FORTRN (just before 77): an Evaluation→

It is a pioneer HLL that aimed solely at the ***efficiency*** of solving scientific analytical (**numerical**) problems. In FORTRAN *true* HLL power, security, type abstraction, clarity "took the backseat"!

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

1- Parameterized subroutines/functions as abstraction modules.
2- Library **intrinsic** (built-in) functions compiled separately and kept in a system **library** (**true** abstraction) modules, e.g., sqrt(), abs(), sin(), cos(), exp(), log(), …, other important numerical analysis modules.
3- **Efficiency**→ fast code execution, minimal machine coding and run-time system overhead; **very static**, everything is known at compile time: name-address bindings, number and size of activation records, **no recursion**.
4- Built-in **true** abstract data types (ADTs) arithmetic operators (+, -, /, *, …).
5- Pioneer in the introduction of HLL's **typing system**, though primitive!
In addition, it introduced **high level statements** (flow-control, input/output, imperative statements).

FORTRAN has the following deficiencies:

1- **Insecure** language, **many security loopholes**, in the language definition and its typing system (?)
2- **Not very powerful** language, in general, no recursion, lack of rich typing system.
3- Tough to write code, restricted format (old versions only).
4- Input/output statements are **system defined** modules, restricted and hard to format program output.

Yet, it was one of the most used languages up to early 80's.

We *owe* it to FORTRAN as an early science advancing HLL to be the leaders in the space exploration, science, and technology—NASA/DOD— research advancement.

ALGOL60 (Naur, 1960)

(ALGOrithmic Language)

- **Elegancy** and **generality (power)** 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):**

I. Free format (no FORTRAN restrictions!).

II. Block structuring the code, introducing the following structuring tools:

i) "*Blocks*" and "*compound* statements".

Ex. *Block*: **begin** declaration-sequence; statement-sequence **end**
They define nested scopes (look example code and its contour diagram pages 102 and 103, respectively.

Why do we need *blocks*?

1) controlled visibility.
To gain the functionality of 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 by all subject procedures. Thus, blocks aid in the construction of large software. (pages 105-107)

2) Efficient use of memory.
When a ***large*** data item is needed declare, **use**, and take it **out** the system memory (stack) after finishing with its usage, ***safely***, not as in FORTRAN's EQUIVELANCE mechanism
Blocks 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 is shared among a number of procedures, we simply encapsulate them and their using code in an internal block. **Now the huge array space is part of the block's AR and not the sharing procedure's AR**. Thus when we exit the block, its AR will be deleted from the

stack, de-allocating the large space AR, and returning to a much smaller procedure AR.
On the other hand, if we do not use blocks, the procedure's AR would be allocated huge space (due to the large array), and kept in memory while executing the "entire" procedure code. (pages 112-114).

```
begin
  integer N;
  Read Int  (N);

 begin
     real array  Data [1:N];  -- Dynamic array!
     real  sum,  avg;
     integer  i;
     sum := 0;

     for i  := 1  step  1 until  N  do
        begin   real  val;
          Read  Real (val);
          Data[i] := if val < 0 then -val  else val   -- Conditional assignment
        end;

     for i := 1 step 1  until  N  do
         sum := sum + Data[i];
     avg  := sum/N;
    Print Real(avg)
   end
end
```

Syntax rules:

statement: *simple-statement* | *compound-statement ;*
compound-statement: **begin** statement-sequence **end** ***;***

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 one 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 statement, but did not make
                          the two statement a compound statement,
                          using begin-end)
```

ii) Powerful structuring constructs: "***switch***", "***for***", nested "***if***"s, conditional expressions.

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. *"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 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?

iii) **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?

Name Visibility -*SCOPEs*- Techniques

Static Scoping: The meaning of a name is interpreted according to the static (lexical) structure of the its hosting program module. For example, a non-local variable name "X" which is defined in module M_{use} will have its meaning (type declaration binding) from the environment of M_{use}'s defining module, say M_{def}, according to the static contour diagram of the program, regardless of the caller of M_{use}.

Dynamic Scoping": The meaning of a name is interpreted according to the run time dynamic behavior (calling sequence) of the its hosting module, M_{use}. A non-local name in M_{use} gets its meaning from the environment of the caller of M_{use}, and not the environment of its definer module M_{define}. Hence, the contour diagram is of no use in case of dynamic scoping! Moreover, it forces the inefficient dynamic type checking. Yet, its major advantage is the "power" of polymorphism, where it facilitates the manufacturing of generic polymorphic modules, where the same code is interpreted differently at run time according to their callers.

Question: Can we draw a static name visibility "contour diagram" for the dynamically scoped languages?? Justify your "yes/no" answer!

Question: In which special case(s) the dynamic and static scoping policies will work the same, i.e., no difference? (Hint: two cases)

Newly introduced Feature	"Static Scop."	VS.	"Dynamic Scop."
1) Code readability and understanding-- **The code's static structure agrees more with the prog. run-time behavior.**	YES		NO
2) Enforcement of run-time type checking (**inefficiency**).	NO		YES
Efficiency (execution speed)?	YES		NO
3) Power (genericity-Polymorphism)?	NO		YES
4) Security—the increase of the potential of programming mistakes?	NO		YES

Example of Static vs. Dynamic Scoping:

```
Program TEST;
    var a, b :integer;
    c : real;
    Procedure P1(var x:integer);
       var  a: real;

       Procedure  P11(y :real);
        var r : real;
        begin    r  :=  y  +  a;   writeln (r);  end;

       Procedure P12(var z : integer);
        var a: integer;
        begin(* P12*)
          a := 20;   P11(1.5);
        end(*P12*)

    begin (* P1*)
      a := 2.5;
      P12 (x);
    end;

   begin (* TEST*)
     a := 15;   b := 20;  c := 1.7;    P1(a);
   end (*TEST*)
```

→ The output (value of "r") is ***4.0*** (1.5+2.5) in case of ***static*** scoping, and ***21.5*** (1.5+20) in case of ***dynamic*** scoping

IV. **Parameter passing:** i) ***"by-value"*** (user view of "input" parameter) and ii) the very powerful ***"by-name"*** (default, input/output parameter).

Pass *"by-value"*: The value of the actual parameter, at the caller side, is placed in its corresponding formal, the callee's AR. For the first time we can say that the **user view** is considered when we think of by-value parameter is an "***input***" parameter. Hence, now the compiler can guard against misuse of the input parameter, e.g., when used as an output parameter (l-value), by the programmer. If such protection exists, the compiler, for efficiency considerations, can implement the passing of value or reference (internally) for scalars and composite structures, respectively.

Pass *"by-name"*: The compiler generates a machine code, function like, called ***"thunk"*** for every actual parameter, at the caller side, instead of carrying out the calculation of the final value of the actual. The *thunk* will range from just a very simple single reference (address), in case of a simple variable name actual parameter, to a very complicated code of an expression actual parameter involving many **references** of all involved names in the expression. All references in the *thunk* will point to the caller's AR slots, specifically to where the involved names in the actual parameter expression. You can always think of pass by-name as **textually substituting the formal parameter by an exact "textual" copy of its corresponding actual parameter everywhere** in the callee's code. Hence, it is a very powerful mechanism since the *thunk* can be a very complex construct (e.g., a function call), where its evaluation is delayed until the evaluation of its formal parameter (lazy evaluation). If any of the involved names in the *thunk*'s code changes, the next evaluation of the formal parameter will be different, which will result in a different value from its last invocation (**polymorphic power**??)

"by-name" is very **powerful** (see the **polymorphic** **Jensen's device** page 131):

```
real procedure Sum ( k, L, u, ak);
   value L, u;
   integer k, L, u; real ak;
  begin real S; S:= 0;
    for k := L  step 1 until u do
          S := S + ak;
    Sum := S;
  end;
x := Sum (i, 1 , n, A[i]);
x := Sum (i, 1 , m, Sum (j, 1 , n, A[i,j]));
```

But *by-name* is also **dangerous** (see the "swap" example on page 133).

```
procedure Swap (x, y);
  integer x, y;      -- (by default x & y are passed by name)
  begin integer t;
      t := x;
      x := y;
      y := t
  end
```

Using **Swap:**

1) i := 5; j:= 7;
   Swap (i, j); -- Then, i will be 7 and j becomes 5

2) i:= 1; A[1] := 2 ; A[2] := 3; A[3] := 4;
   Swap (A[i], i); --> **begin integer t; t := A[i]; A[i] := i; i := t end;**
   Now after the Swap(A[i], i) call -- > i is 2 , and A[1] is 1

3) i:= 1; A[1] := 2 ; A[2] := 3; A[3] := 4;
   Swap (i, A[i]); --> **begin integer t; t := i; i := A[i]; A[i] := t end**
   Now after the Swap(i, A[i]) -->

Question: Does by-name facilitate **passing** a "**function**" as a **parameter**?

Question: When would it be the case where by-name and by-reference are the same? [Think the nature of the actual parameter]

V. "0-1-∞" design principle: (page 117)

For any introduced feature in the language, you do not ask the users to remember any specific restricting numbers; or if you must, it should be either 1 or any number.

For example, in ALGOL (theoretically) there is no limit on the *label* length and block/procedure nesting depth.

Generality of Control Structures: (page 121)

- Extending the ***"if"*** statement of FORTRAN to ***"if-then-else"***.
- Extending the "***DO***-loop" statement of FORTRAN to "for" statement
- Definite looping: **for** i := 1 **step** 2 **until** N * M **do** statement;
- Indefinite looping: **for** NewGuess := Improve (OldGuess)
  while abs(NewGuess – OldGuess) > 0.0001 **do** statement;

In my view, it is a bit confusing syntax; mixing the definite and indefinite semantics for looping!

Here is another very powerful "for"! (I do not think any other languages would have a more general one!): (top of page 138)

(An example of power vs. simplicity/readability)

```
for i :=  3,  7,
          11 step 1 until  16,
           i/2 while i ≥ 1 ,
            2  step  i until  32
  do  print (i)
```

the output (values of i) is:

3 7 11 12 13 14 15 16 8 4 2 1 2 4 8 16 32

- The selections statement ***"switch"***:

```
begin
   switch  S  =  L  ,  if  i > 0  then  M  else  N ,    Q ;
          (evaluate i, j)
    goto S[j];   (* if  j=2 then the value of i will decide jumping to either M or N *)

   L:  ****
       goto done
   M:  ****
       goto done
   N:  ****
       goto done
   Q:  ****
  done:
end
```

* Notice that the "***switch***" and "***for***" statements are "baroque".

*ALGOL solved the dangling "else" problem (DE):
If B then if C then S else T;
Does the "else" relates to the first or second "then"
Solution: ALGOL restricted the consequent of the "if" must not be another if statement:
Hence it is **illegal** to write "if A then if B then D else C

* Remember that the lack of defining reserved words in FORTRAN was a major factor that resulted in a security loophole.

Three Lexical conventions for words:

Reserved words: reserved for the language use, can not be used by the user as ids!
Ex: if, procedure, begin, end, …
Used by most modern languages!

Keywords: ALGOL approach, used by the language and unambiguously marked to be used by the language, yet the programmer can use them for ids if they are not marked! (marking: boldface, preceded with #, surrounded by quotes)

Keyword in context: words are keywords only when expected in their context:

Example:

```
IF IF THEN
  THEN = 0
ELSE
   ELSE = 0
```