Subprograms Are Implemented Using Activation Records (AR):

When the main/subprogram S calls subroutine/function F, the execution control will be changing from “caller” S to the “callee” F (upon encountering a subroutine CALL statement). Since we need to resume the execution of S after returning from F, we need to save the “state” of the caller S in some data structure (e.g., record), that is called “Activation Record”. Such saved state will be used to restore the activation of the caller upon the return from the callee subroutine, i.e., after executing the RETURN statement at the callee.

The state of a computation (program in execution) consists of:
1- All of its local&formal parameters (if any), and any temp values in registers (TEMP).
2- The instruction pointer (IP) that points to the next statement to be executed. At the subroutine invocation, the IP will hold the resume (return) address in the caller code.
3- Pointer to the caller’s AR called “dynamic link” (DL).

Steps taken upon subroutine invocation (call):
1- Allocate an AR for the callee.
2- Save the “state” of the caller in its AR.
3- Compute the actual parameters and transfer them to their corresponding formals in the callee AR.
4- Place a pointer to the callee AR into the caller AR (the DL).
5- Transfer the execution to the beginning of the callee code.

Implementation of Non-recursive Call, in FORTRAN:

Code in the caller S:
Call F(P1, P2, P3, …, Pn) →

Save Temp values in M[AR(S)].TEMP;
M[AR(S)].IP := resume address (IP);

M[AR(F)].PARM[1] := compute P1 (address in case of by-reference);
M[AR(F)].PARM[2] := compute P2 (address in case of by-reference);

***
M[AR(F)].PARM[n] := compute Pn (address in case of by-reference);

M[AR(F)].DL := Address of AR(S);

Go to entry (F)

resume:
    Restore registers from M[AR(S)].TEMP;
    If F is a function then get its returned value.
Code in the callee F:
entry (F): …the machine code of F …
........
RETURN →
If F is a function, place the returned value where accessible to the caller;
Go to ( M[M[AR(F)].DL].IP);

Data Structures:

- **Operators are overloaded**: it is when an operator is used with integer, double/float, and complex operands, e.g., the “+” in FORTRAN, i.e., the “+” carries more than one meaning (problem?!? If so, how do we solve it?). It is an ad-hoc polymorphism.

- Early FORTRAN did not allow mixed mode expressions (of more than one type operands), instead of implicit coercion, the user must explicitly use conversion, e.g., I = IFIX (X + FLOAT (I)) when adding real X to integer I.

- Newer FORTRAN versions permit mixed mode, running the risk of getting a wrong results if the user is not careful when writing the expression (oops!, Security Loophole). For example, \( X^{(1/3)} \) should return the cube-root of X, instead it returns \( X^0 \), since the integer division 1/3 returns 0. Moreover, we can easily lose precision when assigning real to integer, e.g., I = X (if X = 0.999999999 we still get 0 stored in I).

- In addition of overworking the integer type with labels, it is overworked with character strings, since they have similar implementation as one word of bits (!?!). FORTRAN (before 77) allows the H-format strings (Hollerith) to be read integer/real variables (!!) and passed as an actual for a corresponding integer formal parameter, where we can easily increment a string by one. look page 60 for an example of such issue (Security Loophole)!

Name Structure:

As shown before, data structure will structure data, and control structure will structure the control flow, and “name” structure will organize the visibility (accessibility) of all names that are used in the program.

**Primitives of name structures**: binding constructs: the declaration statement “INTEGER A, B, C” binds A, B, and C names to the type
INTEGER and the addresses of their corresponding allocated appropriate size memory slots, in the static symbol table.

**Environments Determine Names/Constructs Meanings:**

The context or environment of any language construct (e.g., statement(s)) is the visible set of definitions (e.g., declarations) to such construct that gives its meaning through the definitions of all of its involved variable names. For example, X = COUNT(I) statement could be interpreted in many different ways/meanings depending on the definition of X, COUNT (a function or an array?!), and I.

**FORTRAN Variable Names are Local in Scope:**

The program is divided into disjoint subprograms (environments), for independent abstract implementation. The details of the subprogram concrete implementation, e.g., formals names, local names, etc, are hidden from the user in a separate environment. All the caller knows is the info in the subprogram interface!

In FORTRAN, only subprogram names are global in scope. But, all locally defined names in a subprogram (or the main program) are local in scope to such subprogram; and never visible to the outside world.

The scope of a “name binding” is the region of the code over which such binding (e.g., type declaration) is visible.

For a better understanding of name structure, we will study the “contour diagram” mechanism which visualizes the program modules as boxes; each is made of one-way mirrors that allow inside-out visibility of name binding, but never outside-in!

Look Figs 2.8 and 2.9, pages 79-80

**COMMON Blocks:**

Since there was no global declarations (scoping), except for subprograms names, FORTRAN facilitated inter-subroutines-communication via global blocks of storage. They thought that communicating via the sub’s interfaces (parameters) would constitute a violation to the sub’s abstraction!
Ex:

```fortran
SUBROUTINE SUB1( A, B)                    SUBROUTINE SUB2( H, M)
REAL X(100)                                             REAL C(50), D(100)
INTEGER Y(250)                                       INTEGER E(200)
COMMON /BLOCK1/  X, Y                        COMMON /BLOCK1/  C, D, E
*** ***
END                                                   END
```

Advantages: Better memory utilization via shared memory space, and facilitates inter-subroutines communications, while maintaining subroutines’ abstraction.

Disadvantages: Insecure *aliasing* of more than (possibly different types) name to the same space in memory, hence inadvertent storage sharing between different-types names; especially with *no type/count check* for matching, *lack enforcement of re-initialization before use* (!?), and *lack of run-time tagging* of the shared memory to know the type of the currently residing value, by the compiler, at different block statements (Security Loophole).

**EQUIVELANCE**: Memory sharing within the same subprogram. Yet, same Security Loophole as in COMMON (aliasing and no run-time tagging). It is similar to the “union” in C and variant records in Pascal/Ada.

Both EQUIVELANCE and COMMON are deprecated features to be removed in later versions of FORTRAN (after F90). Yet, they are an example of gaining efficiency & abstraction versus lose of security!