CSCI-C311 Programming Languages

Introduction

Dr. Hang Dinh

1

Outline and Reading

- After this lecture, you will learn
  - The Art of Language Design
  - Programming Language Classification
  - Declarative Languages v.s. Imperative Languages
- Reading
  - Scott 4e - Section 1.1
  - Scott 4e - Section 1.2
  - Scott 4e - Section 1.3

2

Why do we have programming languages?

- What is a programming language for?
  - way of thinking -- way of expressing algorithms
  - languages from the programmer's point of view
  - abstraction of virtual machine -- way of specifying what you want the hardware to do without getting down into the bits
  - languages from the implementor's point of view

3

Why are there so many programming languages?

- Evolution -- we've learned better ways of doing things over time
- Socio-economic factors: proprietary interests, commercial advantage, e.g., Microsoft’s C# and Google’s Golang
- Orientation toward special purposes/hardware
  - Lisp for manipulating symbolic data and complex data structures.
  - Prolog for reasoning about logical relationships among data.
  - C for low-level systems programming
  - Java for devices that communicate over a network
- Diverse ideas about what is pleasant to use

4

The Art of Language Design

- What makes a language successful?
  - easy to learn (BASIC, Pascal, Scheme, Java, Python)
  - easy to express things, easy use once fluent, "powerful" (C, Common Lisp, APL, Algol-68, Perl)
  - easy to implement (BASIC, Pascal, Java, Python)
  - possible to compile to very good (fast/small) code (Fortran)
  - backing of a powerful sponsor (PL/1, COBOL, Ada, C#, Visual Basic, Objective-C)
  - associated with open source (C)
- No single factor determines whether a language is "good"

5

Why study programming languages?

- Help you choose a language.
  - C vs. C++ vs. C# for systems programming
  - Fortran vs. C for numerical computations
  - PHP vs. Ruby for web-based applications
  - Ada vs. C for embedded systems
  - Common Lisp vs. Scheme vs. ML for symbolic data manipulation
  - Java vs. .NET for networked PC programs

6

Why study programming languages?

- Make it easier to learn new languages.
- Some languages are similar; easy to walk down family tree
  - Java and C# are easier to learn if you already know C++
- Concepts (iteration, recursion, abstraction,…) have even more similarity
  - If you think in terms of these concepts, you will find it easier to assimilate the syntax and semantic details of a new language.
  - Think of an analogy to human languages: good grasp of grammar makes it easier to pick up new languages (at least Indo-European).

7

Why study programming languages?

- Help you make better use of whatever language you use
  1. understand obscure features of a language
  2. understand implementation costs: choose between alternative ways of doing things, based on knowledge of what will be done underneath
  3. figure out how to do things in languages that don't support them explicitly

8

Language Classification

- Group languages as
  - **imperative**
    - von Neumann (Fortran, Pascal, Basic, C)
    - object-oriented (Smalltalk, Eiffel, C++?)
    - scripting languages (Perl, Python, JavaScript, PHP)
  - **declarative**
    - functional (Scheme, ML, pure Lisp, FP)
    - dataflow (Id, Val)
    - logic, constraint-based (Prolog, VisiCalc, RPG)
- These categories are fuzzy and open to debate

9

Declarative v.s. Imperative

Declarative Languages

- Focus on *what* the computer is to do
- "high-level": more in tune with programmers' point of view, less with the implementor's point of view
- Example with printing an array:
  - the result of the whole operation can be described as mapping a function representing an operation of printing out a value to this array.

Imperative Languages

- Focus on *how* the computer should do it
- predominate mainly for performance reason.
- Example with printing an array:
  - loop over this array of data.
  - For each element, perform this operation of printing the element.

10

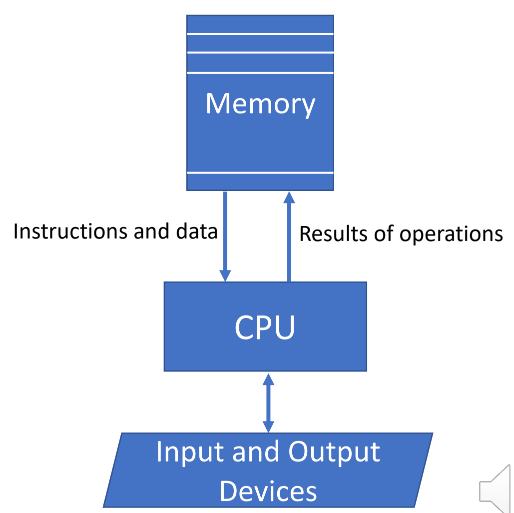
Imperative Subfamily: von Neumann

- The **von Neumann** languages (Fortran, Ada, Pascal, Basic, C,…)
  - Probably the most familiar and widely used
  - Include all languages in which the basic means of computation is the modification of variables.
  - Based on statements (assignment in particular) that influence subsequent computation via the side effect of changing the value of memory.
- Influenced by the well-known computer architecture: von Neumann
  - Developed by mathematician John von Neumann
  - Called *stored program* computing

11

The von Neumann Computer Architecture

- Data and programs stored in memory that is separate from CPU
- Instructions and data are piped from memory to CPU
- CPU repeatedly fetches, interprets, and updates
- Basis for imperative languages
  - Variables model memory cells
  - Assignment statements model piping
  - Iteration is efficient



12

Imperative Subfamily: Object-Oriented (OO)

- Most OO languages are closely related to von Newmann languages
  - But have a much more structured and distributed model of both memory and computation
- Picture computation as interactions among semi-independent objects,
  - Not as the operation of a monolithic processor on a monolithic memory
  - Each object has both its own internal state and subroutine to manage that state
- Examples:
  - **Smalltalk** is the purest of the OO languages.
  - **C++** and **Java** are the most widely used.
  - **Ocaml** is both OO and functional.

13

Imperative Subfamily: Scripting

- Emphasis on coordinating (or "gluing together") components drawn from some surrounding context.
- Several were developed for specific purposes
  - `csh` and `bash` are the input languages of job control (`shell`) programs
  - **PHP** and **JavaScripts** are intended for generating dynamic web content
  - **Lua** is intended for extension/embedded settings (in the gaming industry)
  - **Perl**, **Python**, and **Ruby** are more deliberately general purpose.
- Most emphasize on rapid prototyping, with a bias toward ease of expression over speed of execution.
  - Good for writing programs fast, but not for writing fast programs

14

Declarative Subfamily: Functional

- *Functional* languages employ a computational model based on the recursive definition of functions.
  - Inspired by *lambda calculus*, developed by Alonzo Church in 1930s.
- A program is considered a function from inputs to outputs, defined in terms of simpler functions through a process of refinement.
- Examples:
  - **Lisp**: The original functional language (more on this later)
  - **ML**: Functional language with “Pascal-like” syntax.
  - **Haskell**: The leading purely functional language.

15

Declarative Subfamily: Dataflow

- The *dataflow* languages model computation as the flow of information (*tokens*) among primitive functions (*nodes*).
- Provide an inherently parallel model:
  - Nodes are triggered by the arrival of input tokens
  - Nodes can operate concurrently.
- Examples:
  - **VPL**: Microsoft Visual Programming Language
  - **Val**: Value-oriented Algorithmic Language
  - **Sisal**: a descendant of Val, but more often described as a functional language

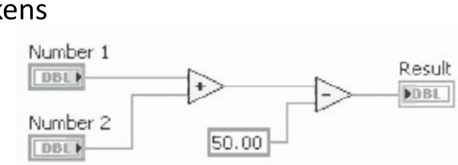


Figure 2.14 Data flow program.

Photo source: https://forum.digilentinc.com/topic/16322-dataflow-programming-and-data-types-in-lab-view/

16

Declarative Subfamily: Logic/Constraint-based

- Inspired from predicate logic
- Model computation as an attempt to find values that satisfy certain specified relationship, using goal-directed search through a list of logical rules.
- Examples:
  - **Prolog**: the best-known logic language
  - **SQL**: database language
  - **XSLT**: scripting language (XSL Translations)
  - Programmable aspects of Excel

17

Lisp and Scheme

- Lisp means **LIS**t **P**rocessor
  - A program in Lisp is itself a list, and can be manipulated with the same mechanisms used to manipulate data.
- Many dialects exists
  - Two most common today are Common Lisp and Scheme.
- **Scheme**: a small, elegant dialect of Lisp
  - Has static scoping and true first-class functions
  - Widely used for teaching.
- We'll focus on **Racket**, a dialect of Lisp and a descendant of Scheme.

18