Course Code 007313 (Spring 2019)

프로그래밍언어의 개념
Concepts of Programming Language
(Lecture 02/03 : Chapter 1-Preliminaries)

Prof. A. P. Shrestha, Ph.D.
Dept. of Computer Science and Engineering, Sejong University

Today

- Chapter 1-Preliminaries
  - Reasons for studying Concepts of PL
  - Programming Domains
  - Language Evaluation Criteria

Next class

- Chapter 1-Preliminaries
  - Influences in Language Design
  - Language Categories
  - Implementation Methods
- Chapter 2-Evolution of Major Programming Languages

2.1.1 Reasons for Studying Concepts of PL

- Increased ability to express ideas

  - Each language has limits on the kinds of control structures, data structures, and abstractions.

  - As such, the forms of constructed algorithms is also limited.

  - Awareness of a wider variety of programming language features can reduce such limitations.

2.1.2 Reasons for Studying Concepts of PL

- Improved background for choosing appropriate languages

  - Programmers tend to use only the most familiar language, even if it is poorly suited for the project at hand.

  - If these programmers were familiar with a wider range of languages and language constructs, they are able to choose the language with the features that best address the problem.

2.1.3 Reasons for Studying Concepts of PL

- Increased ability to learn new languages

  - The process of learning a new programming language can be lengthy and difficult

  - Once thorough understanding of the fundamental concepts of languages is acquired, it becomes far easier to see how these concepts are incorporated into the design of the language being learned.

  - Eg: Understanding the concepts of object-oriented programming will result easier time in learning Java

2.1.4 Reasons for Studying Concepts of PL

- Better understanding of significance of implementation

  - Understanding implementation issues helps in

  i) finding and fixing bugs,

  ii) visualizing execution of various language constructs

  iii) relative efficiency of alternative constructs that may be chosen for a program.

  - For example, programmers unaware about the complexity of the implementation of subprogram calls do not realize that frequent calling of a small subprogram can be a highly inefficient design choice.

2.1.5 Reasons for Studying Concepts of PL

- Better use of languages that are already known

  - It is uncommon for a programmer to be familiar with all features of a language

  - By studying the concepts of programming languages, programmers can learn about previously unknown and unused parts of the languages they already use.

2.1.6 Reasons for Studying Concepts of PL

- Overall advancement of computing

  - Sometimes, the most popular languages are not always the best available

  - A language might become widely used because those in positions to choose languages were not sufficiently familiar with programming language concepts

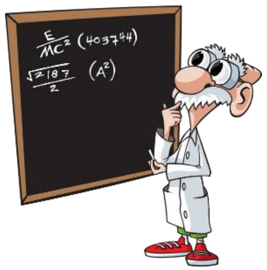
  - Eg. ALGOL 60 vs. FORTRAN

  - ALGOL 60 could not displace FORTRAN although it was more elegant and had much better control statements

  - Many did not clearly understand the conceptual design of ALGOL 60

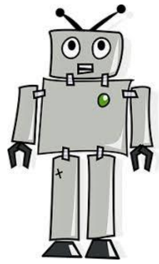
2.2.1 Programming Domains

- Scientific applications
  - Large numbers of floating point computations; use of arrays and matrices
  - Common control structures were counting loops and selections
  - FORTRAN, ALGOL 60, MATLAB

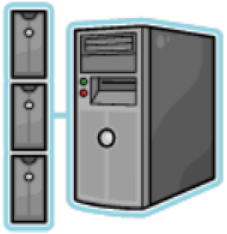


- Business applications
  - Produce reports, use decimal numbers and characters
  - COBOL

- Artificial intelligence
  - characterized by the use of symbolic rather than numeric computations
  - symbols, consisting of names rather than numbers
  - LISP



2.2.2 Programming Domains



- Systems programming
  - System Software: Operating system and the programming support tools
  - Need efficiency because of continuous use
  - C and C++

- Web Software
  - Eclectic collection of languages: markup (e.g., HTML), scripting (e.g., PHP), general-purpose (e.g., Java)

2.3 Language Evaluation Criteria

- Readability
  - the ease with which programs can be read and understood
- Writability
  - the ease with which a language can be used to create programs
- Reliability
  - conformance to specifications (i.e., performs to its specifications)
- Cost
  - the ultimate total cost

2.3.1.1 Evaluation Criteria-Readability

- Overall simplicity

  - A manageable set of features and constructs

  - Minimal feature multiplicity (i.e. more than one way to accomplish a particular operation).

Example i) count = count + 1
ii) count += 1
iii) count++
iv) ++count
(iii) and (iv) have slightly different meanings, but they are same when used as stand-alone expressions

  - Minimal operator overloading

Example
"+" : Addition for both integer and floating-point is acceptable
"+": Sum of all elements of between two single-dimensioned array will be confusing
(Vector addition has different meaning)
"+": Difference between respective elements of array is even worse!!

2.3.1.2 Evaluation Criteria-Readability

- 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

  - Every possible combination is legal

IBM

Instruction:
A Reg1, memory_cell
AR Reg1, Reg2

Semantics:
Reg1 ← contents(Reg1) + contents(memory_cell)
Reg1 ← contents(Reg1) + contents(Reg2)

VAX

Instruction:
ADDL operand_1, operand_2

Semantics:
operand_2 ← contents(operand_1) + contents(operand_2)

Operand can be either a register or a memory cell.

The VAX instruction design is orthogonal in that a single instruction can use either registers or memory cells as the operands

2.3.1.3 Evaluation Criteria-Readability

- Data types
  - Adequate predefined data types

Example: Flag Indicator

Numeric	Boolean
timeOut = 1	*timeOut* = True
Meaning is unclear	Meaning is perfectly clear

2.3.1.4 Evaluation Criteria-Readability

- Syntax Considerations
  - Special words and methods of forming compound statements

C language

```
for
{
  ....
}
```

```
if
{
  ....
}
```

ADA

```
for
  ....
end loop
```

```
if
  ....
end if
```

FORTRAN 95

- Special Words: ***Do* ,** ***End***
- ***Do*** and ***End*** are also legal variable names

- Form and meaning: appearance of statement should indicate their purpose

C language (violation) -**static**

i) Used on definition of a variable inside a function: variable is created at compile time
ii) Used on the definition of a variable that is outside all functions: variable is visible only in the file in which its definition appears; that is, it is not exported from that file.

2.3.2.1 Language Evaluation Criteria-Writability

- Simplicity and Orthogonality

  - large number of constructs→ difficult to familiarize with all

  - accidental use of unknown features → unusual results

  - a smaller number of primitive constructs and a consistent set of rules for combining them (i.e. orthogonality) is better

  - On the other hand, when nearly any combination of primitives is legal, errors in programs can go undetected.

2.3.2.2 Language Evaluation Criteria-Writability

- Support for abstraction

  - The ability to define and use complex structures or operations in ways that allow details to be ignored

Types of abstraction

i) Process abstraction: use of a subprogram

ii) Data abstraction: use of class (group data members using access specifiers)

2.3.2.3 Language Evaluation Criteria-Writability

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

Example

count++ is more convenient and shorter than count = count + 1

Note!!!

Readability demands minimal feature multiplicity

Contradiction??

2.3.3.1 Language Evaluation Criteria-Reliability

- Type Checking

  - Verify that the type of a construct (constant, variable, array, list, object) matches what is expected in its usage context

  - Run-time type checking is expensive, compile-time type checking is more desirable

Example

-Original C language: Passing an **int** type variable in a call to a function that expected a **float** type as its formal parameter was possible, and neither the compiler nor the run-time system would detect the inconsistency.

-Current version of C : eliminated this problem

2.3.3.2 Language Evaluation Criteria-Reliability

- Exception Handling

- The ability of a program to intercept run-time errors, take corrective measures, and then continue

<u>Examples</u>
i) a program attempting to divide by zero
ii) a user providing abnormal input
iii) a file system error being encountered when trying to read or write a file

- Ada, C++, Java, and C# include exception handling capabilities
- C and Fortran do not have such capabilities.

2.3.3.3 Language Evaluation Criteria-Reliability

- Aliasing

-Two or more distinct names that can be used to access the same memory cell

- A dangerous feature in a programming language

- Two pointers set to point to the same variable

- Changing the value pointed to by one of the two changes the value referenced by the other

2.3.4 Language Evaluation Criteria-Cost

- Training programmers to use the language
- Writing programs (closeness to particular applications)
- Compiling programs
- Executing programs
- Language implementation system: availability of free compilers
- Reliability: poor reliability leads to high costs
- Maintaining programs

2.3.5 Language 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

2.4 Influences on Language Design

- Computer Architecture

- Programming Methodologies Influences

2.4.1.1 Influences on Language Design

- Computer Architecture

  - Most languages around 50 years ago were developed around the *von Neumann* architecture

  - These languages are called imperative language

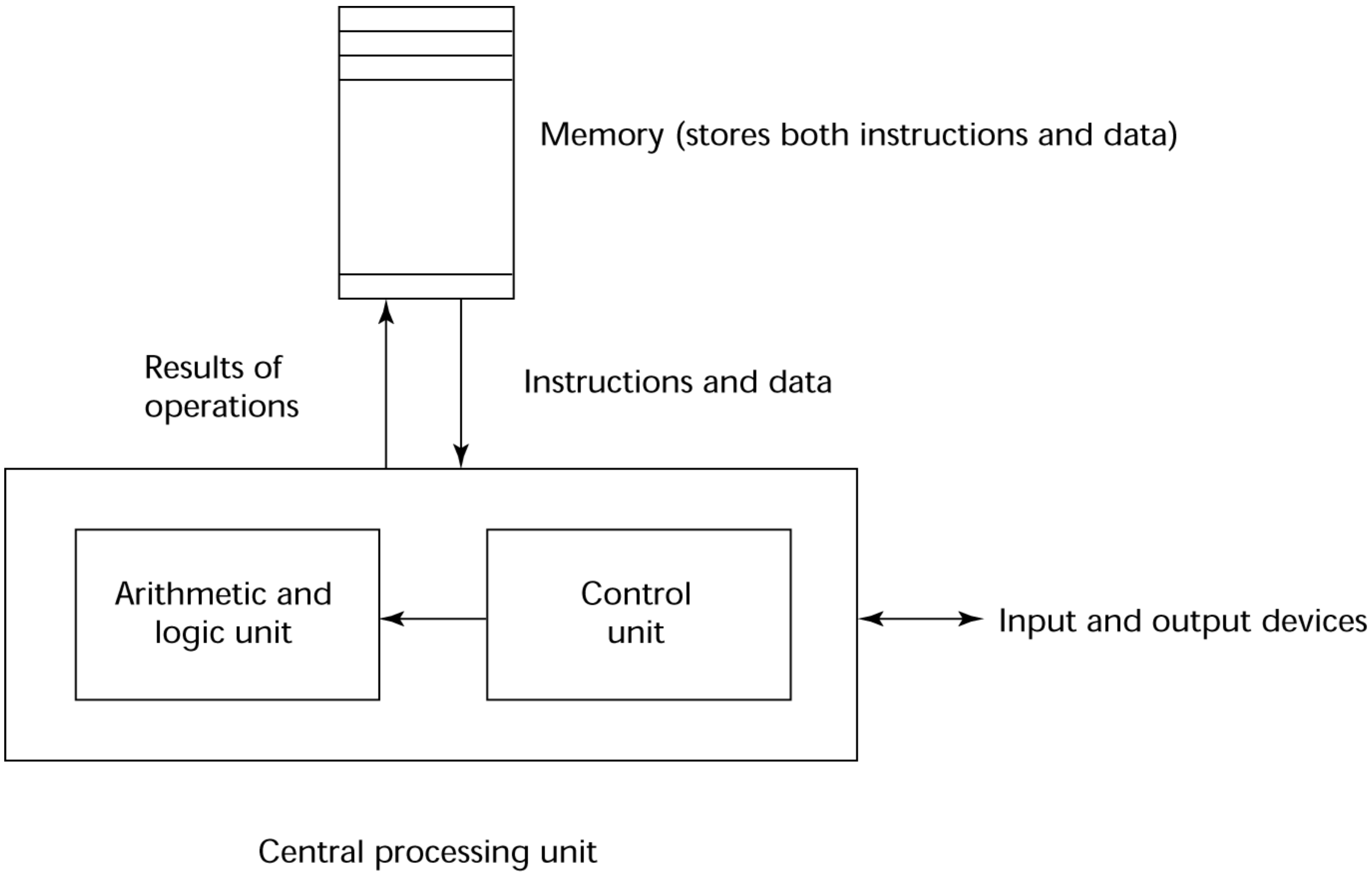
  - Data and programs are stored in the same memory

  - Memory is separate from CPU

  - Instructions and data must be transmitted, or piped, from memory to the CPU.

  - Results of operations in the CPU must be moved back to memory.

2.4.1.2 The von Neumann Architecture



2.4.1.3 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**
- Fetch-execute-cycle (on a von Neumann architecture computer)

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

2.4.2 Programming Methodologies Influences

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures
  - structured programming
  - top-down design and step-wise refinement
- Late 1970s: Process-oriented to data-oriented
  - Data abstraction
- Middle 1980s: Object-oriented programming
  - Data abstraction + inheritance + polymorphism

2.5 Language Categories

- Programming languages are often categorized into:
    1. Imperative,
    2. Functional,
    3. Logic, and
    4. *Object-oriented

*Not considered to form a separate category of languages

(although differs significantly from the procedure-oriented paradigm usually used with imperative languages)

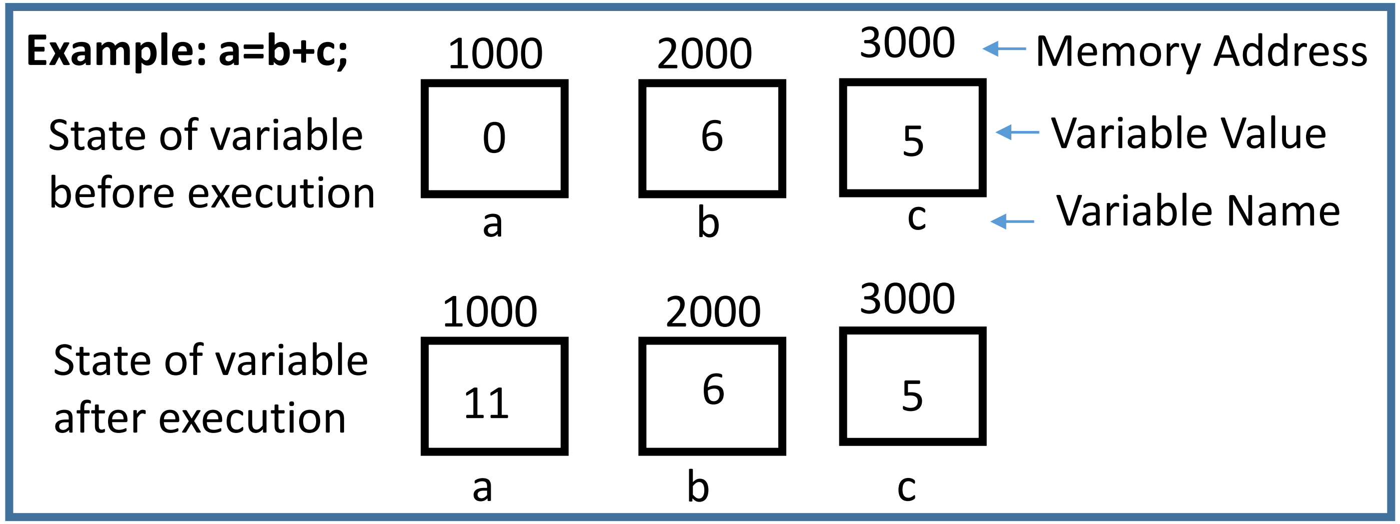
*The extensions to an imperative language required to support object-oriented programming are not intensive

2.5.1 Language Categories

- Imperative
  - Statement-oriented language in which the programmer instructs the machine how to change its state,
  - Consists of a sequence of statements and execution of each statement cause the computer to change the value of one or more location in its memory
  - Examples: C, FORTRAN, PASCAL, ADA etc

- Syntax

  Statement1;
  Statement 2;
  :
  Statement n;



2.5.2 Language Categories

- Functional (Applicative)
  - More emphasis is given to the function that the program represents rather than just the state changes, as the program executes statement by statement
  - Syntax

    $\text{function}_n(..\ \text{function}_n(\text{function}_n(\text{data}).....)$

  - Examples: LISP, Scheme, ML, etc

Example (LISP)

$x^2+y^2 \rightarrow$ (defun sumsqr(x y) (+(*xx) (*yy)))

2.5.3 Language Categories

- Logic (Rule Based)
  - Rule-based (rules are specified in no particular order)
  - Generally used for AI
  - Example: Prolog
  - Syntax enabling condition$_1 \rightarrow$ action$_1$
    enabling condition$_2 \rightarrow$ action$_2$
    ----
    enabling condition$_n \rightarrow$ action$_n$

- Object-oriented
  - Encapsulation
  - Data abstraction
  - Inheritance
  - Polymorphism
  - Message Passing
  - Example C++, JAVA, Python, C# etc

2.6.1 Implementation Methods

- Compilation

  - Programs are translated into machine language

  - The advantage of very fast program execution, once the translation process is complete

  - Use: Large commercial applications

- Pure Interpretation

  - Programs are interpreted by another program known as an interpreter

  - Use: Small programs or when efficiency is not an issue

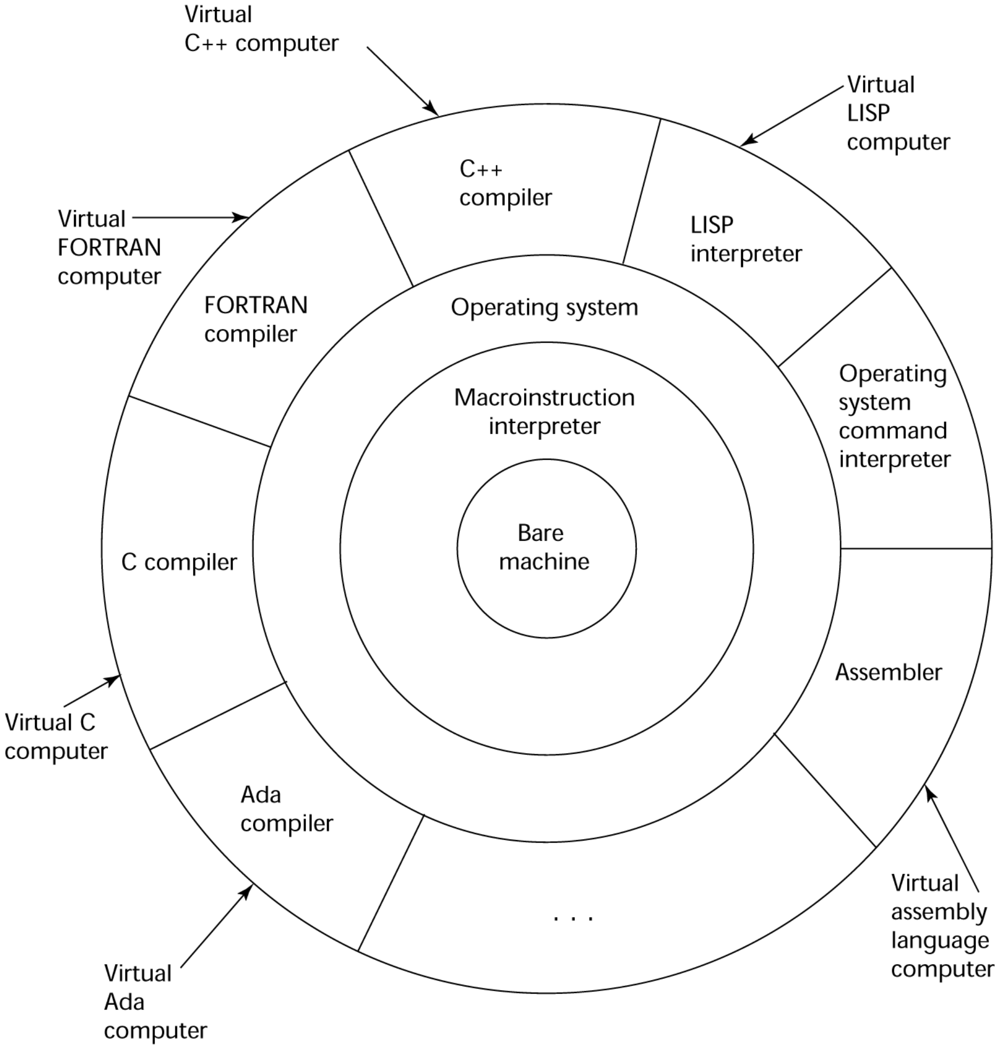
- Hybrid Implementation Systems

  - A compromise between compilers and pure interpreters

  - Use: Small and medium systems when efficiency is not the first concern

2.6.2 Layered View of Computer

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



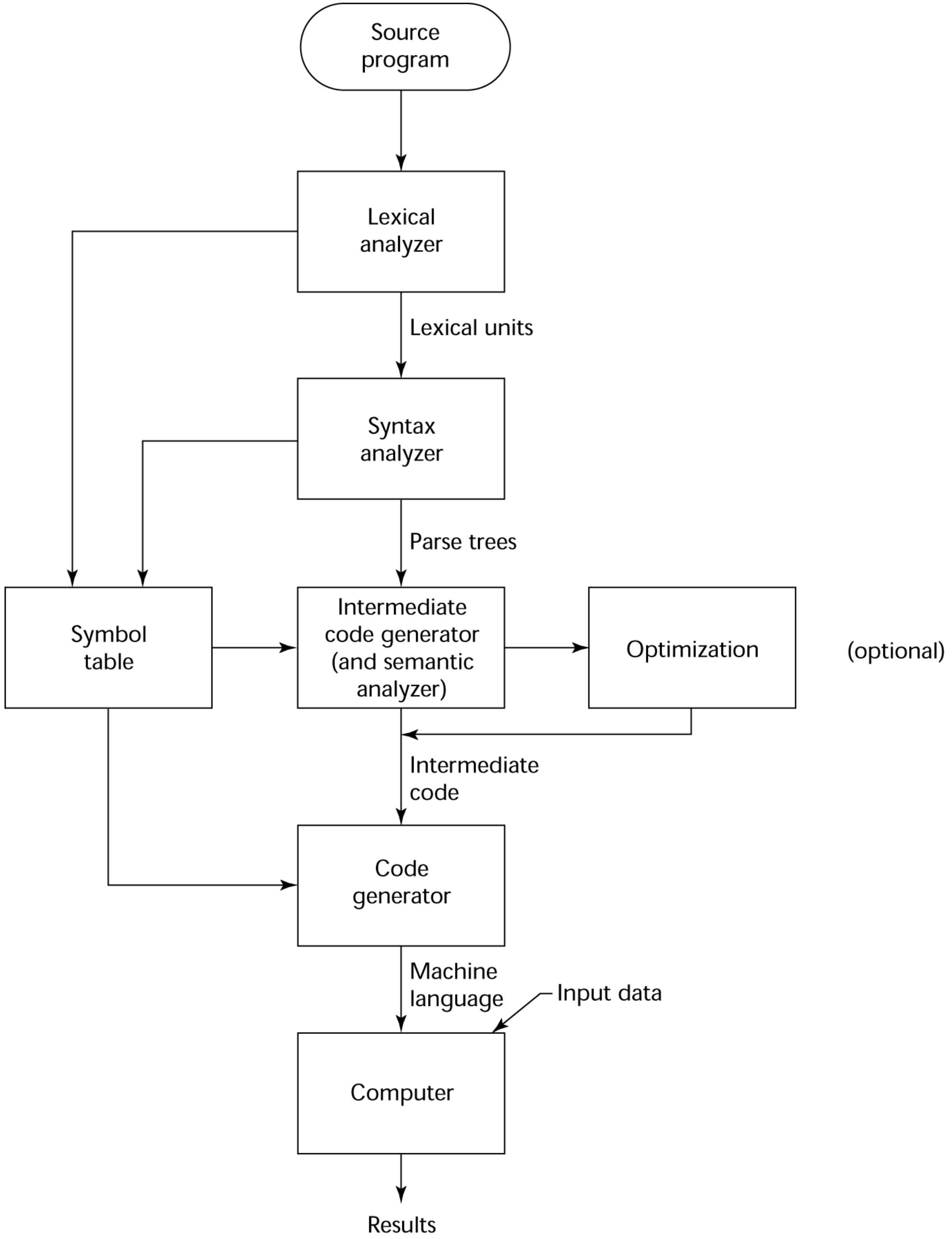
2.6.3 Compilation

- Translate high-level program (source language) into machine code (machine language)

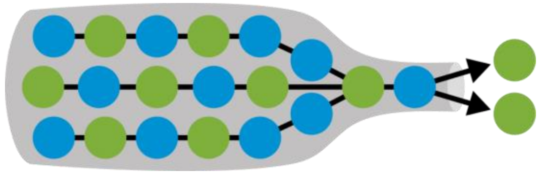
- Slow translation, fast execution

- Compilation process has several phases:
  - lexical analysis: converts characters in the source program into lexical units
  - syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
  - Semantics analysis: generate intermediate code
  - code generation: machine code is generated

2.6.4 The Compilation Process



2.6.5 Von Neumann Bottleneck!!



- Connection speed between a computer's memory and its processor determines the speed of a computer

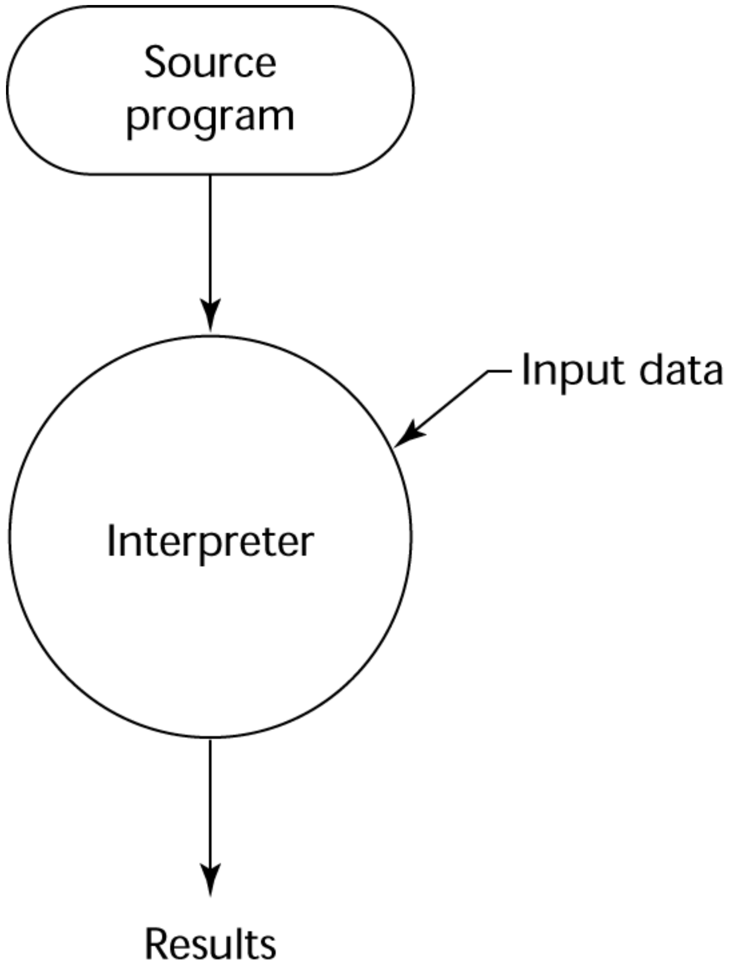
- Program instructions often can be executed much faster than the speed of the connection; the connection speed thus results in a *bottleneck*

- Known as the *von Neumann bottleneck*; it is the primary limiting factor in the speed of computers

2.6.6 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)
- Now rare for traditional high-level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

2.6.7 Pure Interpretation Process



2.6.8 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
- Examples
  - Perl programs are partially compiled to detect errors before interpretation

2.6.9 Just-in-Time Implementation Systems!!

- Initially translate programs to an intermediate language
- Then compile the intermediate language of the subprograms 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
- In essence, JIT systems are delayed compilers

2.6.10 Preprocessors

- Preprocessor 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

Q & A