CSCI312 Principles of Programming Languages

Chapter 1
Overview

Xu Liu
What is a Programming Language (PL)

Notation for

- Describing a computation that is machine-translatable (and human-readable)
- Is capable of expressing any computer program

Why need so many PLs

- Same theoretical power
- Different practical power
- Facilitate or impede certain modes of thought
What We will Learn

• Various paradigms for specifying programs
• How to give (precise) meaning to programs
• How to use programming languages to prevent runtime errors
• Explore these concepts in real-world languages
Why Study PLs?

Help choose a language

- C vs. Modula-3 vs. C++ for system programming
- Fortran vs. APL vs. Ada for numerical computations
- Ada vs. Modula-2 for embedded systems
- Common Lisp vs. Scheme vs. ML for symbolic data manipulations
- Java vs. C/CORBA for networked PC programs
Why Study PLs?

Make It Easier to Learn New Languages

– similar syntax/semantics for languages
  • iteration, recursion, abstraction, function call, …
Why Study PLs?

Help Make Better Use of Whatever Language You Use

- *Understand obscure features*
  - In C, help understand unions, arrays, pointers, separate compilation, varargs, catch and throw
  - In Common Lisp, help understand first-class functions/closures, streams, catch and throw, symbol internals

- *Understand implementation costs*
  - Use simple arithmetic equal (x\*x instead of x**2)
  - Avoid call by value with large data sets
Course Administration

• Instructor: Xu Liu (McGl 117; xliu13@wm.edu)
  – Office hours: 10-11:00am MWF

• TA: Lele Ma (lma03@mail.wm.edu)
  – Office hours: TBD

• Most contents are on
  – http://www.cs.wm.edu/~xl10/cs312

• Assignment submissions and grades are on Blackboard
Textbooks

• Required: None

• Recommended


Course Requirements

• A project-centric course
  – six projects
• A few in-class quizzes
• Midterm and final exams
Grading

• Breakdown
  – *Projects*: 40%
  – *Midterm*: 15%
  – *In-class quizzes*: 10%
  – *Final*: 35%

• 10% late penalty per day

• Final grade is curved
  – 90% guaranteed *A-*.
  – 80% guaranteed *B-*.
  – etc.
Collaboration

• Written assignments
  – Must complete individually

• Programming assignments
  – May complete alone or in pairs
  – If in pairs, must follow “the rules of pair programming”, linked on the course web page
    • can change your partner for each project, but not during one project

• Any cheating will result in an F and referral to honor code violation committee

• Plagiarism-detection software will be used
Contents

1.1 Principles
1.2 Paradigms
1.3 Special Topics
1.4 A Brief History
1.5 On Language Design
   1.5.1 Design Constraints
   1.5.2 Outcomes and Goals
1.6 Compilers and Virtual Machines
Principles of PL

Programming languages have four properties:

- Syntax
- Naming
- Types
- Semantics

For any language:

- Its designers must define these properties
- Its programmers must master these properties
Syntax

The *syntax* of a programming language is a precise description of all its grammatically correct programs.

When studying syntax, we answer questions like:

– *What are the basic statements for the language?*
– *How do I write a ... ?*
– *Why is this a syntax error?*
Naming

Many entities in a program have names:

variables, types, functions, parameters, classes, objects, ...

Named entities are bound in a running program to:

– Scope
– Visibility
– Type
– Lifetime
Types

A *type* is a collection of values and a collection of operations on those values.

- **Simple types**
  - *numbers, characters, booleans, ...*

- **Structured types**
  - *Strings, lists, trees, hash tables, ...*

- A language’s *type system* can help to:
  - *Determine legal operations*
  - *Detect type errors*
Semantics

The meaning of a program is called its semantics. In studying semantics, we answer questions like:

- What does each statement mean?
- What underlying model governs run-time behavior, such as function call?
- How are objects allocated to memory at run-time?
1.2 Paradigms

A programming *paradigm* is a pattern of problem-solving thought that underlies a particular genre of programs and languages.

There are four main programming paradigms:

– *Imperative*
– *Object-oriented*
– *Functional*
– *Logic (declarative)*
Imperative Paradigm

Follows the classic von Neumann-Eckert model:

– *Program and data are indistinguishable in memory*
– *Program = a sequence of commands*
– *State = values of all variables when program runs*
– *Large programs use procedural abstraction*

Example imperative languages:

– *Cobol, Fortran, C, Ada, Perl, ...*
The von Neumann-Eckert Model

Figure 1.1: The von Neumann-Eckert Computer Model
Object-oriented (OO) Paradigm

An OO Program is a collection of objects that interact by passing messages that transform the state.

When studying OO, we learn about:

- Sending Messages
- Inheritance
- Polymorphism

Example OO languages:

Smalltalk, Java, C++, C#, and Ruby
Functional Paradigm

Functional programming models a computation as a collection of mathematical functions.

– Input = domain
– Output = range

Functional languages are characterized by:

– Functional composition
– Recursion

Example functional languages:

– Lisp, Scheme, ML, Haskell, ...
Logic Paradigm

Logic programming declares what outcome the program should accomplish, rather than how it should be accomplished.

When studying logic programming we see:

– *Programs as sets of constraints on a problem*
– *Programs that achieve all possible solutions*
– *Programs that are nondeterministic*

Example logic programming languages:

– *Prolog*
Figure 1.2: A Snapshot of Programming Language History
What makes a successful language?

Key characteristics:

- Simplicity and readability
- Clarity about binding
- Reliability
- Support
- Abstraction
- Orthogonality
- Efficient implementation
Simplicity and Readability

• Small instruction set
  – *E.g.*, *Java vs Scheme*

• Simple syntax
  – *E.g.*, *C/C++/Java vs Python*

• Benefits:
  – *Ease of learning*
  – *Ease of programming*
Clarity about Binding

A language element is bound to a property at the time that property is defined for it.

So a binding is the association between an object and a property of that object

- Examples:
  - a variable and its type
  - a variable and its value

- Early binding takes place at compile-time
- Late binding takes place at run time
Reliability

A language is **reliable** if:

- *Program behavior is the same on different platforms*
  - E.g., early versions of Fortran
- *Type errors are detected*
  - E.g., C vs Haskell
- *Semantic errors are properly trapped*
  - E.g., C vs C++
- *Memory leaks are prevented*
  - E.g., C vs Java
Language Support

- Accessible (public domain) compilers/interpreters
- Good texts and tutorials
- Wide community of users
- Integrated with development environments (IDEs)
Abstraction in Programming

• Data
  – *Programmer-defined types/classes*
  – *Class libraries*

• Procedural
  – *Programmer-defined functions*
  – *Standard function libraries*
Orthogonality

A language is orthogonal if its features are built upon a small, mutually independent set of primitive operations.

- Fewer exceptional rules = conceptual simplicity
  - E.g., restricting types of arguments to a function
- Tradeoffs with efficiency
Efficient implementation

- **Embedded systems**
  - *Real-time responsiveness (e.g., navigation)*
  - *Failures of early Ada implementations*
- **Web applications**
  - *Responsiveness to users (e.g., Google search)*
- **Corporate database applications**
  - *Efficient search and updating*
- **AI applications**
  - *Modeling human behaviors*
1.6 Compilers and Virtual Machines

Compiler – produces machine code

Interpreter – executes instructions on a virtual machine

• Example compiled languages:
  – Fortran, Cobol, C, C++

• Example interpreted languages:
  – Scheme, Haskell, Python

• Hybrid compilation/interpretation
  – The Java Virtual Machine (JVM)
The Compiling Process

Figure 1.4: The Compile-and-Run Process
The Interpreting Process

Figure 1.5: Virtual Machines and Interpreters