Data Control -

One aspect to the semantics of a programming language concerns access to the data objects of the program; language features related to this aspect are known as data control features.

Examples of such features include scope rules and parameter transmission. It is the control of the transmission of data among sub-programs of a program.

Attributes of a Data Control :- The data-control features of a programming language are those parts concerned with the accessibility of data at different points during program execution. The sequence-control mechanisms provide the means to coordinate the sequence in which operations are invoked during program execution. Once an operation is reached during execution, it must be provided with the data on which it is to operate. The data-control features of a language determine how data may be

provided to each operation, and how a result of one operation may be saved and retrieved for later use as an operand by a subsequent operation.

⇒ When writing a program, one ordinarily is well aware of the operations that the program must execute and their sequence, but seldom is the same true of the operands for those operations.

Names & Referencing Environment

There are two ways that a data object can be made available as an operand for an operation:-

* Direct Transmission - A data object computed at one point as the result of the operation may be directly transmitted to another operation as an operand.

eg:- X := Y + 2 * Z ;

The result of the multiplication 2 X Z is transmitted

directly to the addition operation as an operand in the statement. The data object is allocated storage temporarily during its lifetime and may never be given a name.

* Referencing through a named data object :-

A data object may be given a name when it is created, and the name may then be used to designate it as an operand of an operation. Alternatively, the data object may be made a component of another data object that has a name so that the name of the larger data object may be used together with a selection operation to designate the data object as an operand.

eg:- a = 2 * z;
x = y + a;

Direct transmission is used for data control within expressions, but most data control outside of expressions involves the use of names and the referencing of names. The problem of the meaning of names forms the central concern in data control.

Program Elements that may be named :-

What kinds of names are seen in programs? Each language differs, but some general categories seen in many languages are as follows:-

1. Variable names
2. Formal parameter names
3. Subprogram names
4. Names for defined types
5. Names for defined constants
6. Statement labels
7. Exception names
8. Names for primitive operations
9. Names for literal constants

Categories 1 to 3, names for variables, formal parameters and subprograms form the center of our concern here. Of the remaining categories, most references to name in these groups are resolved during translation rather than during program execution.

Simple names: They are represented by identifiers such as X, Z2 and Sub1.

Composite names:- A composite name is a name for a component of a data structure.

for eg:- If A is the name of an array,
then A is a simple name,
A[3] is a composite name.

Associations and Referencing Environments:-

Associations:- Data control is concerned with the binding of identifiers in large part to particular data objects and subprograms. Such a binding is termed an association and may be represented as a pair consisting of the identifier and its associated data object or subprogram.

Referencing Environment: Each program or subprogram has a set of identifier associations available for use in referencing during its execution. This set of identifier

associations is termed the referencing environment of the subprogram. The referencing environment of a subprogram is ordinarily invariant during its execution. It is set up when the subprogram activation is created, and it remains unchanged during the lifetime of the activation. The values contained in the various data objects may change, but the associations of names with data objects and subprograms do not.

Several Components of the referencing environment of a subprogram :-

1. Local Referencing Environment (Simply local Environment) :-

The set of associations created on entry to a subprogram that represent formal parameters, local variables, and subprograms defined only that subprogram forms the local referencing environment of that activation of the subprogram.

The meaning of a reference to a name

in the local environment may be determined without going outside of the subprogram activation.

2. Non-local referencing environment :-

The set of associations for identifiers that may be used within a subprogram but that are not created on entry to it termed the non-local referencing environment of the subprogram.

3. Global referencing environment -

If the associations created at the start of execution of the main program are available to be used in a subprogram, then these associations form the global referencing environment of that subprogram. It is a part of the non-local environment.

4. Predefined referencing environment - Some identifiers have a predefined association that is defined directly in the language definition. Any program or subprogram may use these associations without explicitly creating them.

Visibility of associations:- An association for an identifier is said to be visible within a subprogram if it is part of the referencing environment of the subprogram currently in execution is said to be hidden from that subprogram.

Dynamic Scope of associations:- Each association has a dynamic scope, which is that part of program execution during which it exists as part of a referencing environment. Thus, the dynamic scope of an association consist of the set of subprogram activations within which it is visible.

Referencing Operations:- A referencing operation is an operation with the signature

ref-op : id x referencing-environment → data object or subprogram

where ref-op, given an identifier and a referencing environment, finds the appropriate association for that identifier in the environment and returns the associated data object or subprogram definition.

Aliases for Data Objects:- When a data object is visible through more than one name (simple or composite) in a single referencing environment, each of the names is termed an alias for the data object.

Aliasing in a Pascal program

No aliasing

```
program main(output);
procedure Sub1(var J: integer);
  begin
    ... { J is visible, I is not }
  end;
procedure Sub 2;
  var I: integer;
  begin
    ...
    Sub 1(1); { I is visible, J is not }
    ...
  end;
begin
  ...
```

I & J are aliased in Sub1

```
program main(output);
var I: integer;
procedure Sub 1(var J:
                integer);
  begin
    ... { I and J refer
          to same }
  end ; { data object
          here }
procedure Sub 2;
  begin
    ...
    Sub 1(1); { I is visible,
                J is not }
    ...
  end;
begin
  ...
```

```
Sub 2 { Neither is visible }          Sub 2 { I is visible,
  ...                                          J is not }
end                                    ...
                                       end
```

Problems with aliasing :-

* can make code difficult to understand for the programmer.
* Implementation difficulties at the optimization step - difficult to spot interdependent statements - not to order them.

Static and Dynamic Scope :-

The dynamic scope of an association for an identifier, is that set of subprogram activations in which the association is visible during execution. It always includes the subprogram activation in which that association is created as part of the local environment. It may also be visible as a non-local association in other subprogram activations.

Dynamic Scope Rules -

A dynamic scope rule define the dynamic scope of each association in terms of the dynamic course of the program execution.

for eg:-

A typical dynamic scope rule states that the scope of an association created during an activation of subprogram P includes not only that activation but also any activation of a subprogram called by P, or called by a subprogram called by P, and so on, unless that later subprogram activation defines a new local association for the identifier that hides the original association.

Static scope -

The static scope of a declaration is that part of the program text where a use of the identifier is a reference to that particular declaration of the identifier.

Static Scope Rules -

Rules for determining the static

scope of a declaration.

for eg :- In Pascal, a static scope rule is used to specify that a reference to a variable x in a subprogram P refers to the declaration of x at the beginning of P, or if declared there, then to the declaration of x at the beginning of the subprogram Q whose declaration contains the declaration of P, & so on.

⇒ Static scope rules relates references with declarations of names in the program text; dynamic scope rules relate references with associations for names during program execution.

Block Structure:-

The concept of block structure - as found in block structured languages such as Pascal, PL/I, and Ada - deserves special mention. Block-structured languages have a characteristics program structure and associated set of static scope rules.

In a block-structured language, each program or subprogram is organized as a set of nested blocks. The chief characteristics of a block is that it introduces a new local referencing environment. A block begins with a set of declarations for names followed by a set of statements in which those names may be referenced.

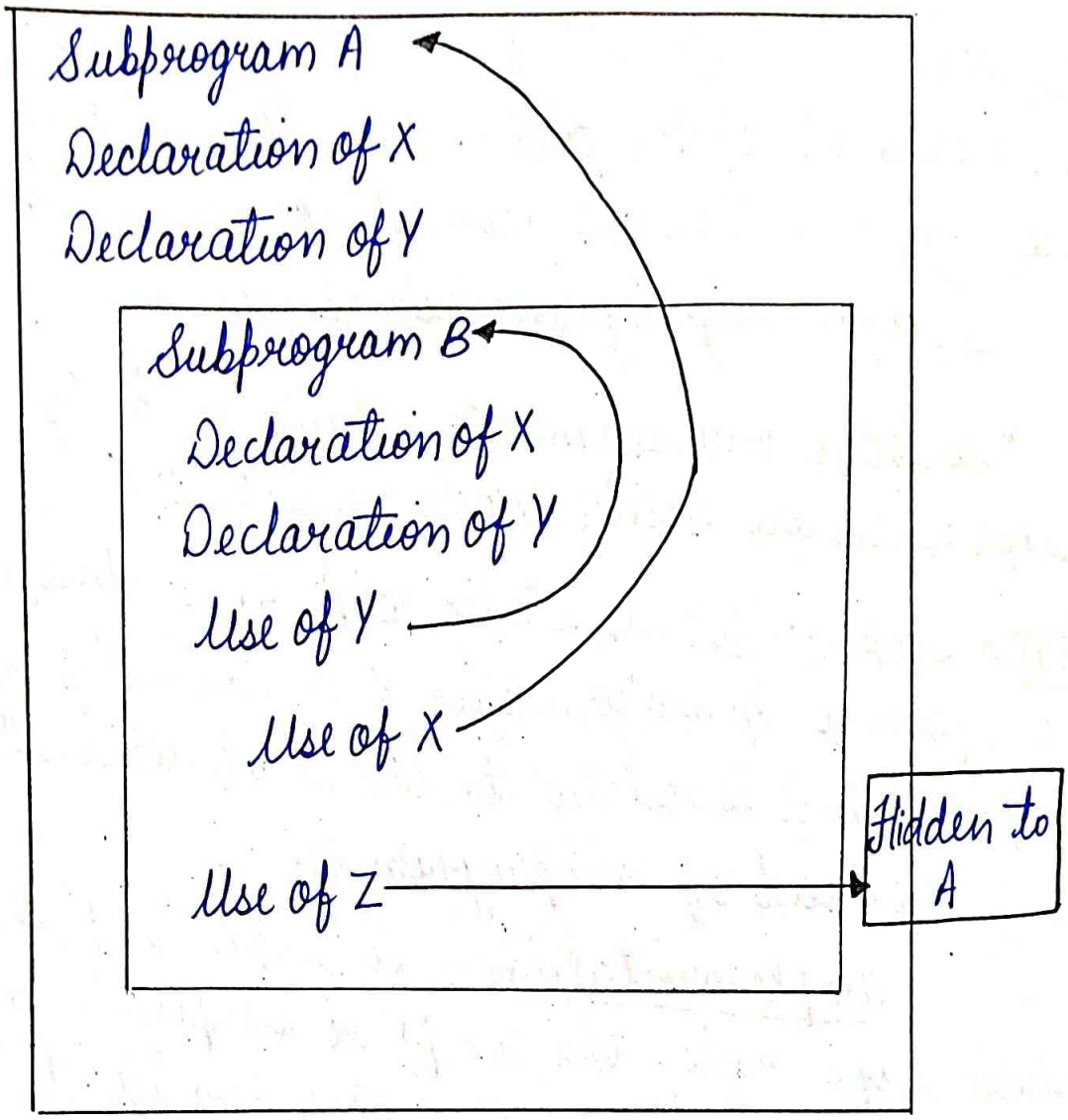
The nesting of blocks is accomplished by allowing the definition of one block to entirely contain the definitions of other blocks. At the outermost level, a program consists of a single block defining the main program. Within this block are other blocks defining subprograms callable from the main program; within these blocks may be other blocks defining

Subprograms callable from within the first-level subprograms, and so on.

The static scope rules associated with a block-structured program are as follows:-

* The declarations at the head of each block define the local referencing environment for the block. Any reference to an identifier within the body of the block is considered a reference to the local declaration for the identifier if one exists.
* If an identifier is referenced within the body of the block and no local declaration exists, then the reference is considered a reference to a declaration within the block that immediately encloses the first block.
* If a block contains another block definition, then any local declarations within the inner block or any blocks it contains are completely hidden from the outer block and cannot be referenced from it.
* A block may be named. The block name becomes part of the local referencing environment of the containing block.

Concept of block structure



Local Data and local Referencing Environments -

The local environment of a subprogram Q consists of the various identifiers declared in the head of the subprogram Q. Variable names, formal parameter names, and subprogram names are the concern here.

For local environments, static and dynamic scope rules are easily made consistent.

Static scope rule: The static scope rule specifies that a reference to an identifier X in the body of subprogram Q is related to the local declaration for X in the head of subprogram Q.

Implementation:- To implement the static scope rule, the compiler simply maintains a table of the local declarations for identifiers that appear in the head of Q, and while compiling the body of Q, it refers to this table first whenever the declaration of an identifier is required.

Dynamic Scope Rule: The dynamic scope rule specifies that a reference to X during execution of Q refers to the association for X in the current

activation of Q.

Implementation: Implementation of the dynamic scope rule may be done in two ways, and each gives a different semantics to local references.

* Retention: The association and the bound values are retained after execution.
* Deletion: The associations are deleted.

Retention and deletion are two different approaches to the semantics of local environments and are concerned with the lifetime of the environment.

⇒ C, Java, Pascal, Ada, LISP, APL and SNOBOLA use the deletion approach.

⇒ COBOL and many versions of FORTRAN use the retention approach.

Implementation of dynamic scope rules in local referencing environments:-

In the implementation of referencing environments, it is convenient to represent the local environment of a subprogram as a local

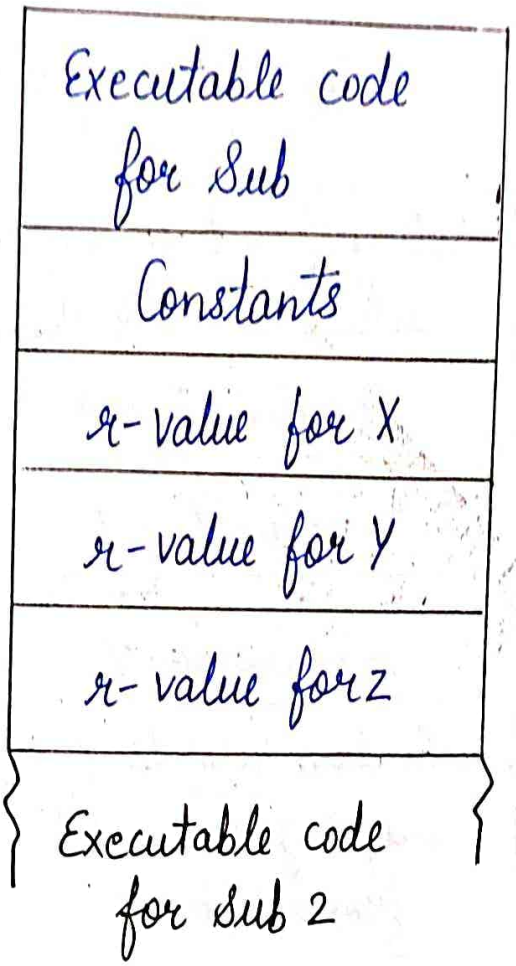
environment table consisting of pairs, each containing an identifier and the associated data object. The storage of each object is represented as a type and its location in memory as an L-value.

Using local environment tables, implementation of the retention and deletion approaches to local environments is straightforward.

RETENTION – If the local environment of subprogram of 'Sub' is to be retained between calls, then a single local environment table containing table containing the retained variables is allocated as part of the code segment of Sub.

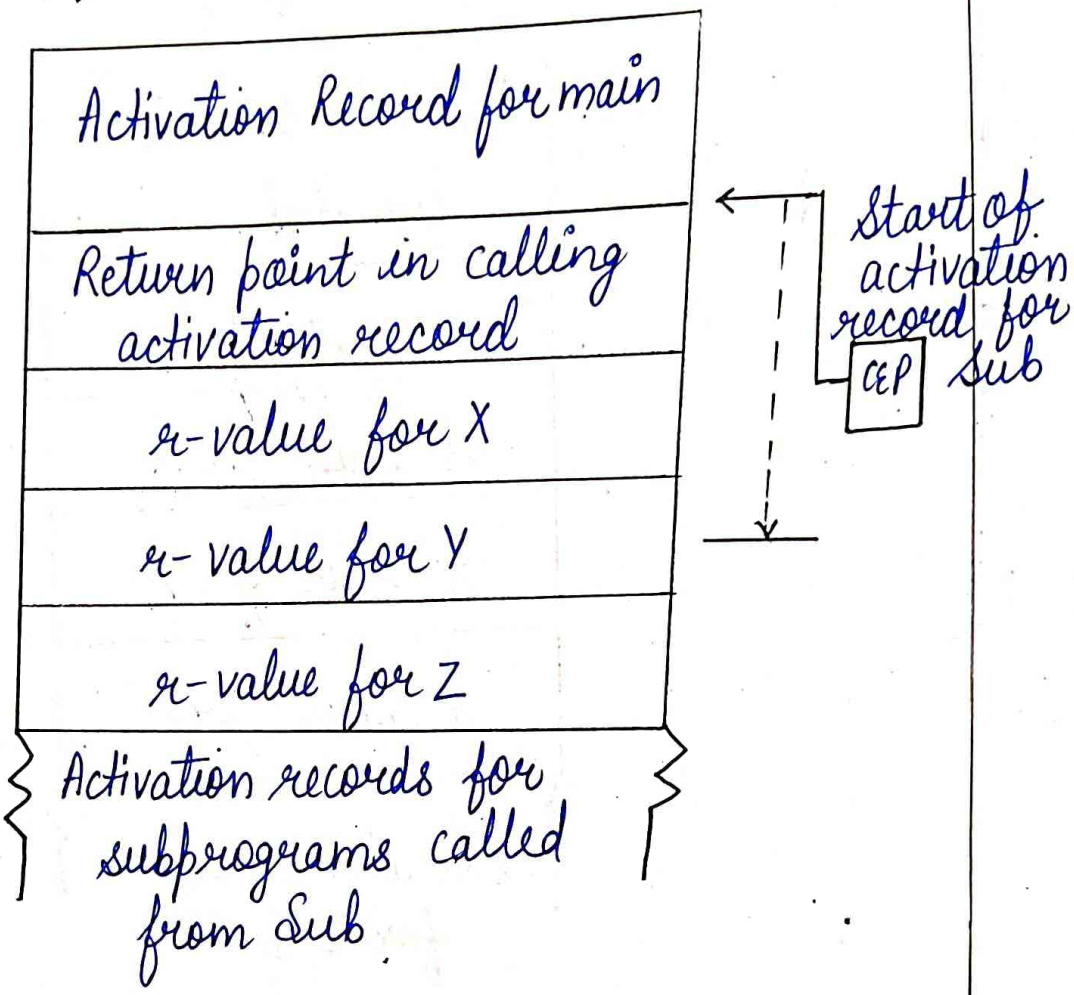
With the implementation of retention for the local environment, no special action is needed to retain the values of the data objects; the values stored at the end of one call of Sub will still be there when the next call begins. Also no special action is needed to change from one local environment to another as one subprogram calls another.

Allocation and Referencing of retained local variables



DELETION:- If the local environment of sub is to be deleted between calls and recreated anew on each entry, then the local environment table containining the deleted variables is allocated storage as part of the activation record for sub.

Allocation and referencing of deleted local variables -



Advantages & Disadvantages :- Both retention & deletion are used in a substantial number of important languages.

⇒ The retention approach allows the programmer to write subprograms that are history sensitive in that their results on each call are partially determined by their inputs and partially determined by the local data values.

⇒ The deletion approach does not allow any local data to be carried over from one call to the next, so a variable that must be retained between calls must be declared as nonlocal to the subprogram.

Data Sharing :-

A data object that is strictly local is used by operations only within a single local referencing environment. Data objects, however, are often shared among several subprograms so that operations in each of the subprograms may use the data. A data object may be transmitted as an explicit parameter between subprograms.

- Parameter and Parameter Transmission –

Parameter: A variable in a procedure that represents some other data from the procedure that invoked the given procedure.

Parameter Transmission :- How that information is passed to the procedure.

Subprograms need mechanisms to exchange data.

Arguments: data objects sent to a subprogram to be processed. This is obtained through
- parameters
- non-local references

Results: data object or values delivered by the subprogram. Returned through
- parameters
- assignments to non-local variables
- explicit function values

1. Actual and Formal Parameters-

A formal parameter is a particular kind of local data object within a subprogram. It has a name, the declaration specifies its attributes.

An actual parameter is a data object that is shared with the caller subprogram. It might be:
* a local data object belonging to the caller
* a formal parameter of the caller

* a non local data object visible to the caller
* a result returned by a function invoked by the caller and immediately transmitted to the called subprogram.

Establishing a Correspondence –

When a subprogram is called with a list of actual parameters, a correspondence must be established between the actual parameters and the formal parameters listed in the subprogram definition. Two methods are described in turn :-

* Positional correspondence
* Correspondence by explicit name

⇒ Methods for Transmitting parameters :-

When a subprogram transfers control to another subprogram, there must be an association of the actual parameter of the calling subprogram with the formal parameter of the called program.

* Call by name: The actual parameter is substituted in the subprogram.

The basic call-by-name rule may be stated in terms of substitution: The actual parameter is to be substituted everywhere for the formal parameter in the body of the called program before execution of the subprogram begins.

* Call by refrence: To transmit a data object as a call-by-reference parameter means that a pointer to the location of the data object is made available to the subprogram.

Passing call by referencing parameters occurs in two stages:-

1) In the calling sub-program, each actual parameter expression is evaluated to give a pointer to the actual parameter data object. A list of these pointers is stored in a common storage area that is also accessible to the subprogram being called. Control is then transferred to the subprogram.

2) In the called subprogram, the list of pointers to actual parameters is accessed to retrieve the appropriate r-values for the actual parameters.

Call by value :- If a parameter is transmitted by value, the value of the actual parameter is passed to the called formal parameter.

1) On invoking a subprogram, a call-by-reference parameter passes it l-value, whereas a call-by-value passes its r-value.

2) In call by value, the formal parameter contains the value that is used.

Call by value result :- If a parameter is transmitted by value result, the formal parameter is a local variable of the same data type as the actual parameter.

Call by constant value :- If a parameter is transmitted by constant value, then no change in the value of the formal parameters is

allowed during program execution. The formal parameter thus acts as a local constant during execution of the subprogram.

Call by Result:- A parameter transmitted by result is used only to transmit a result back from a subprogram. The formal parameter is a local variable with no initial value.

3. Transmission Semantics:-

Types of parameters: input information
output information
both input and output

The three types can be accomplished by copying or using pass-by-reference.

Return results: using parameters
using functions with a return value.

Implementation of parameter Transmission:-

Implementing formal parameters:

Storage: in the activation record

Type:

local data object of type T in case of pass by value, pass by value-result, pass by result.

local data object of type pointer to T in case of pass by reference.

Call by name implementation: the formal parameters are subprograms that evaluate the actual parameters.

Actions for parameter transmission:-

⇒ associated with the point of call of the subprogram.

Each actual parameter is evaluated in the referencing environment of the calling program, and list of pointers is set up.

⇒ associated with the entry and exit in the subprogram.

on entry:
copying the entire contents of the actual parameter in the formal parameter, or copying the pointer to the actual parameter

on exit:
copying result values into actual parameters or copying function values into registers.

These actions are performed by prologue and epilogue code generated by the compiler and stored in the segment code part of the activation record of the subprogram.

Thus, the compiler has two main tasks in the implementation of parameter transmission-
. it must generate the correct executable code for transmission of parameters, return of results, and each reference to a formal-parameter name.
. it must perform the necessary static type checking to ensure that the type of each actual parameter data object matches that

declared for the corresponding formal parameter.