Chapter 1

Preliminaries

Topics

- Why Studying *Concepts* of Programming Languages?
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

Why Studying Concepts of Prog. Languages?

- Increased ability to express ideas.
- Improved background for choosing appropriate languages.
- Increased ability to learn new languages.
- Better understanding of significance of implementation.
  - Implementation issues → understand a way that a language is designed → intelligent use of a language
- Better use of languages that are already known.
- Overall advancement of computing.

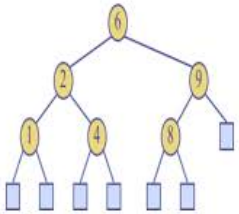
Input (data)

Algorithm

Output

Algorithm

+



Data Structure

+

python

Programming Language

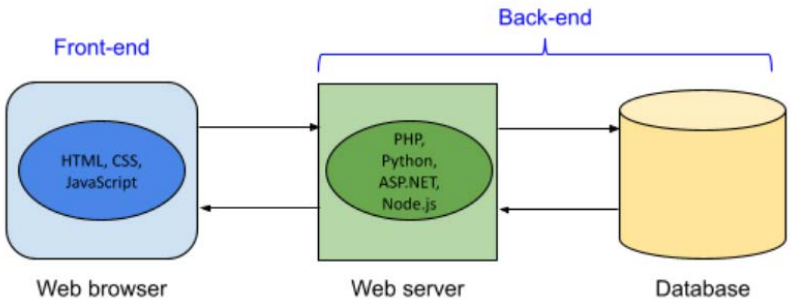
=

```
>>> def fib(n):
>>>     a, b = 0, 1
>>>     while a < n:
>>>         print(a, end=' ')
>>>         a, b = b, a+b
>>>     print()
>>> fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Program

Programming Domains

- Scientific applications
  - Large numbers of floating-point computations; use of arrays
  - Fortran
- Business applications
  - Produce reports, use decimal numbers and character data
  - COBOL
- Artificial intelligence
  - *Symbolic* computation:
    - Manipulation of symbols rather than numbers; use of linked lists
  - LISP, (Prolog, Python)
- Systems programming
  - Need efficiency because of continuous use
  - 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 most important criterium
  - 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 under all conditions.

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

Evaluation Criteria: Readability

- Overall simplicity
  - A *manageable set* of features and constructs – too many are bad!
  - *Minimal* feature multiplicity :

    e.g.) `count = count + 1, count +=1,  count++`

  - Minimal operator overloading: e.g.) +
- Orthogonality
  - A relatively *small set* of primitive constructs can be combined in a relatively *small number of ways* to build the control & data structures of language.
  - Every possible combination is legal/meaningful.
    - E.g.) 4 data types + 2 type operators → eight data structures.
  - The more orthogonal the design of a language, the fewer exceptions (i.e. a higher regularity) the language requires

    → easier to learn, read, and understand.
  - Meaning is context-independent.

Evaluation Criteria: Readability (cont.)

- Data types
  - Adequate predefined data types
  - E.g.) `timeout = 1` vs. `timeout = true`
- Syntax considerations
  - Identifier forms: flexible composition
  - *Special words* and methods of forming compound statements:

    e.g.) `while, class, for, begin/end, if/end if`
  - Form and meaning: self-descriptive constructs, meaningful keywords for their purpose

    e.g.) `static` – its meaning depends on the context of its appearance.

    If used on the definition of a variable *inside* a function,

    it means the variable is created at compile time.

    If used on the definition of a variable that is *outside* all functions,

    then the variable is visible only in the file where it is defined; non-exportable

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.
    - E.g.) In Modula2: definition module (`file.def`)
      vs. implementation module (`file.mod`)
- Expressivity
  - A set of relatively convenient ways of specifying operations.
    - E.g.) `count++ (in C)`
  - Number of operators and predefined functions.

Evaluation Criteria: Reliability

- Type checking
  - Testing for *type errors* either at compile time or run time (i.e. during execution).
- Exception handling
  - *Intercept* run-time errors, take corrective measures, then continue.
  - Ada, C++, Java, C#, Python (`try: … except: …`).
- Aliasing
  - Presence of two or more distinct referencing methods for the same memory location. -- unexpected change of a value
- 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.
  - The easier a program is to write, the more likely it is to be correct.

Evaluation Criteria: (cont.)

Table 1.1 Language evaluation criteria and the characteristics that affect them

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Evaluation Criteria: Cost

- Training programmers to use the language
- Writing programs (closeness to particular applications)
- Executing programs
- Reliability: poor reliability leads to high costs
- Maintaining programs

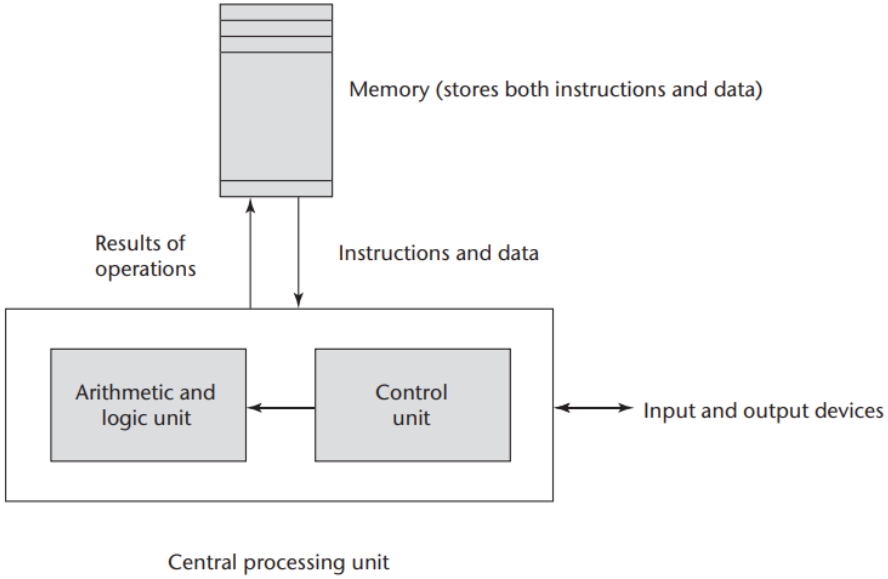
Evaluation Criteria: Others

- Portability
  - The ease with which programs can be moved from one implementation to another.
  - Most strongly influenced by the degree of *standardization* of the language

  $\Rightarrow$a standard version of the language: C++ in 1989 $\rightarrow$ 1998.
  (by an ISO)
- Generality
  - The applicability to a *wide range* of applications.
- Well-definedness
  - The completeness and precision of the language's official definition.

Other Influences on Language Design

- Computer Architecture
  - Languages are developed around the prevalent computer architecture, known as the *von Neumann* architecture.
    - CPU: ALU + Control Unit
    - Memory
    - Input/Output Devices
  - Future: quantum computer architect

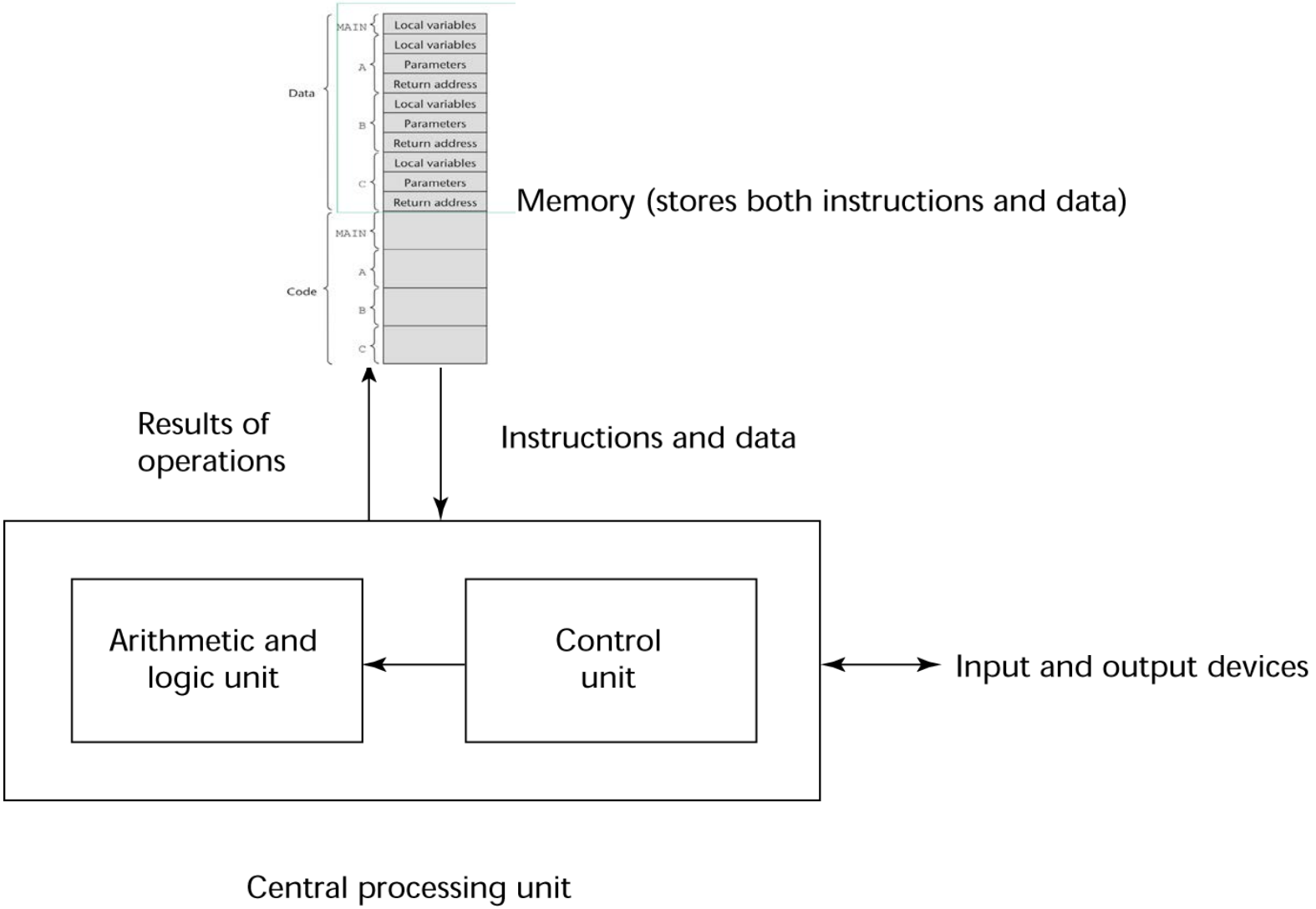


- Program Design Methodologies
  - New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages.

Computer Architecture Influence

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
  - Data and programs stored in memory.
  - Memory is separate from CPU.
  - Instructions and data are piped from memory to the CPU.
  - Basis for imperative languages
    - Variables model memory cells
    - Assignment statements model piping
    - Iteration is efficient

The von Neumann Architecture



The von Neumann Architecture

- Fetch-Execute-cycle

initialize the program counter

repeat forever

fetch the instruction pointed by the counter

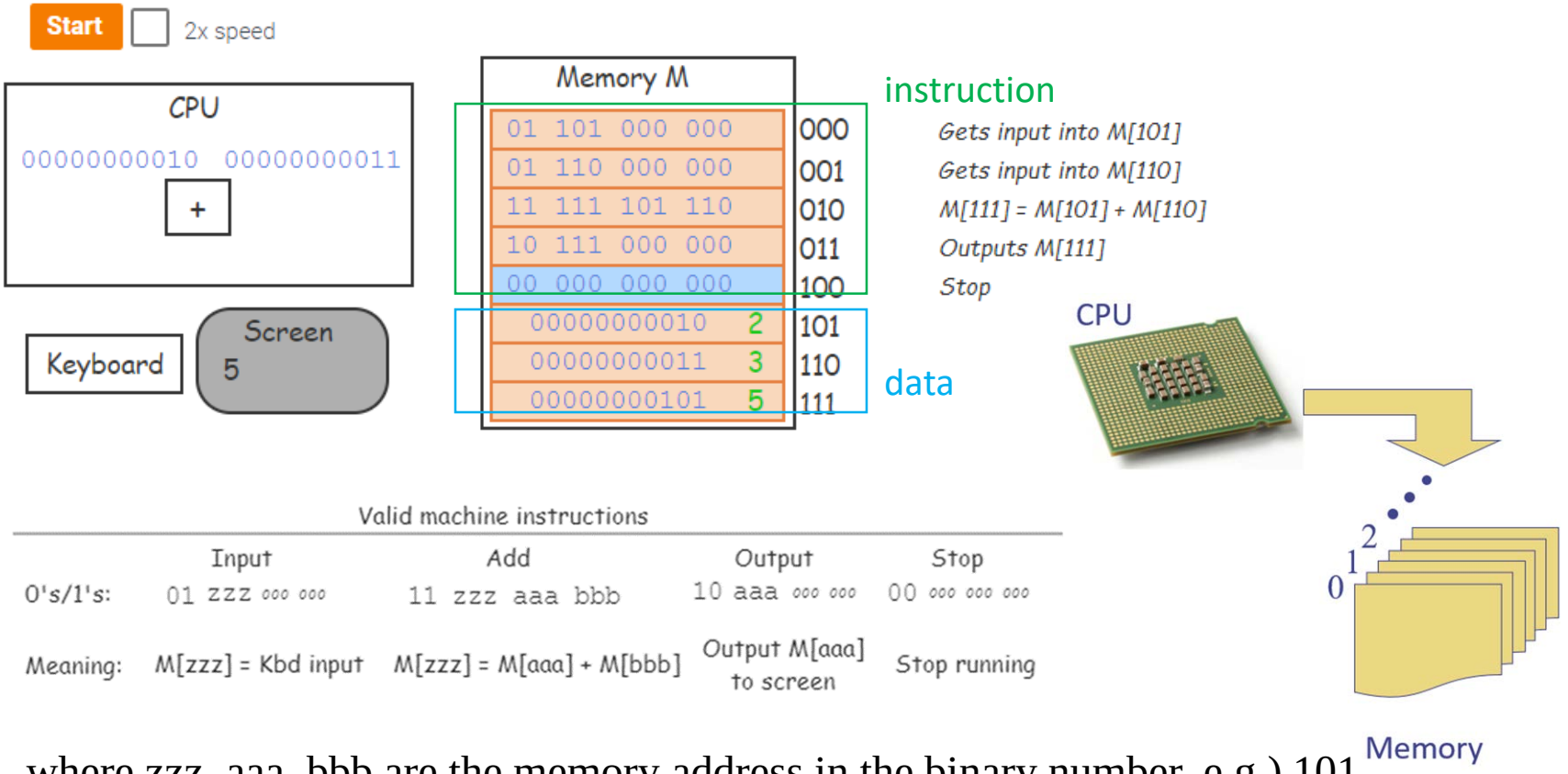
increment the counter

decode the instruction

execute the instruction

end repeat

2.8-2.9. Programming (adopted from zyBooks): in Machine Language



Valid machine instructions

	Input	Add	Output	Stop
0's/1's:	01 zzz ooo ooo	11 zzz aaa bbb	10 aaa ooo ooo	00 ooo ooo ooo
Meaning:	M[zzz] = Kbd input	M[zzz] = M[aaa] + M[bbb]	Output M[aaa] to screen	Stop running

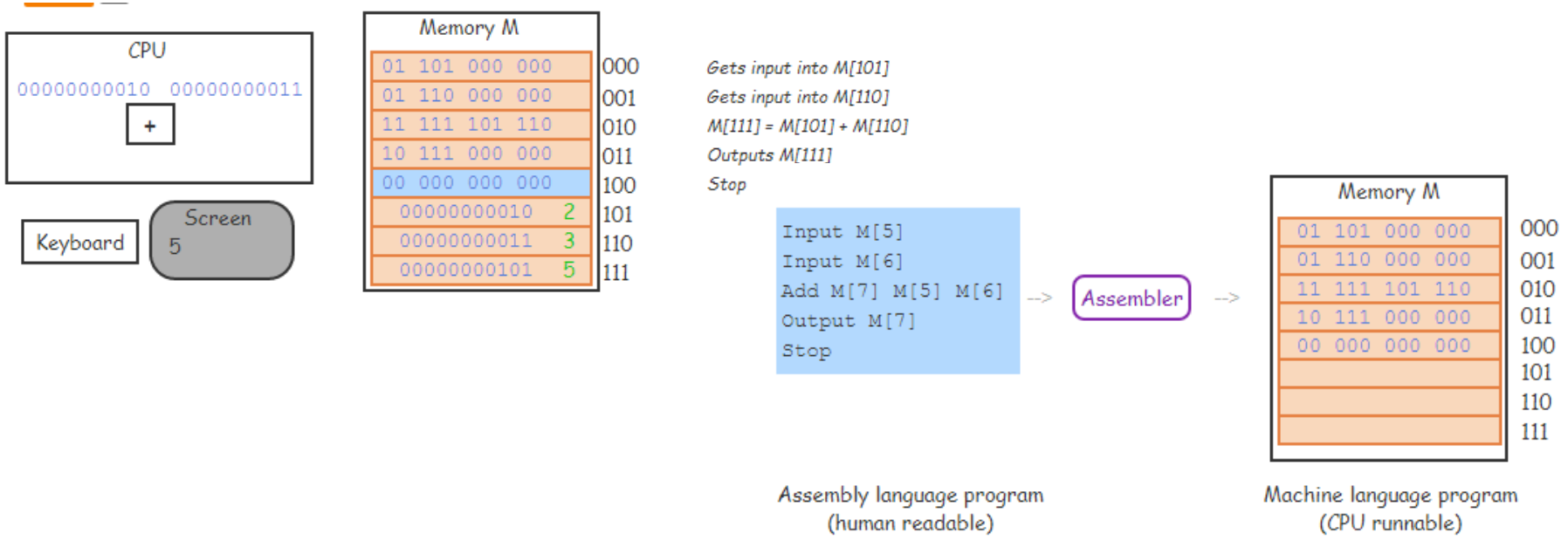
where zzz, aaa, bbb are the memory address in the binary number, e.g.) 101

2.8-2.9. Programming: Machine lang., Assembly lang. (adopted from zyBooks)

- Assembly Language: a textual human-understandable representation of a machine language's 0's and 1's, as in:

  Add M[7] M[5] M[6] (i.e. 11 111 101 110)

- Assembler: a program that automatically converts an assembly language program into machine language.

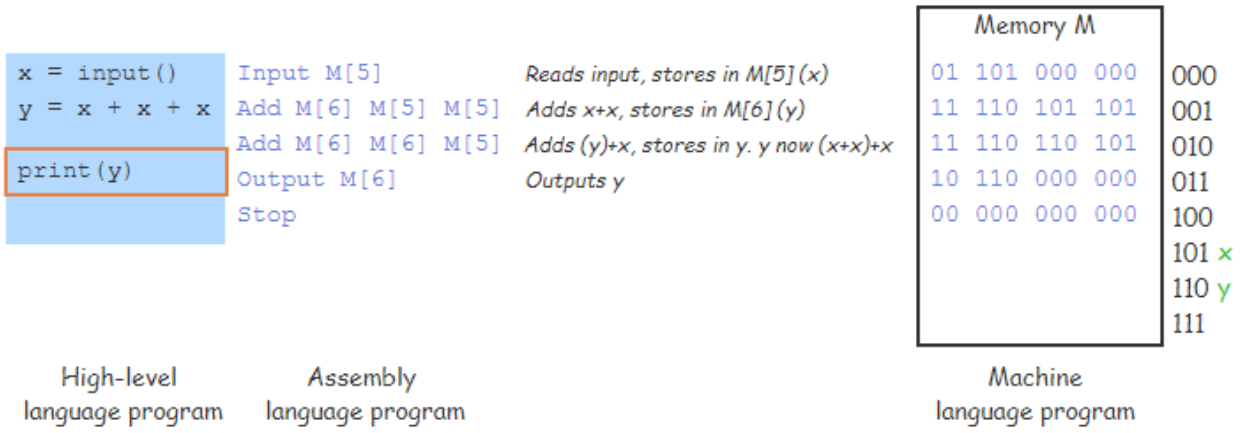


2.10. Programming: High-Level Language (adopted from zyBooks)

- High-level language: a programming language having higher-level instructions than assembly language, enabling greater programmer productivity.

  e.g.) all the modern prog. language such as Python, Java, C, etc. etc.

- Fortran (Formula Translation):
  - The 1st mainstream high-level language, from IBM in 1957,
- Compiler: a program that converts a high-level language to assembly /machine language.



Programming Design Methodologies

- 1950s - early 1960s:
  - Simple applications; worry about machine efficiency
- Late 1960s: 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 (first in SIMULA67)
- Middle 1980s: Object-Oriented Programming
  - Data abstraction + Inheritance + Polymorphism
    - Encapsulation of {data objects, process} + Control of access to data
    - Enhancing the potential reuse of existing SW.
    - Dynamic binding
  - OOP concept in Smalltalk's support Java, C++, C#.

Language Categories

- Imperative Programming Language (PL)
  - Central features: variables, assignment statements, and iteration
    - It uses statements of changing program's state by describing every step.
  - *Procedure-oriented* paradigm
  - Include languages that support object-oriented programming
  - Includes scripting languages
    - being bound by their implementation method than a language design – Perl, JavaScript, Ruby.
  - Includes the visual languages
  - Examples: Fortran, Pascal, Ada, C;

    Java, Perl, C++; JavaScript; Visual BASIC, .NET

Language Categories

- Object-Oriented PL
  - Popular OOPL grew out of Imperative PL.
  - An object consists of some internal data items plus operations that can be performed on that data.
  - OOPL supports decomposing a program into objects.
  - E.g.) C++, Python, Java, C# (extensive OO support).
  - Not OOPL: C, MATLAB, Javascript, Fortran, COBOL, Ada, etc.

Language Categories

- Functional PL
  - Main means of making computations is by applying functions to given parameters.
  - Examples: LISP, Scheme, ML, F#
- Logic PL
  - Rule-based
    - Designed for inference with knowledge in the form of rules.
    - Rules in no particular order - the language implementation system must choose an order in which the rules are used to produce the desired result.
  - Example: Prolog

Language Categories

- Markup/programming hybrid PL
  - A Markup language allows a developer to describe a document's content, desired formatting, or other features. E.g.) HTML
  - HTML (Hyper-Text Markup Language):
    - a textual language for creating web pages allowing a developer to describe the text, links, images, and other features of a web page.
    - Extension: .html or htm
    - not executed statement-by-statement - HTML file is not a program.
    - A web browser reads an HTML file and renders the corresponding web page.
    - An HTML file surround text by different tags to yield different formatting.
    - Tags: an HTML file contains normal text surrounded by tags that indicate formatting, links, or other items: e.g.) <head> </head> <body> </body>, etc.
    - Link: text that can be clicked to jump to another web page.
  - Markup languages extended to support some programming
  - Examples: HTML, XML, JSTL(Java Server Pages Standard Tab Library), XSLT (eXtensible Stylesheet Language Transformation)

Tags	Purpose	Example HTML
h1, h2, h3	Headers. h1 is largest.	`<h2> Puppies are cute </h2>`
p	Paragraph	`<p> Humans seem designed to see puppies as cute. </p>`
b, i, u	Bold, italic, underline	`<p> Humans seem <u>designed</u> to see puppies as cute. </p>`

```
<!DOCTYPE html>
<html>
<head>
  <title>My Awesome Page's Title</title>
</head>
<body>
   <h1> This is a Header </h1>
   <p> Here is a paragraph, with <i>some italicized words</i>
       and <b>some bolded words</b> too.
   </p>
</body>
</html>
```

This is a Header

Here is a paragraph, with *some italicized words* and **some bolded words** too.

```
<body>
   <p> My favorite web page is <a href="https://wikipedia.org"> this one</a>. My second favorite is FINISH THIS. </p>
</body>
```

My favorite web page is this one. My second favorite is FINISH THIS.

Language Design Trade-Offs

- Reliability vs. Cost of execution
  - Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs.
- Readability vs. Writability
  - Example: APL 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.

Implementation Methods

- Compilation (slide-#29)
  - Programs are translated into machine code.
  - Slow translation but Fast execution
  - includes JIT (Just-In-Time) systems:

    source language →translate intermediate language

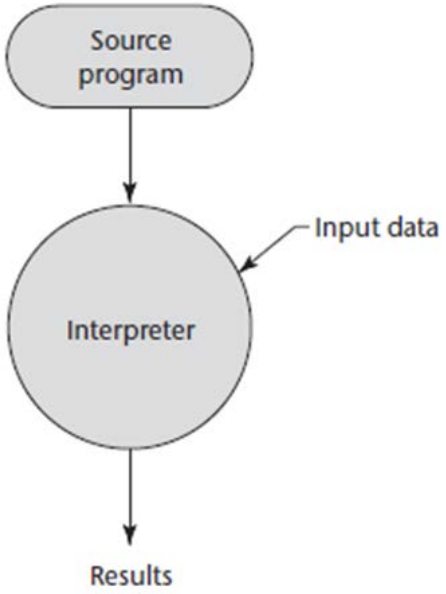
    →during execution machine code.

    JIT:

    - Compile the intermediate language of the subprograms into machine code *when they are called*. -- Delayed Compiler
    - Machine code version is kept for subsequent calls.
    - Widely used for Java programs
    - .NET languages are implemented with a JIT system.
  - Use: Large commercial applications

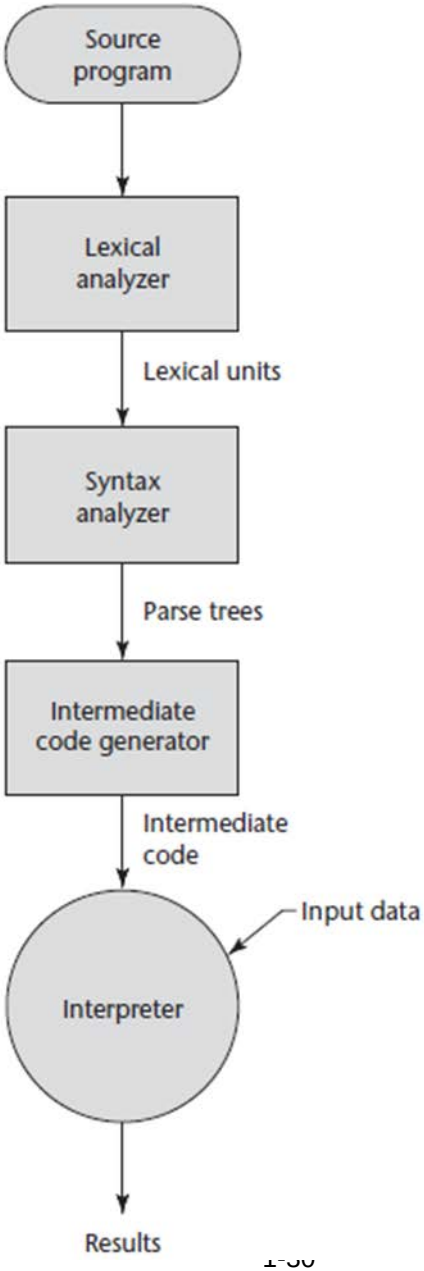
Implementation Methods

- Pure Interpretation
  - Programs are interpreted by another program known as an interpreter
  - No translation – Slow execution
  - Easier implementation of programs
  - Becoming rare
  - Use: Small programs or
    when efficiency is not an issue



Implementation Methods

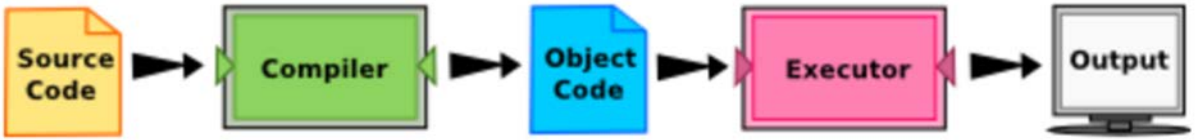
- *Hybrid* Implementation Systems (slide-#33)
  - A compromise between compilers and pure interpreters
  - A high-level language program is translated to an intermediate language that allows easy interpretation.
  - Small translation cost.
  - Use: Small and medium systems when efficiency is not the first concern
  - Examples
    - Perl programs are partially compiled to detect errors before interpretation.
    - Initial implementation of Java Initial - the intermediate form, *byte code*, is portable to any machine with a byte code interpreter and a run-time system.



Compiler vs. Interpreter

- Interpreter: it reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.

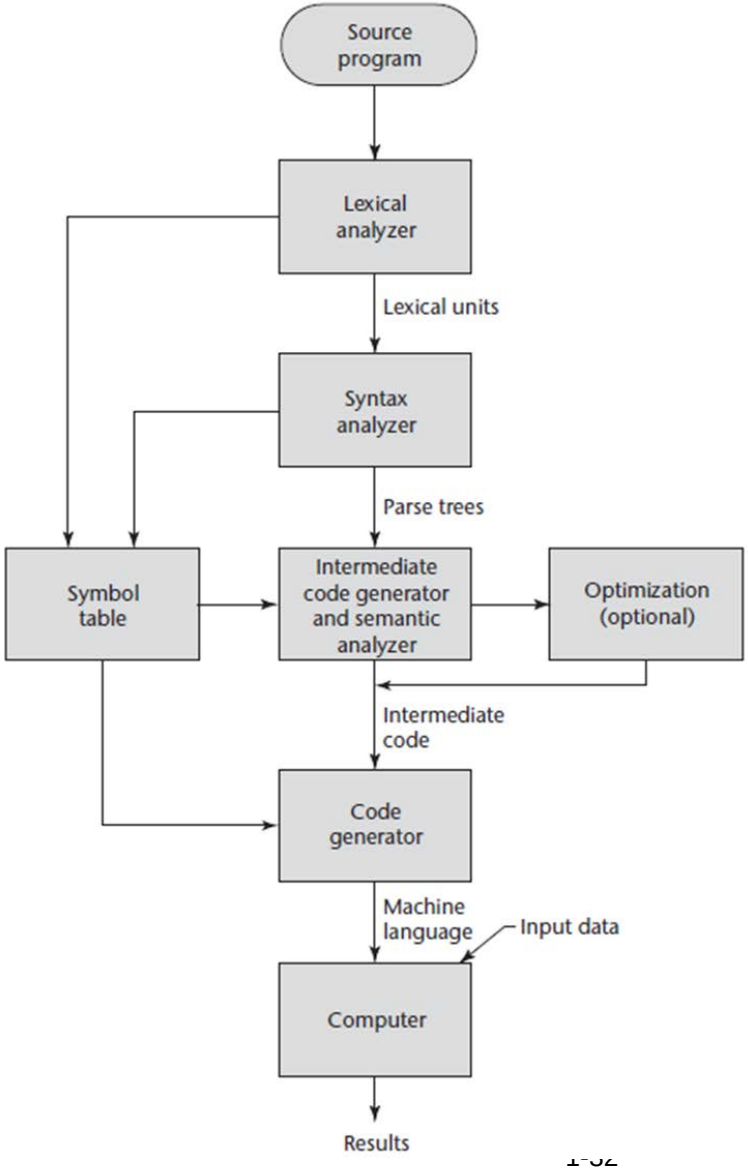
- Compiler: it reads the program and translates it completely before the program starts running. In this case, the high-level program is called the source code, and the translated program is called the object code or the executable.



- *Python, Perl, Matlab,* and *Lisp* uses both interpreter and compiler.

Compilation

- Several phases in Compilation:
  - Lexical analysis: converts characters in the source program into lexical units – identifier, operator, etc.
  - Syntax analysis: transforms lexical units into *parse trees* which is a hierarchical syntactic structure of a program.
  - Semantics analysis: generates intermediate code and checks errors that couldn't be detected during syntax analysis.
  - Code generation: machine code is generated.



Additional Compilation Terminologies

- **Load module** (executable image):
  - the user and system code together

- **Linking (and loading)**: by a ***linker***
  - the process of collecting system program units and linking them to a user program.
  - The linking operation connects the user program to the system programs by placing the addresses of the entry points of the system programs in the calls to them in the user program.

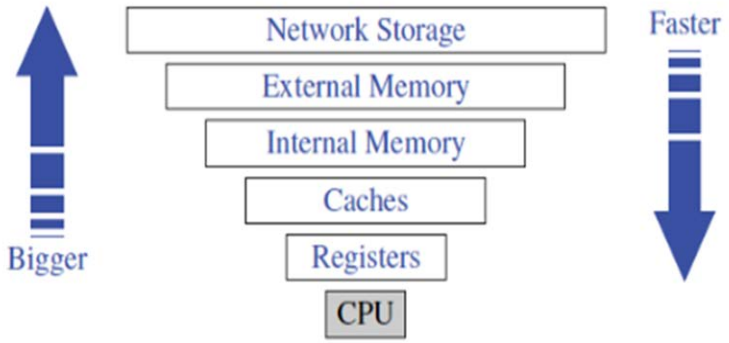
Von Neumann Bottleneck

- 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

Ref.) Memory Hierarchies

(from Introduction to Algorithms by Cormen, et.al.)

- A large amount of data for computer applications – often too large to fit in the internal memory.
- Computers have a hierarchy of different kinds of memories, which vary in terms of their size and distance from the CPU.
- The internal **registers** – the closest to CPU, very fast access, but relatively few such locations.
- The **cache** memory.
- The (**internal) memory** – a.k.a. main memory, core memory, RAM.
- The **external memory (Drive)** – HD, SDD, CD, DVD, tapes, etc.
- The network storage - iCloud, Dropbox, OneDrive, Google drive, etc.



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
  - E.g.) `#include “myLib.h”,`
  - `#define max(A, B) ((A)>(B)?(A):(B))`
  - `x=max(2*y, z/1.7)`

    ⇒expanded to `x= ((2*y)>(z/1.7)?(2*y):(z/1.7))`

  where myLib.h is a header file that contains a function declaration and the macro definitions to be shared b/t several source files.

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.
  - Cf) Linux: open-source Unix-like OS based on Linux kernel.
- Microsoft Visual Studio .NET
  - A large, complex visual environment
  - Used to build Web applications and non-Web applications in any .NET language - Visual Basic.NET, C#, C++
- NetBeans
  - IDE (Integrated Development Environment) for Java that has extensions for PhP, C, C++, HTML5, JavaScript.

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:

  compilation, pure interpretation, and hybrid implementation

Programing Language Tutorials

- w3schools.com
- javatpoint.com
- geeksforgeeks.org/
- www.tutorialspoint.com/computer_programming_tutorials.htm