# 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 an artificially added (*ad-hoc*) feature. : Ada, Pascal,Modula-2,....

***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 *abstract* code sharing the concept of reusability is introduced via the ***inheritance*** mechanism. C++, Java, Smalltalk.

## B) ***Declarative:***

Languages of such category have Higher level abstraction than the above the von Neumann block-structured and OOL languages; defining more of "**what?**" is problem that the computer is suppose 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***:

- A program is just a **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.
- "**atoms**" with **powerful** ***property*** structure.

**Examples:** Lisp, ML, Scheme, Miranda, id, val, 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).

Producing an output, as in the imperative domain, is not explicitly coded when we write logic programs (e.g., write, print, …); instead 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*".

# HLLs Translation and Software Simulation (HLL Virtual Machine Interpreters)

## A) HLLs Translators:

### i) *Compilers*:

**HLL program** → **Scanner (Lexical Analyzer): (Token Stream)** →

**Syntactic Analysis:** (Abstract Syntax Parse Tree) →

**Semantic Analysis & intermediate Code generation:** (Intermediate code) →

**Optimization** → **Code Generation** → Machine Code

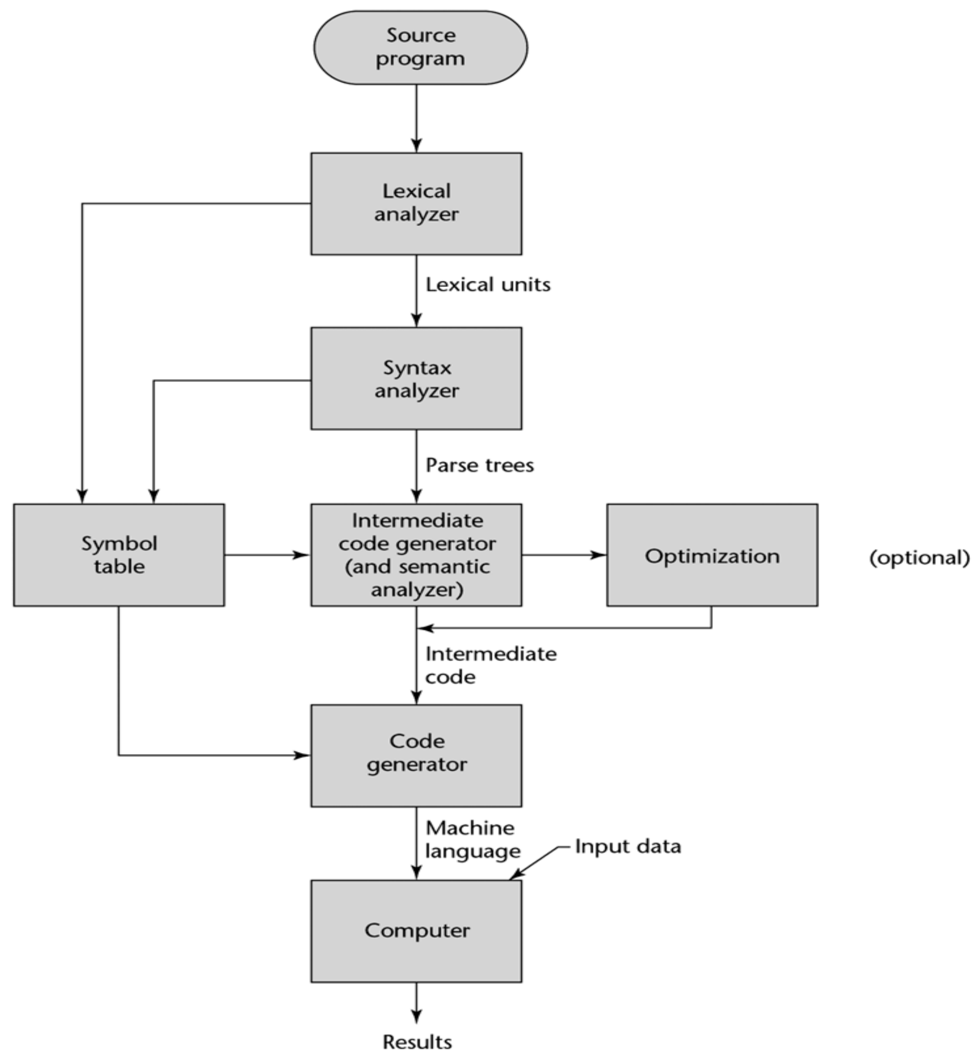



**Figure 1.3**
The compilation process

**Concepts of Programming Languages**, Robert W. Sebesta, 8th Edition

**Scanner (lexical-analyzer)**: scans the input program statements and extracts its composing tokens, e.g., keywords, ids, … .

Example: Y := X + Z * 155;
Output tokens: Y, := , X , + , Z , 155 ;

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

**Semantic Analyzer:** takes the above AST augments it with semantics info and build the symbol table; then checks for semantic consistency (e.g., consistent use of operators and data types).

**Optimization (*optional*):** optimizes the parsed tree for efficient intermediate code generation.

**Code Generation:** uses the resultant augmented AST with the intermediate code to generate the target machine code which is not directly executable, it still needs to go through the operating system link/load phases.

ii) ***Assemblers*:** Assembly program: mnemonic code→
Assembler: target machine code→
Linker/Loaders: executable machine code

The machine code produced by the compiler (or assembler) is not directly executable, all unresolved addresses must be massaged first and all needed external units are to be linked; then the linked code is to be loaded in the memory before execution begins.

## The Machine instruction cycle (von Neumann)

**Initialize the program counter**

**Repeat forever**

  **Fetch** the instruction (^ PC)

  **Increment PC++**

  **Decode** the instruction

    (get all operands, if any)

  **Execute** the instruction

    (store results, if any)

**End repeat**

ii) ***Preprocessors*:** C++ → Preprocessor**:** C code

C → Preprocessing to handle directives ("include"), macros ("#define"), and conditional macros ("#if")

## B) Software Simulators (Interpreters):

HLL program →intermediate (or compiled) code→ program output

Examples: Java (precompiled code), Lisp, ML, Smalltalk, Prolog.

I) **Pure:** (APL, SNOBOL, LISP, PHP, JavaScript):

**What?** The HLL instructions are treated as if low level machine instructions: fetch/decode/execute.

**Why?** **Powerful and useful in debugging the program; direct pointers to erroneous HLL statements!**

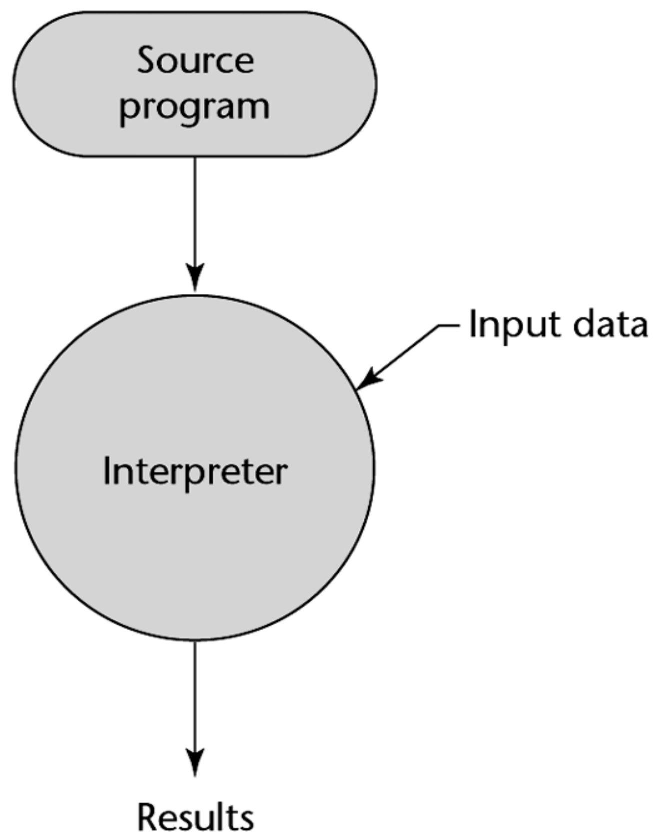
**Why Not?** **1) very Slow!**

**2) needs large run-time space to store tables and AST**

Yet, they are back to use as Web Scripting languages!

**Figure 1.4**

Pure interpretation




**Concepts of Programming Languages**, Robert W. Sebesta, 8th Edition

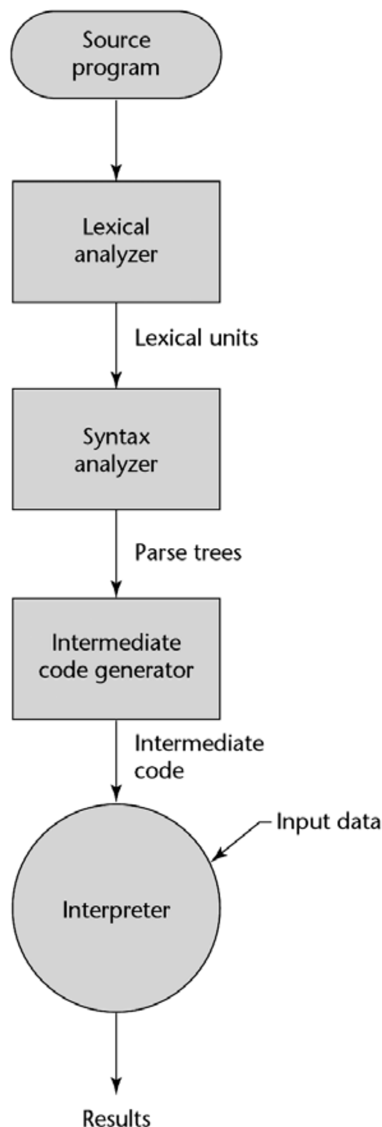
II) **Hybrid:** (Perl, and early Java VM)

**What?** Go with the compilation until the "intermediate" code is generated, then feed it to ***byte-code*** interpreter. (compilation of intermediate code into the native machine language, for plate form independency.)

**Why?** To **speed up the execution** of the program, it is much easier and faster to deal (fetch/decode/execute, much smaller tables) with well designed "intermediate code" than the original HLL, also platform independency.

**Figure 1.5**

Hybrid implementation system




**Concepts of Programming Languages**, Robert W. Sebesta, 8th Edition

- **Remember, each HLL represents a Virtual Machine (VM) that encapsulates the "real" hardware in-use and the corresponding language translator.**

- **The programmer thinks that the hardware understands C++, C, Pascal, Ada, Java, etc, which is not true (why?).**

- **Not all paradigms members' HLLs are "pure", i.e., *rigidly* following the paradigms' main philosophy. In the literature, here are some examples of pure languages:**

  **Object Oriented Paradigm:**
  **Eiffel, Smalltalk, Suneido, and Ruby.** (**Java** is almost there!)

  **Functional paradigm:**
  **Concurrent/Clean, Curry, FP, Haskell, Hope, Miranda, Charity, and ML.**

  **Logic paradigm:**
  **Mercury (a pure subset of Prolog!), Starlog (bottom-up evaluation).**