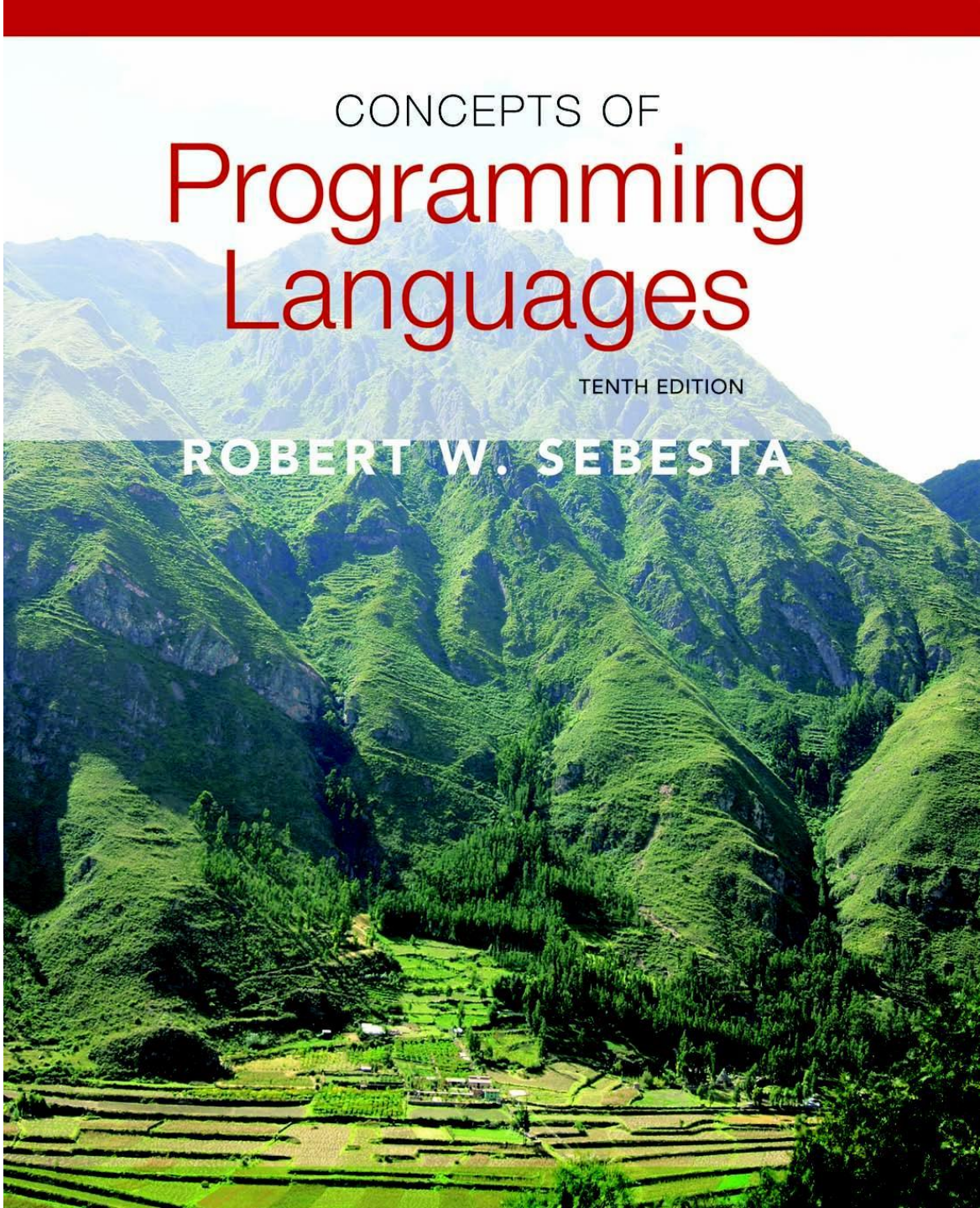
Chapter 1

Preliminaries



Chapter 1 Topics

- Reasons for Studying Concepts of Programming Languages
- Programming Domains
- Language Evaluation Criteria (계산)
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
  - the most common general approaches to implementation
  - compilation, interpretation, hybrid implementation systems, preprocessors
- Programming Environments

Compile vs Interpret
〉
〉
혹은 이를 혼합한 경우
〉

Reasons for Studying Concepts of Programming Languages

=〉언어를 선택하는 정확한 안목을 기르기 위해서

- Increased ability to express ideas
  (associative arrays in Perl, simulate them in c)
  ㄴ해쉬테이블의 형태
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
  (Object Orient Programming concept -> Java)
- Better understanding of significance of implementation (program bugs can be fixed)
- Better use of languages that are already known
- Overall advancement of computing
  (ALGOL 60 vs Fortran)

Programming Domains
(Computer applications & their associated languages)

- **Scientific applications**
  - Large numbers of floating point computations; use of arrays
  - Fortran
- **Business applications**
  - Produce reports, use decimal numbers and characters
  - COBOL (a management system of bank)
- **Artificial intelligence**
  - Symbols rather than numbers manipulated; use of linked lists
  - LISP =〉Machine learning등에 사용 function language임
- **Systems programming**
  - Need efficiency because of continuous use (OS & Kernel)
  - C (UNIX is OS implemented by C)
- **Web Software**
  - Eclectic collection of languages: markup (e.g., HTML), scripting (e.g., PHP), general-purpose (e.g., Java)

Language Evaluation Criteria

- **Readability**: the ease with which programs can be read and understood **(ease of maintenance)**
- **Writability**: the ease with which a language can be used to create programs

PL에서 orthogonality
〉중복의 의미를 많이 가지고 있으면 안된다.
〉〉즉, 너무 중복되게 사용되지 않도록 만드는 것.

- **Reliability**: conformance to specifications (i.e., performs to its specifications)
- **Cost**: the ultimate total cost

Evaluation Criteria: Readability

- **Overall simplicity**
  - A manageable set of features and constructs
  - Minimal feature multiplicity **(p.29, example)**
  - Minimal operator overloading (적은 연산자)

- **Orthogonality**
  - A relatively **small set of primitive constructs** can be combined in a relatively **small number of ways** to build the control and data structures of the language
  - 예제)저급언어(IBM, VAX machine – p.30), 고급언어(C – p.31)
  - Good combination of simplicity and orthogonality (LISP, 함수형 언어)

- **Data types:** Adequate predefined data types (P.32, timeOut = true)

- **Syntax considerations**
  - Identifier forms: flexible composition
  - Special words and methods of forming compound statements
  - Form and meaning: self–descriptive constructs, meaningful keywords

Evaluation Criteria: Writability

- **Simplicity and orthogonality**
  - **Few** constructs, a **small** number of primitives, a **small set** of rules for combining them

- **Support for abstraction**
  - The ability to define and use complex structures or operations in ways that allow details to be ignored **(process & data)**

Abstraction of Process – Subprogram

Abstraction of Data – Binary Tree

- **Expressivity**
  - A set of relatively convenient ways of specifying operations
  - Strength and number of operators and predefined functions

(ex) count = count+1 → count++

while → for (for counting loops)

Evaluation Criteria: Reliability

- **Type checking (ch6)**
  - Testing for type errors (by compiler or during program execution)
  - Run-time type checking is expensive
  - **Compile-type checking** more desirable **(Java)**
- **Exception handling (ch. 14)**
  - Intercept run-time errors & take corrective measures **(C++, Java, C#)**
- **Aliasing (ch.5 & 9)**
  - Presence of **two or more distinct referencing methods** for the **same memory location** (dangerous feature) =〉여러 함수들, 한 메모리 위치공유
  - Other languages **restricts aliasing** to increase their reliability
- **Readability and writability**
  - A language that does not support **"natural" ways of expressing an algorithm** will require the use of "unnatural" approaches, and hence **reduced reliability**

Evaluation Criteria: Cost

- **Training programmers to use the language**
- **Writing programs** (closeness to particular applications)
- **Compiling programs**
- **Executing programs** (**Optimization**)
  - **Reduction** of the code size
  - **Increase** of execution speed of the code that compilers produce
- **Language implementation system**
  - Availability of free compilers
  - Free compiler/interpreter systems of Java became available
- **Reliability**: poor reliability leads to high costs
- **Maintaining programs**
  - poor readability can make the task extremely challenging

Evaluation Criteria: Others

- **Portability**
  - The ease with which programs can be moved from one implementation to another
- **Generality**
  - The applicability to a wide range of applications
- **Well-definedness**
  - The completeness and precision of the language's official definition

Influences on Language Design

- **Computer Architecture**
  - Most of the popular languages of the past 50 years are developed around the prevalent computer architecture, known as the *von Neumann* architecture

여기까지 함 (3/7)

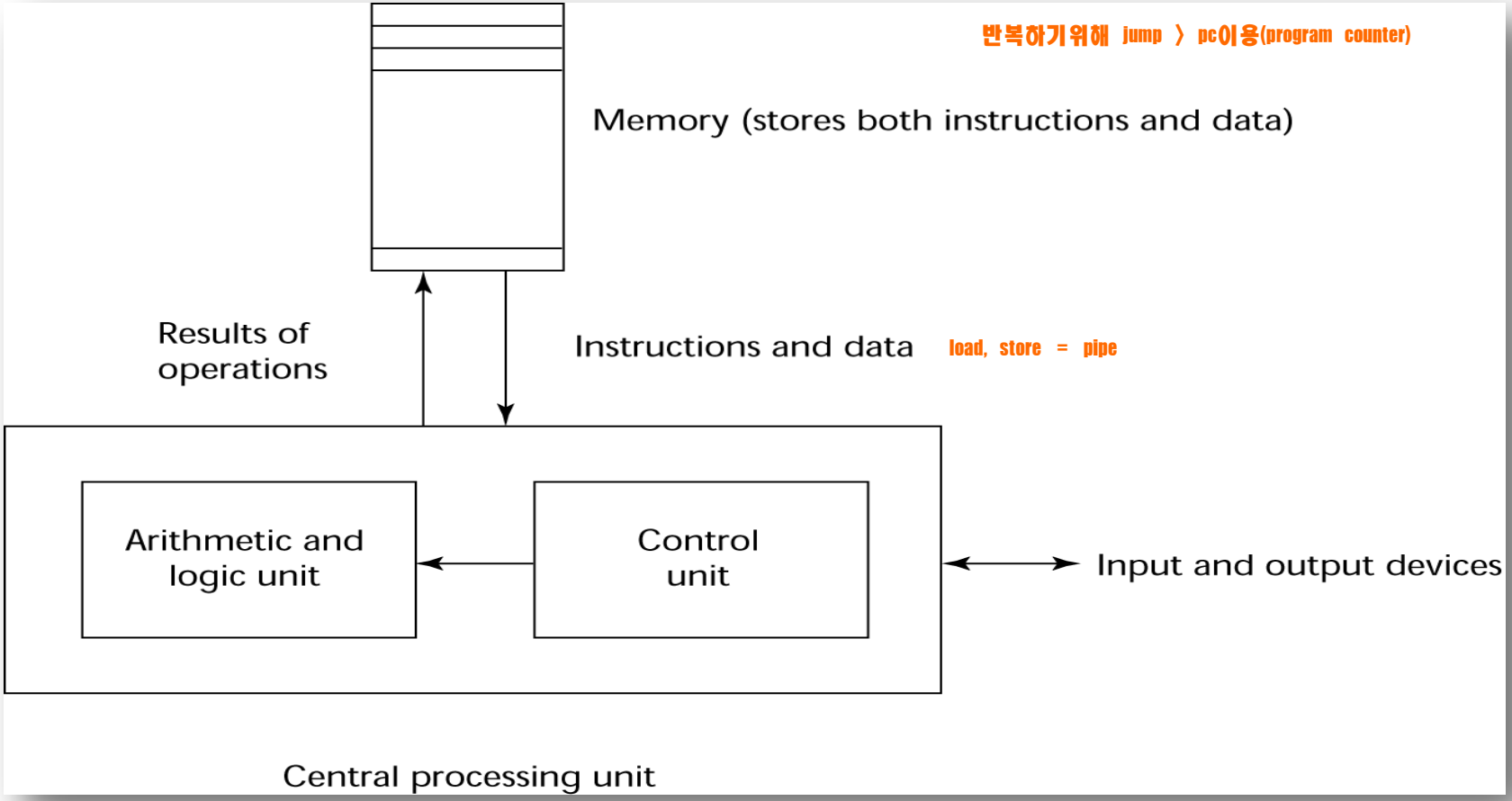
- **Program Design Methodologies**
  - New software development methodologies (e.g., object–oriented software development) led to new programming paradigms and by extension, new programming languages

Computer Architecture Influence

- Well-known computer architecture: **Von Neumann**
- Nearly all digital computers built since 1940s have been based on the von Neumann architecture
- **Imperative**(명령형 언어)**(or procedural) languages**, most dominant, because of von Neumann computers
  - Data and programs stored in the same memory
  - CPU, which executes instructions, is separate from memory
  - Instructions and data must be transmitted, or piped, from memory to CPU

Computer Architecture Influence

- Well-known computer architecture: **Von Neumann**



Computer Architecture Influence

- **The central features of imperative languages**
  - **Variables** model memory cells
  - **Assignment statements** model piping
    - Based on piping operation between CPU and memory cells
  - **Iteration :** the most efficient way to implement repetition on von Neumann architecture
    - Instructions stored in a adjacent cells of memory
    - Repeating the execution of a section of code requires only a branch instruction, which is used to implement IF statements and loops in assembly language

로컬리티사용

Computer Architecture Influence

- **The central features of imperative languages**

❖ **명령형 언어(imperative language) 또는 절차 언어(procedural language)**

- ✓ 프로그래밍 언어는 컴퓨터의 연산을 모방하고 추상화하는 데서 비롯됨
- ✓ 따라서 컴퓨터의 구조가 언어 설계에 영향을 미친 것은 당연함
- ✓ 프로그래밍 언어의 특징
  - **명령의 순차적 실행**
  - **기억 장소 위치를 표현하는 변수의 사용**
  - **변수의 값을 변경하기 위한 배정문(assignment statement)의 사용**

The von Neumann Architecture

- The execution of a machine code program on a von Neumann architecture computer occurs in a process called the **fetch–execute cycle**

Algorithm

```
initialize the program counter
repeat forever
    fetch the instruction pointed by the counter
    increment the counter
    decode the instruction
    execute the instruction
end repeat
```

※ The address of the next instruction to be executed is maintained in a register called **the program counter**

Programming Methodologies Influences

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures
  (caused by computing costs: HW → SW)
  - structured programming for large and complex tasks
  - top-down design and step-wise refinement (단계적 세분화)
    - Requiring the extensive use of **goto**s
- Late 1970s: Process-oriented to data-oriented
  - **data abstraction**
- Middle 1980s: Object-oriented programming
  - Data abstraction + inheritance + polymorphism

Language Categories (more details in chapter 2)

- **Imperative**
  - Central features are variables, assignment statements, and iteration (ex. these features, both in C and Java, are used in the almost same way)
  - Include languages that support object-oriented programming
  - Include scripting languages
  - Include the visual languages
  - Examples: C, Java, Perl, JavaScript, Visual BASIC .NET, C++
- **Functional**
  - Main means of making computations is by applying functions to given parameters
  - Examples: LISP, Scheme, ML, F#
- **Logic**
  - Rule-based (rules are specified in no particular order)
  - Example: Prolog (chapter 16) prolog 〉 발표할 만한 내용인지 찾아 볼 것
- **Markup/programming hybrid**
  - Markup languages extended to support some programming
  - Examples: JSTL, XSLT (extension version of HTML and XML)

Language Design Trade-Offs

- Language evaluation criteria provides a framework for language design
  - **Reliability** vs. **cost of execution**

c와 java를 비교한다면 java의 reliability가 더 좋음
〉〉사용하는 도중 문제가 발생할 소지가 있는 부분을 막아두니까
(ex)컴파일은 됐는데 메모리 할당잘못해서 seg뜰때

    - **Example**: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs
      (**C language : no index range checking, more faster execution)**
  - **Readability** vs. **writability**
    - **Example**: APL(A Programming Language) provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at **the cost of poor readability**
  - **Writability** (flexibility) vs. **reliability**
    - **Example**: C++ **pointers** are powerful and very flexible but are unreliable (**Not included in Java**)

Language Design Trade-Offs

- No range index checking in C language

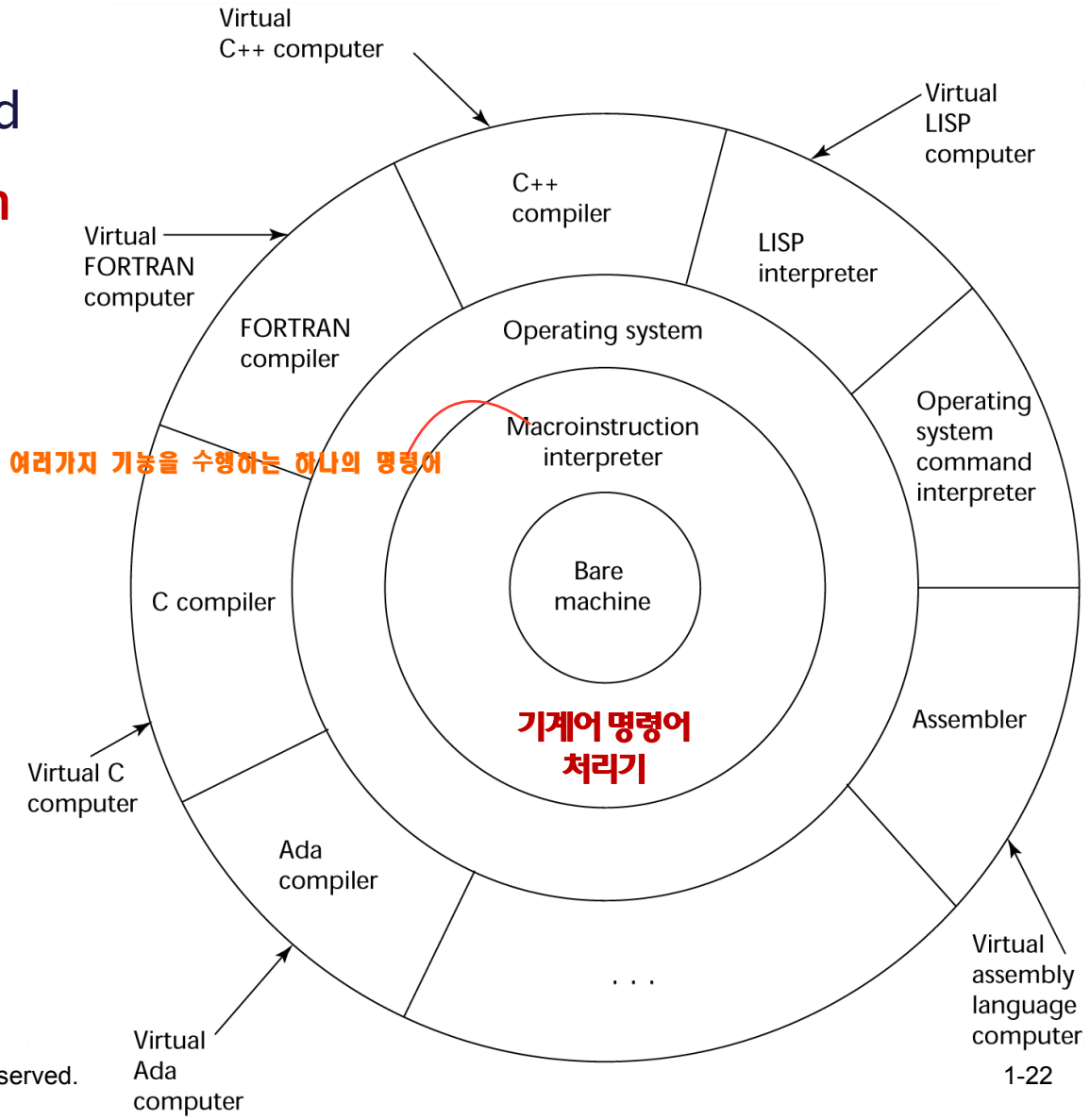
```
#include <stdio.h>
int main()
{
   char hello[12] = "No index range checking! Segmentation fault??";
   int i = 0;
   printf("%s", hello[i]);
   printf("bye bye\n");
}
```

Implementation Methods

- **The primary components of a computer**
  - **Internal memory** : storing programs and data
  - **Processor** : collection of circuits that provides a realization of machine instructions
- **Operating System**
  - Supplies **higher–level primitives** than those of the machine language
    - System resource management, I/O operations
    - File management system, Text / program editors
    - A variety of other commonly needed functions
- **Language implementation systems**
  - Need many of the OS facilities (Managing Memory, controlling I/O, handling Interruptions, securing Data, Processing, such as copy and paste etc.)
  - Interface with the OS rather than directly with the processor (in machine language)

Layered View of a Computer

The **operating system** and **language implementation** are layered over **machine interface** of a computer



Implementation Methods

- **Compilation (compiler implementation)** 코드 재사용성 up
  - Programs are translated into machine language; includes JIT systems
  - Use: Large commercial applications (C, C++, COBOL, Ada)

- **Pure Interpretation** 한줄 한줄 수행할때마다 해석
  - Programs are interpreted by another program known as an interpreter
  - Use: Small programs or when efficiency is not an issue

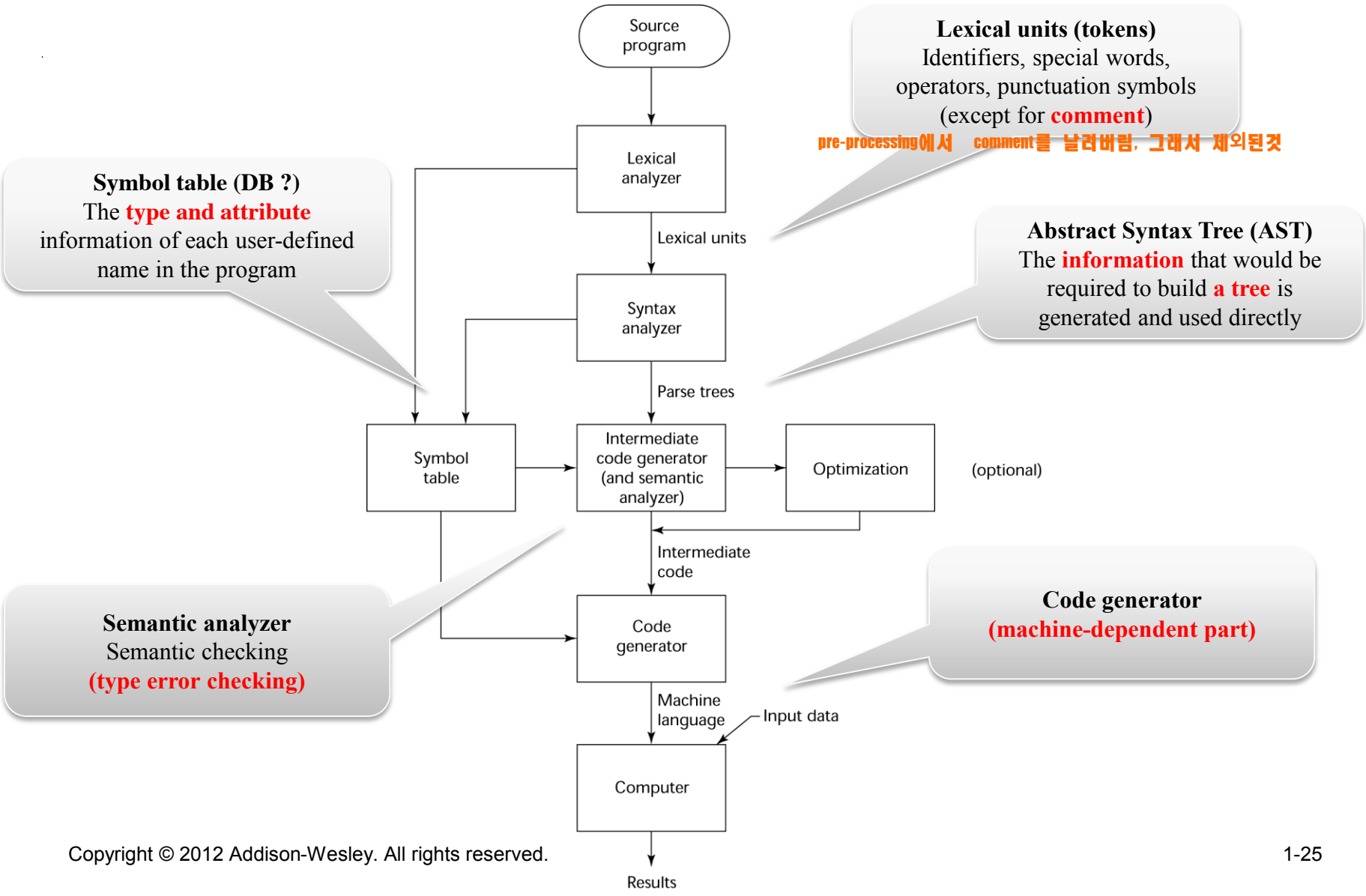
- **Hybrid Implementation Systems** 재사용성을 높이고 기계의존성을 낮춤 〉byte code로 바꿈
  - A compromise between compilers and pure interpreters
  - Use: Small and medium systems when efficiency is not the first concern

Compilation

한번기계어로 변환되면 메모리에 올려놓고 재사용을 할 수 있음,
but 해석언어의경우 그때마다 다시 읽어서 해석해야하기 때문에
일회성의 목적을 가짐

- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases:
  - lexical analysis: converts characters in the source program into lexical units (**tokens**)
  - syntax analysis: transforms lexical units into ***parse trees*** which represent the syntactic structure of program
  - Semantics analysis: generate intermediate code (**type checking**)
  - code generation: machine code is generated (**target machine dependent**)

The Compilation Process



Additional Compilation Terminologies

- **Load module** (executable image): the user and system code together
- **Linking and loading**: the process of collecting system program units and linking them to a user program

❖ 링킹(linking), 링커(linker)
  ▪ 여러 개의 목적 프로그램을 연결하여 하나의 실행 가능한 프로그램을 만드는 과정

❖ 로더(loader)
  ▪ 프로그램의 실행을 위하여 메모리에 적재

Von Neumann Bottleneck

cpu와 memory사이에서 속도차 때문에 느려지는것을 의미

- Connection speed between a computer's memory and its processor determines the speed of a computer
- Program instructions often can be executed much faster than the speed of the connection; the connection speed thus results in a *bottleneck*
- Known as the *von Neumann bottleneck*; it is the primary limiting factor in the speed of computers
  - **One of the primary motivations** for the research and development of **parallel computers**

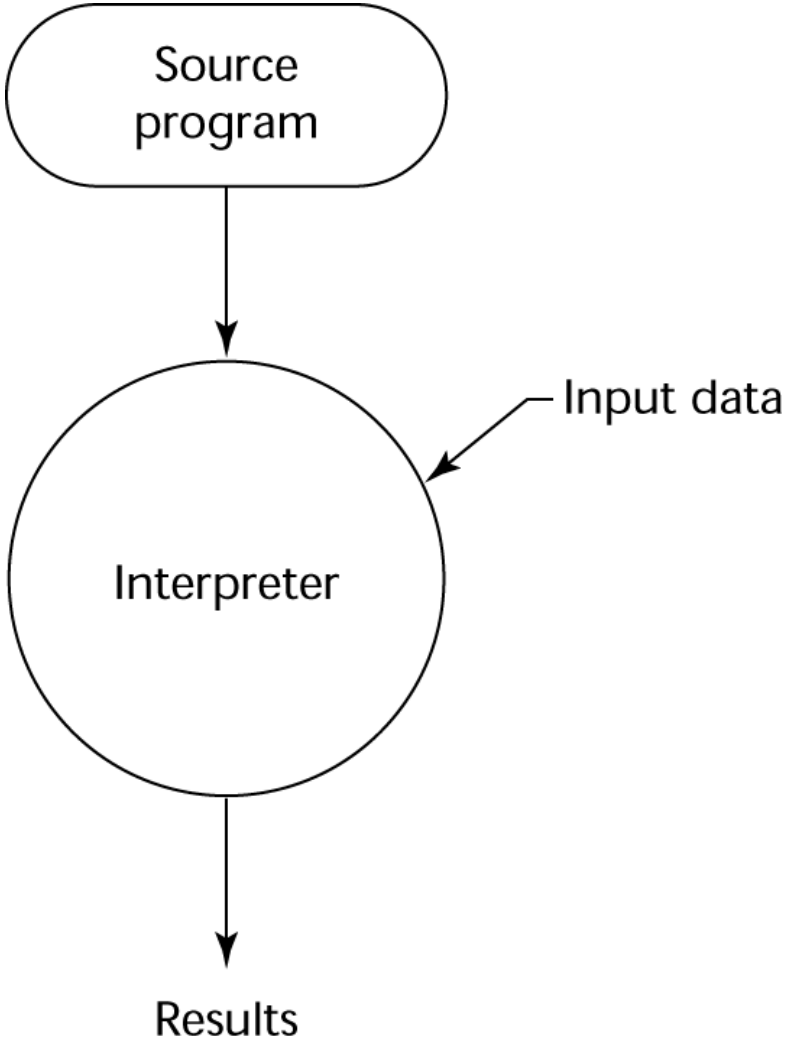
Pure Interpretation

- No translation
- Easier implementation of programs
  (run-time errors can easily and immediately be displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
  (source program & symbol table during interpretation)
- Early languages of the 1960s(APL, SNOBOL, and LISP)
  (Now rarely used on high-level languages)
- Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

빠른속도를 요구할때는 사용하지 않음
〉컴파일레이션보다 느리니까

+ 알파
클라이언트 서버관계가
예전에 2티어였다가
요즘 3티어로 올라옴
〉dependent를 줄여줌

Pure Interpretation Process



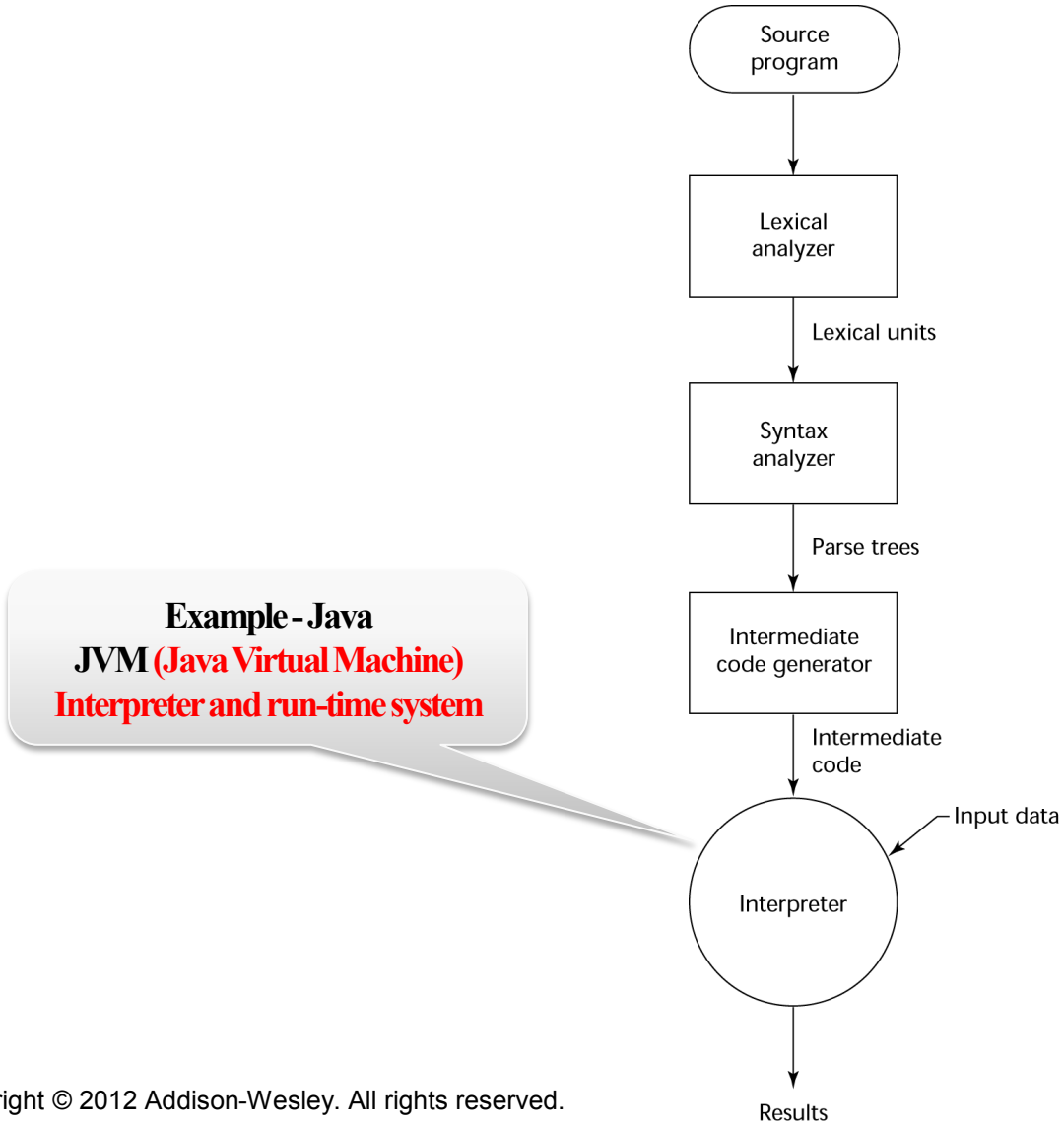
Hybrid Implementation Systems

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation

compiler의 역할을 하므로 재사용성이 늘어나서 해석언어보다 빠름

- Examples
  - **Perl programs** are partially compiled to detect errors before interpretation
  - Initial implementations of **Java** were hybrid; the intermediate form, **byte code**, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called ***Java Virtual Machine***)

Hybrid Implementation Process



Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then, during execution, JIT compiles the intermediate language methods into machine code when they are called
- Machine code version is kept for subsequent calls
- JIT systems are widely used for **Java programs**
- **.NET languages** are implemented with a JIT system

Preprocessors

- Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included
- A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros
- A well–known example: C preprocessor
  - expands **`#include`**, **`#define`**, and similar macros

Programming Environments

- A collection of tools used in software development
- UNIX
  - An older operating system and tool collection
  - Nowadays often used through a GUI (e.g., CDE, KDE, or GNOME) that runs on top of UNIX
- Microsoft Visual Studio.NET
  - A large, complex visual environment
- Used to build Web applications and non–Web applications in any .NET language
- NetBeans
  - Related to Visual Studio .NET, except for applications in Java

Summary

- The study of programming languages is valuable for a number of reasons:
  - Increase our capacity to use different constructs
  - Enable us to choose languages more intelligently
  - Makes learning new languages easier
- Most important criteria for evaluating programming languages include:
  - Readability, writability, reliability, cost
- Major influences on language design have been machine architecture and software development methodologies
- The major methods of implementing programming languages are: compilation, pure interpretation, and hybrid implementation