FORTRN (just before 77): an Evaluation

Pioneer HLL that aimed solely for efficiency of solving scientific analytical (numerical) problems. In FORTRAN power, security, type abstraction, clarity were in the back seat!

Major contributions: (compared to the low level assembly language)

1- Subroutines/functions as abstraction modules.
2- Library **intrinsic** (built-in) functions compiled separately and kept in a system library (true module abstraction), e.g., sqrt(), abs(), sin(), cos(), exp(), log(), …
3- Efficiency → fast code execution, minimal machine coding and run-time system overhead; very static, everything is known at compile time: name bindings, number and size of activation records, no recursion.
4- Built-in true abstract data types (ADTs) arithmetic operators (+, -, /, *, …).
5- Language typing system.
6- High Level statements (flow-control, input/output, imperative statements.

FORTRAN has the following deficiencies:

1- Insecure language, many security loopholes, in the language defined semantics and its typing system (?)
2- Not very powerful language, in general.
3- Tough to write code, restricted format (old versions only).
4- Input/output is system defined modules, with restricted and hard formatting for the users.

Yet, it was one of the most used languages, maybe until now!

----------------------------------------- 2.8.2007 ---------------------------------------

**ALGOL60** (Naur, 1960)

(ALGOrithmic Language)

- **Elegancy** and **generality** are the main design goals of the ALGOL (general-purpose, powerful, and universal language).
- (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:** \begin{verbatim}
begin declaration-sequence; statement-sequence end
\end{verbatim}

They define nested scopes (look example code and its contour diagram pages 102 and 103, respectively.

**Why do we need 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:**

\begin{verbatim}
statement: simple-statement | compound-statement ;
compound-statement: begin statement-sequence end ;
\end{verbatim}

**Why do we need 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.

\begin{verbatim}
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)\end{verbatim}
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 enforce dynamic binding of names to memory locations?

iv) Nested scopes, allowing subprograms nesting, for better abstraction.
Remember, the scope of all declarations at the most inner box is not visible to any other outer boxes. Whereas, the most outer scope (all declarations in the box that encloses all inner boxes) is visible to all enclosed boxes. When we nest subprograms (sub1, sub2, …), within a hosting module (say M), we mean to create separate abstractions, i.e., sub1, sub2, …, within M. Hence, any declaration inside sub1 and sub2 should not be visible to the outside of sub1, sub2.

Question: Why do not we just write M, sub1, sub2, … at the same level without nesting, i.e., why nesting subs’ abstractions?