Overview of Programming Languages

History of Programming Languages (Stephanie used a different approach during lecture 1, but doesn’t want to delete these notes about the history of PL because they are so informative)

- **Earliest digital computers without memory** (The imitation Game Clip “Turing’s Machine” 2014 https://www.youtube.com/watch?v=nmXzPgVjxRw)
  - adjust gears, connect cables and flip switches

- **Later computers with punch card reader** (Punch Card from a Fortran program https://en.wikipedia.org/wiki/Computer_programming_in_the_punched_card_era)
  - punch card: card-stock piece of paper with holes punched in it
  - data and instructions are holes
  - card reader reads holes
  - programmers had to know the machine code (different machines have different instructions)
  - not easy to read and write

- **Assembly language**
  - machine-dependent, CISC, RISC, close related to the machine architecture
  - the first abstraction of machine code, using text mnemonics to represent binary instructions and symbols to represent binary sequence
  - punch card reader became interpreters
  - more readable than pure machine code
  - inefficient and error-prone (count lines, no efficient flow control but JMP to lines)

- After 1950s
  - PL were intended to bridge the gap between natural language and the machine instructions
  - **Higher-ordered languages** (our focus in CS333)
    - Independent of any particular machine architecture
    - Closer to natural language
    - **Compilers/interpreters** translate the programs into assembly languages/machine code

- Early PL
  - developed to satisfy particular needs
  - influenced later language development
  - some survived and heavy used, some evolved or replaced by other languages (why)
    - too complex implement, too slow, or not general enough caused the language not survived

  - some early PLs:
**TIMELINE OF PL HISTORY**

- Fortran: scientific computing
  - long term used
  - fast
- Cobol: business computing
  - long term used
  - user friendly
- Algol: general purpose programming
  - short term
- Lisp: AI programming
  - long term
  - functional PL, 63 years old, treat computation as the evaluation of mathematical function, avoid changing status and mutable data
- C: system programming
  - long term
  - explicit memory management
- Prolog: theorem proof
  - long term
- SEQUEL: database management
  - SQL
- **Tools** to Enable and Facilitate the use of PL
  - compilers/interpreters to convert the language into machine language/assembly language
  - languages must be described in an unambiguous manner (grammar)
  - underlying computer architecture supports the features of a language (e.g., precision 32- and 64-bit)

---

**Nature of PLs** (*notes not covered directly by Stephanie*)

- PLs enable communication between programmers and computers
  - PLs describe the tasks in a way both programmers and machines can understand.
- All applications in your laptop are written in certain PLs including OS, Office Word, PPT, Excel, Web browsers, text editors, IDEs (Intergrated development environments, e.g. Visual Studio), etc.
- All these programs share the same hardware resources (CPU, memory, etc).
- Therefore, it’s necessary to be aware of the resource constrains and write efficient programs.
- In this course, I hope to help you understand PLs better so that you can write more efficient programs.

---

**PL Concepts** (*Stephanie presents these in class*)

- **Syntax**: defines the structure of the language
  - Syntax is defined by a vocabulary that specifies the set of possible symbols and a grammar that defines the set of possible valid programs. Note that a valid program is not necessarily a correct or useful program.
  - *Project 2 addresses lexical analysis and syntax.*
  - *Project 3 addresses syntax*

- **Names**
  - Programs requires that we give names to entities in the program.
    - *Project 3 addresses naming*
  - Entities include constants, variables, expressions, functions, libraries, and programs.
  - Names permit to manipulate entities. (So, name is important!)
    - *Scope*: defines the part of the program in which a name refers to a specific entity.
      - It is important to know when the name of an entity is available.
    - *Visibility*:
      - Sometimes we use the same name for different entities.
      - The visibility rules define which entity a particular name usage accesses.

- **Binding**
- The entity referred by a name is not always defined when the programmer writes the code.
- Binding specifies when the connection between entity (variable) and its property (value) is made.

- Types
  - All data in a computer is a sequence of binary values.
  - Project 3 addresses types
  - The data types of a programming language define the abstractions built on top of binary sequences to permit a programmer to generate and manipulate information.
  - Data types can be simple, such as integers or characters, or more abstract such as lists, hash tables and functions.

- Semantics
  - The meaning of a program is defined by its semantics.
  - Generally, semantics are defined in terms of the behavior of a particular architecture, or computing model that is independent of the actual underlying hardware (but not always).
  - Project 4 addresses semantics

- Organization
  - All programming languages contain constructs that permit us to build abstractions. In some languages this is easier than others.
  - Macros, functions, classes, interfaces, and packages examples of organization constructs that can exist in a language.
  - Projects 5 and 6 address organization with functions and polymorphism.

- Memory Management
  - Allocating, freeing, and making use of memory are central to writing programs.
  - In some language memory management is hidden from the programmer, while in others the programmer is responsible for managing its use.
  - The most important concepts in memory management are the system stack - which is generally used for handling local variables and function calls - and the heap - which is generally used for dynamically allocated objects. The system stack is rarely explicitly managed by the programmer, while the heap is often at least partly exposed.
  - Project 7 addresses memory management.

Disclaimer: Notes adapted from previous CS333 lecture materials at Colby College