Discussion:

Why programming languages?
What is a programming language?
How a programming language is used?

Threads that guided the development of HLL's from Early FORTRAN until now!

1) ***Data*** **manipulation and** ***security*** **of usage.**

2) ***Logic*****&*****Control*** **of the code.**

3) ***Abstraction*** **of the host platform (computer system details), i.e., height above the hardware.**

4) ***Separation*** **between** ***regular*** **users and** ***professional*** **designers of software.**

1- The position of High Level Languages(HLLs) in the Computer System:

HLLs allow us to feasibly utilize the hardware (black-box) for the implementation of the toughest algorithmic solutions of most problems that we face.

Three levels of machine programming:

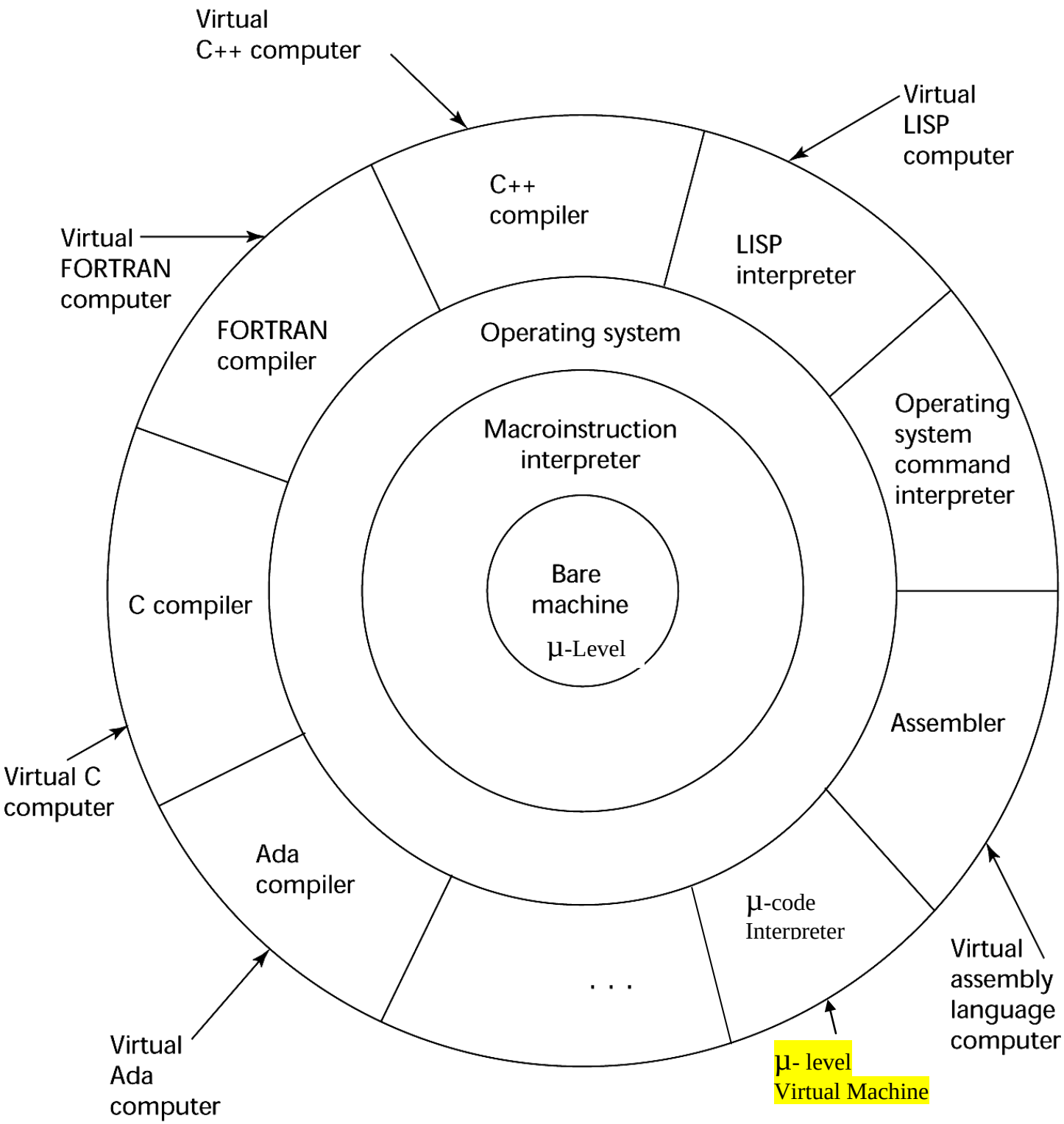
I) **Micro-level**: Micro coding at the micro architecture, in the micro "programmable" machines (where a ROM stores all micro routines that encodes the macro-level machine instruction set). Very fast execution of code, yet very tough to code, read, and maintain.

II) **Macro-level**: Macro (assembly) level architecture, easier to read the individual mnemonic assembly instruction, but tough to guess the program purpose (goal). It is a pseudo ***virtual*** machine that abstracts the micro level to the macro level programmers.

III) **HLL-level**: The third layer that works, again, as a ***virtual*** machine. It abstracts the details of the underlying micro/macro levels to the higher level programmers, presenting English like programming language (HLL) syntax. It is called HLL since it is up high above the hardware level, making the language much lesser hardware dependent.

Every mnemonic language the computer can process represents a virtual machine over the same hardware, including the mnemonic µ-code!

Remember -- As we move up from the core hardware (H/W) we get "higher" in the language *abstraction*, i.e., lesser dependency on the H/W, for better readability and ease of software (S/W) design (modularity and maintenance); yet we lose speed of code execution (why?).


Virtual
C++ computer
Virtual
LISP
computer
Virtual
FORTRAN
computer
C++
compiler
LISP
interpreter
FORTRAN
compiler
Operating system
Operating
system
command
interpreter
Macroinstruction
interpreter
Bare
machine
μ-Level
C compiler
Assembler
Virtual C
computer
Ada
compiler
μ-code
Interpreter
. . .
Virtual
assembly
language
computer
Virtual
Ada
computer
μ- level
Virtual Machine

2- Why Should we 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 (implementation of problems' solutions).

iii) **Future design of new** HLLs and special purpose languages.

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 language tools, 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) Programs portability between different platforms.

f) The cost of program:
1) development 2) translation 3) maintenance

g) Its support of useful software development environment (to the user), e.g., editors, interpreters, and graphical user interface.

4- Major Factors That Characterize a HLL

a) **Power:**

1) Short syntax that encodes powerful semantics.

2) Providing enough tools to the programmer to carryout anything he/she “dreams" to do!

3) Recursion – utilizing the system run dynamic semantics and resources to implement the problem solution.

4) Polymorphism – overloading and genericity.

b) **Modularity** and **Abstraction**:

Modularity help at all phases of S/W design, code reusability, easy maintenance, testing, separate module design.

In addition to facilitating the clean and safe separation between the user and implementer domains, **abstraction** has an important impact on a language via the amount of algorithmic solution encoding burden that will be shifted from the user domain to the system (HLL's implementer) domain.

c) **Security** -- of language (user and encoding)
-- of applications (run time exceptions).

d) Programs' **speed of execution:** programs that run faster (independent of the programmer skills).

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!

Gaining in one direction (factor) leads to losing in one or more of the other directions (*No Free Lunch!!*).

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.
Yet, in **general purpose** languages we try to combine most of the above features, especially those related to users – Security of coding, readability, and abstraction/modularity. **What we pay depends on the degree of implementing any of the desired feature(s).**

High Level Languages Paradigms

A) *Imperative*:

Action oriented, the programmer dictates to the CPU **how** to execute the code via a **sequence of commands** (instructions), where the execution control flow is govern by an instruction counter, and possibly, changing the computer state with instruction's execution.

Example HLLs: FORTRAN, PASCAL, BASIC, C, Ada, Modula, C++, Java, Smalltalk,…

i) ***Block-Structured*** (non-object_oriented). Data are passive and abstraction is weak and is an artificially added (ad-hoc) feature.

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*** (true ADT).

For ***abstract*** code sharing the concept of reusability is introduced via the ***inheritance*** mechanism. Inheritance is defined via a **static**

sub/class hierarchy, where a subclass can inherit **all** components of its parent class (**state** and **behavior**) **violating** the ***security*** (by the introduction of fare reaching changes to its state variables) and ***abstraction*** (hiding the details of the parent from the outside world) of the parent object (e.g., *class* in smalltalk)!

B) ***Declarative:***

Languages of such category have higher level than the above HLLs of von Neumann and OOLs; defining more of "**what?**" is problem that the computer is to solve more than "**how?**" to solve it. The problem to be solved is described via a set of function calls or rules of inference, hence there is no "how" to do it via the CPU, and no higher-level notion of intermediate changing to the system state i.e., side effects on system memory.

Examples of Declarative HLLs are:

i) ***Functional***:

The following is true in ***pure*** Functional definition:

- 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.

None pure Function HLLS examples: Lisp, ML, Scheme, Miranda, id, Haskell

ii) Logic:

The program is a set of axioms and rules that describes 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.

It is the most abstract domain, replacing “***how** to solve the problem*” with “***what** is the assertion of the problem to be solved*”.

In the pure domain, there is a total separation between the "***How***" (control) and the "***What***" (logic).