CMSC 330: Organization of Programming Languages

Introduction

Instructors: Mike Hicks, Dave Levin
Course Goal

Learn how programming languages work

- Broaden your language horizons
  - Different programming languages
  - Different language features and tradeoffs
    - Useful programming patterns

- 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 help 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
  - A language not only allows you to express an idea, it also shapes how you think when conceiving it
    - There are some fundamental computational paradigms underlying language designs that take getting used to
  - You may need to learn a new (or old) language
    - Paradigms and fads change quickly in CS
    - Also, may need to support or extend legacy systems
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
  - Automata theory
  - Context free grammars
  - Parallelism & synchronization

- Improve programming skills
  - Practice learning new programming languages
  - Learn how to program in a new style
Syllabus

- Scripting languages (Ruby)
- Regular expressions and finite automata
- Context-free grammars
- Functional programming (OCaml)
- Formal semantics
- Concurrency
- Logic programming (Datalog)
- Environments, scoping, and binding
- Comparing language styles; other topics
Calendar / Course Overview

Tests
- 5 quizzes, 2 midterms, final exam

Projects
- Project 1 – Ruby
- Project 2 – Ruby
- Project 3 – OCaml
- Project 4 – OCaml / Multithreading
- Project 5 – Datalog

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 3.12.1
Rules and Reminders

► Use lecture notes as your text
  • Supplement with readings, Internet
  • You will be responsible for everything in the notes, even if it is directly covered in class!

► Keep ahead of your work
  • Get help as soon as you need it
    ➢ Office hours, Piazza (email as a last resort)

► 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
  - We’re using Moss; cheaters will be caught
- Work together on high-level project questions
  - Do not look at/describe another student’s code
  - If unsure, ask an instructor!
- Work together on practice exam questions
Changing Language Goals

1950s-60s – Compile programs to execute efficiently

- Language features based on hardware concepts
  - Integers, reals, goto statements
- Programmers cheap; machines expensive
  - Computation was the primary constrained resource
  - Programs had to be efficient because machines weren’t
    - Note: this still happens today, just not as pervasively
Changing Language Goals

Today

• Language features based on design concepts
  ➢ Encapsulation, records, inheritance, functionality, assertions

• Processing power and memory very cheap; programmers expensive
  ➢ Scripting languages are slow(er), but run on fast machines
  ➢ They’ve become very popular because they ease the programming process

• The constrained resource changes frequently
  ➢ Communication, effort, power, privacy, …
  ➢ Future systems and developers will have to be nimble
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
    ```ocaml
    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 (“B implies A”)
    - `append([], L2, L2).`
    - `append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).`
  - PROLOG (1970)
  - Datalog (1977)
  - Various expert systems
Object-Oriented Languages

- Programs are built from objects
  - Objects combine functions and data
    - Often into “classes” which can inherit
  - “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)
  - Ruby (1993)
  - Java (1995)
Concurrent/parallel languages

- Traditional languages had one thread of control
  - Processor executes one instruction at a time
- Newer languages support many threads
  - Thread execution conceptually independent
  - Means to create and communicate among threads
- Concurrency may help/harm
  - Readability, performance, expressiveness
- Many examples
  - Erlang, Cilk, Java, Conc. Haskell, Fortress, UPC
  - C/C++, Ruby, OCaml, Python, …
Scripting Languages

- Rapid prototyping languages for common tasks
  - Traditionally: text processing and system interaction
- “Scripting” is a broad genre of languages
  - “Base” may be imperative, functional, OO…
- Increasing use due to higher-layer abstractions
  - Not just for text processing anymore

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

```ruby
#!/usr/bin/ruby
while line = gets do
  csvs = line.split /,/
  if(csvs[0] == "330") then
    ...
```
Other Languages

There are lots of other languages with various features:

- **COBOL (1959)** – Business applications
  - Imperative, rich file structure

- **BASIC (1964)** – MS Visual Basic
  - Originally designed for simplicity (as the name implies)
  - Now it is object-oriented and event-driven, widely used for UIs

- **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 (Matz)
  - “Ruby is designed to make programmers happy”
  - Core of Ruby on Rails web programming framework (a key to its popularity)
  - Similar in flavor to many other scripting languages
  - Much cleaner than perl
  - Full object-orientation (even primitives are objects!)
A Small Ruby Example

intro.rb:

```ruby
def greet(s)
  3.times { print "Hello, " }
  print "#{s}!\n"
end
```

% irb       # you'll usually use "ruby" instead
irb(main):001:0> require "intro.rb"
=> true
irb(main):002:0> greet("world")
Hello, Hello, 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

intro.ml:

```ocaml
let greet s =
    List.iter (fun x -> print_string s)
    ["hello"; s; "!\n"]
```

$ ocaml

```
Objective Caml version 3.12.1

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

---

CMSC 330 - Spring 2013
Attributes of a Good Language

• Cost of use
  • Program execution (run time), program translation, program creation, and program maintenance

• Portability of programs
  • Develop on one computer system, run on another

• Programming environment
  • External support for the language
  • Libraries, documentation, community, IDEs, …
Attributes of a Good Language

• Clarity, simplicity, and unity
  • Provides both a framework for thinking about algorithms and a means of expressing those algorithms

• Orthogonality
  • Every combination of features is meaningful
  • Features work independently

• Naturalness for the application
  • Program structure reflects the logical structure of algorithm
Attributes of a Good Language

• Support for abstraction
  • Hide details where you don’t need them
  • Program data reflects the problem you’re solving

• Security & safety
  • Should be very difficult to write unsafe programs

• Ease of program verification
  • Does a program correctly perform its required function?
Program Execution

- 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

- Source program translated ("compiled") to another language
  - Traditionally: directly executable machine code
  - Generating code from a higher level "interface" is also common (e.g., JSON, RPC IDL)
Interpreter executes each instruction in source program one step at a time

- No separate executable
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

- java
  - Interpreter – Java byte code executed by virtual machine

- sh/csh/tcsh/bash
  - Interpreter – commands executed by shell program
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 often compiled, although debuggers provide interpreter support
  - Scripting languages and other special-purpose languages are interpreted, even if general purpose
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: a mathematical way of specifying what programs do, i.e., their semantics
  - Doing so depends on the semantics of the language
Semantic styles

- Textual language definitions are often *incomplete* and *ambiguous*

- *A formal* semantics is basically a mathematical definition of what programs do. Two flavors:
  
  - **Denotational semantics (compiler/translator)**
    - Meaning defined in terms of another language (incl. math)
    - 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
Summary

- Many types of programming languages
  - Imperative, functional, logical, OO, scripting, ...
- Many programming language attributes
  - Clear, natural, low cost, verifiable, ...
- Programming language implementation
  - Compiled, interpreted
- Programming language semantics
  - Proving your program operates correctly