CMSC 330: Organization of Programming Languages

Introduction

Instructors: Rance Cleaveland, Jeff Foster
Course Goal

Learn how programming languages work

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

• Programming is a human activity
  ➢ Features of a language make it easier or harder to program for a specific application

• Using the right programming language for a problem may make programming
  ➢ Easier, faster, less error-prone
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!
Why Study Programming Languages?

- To make you better at using languages you already know
  - Many “design patterns” in Java are functional programming techniques
  - Understanding what a language is good for will help you know when it is appropriate to use
  - The deeper your understanding of a language, the better you will be at using it appropriately
Course Subgoals

- Learn some fundamental programming-language 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)

- **Meet your professor!**
  - 1% of your grade determined by coming to chat with your professor during office hours or at a mutually agreed-upon time
  - Conversation need not be long, or technical … but we would like to get to know you!
Project Grading

- Projects will be graded using the CS submit server
- You may develop your programs on your own machine, but it is your responsibility to ensure that they run correctly on the linuxlab cluster (linuxlab.cs.umd.edu)!
- Software versions
  - Ruby 1.8.6
  - Ocaml TBA
Rules and Reminders

- Use lecture notes as your text
  - To be supplemented by readings, internet
  - You will be responsible for everything in the notes, even if it is not necessarily covered in class!

- 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
  - 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 procedures 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)
Logic-Programming Languages

- Also called rule-based or constraint-based
- Program consists of a set of rules
  - “A :- B” – If B holds, then A holds
  - $\text{append}([], L2, L2)$.
  - $\text{append}([X|Xs], Ys, [X|Zs]) :- \text{append}(Xs, Ys, Zs)$.
- 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
  - Designed with ease of use in mind
  - “Base” may be imperative or functional; may be OO

```perl
#!/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
  ➢ Now a single word oxymoron
• 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!)
A Small Ruby Example

```
intro.rb:
def greet(s)
  print("Hello, ")
  print(s)
  print("!\n")
end
```

```
% irb     # you'll usually use "ruby" instead
irb(main):001:0> require "intro.rb"
=> true
irb(main):002:0> greet("world")
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
A Small OCaml Example

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

```
$ ocaml
    Objective Caml version 3.08.3

# #use "intro.ml";;
val greet : string -> unit = <fun>
# greet "world";;
Hello, world!
- : unit = ()
```
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

5. Naturalness for the application
   - Program structure reflects the logical structure of algorithm
Attributes of a Good Language

4. Support for abstraction
   • Program data reflects problem being solved

5. Ease of program verification
   • Verifying that program correctly performs its required function

8. Programming environment
   • External support for the language
Attributes of a Good 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
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
Source program translated ("compiled") 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**

```python
def greet(s):
    print("Hello, ")
    print(s)
    print("!\n")
end
```

```
"world"
```

```
"Hello, world!"
```
Architecture of Compilers, Interpreters
Front Ends and Back Ends

- Front ends handle syntactic analysis
  - Parser converts source code into intermediate format (“parse tree”) reflecting program structure
  - Static analyzer checks parse tree for errors (e.g. types), may also modify it
  - What goes into static analyzer is language-dependent!

- Back ends handle “semantics”
  - Compiler: back end (“code generator”) translates intermediate representation into “object language”
  - Interpreter: back end executes intermediate representation directly
Compiler or Interpreter?

- 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
Compilers vs. Interpreters

- **Compilers**
  - Generated code more efficient
  - “Heavy”

- **Interpreters**
  - Great for debugging
  - Slow

- **In practice**
  - General-purpose programming languages (e.g. C, Java) are compiled, although debuggers provide interpreter support
  - Scripting languages and other special-purpose languages are interpreted
Formal (Mathematical) Semantics

- What do my programs mean?

- 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