CMSC 330: Organization of Programming Languages

Introduction
Instructor: Chau-Wen Tseng

Course Goal
Learn how programming languages “work”

- Broaden your language horizons
  - Different programming languages
  - Different language features and tradeoffs
- Study how languages are implemented
  - What really happens when I write x.foo(...)?
- Study how languages are described / specified
  - Mathematical formalisms

All Languages Are Equivalent
- A language is Turing complete if it can compute any function computable by a Turing Machine
- Essentially all general-purpose programming languages are Turing complete
  - i.e., any program can be written in any programming language
- Therefore this course is useless?!
  - Learn only 1 programming language, always use it

Why Study Programming Languages?
- To allow you to choose between languages
  - Using the right programming language for a problem may make programming
    - Easier, faster, less error-prone
  - Programming is a human activity
    - Features of a language make it easier or harder to program for a specific application

Why Study Programming Languages?
- To make you better at learning new languages
  - You may need to add code to a legacy system
    - E.g., FORTRAN (1954), COBOL (1959), ...
  - You may need to write code in a new language
    - Your boss says, “From now on, all software will be written in (C++/Java/C#/Python...)”
  - You may think Java is the ultimate language
    - But if you are still programming or managing programmers in 20 years, they probably won’t be programming in Java!
Course Subgoals

- Learn some fundamental CS concepts
  - Regular expressions
  - Context free grammars
  - Automata theory
  - Compilers & parsing
  - Parallelism & synchronization

- Improve programming skills
  - Learn how to learn new programming languages
  - Learn how to program in a new programming style

Calendar / Course Overview

- Tests
  - 3-4 quizzes, 2 midterms, final exam

- Projects
  - Project 1 – Ruby
  - Project 2 – Ruby
  - Project 3 – OCaml
  - Project 4 – OCaml
  - Project 5 – Multithreading (Ruby or Java)

- Programming languages
  - Ruby, OCaml, Java

Rules and Reminders

- Use lecture notes as your text
  - To be supplemented by readings, internet

- Keep ahead of your work
  - Get help as soon as you need it
    - Office hours, CS forum, email

- Don’t disturb other students in class
  - Keep cell phones quiet
  - Use laptops only for school work

Academic Integrity

- All written work (including projects) must be done on your own
  - Do not copy code from other students
  - Do not copy code from the web

- Work together on high-level project questions
  - Do not look at/describe another student’s code
  - If unsure, ask instructor!

- Can work together on practice questions for the exams

Syllabus

- Scripting languages (Ruby)
- Regular expressions and finite automata
- Functional programming (OCaml)
- Context-free grammars
- Formal semantics
- Environments, scoping, and binding
- Object-oriented programming (Java)
- Concurrency
- Advanced topics

Changing Language Goals

- 1950s-60s – Compile programs to execute efficiently
  - Language features based on hardware concepts
    - Integers, reals, goto statements
  - Programmers cheap; machines expensive
    - Keep the machine busy
Changing Language Goals

- Today
  - Language features based on design concepts
    - Encapsulation, records, inheritance, functionality, assertions
  - Processing power and memory very cheap; programmers expensive
    - Ease the programming process
    - To wit: scripting languages are very slow, and yet very popular

Language Attributes to Consider

- Syntax
  - What a program looks like

- Semantics
  - What a program means (mathematically)

- Implementation
  - How a program executes (on a real machine)

Imperative Languages

- Also called procedural or von Neumann
- Building blocks are functions and statements
  - Programs that write to memory are the norm
    
    ```
    int x = 0;
    while (x < y) x = x + 1;
    ```

  - FORTRAN (1954)
  - Pascal (1970)
  - C (1971)

Functional Languages

- Also called applicative languages
- No or few writes to memory
  - Functions are higher-order
    
    ```
    let rec map f = function [] -> []
      | x::l -> (f x)::(map f l)
    ```

  - LISP (1958)
  - ML (1973)
  - Scheme (1975)
  - Haskell (1987)
  - OCaml (1987)

Logical Languages

- Also called rule-based or constraint-based
- Program consists of a set of rules
  - "A -> B" – if B holds, then A holds
    
    ```
    append([], L2, L2).
    append([X|Xs], Ys, [X|Ys]) :- append(Xs, Ys, Ys).
    ```

  - PROLOG (1970)
  - Various expert systems

Object-Oriented Languages

- Programs are built from objects
  - Objects combine functions and data
  - Often have classes and inheritance
  - "Base" may be either imperative or functional
    
    ```
    class C { int x; int getX() {return x;} ... }
    class D extends C { ... }
    ```

  - Smalltalk (1969)
  - C++ (1986)
  - OCaml (1987)
  - Java (1995)
Scripting Languages

- Rapid prototyping languages for “little” tasks
  - Typically with rich text processing abilities
  - Generally very easy to use
  - “Base” may be imperative or functional; may be OO

```bash
#!/usr/bin/perl
for ($j = 0; $j < 2*lc; $j++) {
    $a = int(rand($lc));
}
```

- sh (1971)
- perl (1987)
- Python (1991)
- Ruby (1993)

“Other” Languages

- There are lots of other languages w/ various features
  - COBOL (1959) – Business applications
    > Imperative, rich file structure
  - BASIC (1964) – MS Visual Basic widely used
    > Originally an extremely simple language
  - Logo (1968) – Introduction to programming
  - Forth (1969) – Mac Open Firmware
    > Extremely simple stack-based language for PDP-8
  - Ada (1979) – The DoD language
    > Real-time
  - Postscript (1982) – Printers- Based on Forth

Ruby

- An imperative, object-oriented scripting language
  - Created in 1993 by Yukihiro Matsumoto
  - Core of Ruby on Rails web programming framework (the key to its popularity)
  - Similar in flavor to many other scripting languages (e.g., perl, python)
  - Much cleaner than perl
  - Full object-orientation (even primitives are objects!)

```
def greet(s)
  print("Hello, ");
  print(s);
  print("\n")
end
```

```
Hello, world!
=> nil
```

OCaml

- A mostly-functional language
  - Has objects, but won’t discuss (much)
  - Developed in 1987 at INRIA in France
  - Dialect of ML (1973)
  - Natural support for pattern matching
    - Generalizes switch/if-then-else – very elegant
  - Has full featured module system
    - Much richer than interfaces in Java or headers in C
  - Includes type inference
    - Ensures compile-time type safety, no annotations

```
let greet =
    begin
        print_string "Hello, ";
        print_string s;
        print_string "\n"
    end
```

```

A Small OCaml Example

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
Attributes of a Good Language

1. Clarity, simplicity, and unity
   - Provides both a framework for thinking about algorithms and a means of expressing those algorithms
2. Orthogonality
   - Every combination of features is meaningful
   - Features work independently
3. Naturalness for the application
   - Program structure reflects the logical structure of algorithm
4. Support for abstraction
   - Program data reflects problem being solved
5. Ease of program verification
   - Verifying that program correctly performs its required function
6. Programming environment
   - External support for the language
7. Portability of programs
   - Can develop programs on one computer system and run it on a different computer system
8. Cost of use
   - Program execution (run time), program translation, program creation, and program maintenance
9. Security & safety
   - Should be very hard to write unsafe program

Attributes of a Good Language

Executing Languages

- Suppose we have a program P written in a high-level language (i.e., not machine code)
- There are two main ways to run P
  1. Compilation
  2. Interpretation

Compilation or Translation

- Source program translated to another language
  - Traditionally: machine code, which can be directly executed

Interpretation

- Interpreter executes each instruction in source program one step at a time
  - No separate executable
Translation phases

- Both compilers and interpreters translate textual source code into an easy-to-work-with format
  - Traditionally involved multiple steps: “lexing” and “parsing”
- The parsed format may undergo several transformations before the final result is produced
  - In a sense, a compiler simply stops before the phase where it could execute the program, while an interpreter “goes all the way”

Compiler or Interprete?

- gcc
  - Compiler – C code translated to object code, executed directly on hardware (as a separate step)
- javac
  - Compiler – Java source code translated to Java byte code
- DOS/sh/csh/tcsh/bash
  - Interpreter – commands executed by shell program
- java
  - Interpreter – Java byte code executed by virtual machine

Decision Less Simple Today

- Previously
  - Build program to use hardware efficiently
  - Often use of machine language for efficiency
- Today
  - No longer write directly in machine language
  - Use of layers of software
  - Concept of virtual machines
    - Each layer is a machine that provides functions for the next layer (e.g., javac/java distinction)
    - This is an example of abstraction, a basic building block in computer science

Formal (Mathematical) Semantics

- What do my programs mean?
  - Let rec fact n = if n = 0 then 1
    - else n * (fact n-1)
  - Let fact n = let rec aux i j = if i = 0 then j
    - else aux (i-1) (j*i) in aux n 1

- Both OCaml functions implement “the factorial function.” How do I know this? Can I prove it?
  - Key ingredient: need a mathematical way of specifying what programs do, i.e., their semantics
  - Doing so depends on the semantics of the language

Semantic styles

- A formal semantics is basically a mathematical implementation. Two flavors
  - Denotational semantics (compiler)
    - Meaning defined in terms of another language
    - If we know what C means, then we can define Ruby by translation to C
  - Operational semantics (interpreter)
    - Meaning defined as rules that simulate program execution
    - Show what Ruby programs do directly, using an “abstract” machine, more high-level than real hardware
- Contrast with textual language definitions, which are incomplete and ambiguous

Summary

- Many types of programming languages
  - Imperative, functional, logical, OO, scripting
- Many programming language attributes
  - Clear, orthogonal, natural…
- Programming language implementation
  - Compiled, interpreted
- Programming language semantics
  - Knowing your program is right