Overview of Programming Languages (II)

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:
  - 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
- 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 (Integrated 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**

**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.

**Names**

- Programs require that we give names to entities in the program.
- **Entities** include constants, variables, expressions, functions, libraries, and programs.
- Names permit use 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.
- 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).

- Organization
  • All programming languages contain constructs that permit us to build abstractions. In some languages this is easier than others.
  • Marcos, functions, classes, interfaces, and packages examples of organization constructs that can exist in a language.

- 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.

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