CMSC 330: Organization of Programming Languages

http://www.cs.umd.edu/~atif/Teaching/Fall2007

Introduction

Instructor: Atif M Memon

TAs: Guilherme Fonseca,
Michael Lam, Xun Yuan

Calendar / Course Overview

- Final Exam (25%)
- 2 midterms (30%)
- One project every few weeks (5 total) (40%)
  - Project 1 - Write a web server log analysis tool.
    - Will be posted next week.
  - Project 2 - Write a unit testing framework in Ruby.
  - Project 3 - Write some OCaml code.
  - Project 4 - NFAs and reg exps in OCaml.
  - Project 5 - Write a threaded bank simulation.
- Two homework assignments (one before each exam) (5%)

- Ruby
- OCaml
- Java

Academic Integrity

- All written work (including projects) must be done on your own
- Work together on practice questions for the exams
- Work together on high-level project questions
  - Never see another student's code
  - If unsure, ask instructor!

What if…?

Rules and Reminders

- Quiet cell phones
- Be on time
- Come to class and discussion section
- Laptops in class only if really needed
- Use lecture notes as the book
- Stay organized and ahead of your work
- Get help as soon as you need it (but not when you don't)
  - Office hours
    - http://www.cs.umd.edu/~atif/Teaching/Fall2007/office-hours.shtml
- Use internet resources

Syllabus

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

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
  - 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
- Therefore this course is useless!

Why Study Programming Languages?

Introduce yourself to your neighbor(s) and together write down three of your own reasons…

Why Study Programming Languages?

- Using the right language for a problem may be easier, faster, and less error-prone
  - Programming is a human activity
  - Features of a language make it easier or harder to program for a specific application
- 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 {Ada/C++/Java/…}"
- 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 think 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

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
- Today:
  - Language features based on design concepts
    - Encapsulation, records, inheritance, functionality, assertions
  - Processing power and memory very cheap; programmers expensive
    - Ease the programming process

Language Attributes to Consider

- Syntax -- What a program looks like
- Semantics -- What a program means
- Implementation -- How a program executes

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|Zs]) :- 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
  - Generally very easy to use
  - "Base" may be imperative or functional; may be OO

```
#!/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 around with 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
    - Realtime
  - Postscript (1982) – Printers- Based on Forth
  - …

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

What if, instead of working independently, adjusting the volume on your radio also changed the station? You would have to carefully change both simultaneously and it would become difficult to find the right station and keep it at the right volume. Your radio and tuning work orthogonally. And aren't you glad they do!

Attributes of a Good Language

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

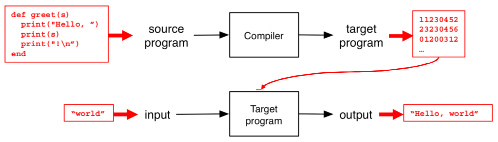
Attributes of a Good Language

6. Programming environment
   - External support for the language
7. Portability of programs
   - Transportability of the resulting programs from the computer on which they are developed to other computer systems
8. Cost of use
   - Program execution, program translation, program creation, and program maintenance

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



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

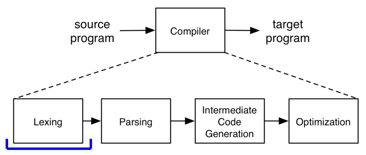
```
11230452
23230456
01200312
…
```

```
"world"
```

```
"Hello, world"
```

- Source program translated to another language
  - Often machine code, which can be directly executed
  - Advantages? Disadvantages?

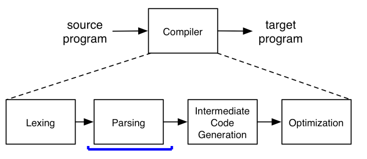
Steps of Compilation



FOR
ID:i
LESSTHAN
NUM:6

1. Lexical Analysis (Scanning) – Break up source code into *tokens* such as numbers, identifiers, keywords, and operators

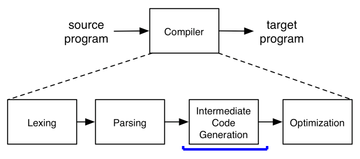
Steps of Compilation



For loop

2. Parsing (Syntax Analysis) – Group tokens together into higher-level language constructs (conditionals, assignment statements, functions, …)

Steps of Compilation



for 0:i:6

or

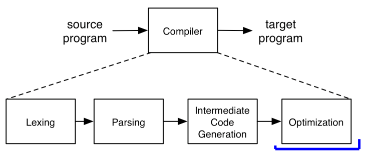
Load 0

Load i

Etc…

3. Intermediate Code Generation – Verify that the source program is valid and translate it into an internal representation
   – May have more than one intermediate rep

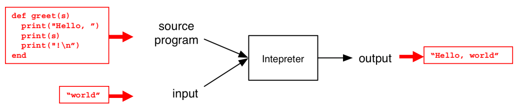
Steps of Compilation



4. Optimization (optional) – Improve the efficiency of the generated code
   – Eliminate dead code, redundant code, etc.
   – Change algorithm without changing functionality (e.g., X=Y+Y+Y+Y ➔ X=4*Y ➔ X = Y shift left 2)

[If interested in compilation, take CMSC 430]

Interpretation



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

```
"world"
```

```
"Hello, world"
```

- Interpreter executes each instruction in source program one step at a time
  – No separate executable
  – Advantages? Disadvantages?

Compiler or Intepreter?

gcc
- Compiler – C code translated to object code, executed directly on hardware

javac
- Compiler – Java source code translated to Java byte code

tcsh/bash
- Interpreter – commands executed by shell program

java
- Interpreter – Java byte code executed by virtual machine

Compilation or Interpretation – Not so 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

Who defines a language?

Is: I = 1 && 2 + 3 | 4; legal in C?

- What is assigned to I if it is?
- Who makes this determination?

3 ways typically to answer this:

1. Read language manual (Problem: Can you find one?)
2. Read language standard (Problem: Have you ever seen it?)
3. Write a program to see what happens. (Easy to do!)

Most programmers do 3, but current compilers may not give correct answer

Creation of standards

Language standards defined by national standards bodies:

- ISO - International Standards organization
- IEEE - Institute of Electrical and Electronics Engineers
- ANSI - American National Standards Institute

All work in a similar way:

1. Working group of volunteers set up to define standard
2. Agree on features for new standard
3. Vote on standard
4. If approved by working group, submitted to parent organization for approval

Creation of standards

- Standards in the US are voluntary:
  - There is no federal standards-making organization.
  - NIST - National Institute for Standards and Technology develops standards that are only required on federal agencies, not for commercial organizations.
- Consensus is the key to standards making:
  - Contentious features often omitted to gain consensus
  - Only vendors have a vested interest in the results
  - Users don't care until standard approved, and then it is too late!

Standards conforming programs

- Standards define behavior for a *standards conforming* program - one that meets the rules of the language standard
- In general (except for Ada), behavior of non-conforming program is not specified, so any extensions to a standards conforming compiler may still be standards conforming, even though the program is not standards conforming.
- Standards supposed to be reviewed every 5 years
  - Examples:
  - FORTRAN 1966, 1977, 1990
  - Ada 1983, 1995
- Not quite 5 years, but at least periodically

When to standardize a language?

- Problem: When to standardize a language?
  - If too late - many incompatible versions
    - FORTRAN in 1960s was already a de facto standard, but no two were the same
    - LISP in 1994, about 35 years after developed.
  - If too early - no experience with language - Ada in 1983 had no running compilers
  - Just right - Probably Pascal in 1983, although it is now mostly a dead language
- Other languages:
  - C in 1988
  - De facto standards: ML, SML, OCaml, Ruby
  - Smalltalk - none
  - Prolog - none

Internationalization

- Programming has become international
  - I18N issue - Internationalization - How to specify languages useful in a global economy? *internationalization* vs. *internationalisation*
- Character sets:
  - 1950s1960s – 6 bit sufficient (upper case, digits, special symbols …)
  - ASCII is a 7 bit 128 character code
  - Single 8-bit byte; usual format today - 256 character values. A lot in 1963, but insufficient today
- What about other languages?
  - Additional letters: German umlaut-ä, French accent-é, Scandanavian symbols-ö,
  - Russian, other alphabets (Greek, Arabic, Hebrew), ideographs (Chinese, Korean)?
  - Unicode - 16 bit code allows for 65K symbols. 8-bit byte is insufficient

Internationalization

- Some of the internationali*ation issues:
  - What character codes to use?
  - Collating sequences? - How do you alphabetize various languages?
  - Dates? – If I said your exam was on 10/12/07 when would you show up?
- Time? - How do you handle time zones, summer time in Europe, daylight savings time in US, Southern hemisphere is 6 months out of phase with northern hemisphere, Date to change from summer to standard time is not consistent. Some zones 30 minutes off.
- Currency? - How to handle dollars, pounds, euros, etc.

Summary

- Language design today must:
  - Allow program solution to match physical structure of problem
  - Allow for world-wide use
  - Be easy to prove solution correct

Ruby

- An imperative, object-oriented scripting language
  - Created in 1993 by Yukihiro Matsumoto
  - 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
  - Makes writing certain programs very elegant
- Has a really nice module system
  - Much richer than interfaces in Java or headers in C
- Includes type inference
  - Types checked at compile time, but no annotations

A Small OCaml Example

intro.ml:

```
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 = ()
```