CSE505: Graduate Programming Languages

Lecture 1 — Course Introduction

Dan Grossman
Winter 2012

Today

- Administrative stuff
- Course motivation and goals
  - A Java example
- Course overview
- Course pitfalls
- Start Caml tutorial (see separate notes)
  - Advice: start playing with it soon (e.g., hw1, problem 1)

Course facts

- Dan Grossman, CSE574, djg at cs
- Adrian Sampson, CSE352, asampson at cs
- Office hours: posted on web page and "negotiable"
  - Or by appointment or just stop by my office if I'm around
- Web page for:
  - Mailing list
  - "Homework 0"
  - Homework 1, fairly carefully pipelined with first lectures
    - Do not wait to do it all

Coursework

- 5 homework assignments
  - "Paper/pencil" (LaTeX recommended?)
  - Programming (Caml required)
  - Where you'll probably learn the most
  - Do challenge problems if you *want* but not technically "extra"
- 1 "introduction/summary" to a published research paper
  - More details later; high work/length ratio
- 2 exams
  - My reference sheet plus your reference sheet; samples provided
- Optional textbook: Types and Programming Languages by Pierce

Exam Scheduling

- Midterm: In class on February 7
- Final:
  - University-proposed time of Wednesday March 14, 10:30-12:20 is horrible for the CSE PhD students (second day of visit day)
  - So I'd like to have the final exam on Monday March 12 instead
  - Expect poll to choose a time
  - Will be confirmed and "set in stone" by next week
  - Willing to make special accommodations if Monday is unworkable for someone

Academic integrity

- Don't cheat in my class
  - I'll be personally offended
  - Being honest is far more important than your grade
- Rough guidelines
  - Can sketch idea together
  - Cannot look at code solutions
- Ask questions and always describe what you did
- Please *do* work together and learn from each other...

Graduate-School Success

- Success in 505 (a graduate course) comes from:
  - Learning and enjoying the material
  - Challenging yourself
  - Managing the "big picture" and the details
- Success has nothing to do with:
  - Scrounging for grading points
  - "Doing better than the person next to you"
- *The person next to you is your colleague for the next 5–50 years*

Logistical Advice

- Take notes:
  - Slides/proofs posted, but they are enough to teach from not to learn from
  - Will often work through examples by hand
- Arrive on time:
  - Missing the first N minutes is so much less efficient than missing the last N minutes
  - I *know* you can get here on time (cf. exam days)

Programming-language concepts

Focus on *semantic* concepts:

What do programs mean (do/compute/produce/represent)?

How to define a language *precisely*?

English is a poor *metalanguage*

Aspects of meaning:

equivalence, termination, determinism, type, ...

This course does *not* gives superficial exposure to N weird PLs

- But it will help you learn new languages via foundations
- And build rigorous models for any area of CS research

Does it matter?

Novices write programs that "work as expected," so why be rigorous/precise/pedantic?

- The world runs on software
  - Web-servers and nuclear reactors don't "seem to work"
- You buy language implementations—what do they do?
- Software is buggy—semantics assigns blame
- Real languages have many features: building them from well-understood foundations is good engineering
- Never say "nobody would write that" (surprising interactions)

Is this Really about PL?

Building a precise model is a hallmark of quality research

The value of a model is in its:

- Fidelity
- Convenience for establishing (proving) properties
- Revealing alternatives and design decisions
- Ability to communicate ideas concisely

Why we mostly do it for programming languages:

- Elegant things we all use
- Remarkably complicated (need rigor)

I believe this "theory" makes you a better computer scientist

- Focus on the model-building, not just the PL features

APIs

Like almost anything in computing, we can describe the course in terms of designing an API

Many APIs have 1000s of functions with simple inputs

- Kernel calls take a struct or two and return an int

A typical language implementation more or less has just

- $\mathit{typecheck} : \mathit{program} \rightarrow \mathit{bool}$
- $\mathit{compile} : \mathit{program} \rightarrow (\mathit{string} \rightarrow \mathit{value})$

But defining $\mathit{program}$ and these functions is subtle, hard

- Conversely, "a data structure is just a really dumb PL"
- Every extensible system ends up defining a PL (game engines, editors, web browsers, CAD tools, ...)

Java example

```
class A { int f() { return 0; } }
class B {
   int g(A x) {
     try { return x.f(); }
     finally { s }
   }
}
```

For all s, is it equivalent for g's body to be "`return 0;`"?
Motivation: code optimizer, code maintainer, ...

Punch-line

Not equivalent:

- Extend `A`
- `x` could be `null`
- s could modify global state, *diverge*, throw, ...
- s could return

A silly example, but:

- PL makes you a good adversary, programmer
- PL gives you the tools to argue equivalence (hard!)

Course goals

1. Learn intellectual tools for describing program behavior
2. Investigate concepts essential to most languages
   - mutation and iteration
   - scope and functions
   - types
   - objects
   - threads
3. Write programs to "connect theory with the code"
4. Sketch applicability to "real" languages
5. Provide background for current PL research
   (less important for most of you)

Course nongoals

- Study syntax; learn to specify grammars, parsers
  - Transforming $\mathbf{3+4}$ or $\mathbf{(+\ 3\ 4)}$ or $\mathbf{+(3,4)}$ to "application of plus operator to constants three and four"
- Learn specific programming languages (but some ML)

What we will do

- Define really small languages
  - Usually Turing complete
  - Always unsuitable for real programming
- Extend them to realistic languages less rigorously
- Digress for cool results (this is fun!?!)
- Study models very rigorously via *operational models*
- Do programming assignments in Caml

Caml

- Caml is an awesome, high-level language
- We will use a tiny core subset of it that is well-suited for manipulating recursive data structures (like programs!)
- You mostly have to learn it outside of class
  - Don't procrastinate
  - Don't hesitate to ask questions
- Resources on course webpage
- I am not a language zealot, but knowing ML makes you a better programmer

Pitfalls

How to hate this course and get the wrong idea:

- Forget that we made simple models to focus on the essence
- Don't quite get inductive definitions and proofs when introduced
- Don't try other ways to model/prove the idea
  - You'll probably be wrong
  - And therefore you'll learn more
- Think PL people focus on only obvious facts
  - Need to start there

Final Metacomment

Acknowledging others is crucial...

This course draws heavily on pedagogic ideas from at least: Chambers, Chong, Felleisen, Flatt, Fluet, Harper, Morrisett, Myers, Pierce, Rugina, Walker

And material covered in texts from Pierce, Wynskel, and others

(This is a course, not my work.)

Caml tutorial

- "Let go of Java/C"
- If you have seen SML, Haskell, Scheme, Lisp, etc. this will feel more familiar
- If you have seen Caml, focus here on "how I say things" and what subset will be most useful to us in studying PL
- Give us some small code snippets so we have a common experience we can talk about
- Also see me use the tools