Topics:

1- The position of High Level Languages (HLLs) in the computer system (core hardware, micro-level, macro-level (assembly), OS/HLLs).

HLLs allow us in the feasibility to utilize the hardware (black-box) for the implementation of toughest algorithmic solutions of most problems that we face. Three levels of programming:

1) **Micro-level**: micro coding at the micro architecture, in the micro programmable machines. Very fast execution of code, yet very tough to code, read, and maintain.

2) **Macro-level**: macro-assembly level architecture, a bit easier to read and understand each individual instruction, but at the program level still tough to code, read, and maintain.

3) **HLL-level**: the third layer that works as a virtual machine. At that level the user thinks that the machine understands the English like programming language (HLL). It is called HLL since it is up higher, above the hardware level, than the other two languages (micro and macro languages). It runs slower than the other two levels’ languages, yet it is much easier to code, read, and maintain. At that level the details of the hardware is abstracted away from the programmers with the advantage of making programs machine independent. Each HHL defines a virtual machine (user point of view), as if the system understands the HLL directly!

2- **Why Study HLLs?**

   i) Better understanding of their features, allowing us, when the need arises, to have a smart choice of the best HLL for the implementation of any algorithmic solution!

   ii) Efficient improvement of existing HLLs to fit better our needs.

   iii) Future design of new HLLs.

3- **What makes a “good” HLL?**

   a) Clarity of its syntax and semantics.
   b) Richness and independency of its features and constructs that makes it easy to find suitable one, mix and combine many, for more efficient software implementation.
   c) Its support of abstraction: i) user defined ii) built-in.
   d) Its support of security at: i) development of software ii) run time robustness
   e) Program portability between different platforms.
   f) The cost of program: i) development ii) translation iii) maintenance
   g) Its support of useful software development environment (to the user), e.g., editors, interpreters, graphical user interface.
4- What are the major factors that characterize a HLL?

  a) **Power** (e.g., short syntax that reflects powerful semantics, the programmer can do anything he/she dreams to do! recursion, polymorphism).
  b) **Abstraction** and **modularity**: it helps at all phases of SW design, code reusability, easy maintenance, testing, separate module design.
  c) **Security**: language & applications.
  d) Program’s **speed of execution** (independent of the programming style!).
  e) **Readability**: how easy to get the semantics from its corresponding syntax.

In the design of a general purpose HLL, our main challenge is to find the optimal point between the above **contradicting** factors!
In case of the design of a special purpose language, we tailor the language based on what we need for specific application, where some of the above factors might turn obsolete.

**High Level Languages Paradigms:**

A) **Imperative**: Action oriented, the programmer dictates to the CPU how to execute the code via **sequence of commands** (instructions), where the execution control flow is govern by an instruction counter, and possibly, changing the computer state with every instruction’s execution.
Example HLLs: FORTRAN, PASCAL, BASIC, C, Ada, Modula, C++, Smalltalk,…

  **Categories:**
  i) **Block-Structured** (non-object oriented). Data are passive and abstraction is weak and artificially added.
  ii) **Object-Oriented** HLLs (OOLs): Data are active, they have behavior, acting on themselves and the other data in the system, via a message based mechanism. Abstraction is inherent in the language; every datum is an object ADT.
   For code sharing the concept of reusability is introduced via the **inheritance** mechanism.

B) **Declarative**: Higher level than the above von Neumann and OOLs. Defining more of “what” the computer is to do than “how” to do it. The problem to be solved is described via a set of function calls or rules of inference, hence there is now “how” to do it CPU, and no intermediate changing of the system memory values (i.e., state!).
Examples are:
Categories:

i) Functional:

- A program is one function composition call.
- The equivalency of programs and data (data and programs are lists).
- No intermediate memory side effect (change of system state).
-Recursion replaces iteration.

Examples: Lisp, ML, Scheme, Miranda, id, Haskell

ii) Logic: The program is a set of axioms and rules that describe the programming environment; then the system evaluate the assertion of a “goal” (theorem”).

The program output(s) are obtained as a side effect of the goal evaluation.

HLLs Translation and Software Simulation (HLL Virtual Machine Interpreters)

A) HLLs Translators:

i) Compilers:

HLL program →

Lexical Analyzer (Scanner) → Token stream →….> (Symbol
Syntactic Analysis (Parser) → Parse tree → ...............>Table, ST)
Semantic Analysis [ &Intermediate Code generation (optional)]→
Abstract syntax parse tree and intermediate form → <--| ST

Code Optimization → Modified intermediate form
Code Generation → Assembly/P-code/Machine language → <--| ST

Scanner: scans the input program statements and extracts its composing tokens.
Example: Y := X + Z;
Output tokens: Y, :=, X, +, Z, ;

Syntax Analyzer (Parser): develops an abstract syntax parse tree (AST) detecting syntax errors.

Semantic Analyzer: takes the above AST and parse it for semantic consistency (e.g., consistent use of operators and data types).
A “symbol” table will be generated to maintain info (attributes) about each identifier (name) used in the program, e.g., its type, internal structure, scope, etc. It uses the symbol table to enforce the language static semantics rules, for example:
1) A name must be declared before use.
2) Type checking
3) Formal/actual parameter matching
4) A function has to have a “return” statement
For the language dynamic semantics rules (hard!), the compiler generates code to check them at run time, and produce run-time error in case of any violation, for example:

1) variable initialization with value before use in an expression.
2) Pointers has to be pointing to a valid object before being de-referenced.
3) Array subscript values are within the declared array boundaries.
4) A function has to return a value.

An optional intermediate form of code (abit higher than assembly) might be generated.

Optimization: optimizes the intermediate code and/or the parsed tree for efficient code generation.

Code Generation: uses the resultant abstract representation for generating executable machine code.

ii) Assemblers: Assembly program → Assembler → executable machine code

iii) Preprocessors:

a) C code with conditional macros → Preprocessor → Modifies C source code
   Modifies C source code → Compiler → “Assembly language”/”Machine code"

b) extended HLL program (C++) → Preprocessor → HLL code (C)

B) Software simulators (Interpreters):

HLL program → The HLL’s Virtual Machine → program output

Examples: Java, Lisp, ML, Smalltalk, Prolog.
FORTRAN
(OLD VERSIONS) [Backus 1956, IBM]

• First widely used HLL.
• The main design goals: 1) Efficiency  2) Amenability for Numerical Engineering Applications.
• All needed storage of the program is decided at compile time, minimal invocation of the OS.
• Simple typing system, static, and few types: integer, real, complex, double, and arrays (new versions added: Boolean, character string and file types).
• No type definition or facilitating user defined data abstractions, only arrays and fixed size char string. In addition, Subroutines and functions.
• Parameter passing by “reference” or “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, in its allocated memory space.
  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, 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 (L1, L2, …, Ln), I
      If \( I = k \), jump to \( L_k \) label in the label list, \( 1 \leq k \leq n \); otherwise no jump.
  iii) Assigned:   GO TO N, (L1, L2, …, Ln)  [ the label list is not used by the compiler!!!]
      go to the address contained in N, N must be pre-assigned some label address, via the assign statement  ASSIGN  label  TO N, that places the address of label into N. It is the responsibility of the programmer to do so (leads to insecurity).
**Language’s Security Loophole:** The similarity of the computed and assigned “GO TO” above, 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, otherwise we get what seems “good” program result, which is not)

---

**Language Typing Systems**

- **What is a “Type”**? A type of a variable is the set of values that such variable can have and the set of operations that can work on such values.
- **Why do we need Types in some HLLs?** Mainly for the following reasons:
  1) Efficient allocation of memory
  2) Type checking (security)
- There are two major type classifications:
  1) **Built-in** system: basic types—integer, real, etc and structure—array, records, files
  2) **User defined**: ADTs, Classes, Modules, Packets
- Type “**Coercion**”: implicit type conversion based on the context of use.

\[
X + I \rightarrow X + \text{FLOAT}(I) \quad X = I \rightarrow X = \text{FLOAT}(I) \quad I = X \rightarrow I = \text{IFIX}(X)
\]

---

- **Explicit vs. Implicit 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 reals. Implicit declarations are power in the language, but it could easily lead to a **language’s security loophole** as we will see later, when the programmer when the compiler considers a misspelled name to be a valid name and use its incorrect value instead of the correct intended name’s value.

- **Strong (secure) vs. Weak (insecure) typing system:**

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

Not allowing any error to escape the all line of defenses (syntactic checking, type checking, run time checking) via the aid of the following: (“**defense on depth**” page 51)
i) A rich enough set of types not to overwork any of its types to represent more than one set of values, e.g., FORTRAN’s weak typing system overworked the type integer by having it also to carry labels’ addresses, and strings (Hollerith strings are considered integers)! The programmer can make a silly mistakes without the system being able to detect them; i.e., N = ISUCC (6HCARAMEL).

Ex: The rich Pascal’s 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!

- 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 syntax/semantics error can go undetected.

Such “type checking” might be of one the following types:

1) Static: at compile time for efficiency.
2) Dynamic: at run time for polymorphic power.

iii) The language must NOT ever trust the user of doing the safe/correct 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 (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 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 guaranteed that the user will make undetectable errors, neither at compile or run times.

- The following three HLL design factors are contradicting (you gain in one you lose in the others): Power, Efficiency, and Security. Discuss via examples from existing HLLs’ features. [Hint: Infinite program execution using recursion, indefinite looping]
- Is it possible that a language with a secure type checking mechanism to be “insecure”?
Parameter Passing Modes in FORTRAN

Call “by-reference” parameter passing only (until FORTRAN 77)

- If the actual parameter is an l-value, e.g. a variable, its reference is passed to the subroutine
- If the actual parameter is an 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. Such addresses would be passed to the corresponding actual!! And, guess what, if we mistakenly change any formal, it 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:

```fortran
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)
Pros: 1- Efficient when passing huge structure (arrays)
      2- the power of aliasing (if you do not care about security!!)
Cons: 1- 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 modern FORTRAN versions, “by-reference” and “global name access” could lead to aliasing two or more names on the same memory location. Example:

```fortran
SUBROUTINE ALIAS(A, B, C, Z)
    INTEGER A, B, C, Z  (remember there is an implicit visibility of S from now on)
    (Now, we also have five 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 in modern FORTRAN*)
Z = A + B + C
RETURN
END

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

INTEGER S, H
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!)

Call “by-value-result” parameter passing frequently used after FORTRAN 77

- 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.
  - 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!)

- Disadvantages:
  1- requires multiple storage for parameters and the time for copying values.
  2- can be 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.

Homework #1: Exercise 2-15 and 2-16 p. 60-61 in your text.
Due: Feb. 6th (in class)