CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2003

Lecture 1— Course Introduction

Today

- Administrative stuff
- Course motivation and goals
  - A Java example
- Course overview
- Course pitfalls
- Our first simple language: IMP

Course facts

- Dan Grossman, CSE556, djg@cs.washington.edu
- TA: Andy Collins, CSE302, acollins@cs.washington.edu
- Office hours: TBD (Tuesday 2-3 plus appt.) ?
- Conventional wisdom on new profs:
  - course too hard
  - no good at admin details
  - so I'll try to avoid this fate
- Web page for mailing list and homework 1 (start problem 0 after Thursday's lecture)

Coursework

- 4–5 homeworks
  - "paper/pencil" (LaTeXrecommended?)
  - programming (OCaml required)
  - where you'll probably learn the most
- 2 exams
  - open notes/book, closed web
- Lecture notes usually available online
- Textbook: mostly for "middle of course"
  - won't follow it too closely

Academic integrity

- If you violate the rules, I will enforce the maximum penalty allowed
  - and I'll be personally offended
  - 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

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, . . .

Does it matter?

Freshmen 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
- Never say "nobody would write that"

Also: Rigor is a hallmark of quality research

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`
- `a` 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
   - objects
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"
  - stop me when I get too sloppy
- Learn specific programming languages (but some ML)
- Denotational and axiomatic semantics
  - Would include them if I had 25 weeks
  - Will explain what they are later

What we will do

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

OCaml

- OCaml 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 have to learn it outside of class, but next lecture will be a primer
- Today, go to `www.ocaml.org` and `caml.inria.fr/oreilly-book/`
- 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 essentials
- Don't quite get inductive definitions and proofs
- 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 will draw heavily on:

- Previous versions of the course (Borning, Chambers)
- Similar courses elsewhere (Harper, Morrisett, Myers, Pierce, Rugina, Walker, ...)
- Texts (Pierce, Wynskel, ...)

This is a course, not my work.

Finally, some content

For our first *formal language*, let's leave out functions, objects, records, threads, exceptions, ...

What's left: integers, assignment (mutation), control-flow

(Abstract) syntax using a common meta-notation:
"A program is a statement s defined as follows"

$$
\begin{array}{rcl}
s & ::= & \textbf{skip} \mid x := e \mid s; s \mid \textbf{if}\ e\ s\ s \mid \textbf{while}\ e\ s \\
e & ::= & c \mid x \mid e + e \mid e * e \\
(c & \in & \{\ldots, -2, -1, 0, 1, 2, \ldots\}) \\
(x & \in & \{x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots, \ldots\})
\end{array}
$$

Syntax definition

$$\begin{array}{lcl} s & ::= & \textbf{skip} \mid x := e \mid s; s \mid \textbf{if}\ e\ s\ s \mid \textbf{while}\ e\ s \\ e & ::= & c \mid x \mid e + e \mid e * e \\ (c & \in & \{\ldots, -2, -1, 0, 1, 2, \ldots\}) \\ (x & \in & \{x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots, \ldots\}) \end{array}$$

- Blue is metanotation ($::=$ "can be a", $\mid$ "or")
- *Metavariables* represent "anything in the *syntax class*"
- Use parentheses to *disambiguate*, e.g.,
  $\textbf{if}\ x\ \textbf{skip}\ y := 0; z := 0$

E.g.: $y := 1; (\textbf{while}\ x\ (y := y * x; x := x - 1))$

Inductive definition

With care, our syntax definition is *not* circular!

$$s ::= \textbf{skip} \mid x := e \mid s; s \mid \textbf{if } e\ s\ s \mid \textbf{while } e\ s$$

$$e ::= c \mid x \mid e + e \mid e * e$$

Let $E_0 = \emptyset$. For $i > 0$, let E_i be E_{i-1} union "expressions of the form c, x, $e + e$, or $e * e$ *where* $e \in E_{i-1}$". Let $E = \bigcup_{i \geq 0} E_i$. The set E is what we mean by our compact metanotation.

To get it: What set is E_1? E_2?

Explain statements the same way. What is S_1? S_2? Stop only when you're bored.

Summary

- Did that first-day stuff
  - Install and play with OCaml
  - Ask questions
- Motivated precise language definitions
- Defined syntax
  - For a very small language
  - Very carefully

Next: Syntax proofs, Then: Caml primer, Then: Semantics

Proving Obvious Stuff

All we have is syntax (sets of abstract-syntax trees), but let's get the idea of proving things carefully...

Theorem 1: There exist expressions with three constants.

Our First Theorem

There exist expressions with three constants.

Pedantic Proof: Consider $e = \mathbf{1} + (\mathbf{2} + \mathbf{3})$. Showing $e \in E_3$ suffices because $E_3 \subseteq E$. Showing $\mathbf{2} + \mathbf{3} \in E_2$ and $\mathbf{1} \in E_2$ suffices...

PL-style proof: Consider $e = \mathbf{1} + (\mathbf{2} + \mathbf{3})$ and definition of E.

Theorem 2: All expressions have at least one constant or variable.

Our Second Theorem

All expressions have at least one constant or variable.

Pedantic proof: By induction on i, show for all $e \in E_i$.

- Base: $i = 0$ implies $E_i = \emptyset$
- Inductive: $i > 0$. Consider *arbitrary* $e \in E_i$ by cases:
  - $e \in E_{i-1} \ldots$
  - $e = c \ldots$
  - $e = x \ldots$
  - $e = e_1 + e_2$ where $e_1, e_2 \in E_{i-1} \ldots$
  - $e = e_1 * e_2$ where $e_1, e_2 \in E_{i-1} \ldots$

A "Better" Proof

All expressions have at least one constant or variable.

PL-style proof: By *structural induction* on (rules for forming an expression) e. Cases:

- c ...
- x ...
- $e_1 + e_2$ ...
- $e_1 * e_2$ ...

Structural induction invokes the induction hypothesis on *smaller* terms. It is equivalent to the pedantic proof, and the convenient way.