Introduction to Programming Language Theory

What is this course about?

The subject of this course is semantics of programming languages—how is it that we can give a precise, mathematical meaning to a programming language. We look at this subject from two perspectives: syntactic proof theory, and denotational model theory.

The first perspective is often referred to under the rubric of the Curry-Howard correspondence, named after two of its major proponents, Haskell Curry [1900–1982] and William Howard [1926–]. The fundamental idea is that data types are theorems, (typed) programs are proofs of theorems and running (evaluating) programs is normalization of proofs. This correspondence suggests that programming is a good medium for understanding logic, and vice versa.

For example, though $27 \times 37 = 999$ has a denotational sense (both sides of the equation refer to the same number), there is a finite computation which shows (proves) that the denotations are the same. In other words, study the finitary dynamics. When we look at logical expressions, rather than ask (denotationally) “is $\phi$ true?” we should ask “what is the proof of $\phi$?” Writes Jean-Yves Girard in the introduction to Proofs and Types, one of the books we’ll use, by proof we understand not the syntactic formal transcript, but the inherent object of which the written form gives only a shadowy reflection [like in Plato’s cave]. We take the view that what we write as a proof is merely a description of something which is already a process in itself.

In contrast, the denotational study of programming language semantics has largely followed Gottlob Frege [1848–1925] (and before him, George Boole [1815–1864]). The semantics of logical expressions (or even natural language) is often modelled this way: a piece of syntax denotes some semantic object which is invariant under “equivalent” rewritings of the syntax. An example from computer science is the language (syntax) of regular expressions, each of which denotes a regular set; that $x^* = \epsilon \cup xx^*$ means each side of the equation denotes the same regular set. This denotational semantics associates programs (including higher-order ones) with functions on well-specified domains—typically (infinite) sets with an underlying ordering on its elements. Recursion is interpreted as a limit (in the naive calculus sense) on these sets.

$\lambda$-calculus: The programming language we’ll use as a lingua franca in this course is the $\lambda$-calculus, which is just a kind of mathematically idealized Scheme. (It’s the blueprint prototype from which functional programming languages were built.)

---

1“A logic without cut elimination,” wrote Jean-Yves Girard, “is like a car without an engine.”
A good slogan for *why* is this: \(\lambda\)-calculus is to programming languages as Turing machines are to computers. If you want to understand computational complexity of algorithms, you have to look at Turing machines (or some variant of them). If you want to understand how programming languages are put together, you have to understand \(\lambda\)-calculus. (That’s why CS21b uses Scheme to describe interpreters and compilers.)

We’ll examine its *confluence* (the Church-Rosser theorem, that evaluation order doesn’t really affect returned answers, like in “normal” mathematical calculi), and Böhm’s theorem, that if any two programs with a different *normal form* (roughly, the “evaluated” answer they produce), are equated, then as a consequence, all programs become equated.\(^2\)

*Simply-typed \(\lambda\)-calculus:* The simply-typed \(\lambda\)-calculus has a type system with variables, products, and functions. Following the Curry-Howard correspondence, \(\lambda\)-terms in this language represent proofs in a logic with propositional variables, conjunction, and implication; the normalization of terms corresponds to cut elimination on the analogous proofs. We will prove that typed terms have a normal form—that they are programs guaranteed to terminate. The first, *weak normalization theorem* identifies a specific reduction strategy which always yields a normal form; the second *strong normalization theorem* shows that any evaluation strategy must produce a normal form. The proof of the latter theorem is interesting in that it presents a general technique which can be generalized to more complex type systems.

*Polymorphically typed \(\lambda\)-calculus (System F):* This typed \(\lambda\)-calculus generalizes the simply-typed system by allowing quantification over type variables, which facilitates a certain *type polymorphism* in programming. The Curry-Howard correspondence is to a logic allowing universal quantification over propositional variables, or (equivalently) second-order quantification over a first-order universe. This calculus allows a very straightforward *coding* of many familiar data types and algorithms for integers, lists, trees, and so on. We’ll prove a strong normalization theorem for System F, using a clever variant of the technique used in the simply-typed case, Girard’s *candidats de reductibilité*. (Via Curry-Howard, this also proves Gentzen’s cut-elimination theorem for second-order logic.) We further show how second-order proofs that functional, equational specifications define *total functions* can then be mechanically analyzed, extracting (strongly normalizing) System F programs which realize the specifications. A corollary of this realizability, and the strong normalization theorem, is a Gödel-style incompleteness theorem that the strong normalization theorem cannot itself be proven in second-order logic, and (by the Curry-Howard correspondence) that a System F interpreter cannot be coded as a System F term. (Compare, in contrast, the metacircular evaluator for Scheme, coded itself in Scheme.)

*Linear logic:* The resource conscious logic constrains the use of *contraction* (that two hypotheses \(A\) are the same as one) and *weakening* (that hypotheses can be added at discretion). To recover the effect of contraction and weakening, an *exponential modality* \(!A\) is introduced which makes their use explicit in the logical formulas. We’ll look at sequent calculus for linear logic, as well as a variant formulation called *proofnets*, and examine the complexity of normalization and of parsing (the so-called *correctness criterion*), show how \(\lambda\)-calculus is coded in this logic (again, following Curry-Howard intuitions).

*Calculus of constructions:* The course concludes with an introduction to this calculus through the Coq programming environment, a subtle mixture of programming and logical notation (even in the above, they are largely separate), through the idea of *dependent products*. This makes was for another kind of logical expressiveness.

\(^2\)This theorem evokes Bertrand Russell’s explanation that from contradiction, any truth results; when challenged to prove that \(1 = 2\) implied he was the Pope, Russell answered, “The Pope and I are two, but two is one, therefore the Pope and I are one.”
**How hard will this course be?**

The course will require no programming, but a certain mathematical maturity—it’s not for the faint-hearted. Students should have completed (or have a strong grasp of the ideas in) CS21b (Structure and Interpretation of Computer Programs) and CS30a (Introduction to the Theory of Computation). A course in mathematical logic (PHIL 106b) or algebra (MATH 30a) would also be good preparation.

**Grading and homework policy**

The grading on this course is pretty flexible—it’s not like the undergraduate courses I teach. The main work is understanding what we’re talking about and doing the reading. Understanding the readings will require *real mathematical maturity*. There will likely be a couple of problem sets, but no exams. You may show further due diligence by presenting some of the material in the lecture syllabus (see the ** items), or by writing a 10-15 page expository paper on a subject from or connected to the course (in consultation with the instructor). Class attendance is required.

**Tentative syllabus**

26 lectures overall.

**Beginning [1 lecture]**

**September 1:** Administrivia and course overview.

**Untyped λ-calculus [4 lectures]**

**September 5:** Introducing the untyped λ-calculus.
   Hindley and Seldin, pp. 1–20; Mitchell, ch. 1.

**September 8:** Representing computable functions over inductively defined data.
   Class notes.

**September 12:** Church-Rosser theorem.
   Hindley and Seldin, pp. 282–289.

**September 15:** Church-Rosser (cont.). Introduction to Böhm’s theorem.
   Class notes.

**Simply typed λ-calculus [5 lectures]**

**September 19:** Böhm’s theorem (concluded).

**September 26:** Simply typed λ-calculus: types, derivations, subject reduction. Programming examples and anomalies.
   Girard, Lafont, and Taylor, ch. 3; Sørenson and Urzyczyn, ch. 3; class notes.
September 29: Type reconstruction and complexity; Statman’s theorem.
Mairson (1992, 2004), class notes.

October 6: Constructive classical logic and continuation-passing style.
Sørenson and Urzyczyn, ch. 6 (sections 6.1–6.4).

October 10: Constructive classical logic and continuation-passing style (cont.) [!!]

**System F (polymorphic \( \lambda \)-calculus), representability and undecidability:**
Gödel’s theorem for computer scientists [5 lectures]

Girard, Lafont, and Taylor, ch. 11.

October 17: Contracting proofs to programs: extraction of programs from termination proofs in second-order logic.
Leivant; Girard, Lafont, and Taylor, ch. 15.

October 20: The Curry-Howard correspondence: types are theorems, proofs are programs, evaluation is proof normalization.
Girard, Lafont, and Taylor, ch. 3; Sørenson and Urzyczyn, ch. 4.

October 24: Girard’s theorem: strong normalization of polymorphically-typed \( \lambda \)-terms and the candidats de reductibilité.
Sørenson and Urzyczyn, pp. 287–290; class notes.

October 27: Gödel-style undecidability: strong normalization of System F can be stated, but not proved, in second-order logic.
Girard, Lafont, and Taylor, ch. 15; class notes.

**Linear logic [5 lectures]**

October 31: **An introduction to proof theory: natural deduction and sequent calculus.

November 3: **Introduction to linear logic: multiplicative fragment, exponential fragment.
Girard, section 1 (first half of paper). Wiki site for linear logic:
Linear logic (cont.)

November 7: **Proofnets and paths for multiplicative linear logic: the Danos-Regnier correctness criterion.
Lafont.

November 10: Duality of expressions and continuations.
Wadler.
November 14: Duality cont.
Mairson.......

Software foundations: functional programming in Coq. [6 lectures]
At this point, the course is going to switch into a pseudo-“flipped classroom” and we are going to try working our way through the online notes Software foundations, with notes available at: https://softwarefoundations.cis.upenn.edu. Have a look at the “Logical Foundations” section—I don’t know how far we will get...

November 17:
November 21:
November 28:
December 1:
December 5:
December 8:

Reading
Note that there is no textbook for this class: readings will be xeroxed and handed out.

Jean-Yves Girard, Yves Lafont, and Paul Taylor,
Proofs and Types.

Jean-Yves Girard,
Linear logic: its syntax and semantics.
Proceedings of the workshop on Advances in linear logic, pp. 1–42.

J. Roger Hindley and Jonathan Seldin,
Lambda-calculus and Combinators: an Introduction.

Yves Lafont,
From proofnets to interaction nets.
Proceedings of the Workshop on Advances in Linear Logic,

Daniel Leivant,
Contracting proofs to programs.
Harry Mairson,
*A simple proof of a theorem of Statman.*

Harry Mairson, *Linear lambda calculus and PTIME-completeness.*

Harry Mairson,
*Call-by-value isn't dual to call-by-name,*
*call-by-name is dual to call-by-value!*

John Mitchell,
*Foundations for Programming Languages.*
http://theory.stanford.edu/people/jcm/books/fpl-chap1.ps
http://theory.stanford.edu/people/jcm/books/fpl-chap2.ps
Benjamin Pierce et al.
*Software Foundations.*

Morton Heine Sørensen and Pawel Urzyczyn,
*Lectures on the Curry-Howard Isomorphism.*

Wikipedia,
*Linear Logic.*

Philip Wadler,
*Call-by-value is dual to call-by-name.*