Goals

- Deeper understanding of PL's
- Broader exposure to PL's
- Not a survey course

Why study PL?

- Elegant math, practical impact
  - a study of expressive power
  - caveat: comfort with logic, proofs, Ch. 1
- Better language design
  - how to specify
  - how to prove correct
  - embarrassing questions to ask
- Better language implementation
  - efficient implementation (more in CS 412)
  - correct implementation
- Better programmer
  - understand your tools (and which ones to use)

Schedule

- Operational semantics  5
- Inductive proofs  3
- Lambda calculus  3
dynamic
- Denotational semantics  4
- Interesting language features  8
- Type systems  4
- Interesting types  8
- Miscellaneous topics  3
static

Workload

- Sign-up sheet
- Readings (see course schedule)
- 6 homeworks (about half with programming component, in ML)
- Scribe 3-4 lectures (in pairs)
  - we will provide TeX template
  - meet with me for feedback
- Prelim: tentatively Oct. 26, 7-9:30PM
- Final exam: Dec. 7, 12-2:30PM

Course Staff

- Lecturer: Andrew Myers
  andru@cs.cornell.edu
  Upson 4124
  Office hours: Wed 3-4PM
- TA: Matthew Fluet
  Email: cs611@cs.cornell.edu
  Upson 4162
  Office hours: TBA
  Web site: courses.cs.cornell.edu/cs611
**Texts**

- **Required:**
  - Winskel, *The Formal Semantics of Programming Languages*
- **Recommended:**
  - Gunter, *Semantics of Programming Languages*
  - Mitchell, *Foundations of Programming Languages*
  - Gifford (will be placed on-line; may be used only for this course)

**IMP**

- Winskel, Ch. 2
- Simple imperative language (vs. functional)
- IMP program is a *command*
  - `skip`
  - `X := a`
  - `c_0; c_1`
  - `if b then c_0 else c_1`
  - `while b do c`
- Variables `(X)` take integer values
- Arithmetic exprs `a`, boolean expressions `b`

**Example: GCD**

```plaintext
while x ≠ y do
  if x < y then
    y := y – x
  else
    x := x – y
end
```

- Turing-complete (barely): no functions, data structures

**Issues**

- What is a legal program in IMP?
  - defined by abstract syntax
- What is a legal program execution in IMP?
  - structural operational semantics
- Other properties of interest
  - expressions terminate, commands may not
  - programs never "crash"
  - evaluation is deterministic

**Defining Syntax**

- Three *syntactic sets*:
  - `Aexp`: set of legal arithmetic expressions `a`
  - `Bexp`: legal boolean expressions `b`
  - `Com`: legal commands `c`
- Define legal programs inductively using Backus-Naur form (BNF):
  - `a ::= n | X | a_0 + a_1 | a_0 * a_1 | a_0 - a_1`
  - `b ::= a_0 = a_1 | a_0 ≤ a_1 | b_0 ∧ b_1 | b_0 ∨ b_1 | ¬b`
  - `c ::= skip | X := a | c_0; c_1 | if b then c_0 else c_1 | while b do c`
  - `X ∈ Loc`, `n ∈ Z`

**Abstract Syntax**

- This course: not about parsing
- Elements of syntactic set are *parse trees*, not concrete syntax
  ```plaintext
  3 + 4 * x = \(\frac{3 + 4 \cdot x}{4}\) ≠ "3+4\(\cdot\)x"
  ```
- But...will write expressions that look concrete
  - parentheses used to disambiguate parsing when necessary: `(3+4)`’5` vs. `3+(4`’5`)
  - not part of abstract syntax
Operational Semantics

- Any element of Com is a legal program. How does it evaluate?
- Defining process of program evaluation: operational semantics
- Java language reference manual: verbose, long operational semantics
- Structural operational semantics: legal executions correspond to proofs
  - compact
  - convenient for proving properties of language

Configurations

- A configuration: what we need to know about a running program to define how it executes

Large-step evaluation

- Large-step semantics define complete evaluation of a program or subexpression

Some evaluations

\[ \langle X, \sigma \rangle \Downarrow \sigma(X) \] for any \( \sigma, X \)
\[ \langle n, \sigma \rangle \Downarrow n \] for any \( \sigma, X \)
\[ \langle n_0 + n_1, \sigma \rangle \Downarrow n_2 \] for any \( n_0, n_1, n_2, X \) where \( n_2 \) is sum of \( n_0, n_1 \)
\[ \langle \text{skip}, \sigma \rangle \Downarrow \sigma \] for any \( \sigma, X \)
\[ \langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \Downarrow \sigma' \] if
\[ \langle b, \sigma \rangle \Downarrow \text{true} \text{ and } \langle c_0, \sigma \rangle \Downarrow \sigma' \] (for any ...)

As inference rules

\[ \langle b, \sigma \rangle \Downarrow \text{true} \quad \langle c_0, \sigma \rangle \Downarrow \sigma' \]
\[ \quad \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \Downarrow \sigma' \]
\[ \langle n, \sigma \rangle \Downarrow n \] axiom

Execution as proof

- Legal executions = those that can be proved correct inductively
- Proof = proof tree where every step is application of an inference rule
- Execution = depth-first walk of proof tree
- Collection of inference rules: proof system

\[ \begin{align*}
&x : \langle x \rightarrow 1 \rangle \\
&\text{if } x < y \text{ then } x := 0 \text{ else } \text{skip} \quad \Downarrow \quad \langle x \rightarrow 0, y \rightarrow 2 \rangle \quad \Downarrow \quad \langle x \rightarrow 0, y \rightarrow 2 \rangle
\end{align*} \]
Applying rules

- Inference rule represents a large (infinite) set of rule instances in which meta-variables are consistently substituted.

\[
\begin{align*}
\langle n, \sigma \rangle & \Downarrow n \quad \langle 0, \sigma \rangle & \Downarrow 0 \quad \langle 1, \sigma \rangle & \Downarrow 1 \\
\langle b, \sigma \rangle \Downarrow \text{true} & \quad \langle c_0, \sigma \rangle \Downarrow \sigma' \quad \langle 0=1, \sigma \rangle \Downarrow \text{true} & \quad \langle \text{skip}, \sigma \rangle \Downarrow \sigma \quad \langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \Downarrow \sigma' \quad \langle \text{if } 0=1 \text{ then skip else ...}, \sigma \rangle \Downarrow \sigma 
\end{align*}
\]