Final Exam Content

- **Pre-midterm material**
  - functional programming (OCaml)
    - parametric polymorphism
    - partial evaluation, currying
    - tail-recursion
  - operational semantics (large-step and small-step)
  - denotational semantics
  - structural induction
  - fixpoint induction
  - semantic equivalence

- **Post-midterm material (emphasized)**
  - logic programming (Prolog)
    - backtracking search
    - reversible programming
    - order-sensitivity, confluence, Church-Rosser
  - untyped, simply-typed, and polymorphic $\lambda$-calculus (System F)
    - type-checking vs. type-inference
    - shallow vs. non-shallow types
    - Curry-Howard isomorphism and type-inhabitation
    - $\lambda$-cube
  - axiomatic semantics
    - partial vs. total correctness
    - Hoare Logic
    - loop invariants
    - weakest precondition, strongest postcondition
Summary

APL: The Big Picture
Language Families

- **Programming Language Semantics**
  - **Operational**: defines program behavior
  - **Denotational**: defines program mathematical equivalent
  - **Static**: defines program safety
  - **Axiomatic**: verifies program correctness

- **Three styles of programming**:
  - **Imperative**: programs are sequences of instructions
  - **Functional**: programs are functions from inputs to outputs
  - **Logic**: programs are declarative input-output relations
Which languages are most popularly used in the “real world” (e.g., industry)?

- unquestionably the imperative ones (C/C++/Java)

Why?

- easy to compile (no longer a compelling reason)
- momentum (large labor pool, well-developed tools)
- easy to write code that almost works

The “software crisis”

- Microsoft spends over half of its budget on testing.
- Programs still regularly buggy, exploited, etc.
- “Find better programmers” is not the answer.
Better Programming Languages

What makes a language “advanced”?

- Correctness/reliability over efficiency (within reason)
  - If I want it to run faster, I’ll buy more processors.
  - Compilers as proof-assistants

- Small-gap translation from mathematical spec to code

- Separation of concerns (the “what” vs. the “how”)

- Succinctness
  - less code = fewer bugs
  - code-reuse (parametric polymorphism)

- Modularity
  - object-oriented programming
  - See also: OCaml module system, aspect-oriented programming

- *Programmer efficiency* vs. program efficiency
Language Choice

Should we bury C/C++/Java?

- No! C/C++ is good for certain things:
  - writing the inner loop of a matrix multiplier
  - writing device drivers (but use formal verification)
  - implementing some runtime libraries (e.g., fast string libs)

- But can we please stop implementing huge software ecosystems with it?

- Java was/is a great step forward...
  - brought type-safe programming to the masses
  - popularized automated garbage-collection

- But such languages still have major weaknesses:
  - uncaught exceptions are only slightly better than crashes
  - language definition defies optimization
  - foundation on untyped, unsafe runtime leads to many compromises
Grand Challenges

- How can we make it “easy” (or easier) to construct iron-clad, fully machine-validated software?
  - Example: *Compositional CompCert* [Stewart et al., POPL 2015]

- What kind of languages might segue the imperative world toward strongly typed, functional/declarative programming?
  - Example: F# [Syme et al., 2001]

- Can we use modern PL theory to debug/correct/analyze legacy codes?

- Can we use PL theory to solve security problems like data confidentiality enforcement?
  - Example: Java Information Flow [Myers, POPL 1999]

- How can we create verified, highly parallelized software?
  - Example: $n$-Variant Framework of Verification for Java Source Code on CPU×GPU Hybrid Platform [Duan et al., HPDC 2019]
Relevance/Usefulness

- Practical right now
  - functional and logic programming
  - operational/denotational semantics for compiler design and analysis
  - static semantics for program safety verification
- Fundamental for understanding real-world software verification
  - $\lambda$-calculus
  - Hoare Logic
  - structural/fixpoint induction
- Learning to write formal, rigorous proofs
  - essential if you want to do science, not just programming
  - infrequently taught at the undergraduate level
  - If you can’t prove easy things, you can’t program hard things.
    - every program is a constructive proof (Curry-Howard)
    - Example: If you can’t reason inductively, you can’t program recursively.
Functional Programming at Microsoft

Microsoft’s Functional Language: F# (OCaml for Visual Studio .NET)
Functional Programming at Facebook

Facebook’s Polymorphic, Statically-typed Web Dev Language
### Stack Overflow Developer Survey

#### What Languages Are Associated with the Highest Salaries Worldwide?

<table>
<thead>
<tr>
<th>Language</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>F#</td>
<td>$74,000</td>
</tr>
<tr>
<td>OCaml</td>
<td>$73,000</td>
</tr>
<tr>
<td>Clojure</td>
<td>$72,000</td>
</tr>
<tr>
<td>Groovy</td>
<td>$72,000</td>
</tr>
<tr>
<td>Perl</td>
<td>$69,000</td>
</tr>
<tr>
<td>Rust</td>
<td>$69,000</td>
</tr>
<tr>
<td>Erlang</td>
<td>$67,000</td>
</tr>
<tr>
<td>Scala</td>
<td>$67,000</td>
</tr>
<tr>
<td>Go</td>
<td>$66,000</td>
</tr>
<tr>
<td>Ruby</td>
<td>$64,000</td>
</tr>
<tr>
<td>Bash/Shell</td>
<td>$63,000</td>
</tr>
<tr>
<td>CoffeeScript</td>
<td>$60,000</td>
</tr>
<tr>
<td>Haskell</td>
<td>$60,000</td>
</tr>
<tr>
<td>Julia</td>
<td>$60,000</td>
</tr>
<tr>
<td>TypeScript</td>
<td>$60,000</td>
</tr>
<tr>
<td>C#</td>
<td>$59,000</td>
</tr>
<tr>
<td>Objective-C</td>
<td>$58,000</td>
</tr>
<tr>
<td>R</td>
<td>$58,000</td>
</tr>
<tr>
<td>Swift</td>
<td>$57,000</td>
</tr>
<tr>
<td>Lua</td>
<td>$56,000</td>
</tr>
<tr>
<td>Python</td>
<td>$56,000</td>
</tr>
<tr>
<td>SQL</td>
<td>$56,000</td>
</tr>
<tr>
<td>JavaScript</td>
<td>$55,000</td>
</tr>
<tr>
<td>HTML</td>
<td>$54,000</td>
</tr>
<tr>
<td>CSS</td>
<td>$53,000</td>
</tr>
</tbody>
</table>
### Stack Overflow Developer Survey

What Languages Are Associated with the Highest Salaries Worldwide?

<table>
<thead>
<tr>
<th>Language</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>F#</td>
<td>$74,000</td>
</tr>
<tr>
<td>OCaml</td>
<td>$73,000</td>
</tr>
<tr>
<td>Clojure</td>
<td>$72,000</td>
</tr>
<tr>
<td>Groovy</td>
<td>$72,000</td>
</tr>
<tr>
<td>Perl</td>
<td>$69,000</td>
</tr>
<tr>
<td>Rust</td>
<td>$69,000</td>
</tr>
<tr>
<td>Erlang</td>
<td>$67,000</td>
</tr>
<tr>
<td>Scala</td>
<td>$67,000</td>
</tr>
<tr>
<td>Go</td>
<td>$66,000</td>
</tr>
<tr>
<td>Ruby</td>
<td>$64,000</td>
</tr>
<tr>
<td>Bash/Shell</td>
<td>$63,000</td>
</tr>
<tr>
<td>CoffeeScript</td>
<td>$60,000</td>
</tr>
<tr>
<td>Haskell</td>
<td>$60,000</td>
</tr>
<tr>
<td>Julia</td>
<td>$60,000</td>
</tr>
<tr>
<td>TypeScript</td>
<td>$60,000</td>
</tr>
<tr>
<td>C#</td>
<td>$59,000</td>
</tr>
<tr>
<td>Objective-C</td>
<td>$58,000</td>
</tr>
<tr>
<td>R</td>
<td>$58,000</td>
</tr>
<tr>
<td>Swift</td>
<td>$57,000</td>
</tr>
<tr>
<td>Lua</td>
<td>$56,000</td>
</tr>
<tr>
<td>Python</td>
<td>$56,000</td>
</tr>
<tr>
<td>SQL</td>
<td>$56,000</td>
</tr>
<tr>
<td>JavaScript</td>
<td>$55,000</td>
</tr>
<tr>
<td>HTML</td>
<td>$54,000</td>
</tr>
<tr>
<td>CSS</td>
<td>$53,000</td>
</tr>
</tbody>
</table>

**Functional**: F#, OCaml, Clojure, Groovy, Haskell, Julia, TypeScript, C#, Objective-C, R, Swift, Lua, Python, SQL

**Transitional**: Bash/Shell, CoffeeScript, Ruby, Bash/Shell, CoffeeScript, R, Swift, Lua, Python, SQL, JavaScript, HTML, CSS
Next Steps

- CS 6353: Compiler Construction
  - We learned how to design and analyze a language.
  - CC teaches how to build a compiler for a language.
- CS 6374: Computational Logic
  - Learn about automated theorem proving
  - Tools for doing formal software verification
- CS 6301.nnn: Language-based Security (shamless plug)
  - Type theory for security analysis & enforcement
  - Information flow, access control, etc.
- Independent study research
  - Dr. Gupta: logic programming
  - Me: language-based security
Bottom-up Formal Methods

**Top-down FM:**
- build high-assurance software from sources
- proofs/types driven by source-level info
- certifying, type-preserving compilation
- Example: Coq/Gallina

**Bottom-up FM:**
- obtain high assurance for source-free software
- proofs/types driven by native-level information
- ISA formal semantics specification & recovery

- **(annotated) source code**
  - Certifying Compiler
  - low-level code proofs/types
  - native code proofs/types

- **Automated Theorem Prover**
  - proofs/types

- **IL code**
  - Lifter

- **native code**
Bottom-up Formal Methods

(annotated) source code

Certifying Compiler

Low-level code
proofs/types

Native code
proofs/types

Top-down FM:
- build high-assurance software from sources
- proofs/types driven by source-level info
- certifying, type-preserving compilation
- Example: Coq/Gallina

Bottom-up FM:
- obtain high assurance for source-free software
- proofs/types driven by native-level information
- ISA formal semantics specification & recovery
Picinæ: Platform in Coq for INstruction-level Analysis of Executables

Bridging two technologies:

- ISA Operational Semantics: Binary Analysis Platform (BAP)
- Program-proof Co-development: INRIA Coq

Diagram:
- ISA Lifter
- Picinæ
- Picinæ IL
- Picinæ Theory
  - Picinæ Definitions
  - Coq
- Theorems & Proofs
- Verified Binary Code Analyses
Windows / MacOS in mission-critical environments:

- “≈ 75% of control systems are on Windows XP or other non-supported OSes” [Daryl Haegley, Office of Assistant Secretary of Defense for Energy, Installations and Environment]

- > 25% of all government computers currently run an outdated Windows or MacOS operating system [BitSight, 2017]

- DHS, Coast Guard and Secret Service currently store top secret information on outdated Windows 2003 servers [OIG-18-56, 2018]

- Hundreds of satellites run Windows 95 and/or are controlled by Windows Mobile devices.
Object Flow Integrity

Securing Large, Component-driven Software Products:

- denotational semantics of C++ object hierarchies
- reflective programming for C++
- automated code synthesis from security specifications
- median 0.34% overhead on COTS Windows applications

Feedback is welcome!

- Topics you wish had been included but weren’t?
- Homework/exam difficulty level
- Style of instruction, lecture format
- Course resources (no textbook…)

Reminder: Please take time to submit course evals!