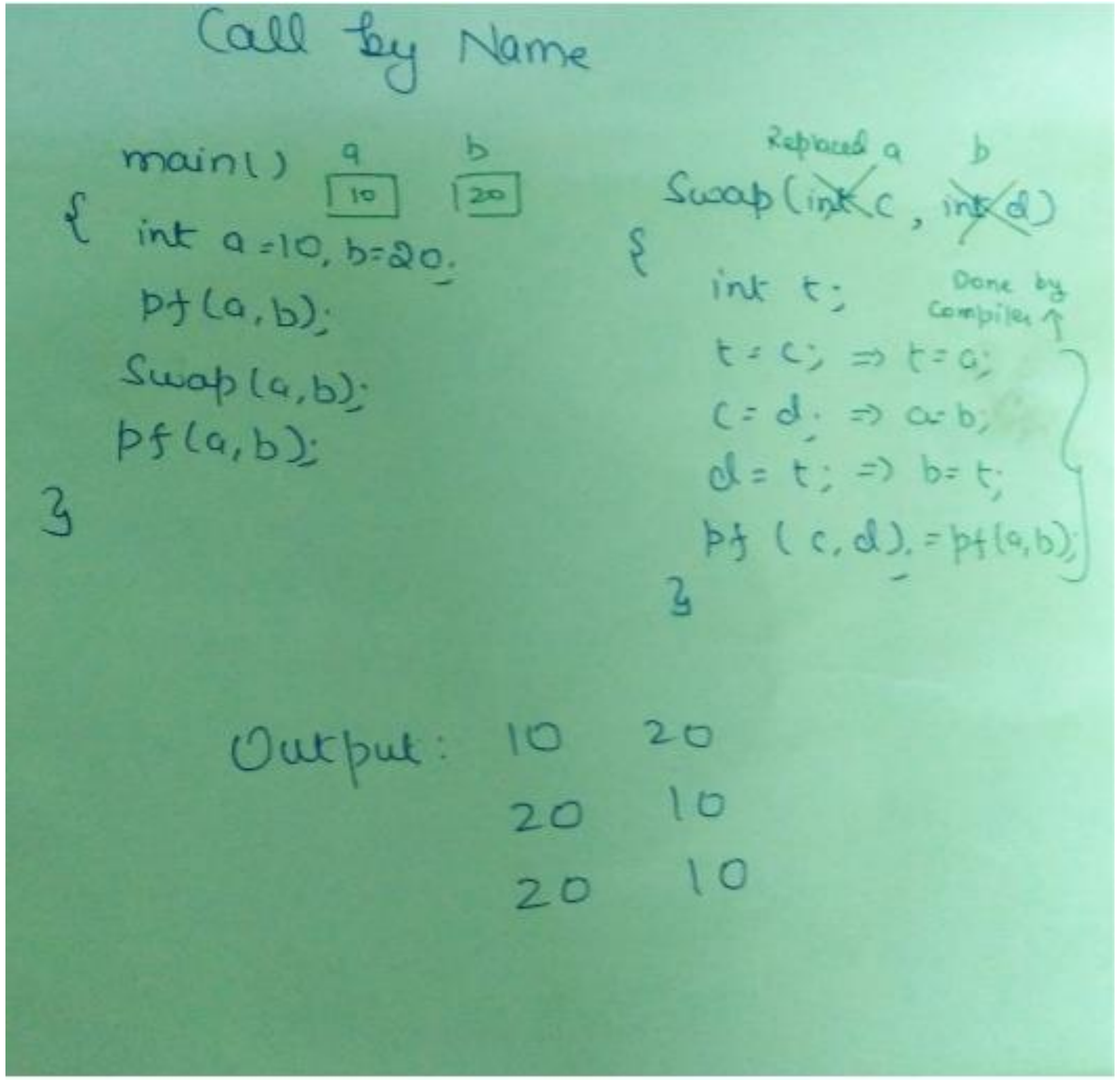
Parameter Passing Methods

4. Call by Name:

- This is less popular method of parameter passing.
- Procedure is treated like macro. The procedure body is substituted for call in caller with actual parameters substituted for formulas.
- The actual parameters can be surrounded by parenthesis to preserve their integrity.
- The local names of called procedure and names of calling procedure are distinct.

Parameter Passing Methods

4. **Call by name:**



Call by Name

```
main()          a      b
               [10]   [20]
{
  int a=10, b=20;
  pf(a,b);
  Swap(a,b);
  pf(a,b);
}
```

```
         Replaced a       b
Swap(~~int~~ c, ~~int~~ d)
{
   int t;              Done by
                       Compiler ↑
   t = c;  => t = a;
   c = d;  => a = b;
   d = t;  => b = t;
   pf(c,d); = pf(a,b);
}
```

```
Output:  10   20
         20   10
         20   10
```

Questions

1) Explain Quadruples and Triples form of three address code with example.
2) Draw a DAG for expression: a + a * (b – c) + (b – c) * d.
3) Construct syntax tree and DAG for following expression.

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

4) Explain quadruples, triples and indirect triples with examples

5) Translate following arithmetic expression (a * b) + (c + d) - (a + b) into

1] Quadruples

2] Triple

3] Indirect Triple

6) Draw syntax tree and DAG for the statement

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

7) Differentiate Static and Dynamic Memory Allocation

8) Explain Activation record.

9) Explain Parameter passing methods with example.

Thanks

Marwadi University
Marwadi Chandarana Group
NAAC A+

Prof. Shilpa Singhal