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:

**Why?** 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! One factor was the lack of powerful aggregate data structure facilities, e.g., records/structures, to express complex objects such as symbol tables without spelling out their implementation details (as being done when using parallel arrays!)

**How?**

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, the lack of enforcement of re-initialization before use (by the compiler), and lack of run-time tagging of the shared memory to know the type of the currently residing value at different block statements (Security Loophole).

**EQUIVELANCE:**

**Why?** 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 & “ad-hoc” abstraction versus lose of security!**

Did we really gain true abstraction via the above two facilities? Were they sort of virtual global declaration facility?

**Syntactic Structures:**

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

Due to ignoring blanks everywhere and the lack of reserved words in FORTRAN, we would have the following problems (**Security Loopholes**):

1- The user might write “DO20I=1.100” instead of “DO20I=1,100” which would give totally different result when executed!

2- DIMENSION IF(100)
   If “IF(I-1)=123” is written instead of “IF(I-1)1,2,3”, such error will not be possibly caught and the (I-1)th element of the array IF will be assigned 123.

The above security loophole is due:
   i) lack of reserved words     ii) ignoring blanks everywhere
   iii) the syntactic similarity of two different semantics.
ALGOL60 (Naur, 1960)

- Elegancy and generality (powerful universal ALGOrithmic Language) are the main design goals of the ALGOL.
- Sample ALGOL60 code: Fig 3.3 page 102,
- Major contributions (new language features):

  1- Free format (no FORTRAN restrictions!).
  2- Block structuring the code, introducing the following structuring tools:
     i) “Blocks” and “compound statements”.
     Ex. Block: \texttt{begin} declaration-sequence; statement-sequence \texttt{end}
     They define nested scopes (look example code and its contour diagram pages 102 and 103, respectively.

Why blocks?

1- To gain what is meant by COMMON in FORTRAN, yet eliminating the disadvantages! In order to share a common declarations among a number of procedures without being part of their interfaces (hence no chance of inconsistency of different types/names/numbers, or violating abstractions), a block will encapsulate all, allowing for efficient and secure access to such declarations among all using procedures. Thus, blocks aid in the construction of large software. (pages 105-107)

2- They define a separate scope with all declarations, of which there might be a huge arrays (large spaces) that they should not be in the system stack (as part of the callee’s AR) when there is no need for them. Hence, if such arrays exists within some procedures, we simply encapsulate them and their using code only in an internal block. Now the huge array space is part of the block AR and not the procedure AR. Thus when we exit the block, its AR will be deleted from the stack, deal locating the large space AR, and returning to a much smaller procedure AR. On the other hand, if we do not use blocks, the procedure AR would be allocated huge space (due to the large array), and kept in memory while executing the “entire” procedure code. (pages 112-114).

Syntax rules:

\texttt{statement: simple-statement | compound-statement ;

compound-statement: begin statement-sequence end ;

Why compound statements? To group multiple statements into a single statement to be used wherever it is needed (e.g., if-then-else, for, ...).
Potential problem! If we start with one statement, then we add more later,
Unless we compound with “begin” “end” we might have an undetected error.

```
x := 0; y := 1;
for i := 1 to 2 do
  x := x + i;
y := y * x;           (* we added this but did not make the two statement a compound statement, using begin-end)
```

ii) Powerful structuring constructs: “switch”, “for”, nested “if”s, conditional expressions. (look point “8” below)

3- **“Stack”** model of computation which facilitates the following new features:

i) **Recursion** (power vs. speed/readability):
   The power of recursion is stemmed from the math proof by induction!
   Hence, the recursion process solves the main problem, in many steps, each with lesser size input, utilizing the same module code (reusability). The major drawback is the overhead of the extra (machine) code for module invocation/return, which slows down the recursive solutions compared to its corresponding iterative approach. Moreover, sometimes recursive solutions are “a bit” harder to read.

ii) **Dynamic Arrays**, with variable subscript range(s), allowing dynamic array size allocation, at run time (for storage efficiency). Instead of committing to a max size array, we dynamically allocate the exact needed size according to each application.

iii) **Dynamic binding of names to memory spaces**, at run time, due to the new recursion feature! (the static type binding still hold).
   **Question:** Why does recursion enforces dynamic binding on names to memory locations?