FORTRAN

(Old Designed Versions-I:IV- Only) [Backus 1956, IBM]

- First widely used HLL, still used in some valuable applications such as weather and climate modeling, and other computational application in fields such as fluid dynamics, economics, physics, and chemistry; with over 57 years of development from FORTRAN-I to FORTRAN 2008.

The main original design goals of old versions of FORTRAN:

1) **Efficiency** 2) A HLL for Engineering Numerical Applications.

- All needed storage of the program is decided at compile time, minimal invocation of the OS.

- Simple *built-in* typing system, static, and few types: integer, real, complex, double, and arrays (newer versions added: Boolean, character string and file types).

- No ***Type*** definition(typedef) or user defined data abstractions, only arrays and fixed size char string (no records--*structures*). In addition, subroutines and functions might be defined (abstraction).

- Parameter passing initially by-"*reference*" (security problem) then by-"*value-result*".

- FORTRAN programs are divided into *disjoint* subprograms' environments, and a main program.

- **Program structure: declarative and imperative sections:**

1) **Declarative section** serving the following functions:
   i) Allocating memory spaces of specified sizes (based on the name declared type) to hold the declared names; bind them statically (in FORTRAN) for the entire life cycle of the program execution.
   ii) Possibly, assigning initial value (if given) to the name's allocated memory space.
   iii) Names' declarations are used by the compiler to secure their manipulation in the imperative code section.

2) **Imperative section** holds one of the following statement types:
   i) Computational: X = Y/Z + F(4)
   ii) Control Flow: GO TO, DO .. CONTINUE, IF (L1, L2, L3) I, CALL SUB1(X,Y,Z),…
   iii) Input Output: READ, PRINT

2.2 Control Structure

- The "**GO TO**" is a simple, yet very powerful statement, known to be the ***workhorse*** of the of the control flow. It led to formation of a "bad" control structure, violating the structure principle (p. 49).
- There are three types of the "GO TO":
  i) **Unconditional**: GO TO *label*
  ii) **Computed**: GO TO (L_1, L_2, …, L_n), I
  --- If I = k, jump to L_k label in the label list, 1 <= k <= n; otherwise no jump.
  iii) **Assigned**: GO TO N, (L_1, L_2, …, L_n) [the list of labels **is not used** by the compiler!!!]
  Go to the address placed in N, hence N must be pre-assigned some label address, via the assign statement "ASSIGN <label> TO <id=N>", that places the address of the label into the id N. **It is the responsibility of the programmer to do so (leads to *insecurity*).**

- **FORTRAN's *Security Loophole*:**
  The **similarity** of the computed and assigned "GO TO" above. In addition to the **overworking** of the integer type to carry label's address and integers (**weak typing**), and **trusting the "*user*"** to use the assign statement before any assigned "GO TO" would introduce a great possibility of the CPU jumping to execute at an unknown place in memory (if we are ***lucky***, we get "segmentation violation" run time error, otherwise we get what seems to be "correct" program result, where it is **not**).

Language Typing Systems (general discussion)

- **Language Typing Systems**: A set of language defined types (INTEGER, REAL, ...), and the associated/provided type checking mechanism by the language compiler.
- **What is a *type*?** A type of a name/variable is the set of values (held in some data structure (DS)-- scalar/aggregate) that such name/variable can have and the associated set of operations (OPs) that can work on or manipulate such set of values.
- A *type* is called an Abstract Data Type (ADT) when its DS and OPs are encapsulated together, where the manipulation of its DS is allowed only via its OPs. (true vs. weak ADTs!)
- **Why do we need Types in some HLLs?** Mainly for the following reasons:
  1) **Efficient** allocation of memory 2) Type checking (**security**)
- Generally speaking, there are **two** major **type classifications**:
  1) **Built-in** system: i) ***basic** types*-- integer, real, etc; and
  ii) ***structure*d**—array, records, files,..
  2) **User defined**: ADTs, Classes, Modules, Packets

- Type ***"Coercion"***: ***implicit*** type conversion based on the context of use.

  X + I → X + *FLOAT*(I) X = I → X = *FLOAT*(I) I = X → I = *IFIX*(X)

Explicit vs. Implicit Name Declarations:

FORTRAN has both implicit (language convention: explicit declarations (via a declaration statement), if not then all names starting with “I through N” are considered *integers*, otherwise they are *real*'s. Implicit declaration contributes to the ease of programming, but it could easily lead to a **language’s security loophole** as we will see later, when the compiler considers a misspelled name to be a valid name and uses its "incorrect" value instead of the correct and intended name’s value.

Strong (secure) vs. Weak (insecure) typing systems:

A ***"strong"*** (secure) typing system has to maintain the following:

Not allowing any error to escape all lines of defenses via the aid of the following:
(***defense on depth*** page 51)

i) A **rich enough set of types, to cover all programmed objects in the user domain, not to "*overwork*" any type to represent another type, i.e., more than one set of values**. For example, FORTRAN's weak typing system (with no LABEL type) overworked the type ***INTEGER*** as a LABLE type, by using integer variables to carry labels' addresses. Moreover, the ***INTEGER*** type is also overworked as STRING (*Hollerith* strings are considered integers)! The programmer can make "silly" mistakes without the system being able to detect them (***security loophole***).

N= ISUCC (6HCARMEL) ---

Example of rich typing systems-- Ada, Pascal, Modula, ; where their typing system introduced of the "**enumerated**" types to secure the user program declaration and manipulation of real-life objects in the programmer environment, e.g., animals, cars' brands, office
supplies, etc! Moreover, they introduced "record" type for aggregation of heterogeneous types fields. Moreover, some languages introduced separate type "**label**" and "**pointer**" to avoid overworking "**integer**" to do their jobs!

- The lack of rich typing system forces the "**overworking"** of existing types to carry more than one meanings, e.g., overworking "integers" as address, string, animals, Boolean, etc. Whence, such overworking of types would lead to "***security loophole***" in the language!

ii) A ***secure*** **"type checking" mechanism** (as a part of the language typing system) guarantees the users' correct and safe manipulation of data at all program constructs, e.g., formal actual parameters type&count matching, operands type matching in all expressions; hence no user typing semantics errors can go undetected.

Such **"type checking"** might be of one the following types (efficiency vs. power):

1) **Static**: at compile time for the early detection of program errors ***efficiency.***

2) **Dynamic**: at run time for **polymorphic** ***power*.**

iii) The language must **NOT ever trust** the user of doing the safe/correct/right coding!

- **A secure language has to have a *strong* typing system, and more!**

  In addition to having a strong typing system, for a HLL to be secure, other language definitions/features **(that aim at other desirable features, e.g., power, abstraction, etc)** have to be secure and not leading to undetectable errors, e.g., ***by-name*** in Algol, and ***dynamic genericity*** in Modula-2 (both are **powerful** features to be covered later in the class).

  A language with a weak typing system will overwork some types, and **trusts** the user to use them wisely (securely), hence the language is **insecure;** because it is possible that the user will make undetectable errors.

Future questions for you to explore answering, as the class progresses:

- The following three HLL design factors are **contradicting** (if you gain in one you will lose in the other(s)): **Power**, **Efficiency**, and **Security**. Discuss via examples from existing HLLs' features. [**Hint**: Infinite program execution using recursion, indefinite looping, "*by-name*" parameter passing such as in ALGOL]
- Is it possible that a language with a **strong**/**secure** **typing** mechanism to be still "**insecure**"??!?!

Parameter Passing Modes in FORTRAN

Call "*by-reference*" parameter passing

- If the actual parameter is a l-value, e.g. a variable, its reference is passed to the subroutine
- If the actual parameter is a r-value, e.g. an expression, it is evaluated and assigned to an invisible temporary variable whose reference is passed to the corresponding actual parameter.
- In early versions of FORTRAN, all constant numbers (e.g., 2, 45, 100, 3, …) were stored in the ***literal table*** and their addresses in memory are known at compile time, for run time efficiency. Such addresses would be passed to the corresponding actual!! And, guess what, if we mistakenly change any *formal* parameter, its corresponding *actual* would change at the caller side. **This is a clear case of security loophole that does not relate to the typing system weakness/strength!**

- For example: In a hypothetical FORTRAN-II coding with pass by-reference:

```
SUBROUTINE ALTERNUMBERS(A)
INTEGER A
A = 99
RETURN
END
```

At the main program:

```
CALL ALTERNUMBERS(7)
 I = 5
 J = I + 7          actually the number 7 is now has 99 on its allocated memory slot!
 PRINT ( J)         the output is 104  !!!!!! Such error is undetectable (security loophole)
```

By Reference:

Pros: 1- Efficient when passing huge structure (arrays)
2- the power of aliasing (if you do not care about security!!)

Cons: 1- As In old FORTRAN's, the lack of the clear distinction, by the language, of **input** versus **output** parameters might easily lead to making meaningless operations, like storing values into expressions and changing literal constants.
Hence, the compiler can not stop any value assignment to a constant/expression actual parameter, since all that the compiler sees is a formal parameter name that might correspond to a constant (right-value) or name (left-value) actual parameter.
!!!!!!!!!!!SECURITY LOOPHOLE!!!!!!!!!

2- In some **block structured** languages, "by-reference" and "**global** names declaration" could lead to the ***aliasing*** of two or more names to the same memory location.

Example: Assume HLL's with global name declaration/visibility, modern FORTRAN's:

```
SUBROUTINE ALIAS(A, B, C, Z)     --  Callee
 INTEGER A, B, C, Z
 (Assuming the "CALL   ALIAS (S, S, S, H)", we will have four names pointing
 (aliasing) into the same allocated slot for S, at the caller side, namely: A, B, C, and
   S, in main program)
    A = 10
    B = 20
    C  = 30
    S =  400        (* S is a global variable seen everywhere*)
    Z =  A + B + C
RETURN
END
```

At the main program code, if we have a name S to be global, i.e., visible everywhere:

```
PROGRAM  MAIN      --  Caller
  INTEGER S, H
       (Remember there is an implicit global visibility of S from now on)
  CALL   ALIAS (S, S, S, H)
 PRINT (H)
```

What is the output of the above code? Easy 60!

The output is not 60 as we expect; instead it is 1200 (why!)

3- Possible parameter collision:

If the procedure is called with the same variable as more than one actual parameter.

The ***order*** that the formal parameters are copied back to the actual parameters (at the end of the subprogram) determines the value of the variable.

Example:

```
SUBROUTINE VALUE-RESULT-PROBLEMALIAS(A, B, C)
 INTEGER A, B, C
 A = 10
 B = 20
 C  = 30
 RETURN
 END

 MAIN Program:
 INTEGER S, H
CALL   ALIAS (S, S, S)
PRINT (S)
```

The value of S depends on the order of assignment statements in the above subroutine, since last assignment was for C= 30 the PRINT(S) will print 30, if we switch C = 30 with A = 10 then S will print 10, and so on.

- A disadvantage-- It requires multiple storage for parameters and the time for copying values.

- **Remember that this is a security loophole problem that does not relate to the typing system, instead it associates with the parameter passing mechanism in the language.**

Call "*by-value-result*" parameter passing (frequently used after FORTRAN 77)

Why? It is because of Cons-2 of by reference above.

- CALL BY VALUE-RESULT (Pass-By-Value-Result):
  - the value of the *actual* parameter is used to initialize the corresponding *formal* parameter, which then acts as a local variable in the callee code.
  - at subprogram termination, the value of the formal parameter is transmitted back to the actual parameter.
  - both of the above operations (value/results) are carried out by the "caller", hence there will be no meaningless operation such as copying back a value into a constant! (Eliminating the above change of 7 to be 99 security loophole !)

Homework #1: Exercise 2-15 and 2-16 p. 60-61 in your textbook.
Due: Tuesday, Feb. 11th (in class)