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***" (***AR***). 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 **I**nstruction **P**ointer (**IP**) that points to the next statement to be executed. At the subroutine invocation (i.e., call time), the **IP** will hold the resume (return) address in the caller code.
3- Pointer to the caller's AR called "***D****ynamic* ***L****ink*" (**DL**).

Steps taken upon subroutine invocation (call), via compiler generated code to be executed at run-time (**overhead!**):

1- Allocate a new **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 ***caller*** AR into the ***callee*** AR (the **DL**).
5- Transfer the execution to the beginning of the ***callee*** code.

Implementation of Call, in FORTRAN [*Non-recursive*] (non-stack model):

Code in the **caller** module ***S***:

Call F(P_1, P_2, P_3, …, P_n) →

1) Create the Activation Record of the Callee [AR(F)], making spaces for all of required fields that hold:
locally declared name, formal parameters, Return Address (IP), other pointers, ...

2) **Save the state of the Caller in its AR(*S*):**
Save Temp values in M[AR(***S***)].TEMP;
M[AR(***S***)].IP := **resume** address (IP); --Save the return Address in the Caller AR(*S*)

3) **Compute the *actual* parameters and transfer them to their corresponding *formals*, in the *callee* AR(F):**

M[AR(***F***)].PARM[1] := compute P_1 (address in case of by-reference);
M[AR(***F***)].PARM[2] := compute P_2 (address in case of by-reference);
***
M[AR(***F***)].PARM[n] := compute P_n (address in case of by-reference);

4) **Place a pointer to the *caller* AR(*S*) into the *callee* AR(*F*) (the DL)**:
M[AR(***F***)].DL := Address of AR(***S***);

5) **Transfer the execution to the beginning of the *callee* code:**
Go to entry (***F***)

resume:
(First, carry out some housekeeping before resuming the execution in the caller code)
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);

- ***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 does the compiler solve it?). "**Operators are overloaded**" 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), the compiler does not warn the user of such problem.

- In addition of ***overworking*** the integer type with labels, the integer type 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 stored in integer/real variables (!!) and passed as actual parameters for their corresponding integer formal parameters, where we can easily increment a string by one (**meaningless operation!**). See page 70 (text) for an example (FUNCTION **ISUCC(N)**) 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. A ***name*** is a *handle* that allows us to manipulate data and abstraction of code (procedures, functions, modules, ...).

Primitives of name structures: binding constructs: the declaration statement "**INTEGER A, B, C**" ***binds*** the names **A, B**, and **C** to the type **INTEGER** and the addresses of their corresponding allocated appropriate size memory slots, in the static symbol table. The run time execution of the compiler generated code for name declarations, and possible initializations, is called "***Elaboration***".

The ***visibility*** of any declared **name** (e.g., id, proc/function, const), which is critical to its legitimate access, is discussed next.

Environments&Scopes

Imagine a name declaration as a flashlight beam, then the ***scope*** of such name declaration is wherever we can see such light beam!
Hence, the ***scope*** of a **name *binding*** is the region of the code over which such *binding* (e.g., type declaration) is "***visible***" (accessible).

Consequently, the ***environment*** of any language construct (at any point of the code) would be the set of different visible names' declarations (light beams) that can be seen at such language construct.
Hence, the ***context*** or ***environment*** of any language's **construct** (e.g., statement, expression) is the set of all visible definitions (e.g., declarations), to such construct, that defines its semantics through the definitions of all of its involved variables' names.

For example, the language construct "**X = COUNT(I)**" (statement) might be interpreted in many different ways (i.e., having different semantics) depending on the definition of the visible names **X**, **COUNT**, and **I**. Hence, the semantics of the statement will depend on the visible definition of **"COUNT"** (a function or an array name).

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.

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

Look Figs 2.8 and 2.9, pages 79-80

Next example if taken from: http://www.csse.monash.edu.au/~lloyd/tildeProgLang/PL-Block/

```
-- Main Program
begin                        --lex level 1
  real v1;                   --level 1 obj 1
  real v2;                   --level 1 obj 2
  proc A =                   --level 1 obj 3
      begin                   --lex level 2
        real v3;              --level 2 obj 1
        proc B =              --level 2 obj 2
            begin             --lex level 3
              real v4;        --level 3 obj 1
              ...; v4 = v1+ v3/v2; ...
            end{B};
        B                     --call B
      end{A};
  proc C =                    --level 1 obj 4
      begin                   --lex level 2
        real v5;              --level 2 obj 1
        proc D =              --level 2 obj 2
            begin             --lex level 3
              real v6;        --level 3 obj 1
              A               --call A
            end{D};
        D                     --call D
      end{C};
  C                           --call C
end
```

-- LA, CS UWA

Test your understanding of the topic. Try to answer the questions:

A) If we are inside B answer the following:

a) Give an example of each:

i) locally defined name.

ii) globally defined name.

iii) non-locally defined name.

b) list all visible names.

c) what is the environment of the assignment statement within the code of B?

d) what is the scope of the name "v4", i.e., all boxes that v4 is visible?

B) If we are inside Main prog. answer the following:

a) what are the names that the Main program can see (use), i.e., visible?

b) what are the procedures that the Main program can call?

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!

Example:

```
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 examples of gaining *efficiency* & *abstraction* versus losing of *security*!**

Did we really gain true abstraction via the above two facilities?

Were they sort of *virtual* global declaration facility?

Scope resolution Operator "::":

Why? To allow the safe use (correct access) of ***local*** names that are similar to previously declared ***global*** (not any non-local) names.

C++ example:

```
#include <iostream>

using namespace std;

int amount = 123;   // A global variable

int main() {
   int amount = 456;   // A local variable

   cout  << :: amount << endl   // Print the global variable
         << amount << endl;    // Print the local variable
}
```

What do you think? Good to have? Free lunch?

Syntactic Structures

- **Restricted format (until FORTRAN 77):** column 1-5 label, 6 for commenting the line, 7-72 for coding statements.

- The following FORTRAN features are **Security Loopholes** in the language:
  i) lack of **reserved** words ii) **ignoring blanks** everywhere
  iii) **implicit** name declaration
  iv) the close ***syntactic* similarity** of two different ***semantics***.

Examples:

1- The user might, inadvertently, just by mistaking the "," for the "."write:

DO20I=1.100 instead of **DO20I=1,10**

The above two constructs are of two totally different meanings (The first one: variable DO20I is assigned the value 1.1, whereas the second represents a "for" loop, changing I from 1 to 100), hence totally yielding different results when executed!

2- **DIMENSION IF**(100)

A user mistake such as writing

"**IF(I-1)=123**" instead of "**IF(I-1)1,2,3**", forgetting to add the commas, will not be caught, by the compiler, and the *(I-1)*th element of the **IF** array will be assigned the value 123.