CSE 307: Principles of Programming Languages

Spring 2015

R. Sekar

Topics

Section 1

Course Organization

Course information sources

Course web page: http://seclab.cs.sunysb.edu/sekar/cse307/
(Redirected from http://www.cs.sunysb.edu/~cse307/)

- Used for general information about the course
  - Instructor and TA information
  - Office hours
  - Lecture notes

Blackboard: Will be used for announcements, emails, assignment submissions, posting grades, etc.

Course Support

- Check the course web pages and Blackboard announcements.
- Follow the discussion forum for the course on Blackboard.
  - Post your questions relating to homeworks (or any other topic discussed in class) on this forum.
    - But check the course web page and previous forum postings before making a new one, as the question may have already been answered.
    - Do not use email except for questions of personal nature.
- Come to my office hours or that of the TAs.
- Grading related questions: send email to the TA who graded your homework.

Course Objectives

- Develop a fundamental understanding of programming language concepts.
- Acquire tools to choose, use, evaluate and design programming languages.
- Learn different flavors of programming languages.

What will you learn in CSE 307?

- Programming Language Concepts
  - Values, Binding, Scopes, Naming, . . .
- Programming Paradigms
  - Object-oriented, Functional, Logic
- Runtime environments, Interpreters and Compilers

Programming Languages Covered

Imperative: C, [Pascal]

Object-oriented: C++, Javascript, Java

Functional: OCAML, Use of functional style in Python, [Haskell]

Logic: Prolog

Textbooks

Required:

- Kenneth C. Louden, *Programing Languages Principles and Practice*, Second Edition, Thomson Publishers.
- You can buy the 3rd edition if you cannot find the 2nd edition
  - But I find used versions of the second edition on Amazon for $1 or so!

How the course is run

- Approximately one homework every two weeks
  - Non-programming homeworks about programming language concepts.
  - Short programming assignments to learn programming in new languages.
  - Larger (around 500 lines) programming assignments: writing interpreters to solidify the understanding programming language concepts.
- You can skip or drop one assignment in the whole semester
- Grading (approximate)
  - 70% exams: Two midterms (15% to 18% each) plus Final (approx. 35%)
  - 30% homeworks, assignments, quizzes and class participation.
  - To receive a good grade, you must do well *individually* in each component

Academic Integrity

- Do not copy from any one, or any source (on the Internet or elsewhere)
- The penalty for cheating is an F-grade, plus referral to graduate school. *No exception,* regardless of the "amount" of copying involved.
- In addition, if you cheat, you will be unprepared for the exams, and will do poorly.
- To encourage you to work on your own, we scale up assignment scores by about 10% to 25%

Section 2

Introduction

Language

- The words, their pronunciation, and the methods of combining them used and understood by a community.
- A formal system of signs and symbols including rules for the formation and transformation of admissible expressions

— Merriam-Webster Dictionary

- **Programming Language:** (from the textbook)

  A notation for describing computation in machine-readable and human-readable form

Programming Languages

- Low-level languages
  - e.g. assembly languages
  - closer to the level of the machine
- High-level languages
  - e.g. Java
  - closer to the level of the human programmer
- Special-purpose languages
  - e.g. SQL
  - Tailored for use in a particular setting
  - ... in constrast to **general purpose** languages

Characteristics of programming languages

- **Human Readability:**
  - Use of *abstractions*

    ... to move the programmer from the domain of the machine to the domain of the problem being solved
  - Different languages differ in the abstractions they facilitate
- **Machine Readability:**
  - Recognizers (syntax checkers)
  - Compilers and interpreters

Data Abstraction

- Basic Data Abstraction:
  - Common, “atomic” data values such as *integers*;
  - places to store such values (e.g. “*variables*”);
  - notation to indicate the association between places, their names and values.
- Structured Data Abstraction:
  - Group or collection of related data values
  - e.g. arrays, records, etc.
- Unit Data Abstraction:
  - Encapsulating related data values and structures into induvidual program units
  - e.g. modules and packages.

Control Abstraction

- Basic Control Abstraction:
  - Most fundamental of control (e.g. data movement)
  - Statements (e.g. `x = x + 1`)
- Structured Control Abstraction:
  - Combine basic controls into more powerful groups
  - Control *structures* such as `if`, `while`, `case`/`switch` etc.
  - Abstraction of a group (sequence) of actions into a single action (e.g. procedures and methods)
- Unit Control Abstraction:
  - Same as in data abstraction: packages and modules

The study of languages

- **Language Definition**
  - Syntax (structure)
  - Semantics (meaning)
- **Language Features**
  - Control structures
  - Data structures
  - Extensions
- **Language Processing**
  - Translation
  - Interpretation
  - Runtime environment
- **Language Design**
  - Understandability
  - Simplicity and Expressiveness
  - Efficiency
  - Portability
  - Security/Error-checking

Language Paradigms

- **Procedural:** Fortran, Algol, PL-1, Pascal, C, . . .
- **Object-oriented:** Simula67, Smalltalk, Ada, Modula-3, C++, Java, . . .
- **Functional:** LISP, ISWIM, Scheme, FP, ML, Haskell, Gofer, . . .
- **Logic:** Prolog, SetL, CLP, Mercury, . . .
- **Scripting / Domain-Specific:** `sh`, AWK, Perl, Tcl/Tk, Postscript, JavaScript, . . .

The more familiar paradigms

- **Imperative/procedural languages**
  - Programs are written with *Control* as the key element.
  - Languages simply abstract the operational aspects of a machine.
    e.g. Pascal, C, Algol.
- **Object-oriented languages**
  - Programs are written with *Data* (objects) as the key element.
  - Languages provide mechanisms to associate operations with specific data values (methods)
    e.g. C++, Java.

Less familiar paradigms

- **Functional languages**
  - Programs are written with focus on operations (functions) on data values
  - Functions combined using *composition*, evaluated at particular values using *application*.
  - Usually have no notion of *assignments*

```
let rec gcd m n =
  let
     r = m mod n
  in
     if r = 0
        then n
        else gcd n r
```

Even less familiar paradigms

- **Logic languages**
  - Programs weitten with focus on *relationships* between data values.
  - Programs specify "what" must be true about a problem's solution; the programming system takes care of achieving the specifications.

Declarative programming

```
mother(mary, joe).          parent(P, C) :- mother(P, C).
father(sam, joe).           parent(P, C) :- father(P, C).
mother(jane, sam).
father(rob, sam).           ancestor(A, D) :- parent(A, D).
...                         ancestor(A, D) :- parent(A, C),
                                              ancestor(C, D).
```


History of Programming Languages (1)

1940s: Programming by "wiring," machine languages, assembly languages

1950s: FORTRAN, COBOL, LISP, APL

1960s: PL/I, Algol68, SNOBOL, Simula67, BASIC

1970s: Pascal, C, SML, Scheme, Smalltalk

1980s: Ada, Modula 2, Prolog, C++, Eiffel

1990s: Java, Haskell

2000s: Javascript, PHP, Python, ...

History of Programming Languages (2)

FORTRAN: the grandfather of high-level languages

- Emphasis on scientific computing
- Simple data structures (arrays)
- Control structures (goto, do-loops, subroutines)

ALGOL: where most modern language concepts were first developed

- Free-form syntax
- Block-structure
- Type declarations
- Recursion

History of Programming Languages (3)

LISP: List-processing language — Focus on non-numeric (symbolic) computation

- Lists as a “universal” data structure
- Polymorphism
- Automatic memory management
- Mother of modern functional languages
- Descedants include Scheme, Standard ML and Haskell
  - Some of these languages have greatly influenced more recent languages such as Python, Javascript, and Scala

History of Programming Languages (4)

Simula 67:

- Object orientation (classes/instances)
- Precursor of all modern OO-languages
  - Smalltalk
  - C++
  - Java

Prolog:

- Back-tracking
- Unification/logic variables

History of Programming Languages (5)

C: “High-level assembly language”

- Simplicity
- Low-level control that enables OSes to be implemented mostly in C
  - Registers and I/O
  - Memory management
  - Support for interspersing assembly code

Java: A simple, cleaner alternative to C++

Reliability: Robustness/Security built from ground-up

Internet focused: “write once, run every where”

Bundled with runtime libraries providing rich functionality

Draws on some concepts from functional languages

Programming Language Design

The primary purpose of a programming language is to support the construction of reliable (and efficient) software

Language Design Criteria:

- Efficiency was the singular focus in the early days (1940s and 1950s).
- Readability
- Complexity
- Reliability
- Expressiveness
- Maintainability
- Portability

Expressiveness

- Ability to write programs focussing on the problem, not on the machine used to solve it.
- Ability to express complex processes and structures.
- Support for *data* and *control* abstraction
- Expressiveness $\neq$ conciseness!

  e.g. `while (*s++ = *t++);`

Simplicity and Extensibility

- Simplicity: ability to express programs concisely, in a manner that is easy to write, read, and understand.
  - Simplicity of learning vs. simplicity of programming vs. simplicity of understanding
  - Small set of basic concepts
  - Constructs can be expressed and used only in one way.
- Extensibility: ability to add new features to the language
  - Data type definition in Pascal, Modula, Ada, ...
  - Definition of new operators (or reuse existing operators such as '+') in SML, Prolog, Haskell, ...

Regularity

- **Generality:** Operations/constructs available for all closely related cases:
  - in C: Compare two integers with == but not two structures/arrays
  - in Java: Can make collections of objects (e.g. `Integer`) but not primitive values (e.g. `int`)
- **Orthogonality:** Constructs can be combined in a meaningful way:
  - in C: All parameters are passed by value *except arrays*
  - in Java: A class can have `static` members but an abstract class cannot.
- **Uniformity:** Constructs appear and behave consistently:
  - in C, Java: = means "assignment" while == is a comparison.
  - *"Law of least astonishment"*

Efficiency

- Efficiency of executable code
- Efficiency of translation
- Efficiency of programming
  - *Reusabilty*
  - *Reliability*
  - *Security*
  - *Maintainability*

Consistency, Precision and Security

- Use of accepted notations and notions
  - `DO 9 I = 1, 10`
  - `DO 9 I = 1. 10`
- Availability of well-specified standards
  - When is `int` same as `long`? `short`?
  - Are structures byte aligned? or word aligned?
- Constructs to build programs that cannot be subverted
  - Array bounds checks
  - Safety in types

Error Detection and Correction

- Catch programming errors at compile-time
  - Strong type system
  - Memory safety
- Constructs to handle usage errors
  - Exception handling mechanism
- Mechanisms to test and uncover errors
  - Reflection ...