Concept of Programming Languages

Preliminaries
- Why Study PL ?
- Evaluation Criteria
- Design Trade-offs

Basic Features (Variable, Expression, and Control)

Names, Binding, Type Checking, Scope
- Variables, Binding
- Type Checking
- Scope, Lifetime

Data Types
- Primitive Types
- Ordinal Types
- Array, Record
- Union, Set, Pointer

Expression & Assignment
- Arithmetic Exp.
- Overloaded Op.
- Type Conversion
- Assignment Stmt

Statement-Level Control Structures
- Compound Stmt, Selection Stmt
- Iterative Stmt, uncond. Branching
- Guarded Commands

Subprograms

Subprograms
- Parameter Passing
- Overloaded Subpgm, Generic Subprg

Implementing Subprogram
- FORTRAN 77
- Algol-like Lang.

Advanced Features

Abstract Data Type
- Abstraction
- Encapsulation, Abstract Data Type

Concurrency
- Coroutine
- Semaphore, Monitor
  Mesage Passing

Exception Handling
- Concepts
- Pl/I, Ada, C++

Non-Imperative Languages

- Functional Programming Languages
- Logic Programming Languages
- Object-Oriented Programming Languages

Chapter 1

Preliminaries

1.1 Reasons for Studying Concepts of Programming Languages
1.2 Program Domain
1.3 Language Evaluation Criteria
1.4 Influences on Language Design
1.5 Implementation Methods
1.6 Programming Environments

“The principal goal of this course is to provide the students with the necessary tools for the critical evaluation of existing and future programming languages and programming language constructs”

1.1 Reasons for Studying Concepts of PLs

- ***Increased capacity to express ideas***
  - It could be argued that learning the capabilities of other languages does not help a programmer who is forced to use a language that lacks those capabilities. That argument does not hold up, however, because there are often ways in which language facilities can be simulated in other languages that do not support those features (in the form of subprograms)
- ***Improved background for choosing appropriate languages***
  - If the programmers were familiar with several languages available and particular features of those languages, they would be in a better position to make informed language choices
- ***Increased ability to learn new languages***
  - Once a thorough understanding of the fundamental concepts of language is acquired, it becomes far easier to see how these concepts are incorporated into the design of the language being learned
- ***Better understanding of the significance of implementation***
  - An understanding of implementation issues leads to the ability to use a language more efficiently (*e.g,* do not use small subprogram)
- ***Better use of language that are already known***
- ***Overall advancement of computing***

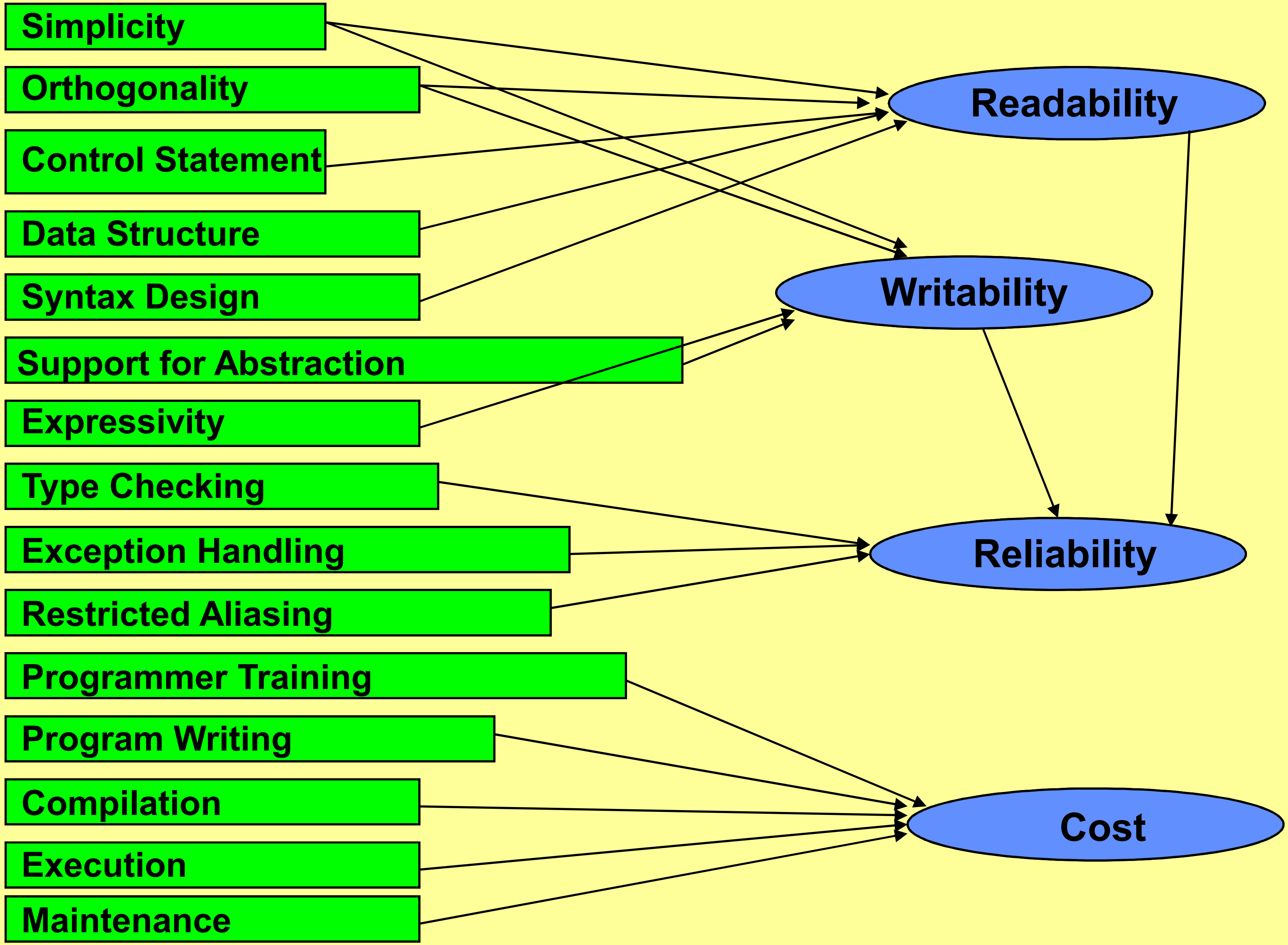
1.2 Programming Domains

– Discussing a few areas of computer application and their associated languages

- ***Scientific Applications***
  - Scientific applications typically have simple data structures but require large amounts of floating-point arithmetic computations
  - For scientific applications where efficiency is the primary concern, like those that are common in the 1950s and 1960s, no subsequent language is better than Fortran (FORTRAN90)
- ***Business Applications***
  - Business languages are characterized, according to the needs of the application, by elaborate input and output facilities and decimal data types
  - There have been only limited developments in business application language other than COBOL
- ***Artificial Intelligence***
  - AI application languages are characterized by symbolic processing and the use of lists as the primary data structure (LISP, PROLOG)
- ***System Programming***
  - System software is used almost continuously and therefore must have execution efficiency. A language for this domain must have low-level features that allow peripheral device driver software to be written
  - UNIX is written almost entirely in C, which has made it relatively easy to port, or move, to different machines
- ***Web Software***
  - XHTML + Scripting Language : (shell, ask, tcl/tk, perl, JavaScript) → dynamic contents
  - Java,…

1.3 Language Evaluation Criteria

- *Programming language evaluation criteria*



(1) Readability

- After software life cycle concept was introduced, readability became an important measure of the quality of programs and programming languages, because ease of maintenance is determined in large part by the readability of programs

• *Overall Simplicity*

- A language that has a large number of elementary components is usually more difficult to learn than one with a small number of elementary components
- Feature multipliciy (having more than one way to accomplish a particular operation) is another problem (예 : *count++, ++count, count +=1,....)*
- Operator overloading, in which a single operator symbol has more than one meaning
- Language statements can also be simplified too much (예 : assembly language)

• *Orthogonality*

- Orthogonality in a programming language means that there is a relatively small set of primitive constructs that can be combined in a relatively small number of ways to build the control and data structures of the language
- The more orthogonal the design of a language, the fewer exceptions the language rules require

  ⇔ 예) `return()` in C

• *Control Statements*

- Programs that can be read from top to bottom are much easier to understand than programs that require the reader to visually jump from one statement to some nonadjacent statement in order to follow the execution order

  ⇔ *GOTO*–less

• *Data Types and Structures*

- The presence of adequate facilities for defining data types and data structures in a language is another significant aid to read (ex: *sum_is_too_big* = *true*)

  ⇔ 예 : `record, boolean type, enumeration type`

- *Syntax Considerations*
  - Identifier forms (ex: SUM_OF_SQUARE, ...)
  - Special words (ex: end_if, end_loop, ...)
  - Form and meaning
    - ⇔ GO TO (10, 20, 30), I (I가 양수, 제로, 음수에 따라..)
    - ⇔ GO TO I, (10, 20, 30) (I는 label 변수)

(2) Writability

- Writability is a measure of how easily a language can be used to create programs for a chosen problem domain

- *Simplicity and Orthogonality*
  - a small number of primitive constructs and a consistent rules for combining them (that is, orthogonality) is much better than simply having a large number of primitives
- *Support for Abstraction*
  - Abstraction means the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored
    - ⇔ Process abstraction : using subprogram
    - ⇔ Data abstraction : using Record type
- *Expressivity*
  - a language has relatively convenient, rather than cumbersome, ways of specifying computations
    - ⇔ *count*++ is more convenient and shorter than *count* = *count* + *1*

(3) Reliability

*a program is said to be **reliable** if it performs its specifications under all conditions*

- ***Type Checking***
  - It is *testing for type compatibility* between two variables or a variable and a constant that are somehow related with one another
  - Because run-time type checking is expensive, compile-time type checking is more desirable
    - ⇔in C : no type checking for parameter passing
    - ⇔in Pascal: *the subscript range checking* is part of type checking, although it must done at run time
- ***Exception Handling***
  - The ability of a program *to intercept run-time errors*, as well as other unusual conditions, to take corrective measures, and to continue is a great aid to reliability
- ***Aliasing***
  - It is, loosely, having two distinct referencing methods, or names, for the same memory cell
  - It is now widely accepted that aliasing is a dangerous feature in programming language
    - ⇔Equivalenced variable in Fortran
    - ⇔the pointers in Pascal
- ***Readability and Writability***
  - Unnatural methods are less likely to be correct for all possible solutions

(4) Cost

- The cost of training programmers to use the language
- The cost of writing programs in the language
- The cost of compiling programs in the language
- The cost of executing programs written in a language
- The cost of maintaining programs

(5) Portability : 얼마나 많은 platform에 그 언어의 컴파일러가 있느냐 ?

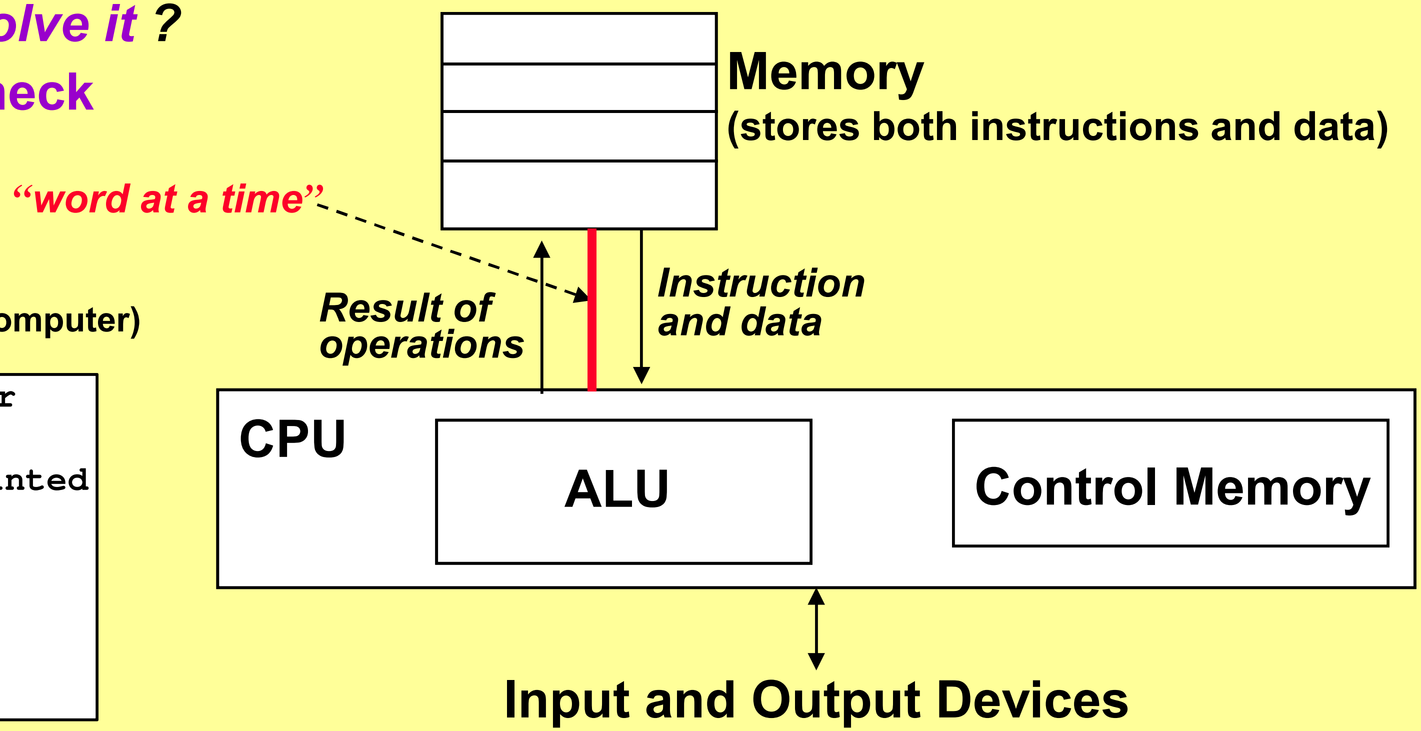
(6) Generality : the applicability to wide range of applications

(7) well-definedness : 언어에 대한 문서화가 얼마나 잘되어 있느냐 ?

1.4 Influences on Language Design

(1) Computer Architecture

- **Imperative (von-Neumann Computer) languages**
  - all language constructs are **high-level abstraction of hardware structure and operations**
    - ⇔ variable (memory cell)
    - ⇔ assignment (fetching and storing)
    - ⇔ expression (arithmetic)
    - ⇔ loop (conditional jump)
  - Focused on ***how to solve it ?***
  - **von-Neumann Bottleneck**



<참고> Fetch-execute-cycle
(on a von Neumann architecture computer)

```
initialize the program counter
repeat forever
  ① fetch the instruction pointed
       by the counter
  ② increment the counter
  ③ decode the instruction
  ④ execute the instruction
end repeat
```

(2) Programming Methodologies (S/W cost가 높아짐에 따라..)

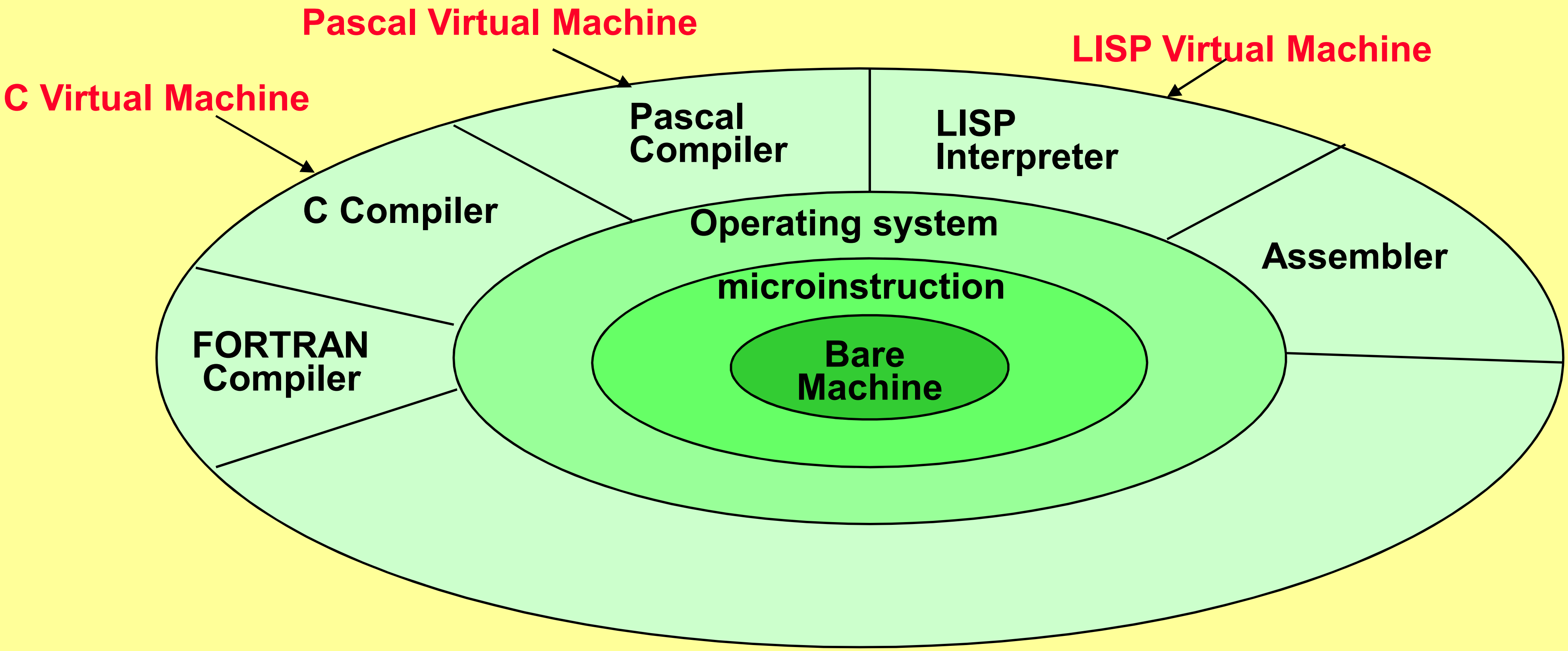
- *Structured programming (goto-less) : 70년대*
  - top-down design, stepwise refinement
  - Procedure-oriented programming
  - Pascal, Algol, ...
- *Data-Oriented programming: 70년대 말*
  - For data abstraction to be used effectively in software system design, it should be supported by the languages used to implement the system
  - Simula67, CLU, Modula-2, C++, Ada
- *Object-Oriented programming : 80년대 초*
  - data abstraction + encapsulation + dynamic type binding
  - Reusing existing software
  - Smalltalk, CLOS, C++
- *Process-Oriented programming*
  - The needs for language facilities for creating and controlling concurrent program units
  - Concurrent Pascal, Ada, ...

1.5 Language Design Trade-Offs

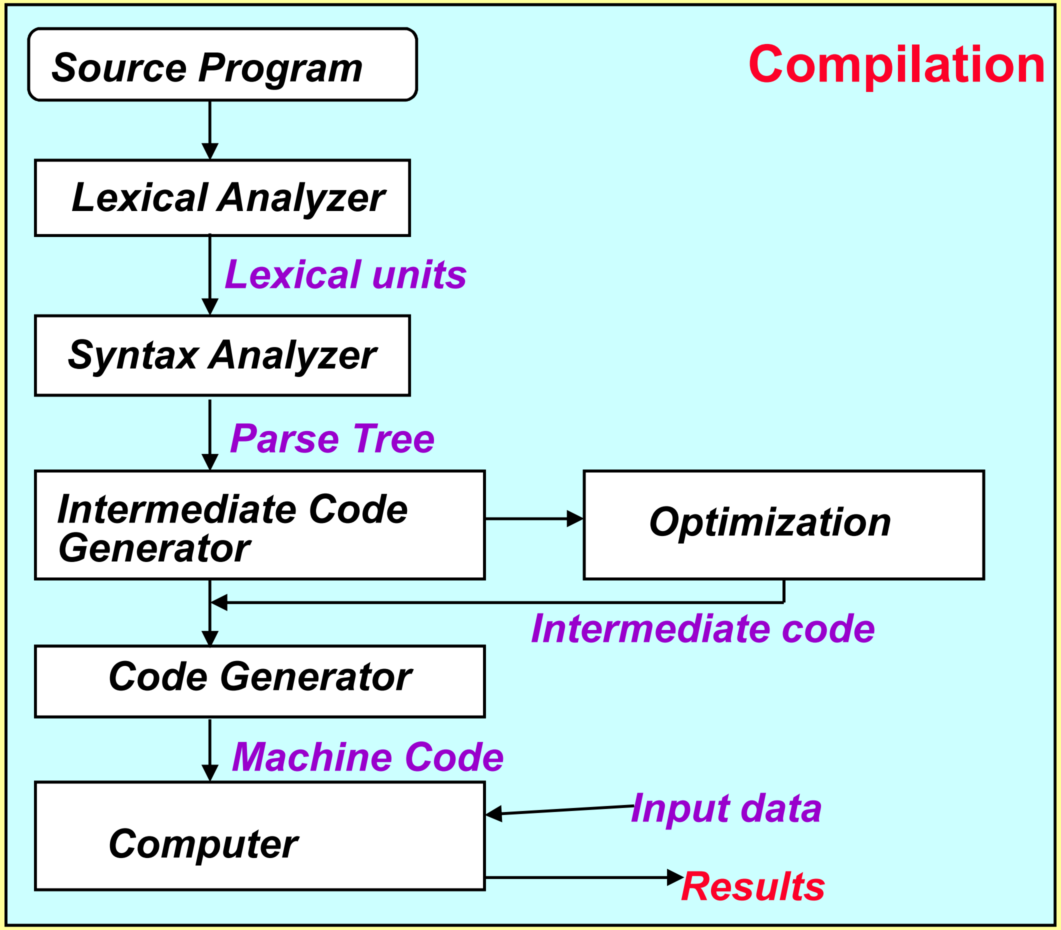
- Conflict between *reliability* and *cost of execution* *(예 :?)*
- The compact and concise expressions have a certain mathematical beauty but are difficult to understand
- Conflict between *flexibility* and *safety* *(예 :?)*
- Conflict between *flexibility* and *efficiency* *(예 :?)*

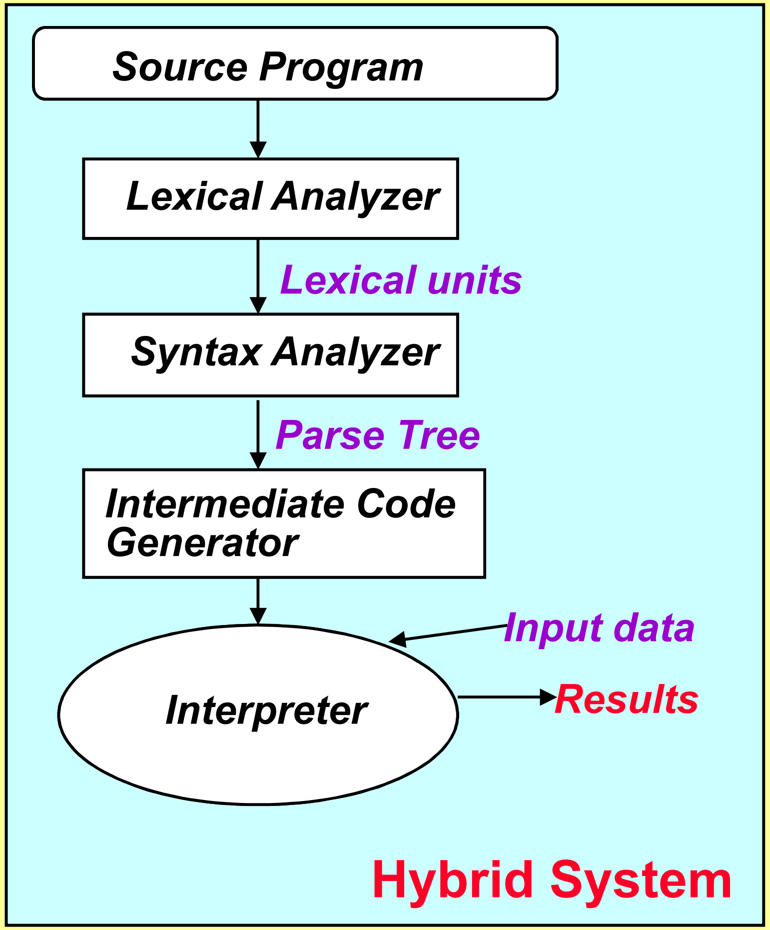
1.5 Implementation Methods

- In the absence of other supporting software, its own machine language is the only language that most hardware computers "*understand*"
- The software that provides the high-level language interface to a computer can take several different forms; compiler, pure interpreter, hybrid systems
- Because high-level language implementation need many of the operating system facilities, they interface to the operating system rather than directly to the processor
- Layered interface, on *virtual computers*



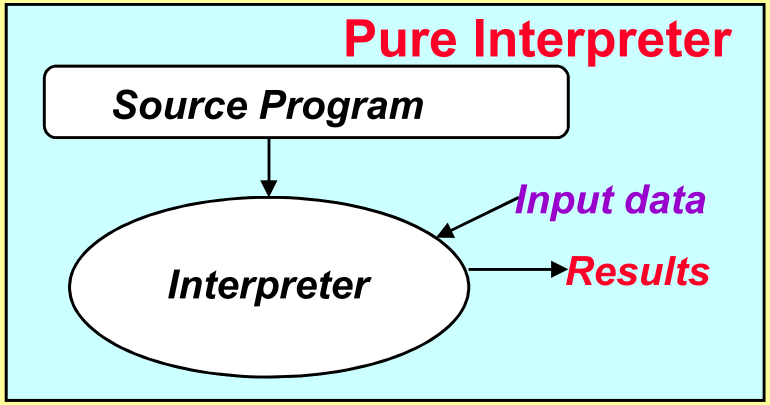
- *Compilation*
  - Programs are translated to machine code, which then can be executed directly on the computer
  - Very fast program execution, once the translation process is complete
  - C, COBOL, Ada





- *Hybrid Implementation System* *(vs. Just-in-Time (JIT) Compilation ➔ widely used in Java)*
  - Compromise between compilers and pure interpreter
  - They translate high-level language programs to an intermediate language designed to allow easy interpretation

- *Pure Interpretation*
  - *Programs are interpreted by another program*, called *interpreter*, with no translation whatever
  - The interpreter program acts as a software simulation of a machine whose fetch-execute cycle deals with high-level language program statements rather than machine instructions
  - Easy implementation of many source-level debugging operations, because all run-time error messages can refer to source-level units
  - The execution is many times slower than in complied systems
  - APL, LISP, Prolog



- Preprocessor
  - a macro expander ➔ `#include, #define by C preprocessor`
  - `#define SUM(A,B) A+B ,      SUM(0.5, 0.7)` ➔ `0.5+0.7`

1.6 Program Environments (IDE : Integrated Development Environment)

- A *program environment* is the collection of tools used in the production of software
  - A file system, editor, a linker, a compiler, debugger, user-interface
  - Example : Microsoft Visual Studio.NET, NetBeans (https://netbeans.org/),
    Eclips (https://eclipse.org/),
    Android Studio (http://developer.android.com/sdk/index.html)