CS 280
Programming Language Concepts

Spring 2025

Bassel Arafeh

GITC 4408

Ph: +(973) 596-3234

CS 280 - Programming Language Concepts

- Prerequisites: CS 114, or CS 116, or IT 114 or equivalent with a grade C or better.
- Conceptual study of programming language syntax, semantics and implementation. Course covers language definition structure, data types and structures, control structures and data flow, runtime consideration, and interpretative languages.
- **Textbook**:
  - **T:** Robert W. Sebesta, Concepts of Programming Languages, 12th Edition., Pearson, 2019.
- **Recommended supporting references**
  - **R1:** Nell Dale; Chip Weems; Tim Richard, Programming and Problem Solving with C++, 6th/7th Edition, Jones & Bartlett Learning.
  - **R2:** Michael D. Adams, Lecture Slides for Programming in C++ [The C++ Language, Libraries, Tools, and Other Topics] (Version 2021-04-01) http://www.ece.uvic.ca/~mdadams/cppbook

Goals of the Course

At the conclusion of this course, the successful (passing) student would have learned:

1. The student will be able to recognize common features of different programming languages.
2. The student will have higher ability for learning new programming languages.
3. The student will gain an appreciation of the strengths and weaknesses of different programming languages.
4. The student will be able to apply programming language syntax and semantics concepts in the implementation of a specified programming language.
5. The student will be able to apply computer science concepts and software development fundamentals to produce computing-based solutions.

Course Format and General Policies

- **Major Topics:**
  - Common features of programming languages
  - Overview of C++ Language (First 3 weeks)
  - Language syntax and semantics
  - Lexical and syntax analysis
  - Variables bindings, scopes and lifetimes
  - Data types
  - Expressions, and assignment statements
  - Control flow statements
  - Subprograms
  - Abstract data types
  - Support for object-oriented programming
  - Exception Handling

Course Format and General Policies

- CS280 Lectures for ALL sections will meet on **Mondays, and Thursdays.**

- **Distribution of Grades**

Component	Percentage
Project Programming Assignments (PAs)	**32%**
Short Programming Assignments (SAs)	**10%**
Mid-term Exam (Online)	**25%**
Final Exam (Online)	**33%**

Course Format and General Policies

- **Short Programming Assignments (SAs)**
  - **They are short programming assignments. They are planned and structured mainly to serve the goals of the programming projects, and to bridge a gap in using certain C++ programming concepts and techniques addressed in the lectures.**
  - There will be **4 SAs** with a weight of **10%** of your total grade.
  - **SA work will be due five days from the posting class day. An extended submission period of two days will be allowed after that with a penalty of 25% deduction from the student's score. Note, no SA submissions will be considered after the extended deadline for submission. See also the course calendar schedule for more details.**

Course Format and General Policies

- **Project Programming Assignments (PAs)**
  - There will be 3 programming projects with a weight of **32%** of your total grade.
  - **Extended submission period of PAs will be allowed after the announced due date with a fixed penalty of 25% deduction from the student's score for 3 days.** The schedule of posting the programming projects and their due dates are shown in the course calendar.

- **Note: All programming assignments must be written in C++, and must be submitted through Canvas to the Vocareum Environment for automatic grading.**

Course Format and General Policies

- **Mid-term and Final Exams**
  - **All Exams are conducted online in the designated exam rooms. All exams require Respondus Lockdown Browser. Please read the NJIT policy in regard to using Lockdown Browser as the proctoring method. The course will use the "New" quizzes engine on Canvas.**
  - The common midterm will be on **Monday 4:00 pm - 5:15 pm – 3/10/2025**. The final exam will be during the week of **May 10-16, 2025,** Both exams are common to ALL sections and MUST be taken by all students and cannot be rescheduled.

Course Format and General Policies

- **YWCC Tutoring for CS 280**

  Tutoring assistance is provided to CS 280 students through the College. Please visit the College page for Fall 2024 undergraduate tutoring at https://computing.njit.edu/tutoring for the scheduled periods for the CS 280 course.

Chapter 1

PRELIMINARIES

Chapter 1 Topics

- Introduction
- An Overview of Programming Languages History
- Reasons for Studying Concepts of Programming Languages
- Programming Domains
- Influences on Language Design
- Language Categories
- Programming Languages Common Concepts
- Implementation Methods
- Programming Environments

Introduction

- How, when and why did programming languages evolve?
  - All processors support a very small set of simple instructions.
  - Instructions are represented as bit patterns that are used to control the operations of the machine.
  - VERY early on, the notion of a stored program (as compared to wired in or fed in from cards or magnetic tapes) developed
  - VERY early on, simple mnemonic languages were developed to make programming a little bit easier

Assembler Language

- Machine code has a series of possible operations and operands, represented by unique bit patterns
- Some mnemonic representation of the operations helps make things readable
- Symbolic names for resources like processor registers help a lot.
- Symbolic names for memory locations help even more

Assembler example

```
;  -----------------------------------------------------------------------;
Writes "Hello, World" to the console using only system calls. Runs on 64-
bit Linux only. ; To assemble and run:
;
;       nasm -felf64 hello.asm && ld hello.o && ./a.out
;   -----------------------------------------------------------------------
        global _start
        section .text
_start: mov     rax, 1              ; system call for write
        mov     rdi, 1              ; file handle 1 is stdout
        mov     rsi, message        ; address of string to output
        mov     rdx, 13             ; number of bytes
        syscall                     ; invoke operating system to do the write
        mov     rax, 60             ; system call for exit
        xor     rdi, rdi            ; exit code 0
        syscall                     ; invoke operating system to exit section
        .data
message: db     "Hello, World", 10        ; note the newline at the end
```

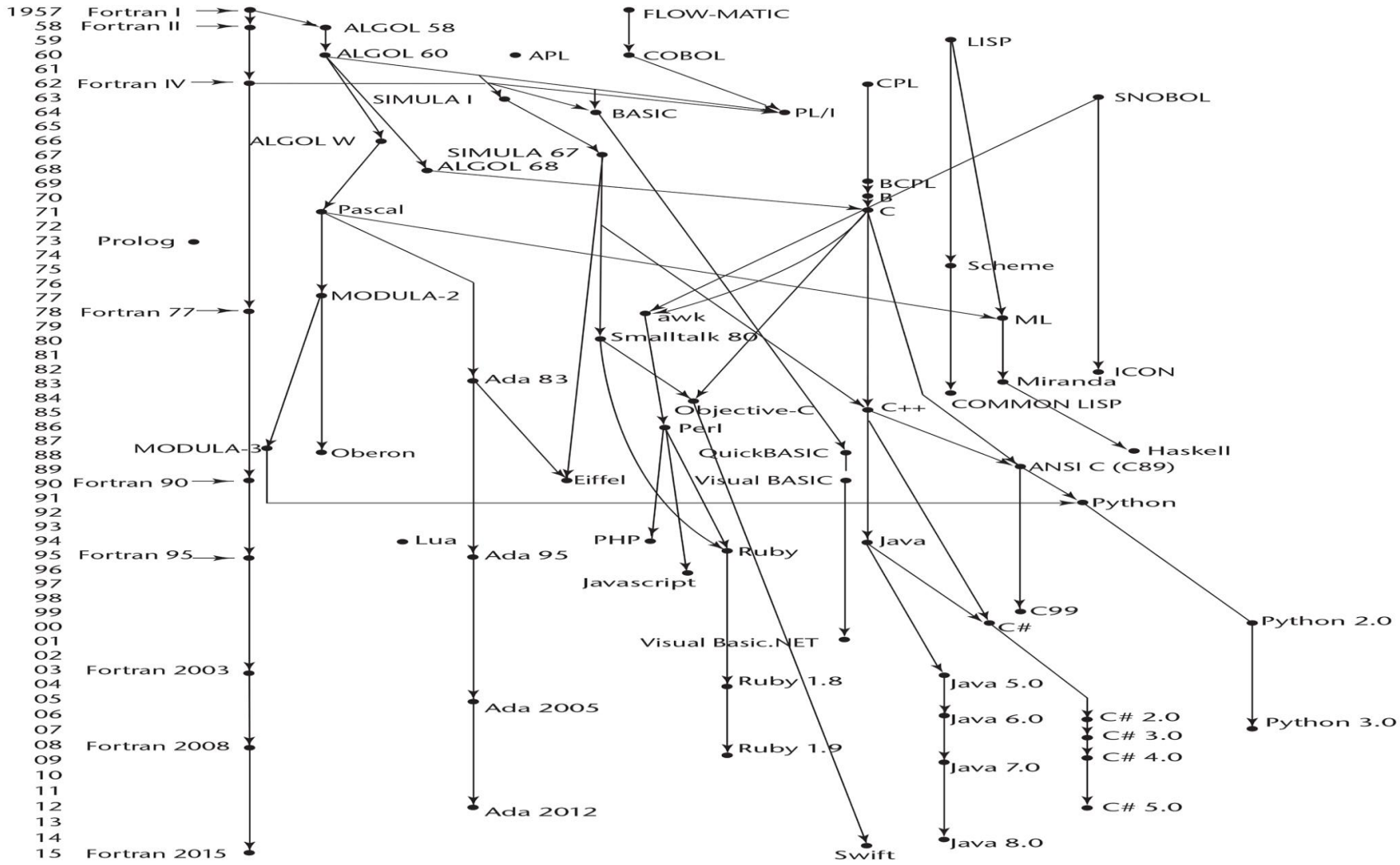
Programming Language History

- **Higher Levels of Language**
  - A higher level of abstraction
  - Easier to read and write
  - Easier to develop systems for more complex uses

- Programming language translation systems (and operating systems) make the hardware usable.

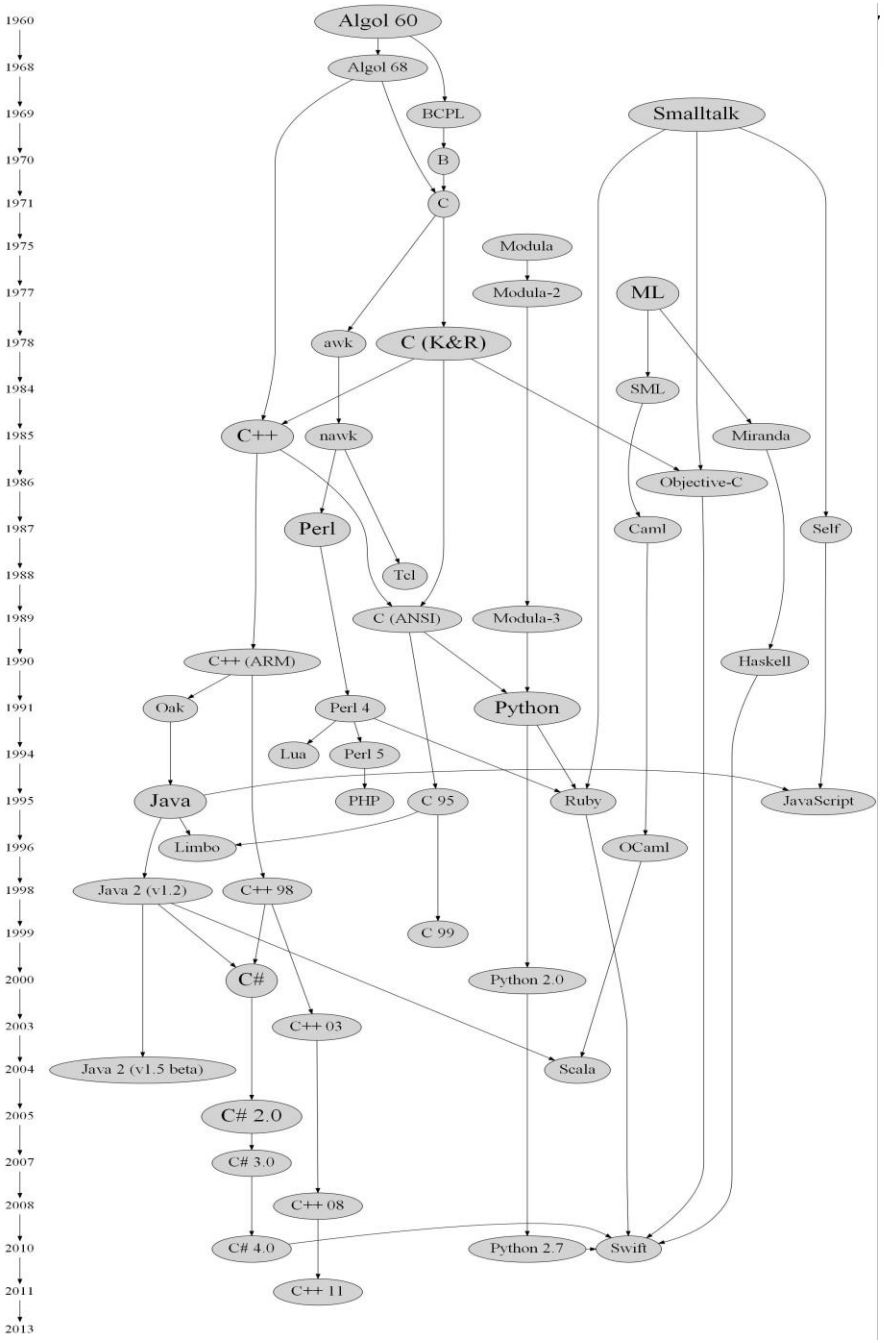
- Each high-level programming language has one or more translators or implementations which translates all programs in that language.

Figure 2-1 Genealogy of common high-level programming languages



Programming Language History

Evolution of the C-based Languages



Reasons for Studying Concepts of Programming Languages

- Increased ability to choose among alternative ways to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Better use of languages that are already known
- Overall advancement of computing

Programming Domains

- Scientific applications
  - Large numbers of floating point computations; use of arrays
  - Fortran
- Business applications
  - Produce reports, use decimal numbers and characters
  - COBOL
- Artificial intelligence
  - Symbols rather than numbers manipulated; use of linked lists
  - LISP
- 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)

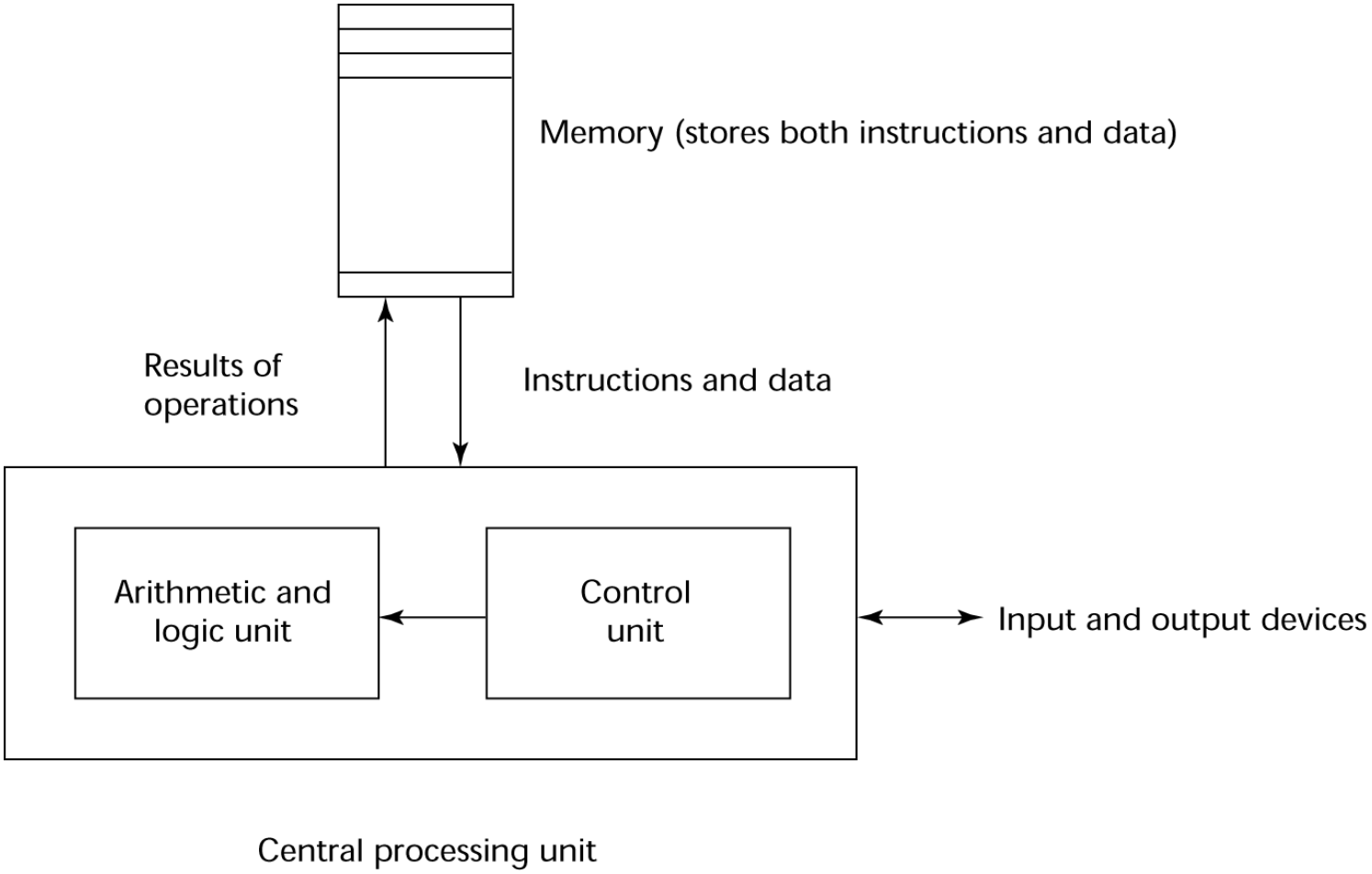
Influences on Language Design

- **Computer Architecture**
  - Languages are developed around the prevalent computer architecture, known as the *von Neumann* architecture
- **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 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 (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
```

The von Neumann Architecture

- **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

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

Language Categories

- **Imperative**
  - Central features are variables, assignment statements, and iteration
  - Including languages that support object-oriented programming, scripting languages, and 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
- **Markup/programming hybrid**
  - Markup language is used to specify the layout of information in Web documents
  - Markup languages extended to support some programming
  - Examples: JSTL (Java Server Pages Standard Tag Library), and XSLT (eXtensible Stylesheet Language Transformations).

Introduction to Programming Languages Common Concepts

- Programming languages have common concepts that are seen across all languages
- This course will discuss and illustrate these common concepts:
  - Syntax
  - Semantics
  - Variables
  - Types
  - Etc.
- The focus will be on C++ and Java features, with examples from other imperative programming languages whenever there is a need for comparison.
  - such as C#, Python and JavaScript.

Syntax

```
int x;
float pi = 3.14159;
```

- What **words and symbols** are valid in the language?
- In what order must those words and symbols appear in order to be valid?
- Incorrect words, symbols, or ordering are errors that prevent a program from running: “Syntax error”.

Semantics

```
int x;
float pi = 3.14159;
```

- What does a valid combination of words and symbols mean?
- Different languages have different rules.
- First line: x is a symbol which can hold an int, and which behaves like an int.
- Second line: pi is a symbol which can hold a float, and which behaves like a float. It has an initial value of 3.14159.
- Note differences: Java semantics say that x has an initial value of 0. C/C++ semantics say x has no predetermined initial value; if you want a value, assign one!

Names

```
int x;
float pi = 3.14159;
```

- Languages usually have **variables**, with rules for how to create names: what characters are valid? How long can names be?
- Some languages (Java/C/C++) require variables to be created by declaring them before using them.
- Other languages (Python) have no such requirement; variables are created by assigning to them.

Types

```
int x;
float pi = 3.14159;
```

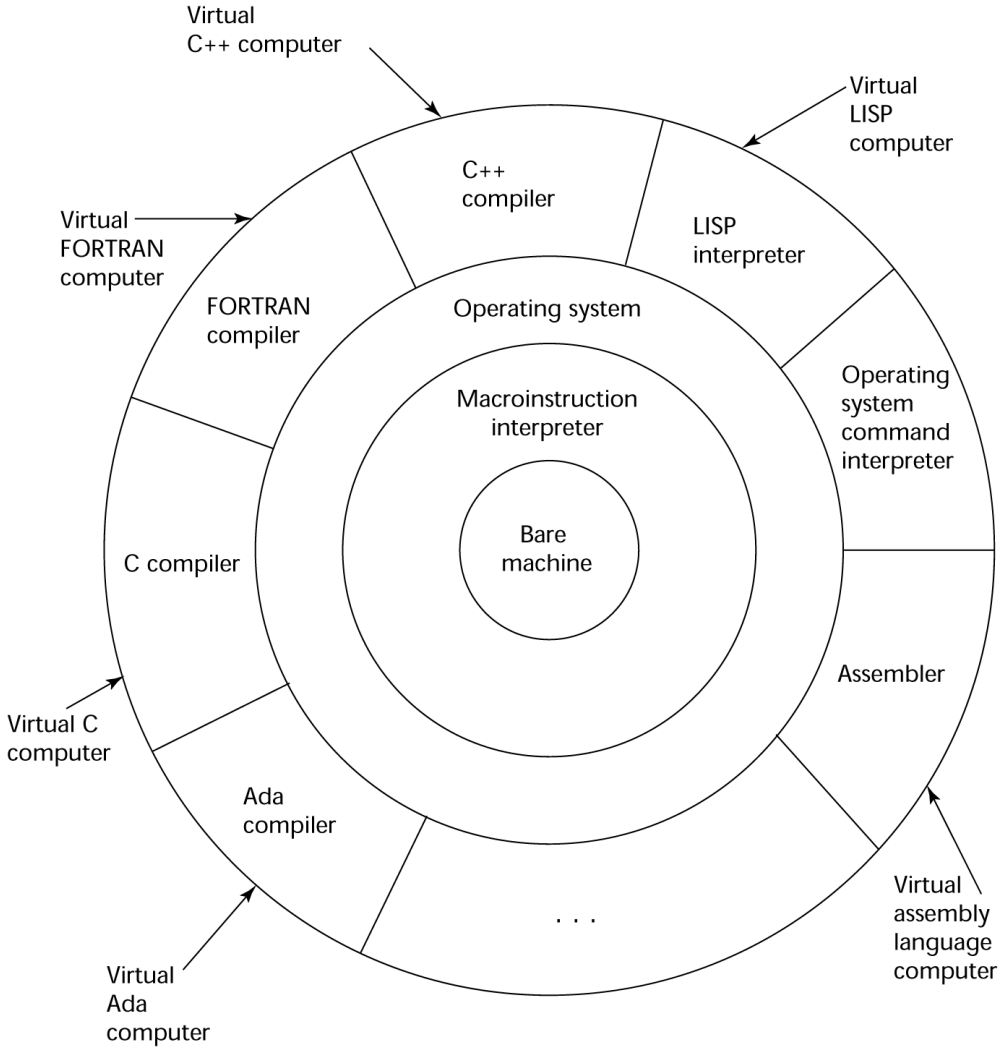
- **Types are mechanisms to define behaviors when variables and constants are used in a program.**
- Types have names (int for integer, float for floating point, etc)
- Constants have types (3 is an int, 3.14159 is a float)
- More complex types are available in many languages. A “class” is just a user defined type.

Implementation Methods

- **Compilation**
  - Programs are translated into machine language; includes JIT systems
  - 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

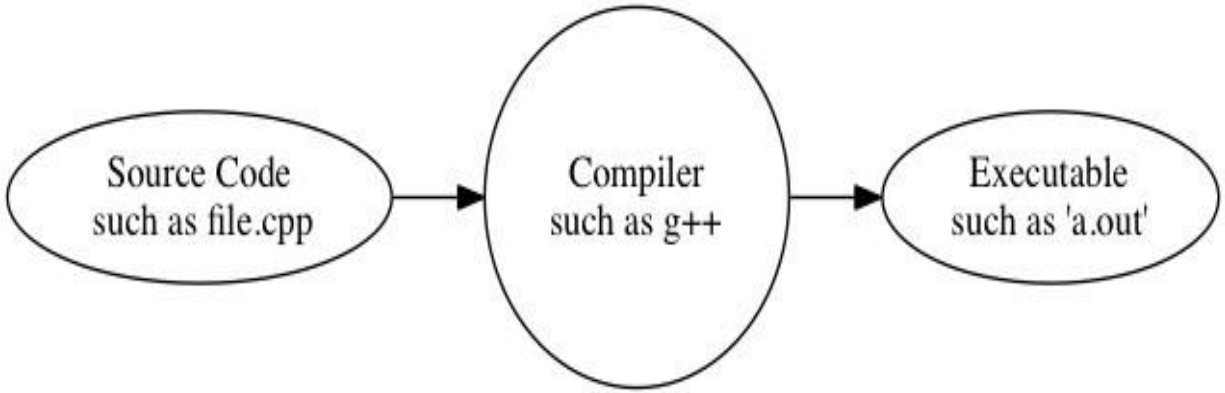
Layered View of Computer

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



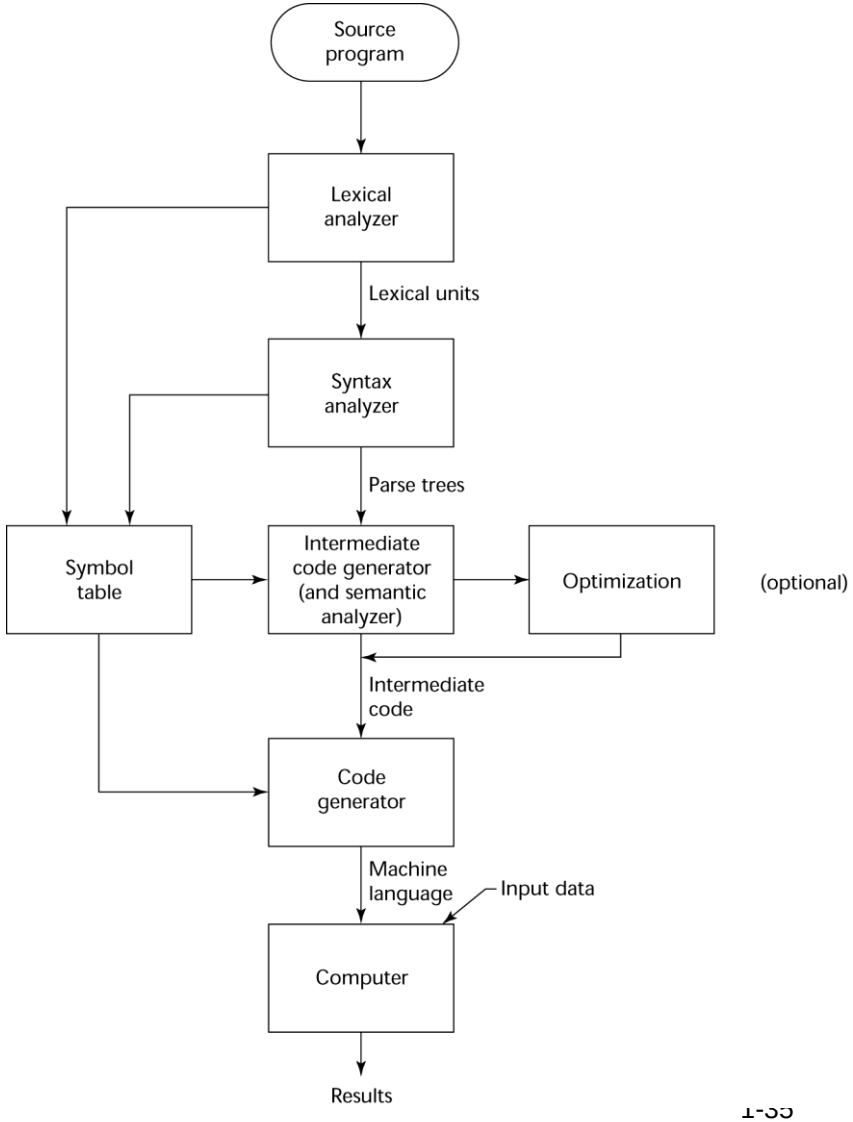
Compilation

- Translate high-level program (source language) into machine code (machine language)
  - An executable file, executable code, executable program, or simply an executable or a binary is a data file that can be executed directly by the hardware over and over again.
  - This is specific to the machine you are compiling for
    - Compiling on one machine to run on another is called “cross-compilation”
  - Slow translation, fast execution



The Compilation Process

- Compilation process has several phases:
  - **Lexical analyzer** converts characters in the source program into lexical units (Tokens)
  - **Syntax analyzer** transforms lexical units into parse trees which represent the syntactic structure of program
  - **Semantics analyzer** enforces the semantic rules of the language
  - **Optimizer** improves the code
  - **Symbol table** keeps a record of all defined variables attributes



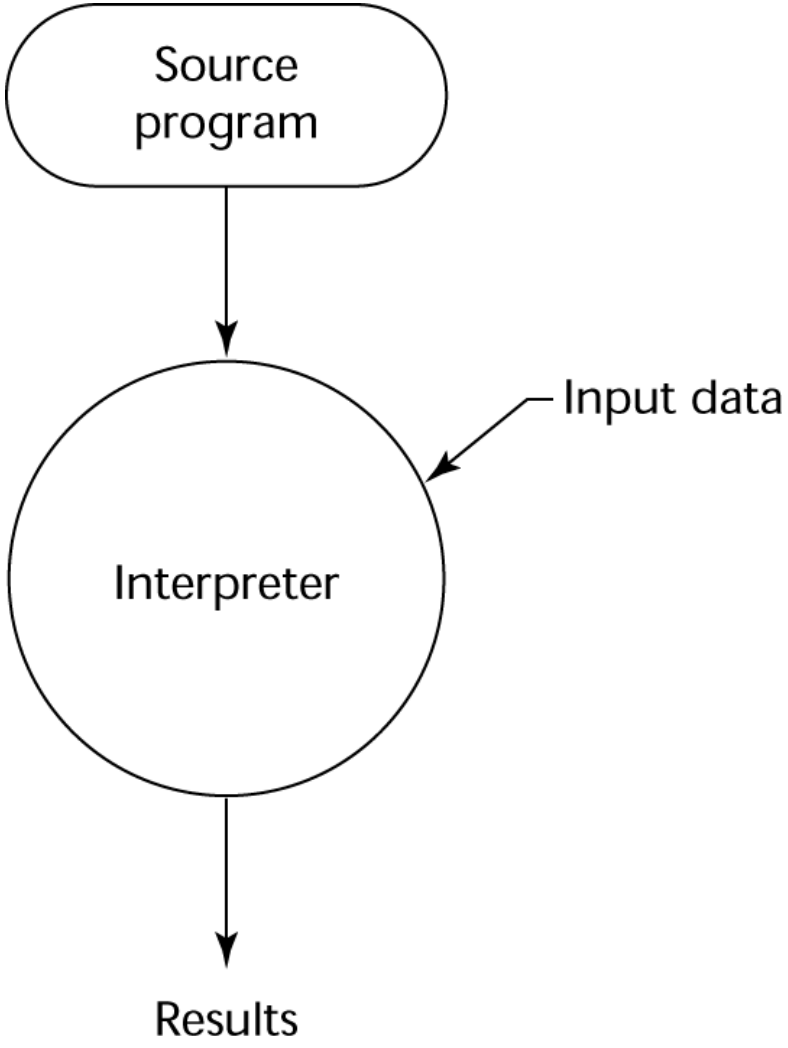
Additional Compilation Terminologies

- **Load module** (executable image): An integrated part of the operating system (which makes it essentially invisible) which loads the content of an executable code file (the user and system code together) into memory.
- **Linking and loading**: the process of collecting system program units and combining them to a user program and probably some library objects files into one executable file.
- **Run-time system**
  - Modern, high-level languages require that a program have additional support during execution. This is sometimes called the run-time system. The run-time system contains lots of code that is not written by the programmer, but was written by others and used when a program in the language is run.

Pure Interpretation

- An interpreter is a program that takes another program as input and executes it, possibly line-by-line, without translating it to an intermediate form.
  - 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
  - Now rare for traditional high-level languages
  - Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

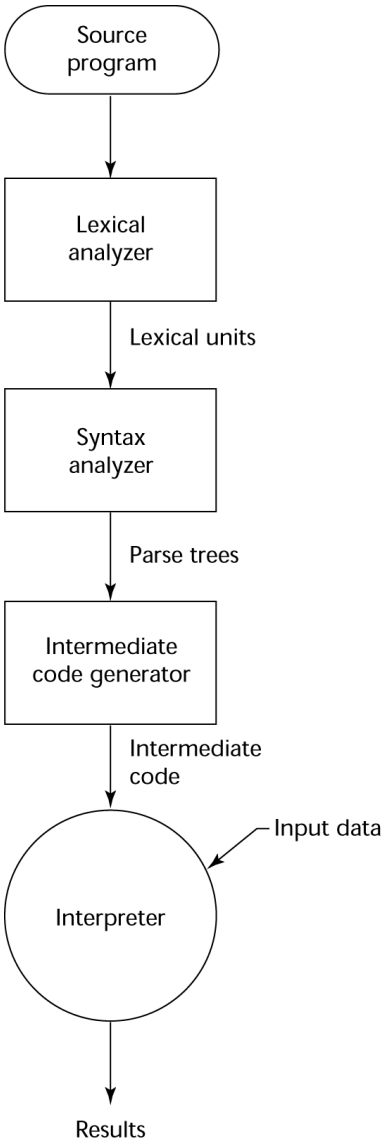
Pure Interpretation Process



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
- 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 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

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