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.

1 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

1 “A logic without cut elimination,” wrote Jean-Yves Girard, “is like a car without an engine.”
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.

**λ-calculus:** The programming language we’ll use as a *lingua franca* in this course is the λ-calculus, which is just a kind of mathematically idealized Scheme. (It’s the blueprint prototype from which functional programming languages were built.) 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 λ-calculus:** The simply-typed λ-calculus has a type system with variables, products, and functions. Following the Curry-Howard correspondence, λ-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 λ-calculus (System F):** This typed λ-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.)

**Full abstraction:** We then pass to the denotational perspective on programming languages, where we’ll give a mathematical model of PCF, a typed λ-calculus that includes recursion (so that programs may loop). We’ll add to this language a parallel-or operation, and prove the following: that two programs \(M, N\) have a different meaning in the model exactly when they can be used in ways that are observationally different. That is, there is some context \(C[\_]\) dependent on the denotational difference between \(M\) and \(N\) where \(C[M]\) halts, but \(C[N]\) loops. This exact correspondence is called full abstraction—that the denotation model

\(^2\)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.”
makes exactly the same distinctions that usage does, no more and no less.

This theorem does not hold for the given model PCF without the parallel-or; we conclude
the course by proving that the theorem does hold for a different model based on two ideas:
linear logic, and games.

**Linear logic:** The *resource conscious* logic constrains the use of *contraction* (that two hy-
potheses $A$ 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).

**Games:** There is yet another logical tradition of understanding logic as a game between
a *prover* and a *skeptic* who interact via a protocol regarding the provability of formulas; a
formula is a theorem when there is a winning strategy for the prover. These games have
intuitionistic and classical variants, with different protocols. A more recent rendition on
these games is through the Curry-Howard correspondence, where the prover and skeptic are,
effectively, a program and its contextual environment. Language features such as state and
control operators can be explained by different game protocols. These games have been used
to build models of programming languages, and address the full abstraction issue.

**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 work for the course will consist of writing a 15-20 page expository paper on a subject
from or connected to the course (in consultation with the instructor), and giving a lecture on
one of the topics in the lecture list (again, in consultation with the instructor). Understanding
the readings will require *real mathematical maturity*, so this presentation is no easy task.
Both the paper and the presentation will be evaluated on the basis of technical accuracy,
but especially on the intuitions that motivate understanding of the subject. There may also
be an occasional problem set. Class attendance is required.

**Tentative syllabus**

26 lectures overall.

**Beginning [1 lecture]**

**August 27:** Administrivia and course overview.
Untyped \(\lambda\)-calculus [4 lectures]

August 31: Introducing the untyped \(\lambda\)-calculus.
    Hindley and Seldin, pp. 1–20; Mitchell, ch. 1.

September 3: Representing computable functions over inductively defined data.
    Class notes.

September 7: Church-Rosser theorem.
    Hindley and Seldin, pp. 282–289.

September 14: Böhm’s theorem.
    Krivine, pp. 67–72; class notes.

Simply and polymorphically typed \(\lambda\)-calculus [4 lectures]

September 17: Simply typed \(\lambda\)-calculus: types, derivations, subject reduction. Programming examples and anomalies. Weak normalization.
    Girard, Lafont, and Taylor, ch. 3; Sørenson and Urzyczyn, ch. 3; class notes.

    Girard, Lafont, and Taylor, ch. 11.

September 24: 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 1: Constructive classical logic and continuation-passing style.
    Sørenson and Urzyczyn, ch. 4 (sections 6.1–6.4).

Representability and Undecidability:
Gödel’s Theorem for Computer Scientists [4 lectures]

October 5: Girard’s theorem: strong normalization of polymorphically-typed \(\lambda\)-terms and the candidats de reductibilité.
    Sørenson and Urzyczyn, pp. 287–290; class notes.

October 8: Contracting proofs to programs: extraction of programs from termination proofs in second-order logic.
    Leivant; Girard, Lafont, and Taylor, ch. 15.

October 12: Contracting proofs to programs (continued).

October 15: 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.
DENOTATIONAL MODELS OF TYPED LANGUAGES [5 lectures]

   Mitchell, ch. 2 (sections 2.1–3, 2.5).

October 22: Domain models, complete partial orders, fixed points.
   Mitchell, ch. 5 (sections 5.1, 5.2).

October 26: Domain models, complete partial orders, fixed points (continued).

October 29: Computational adequacy: the full abstraction problem.
   Mitchell, ch. 5 (section 5.4).

November 2: Full abstraction with parallel-or (continued).

LINEAR LOGIC [2 lectures]

November 5: Introduction to linear logic: multiplicative fragment, exponential fragment.
   Girard, section 1 (first half of paper).

November 9: Proofnets and paths for multiplicative linear logic: the Danos-Regnier correctness criterion, and its complexity.
   Lafont.

LINEAR LOGIC AND GAME SEMANTICS [6 lectures]

November 12: Game semantics for programming languages: an informal introduction.
   Abramsky and McCusker, sections 1 and 2.

November 16: Game semantics: an introduction to category theory and categories of games.
   Abramsky and McCusker, section 3.

November 19: Modelling the language PCF by games.
   Abramsky and McCusker, section 4.

November 23: Games: adequacy, definability, full abstraction.

November 30: Games: adequacy, definability, full abstraction.

December 3: Games: adequacy, definability, full abstraction.
Reading

Samson Abramsky and Guy McCusker, 
*Game semantics.*
NATO Science Series, Series F: Computer and Systems Sciences. 
Springer-Verlag, Berlin, Germany, 1999.

Jean-Yves Girard, Yves Lafont, and Paul Taylor, 
*Proofs and Types.*
http://www.monad.me.uk/stable/Proofs+Types.html 

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.*

Jean-Louis Krivine, 
*Λ-calculus, Types and Models.*
Ellis Horwood, 1993.

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

Daniel Leivant, 
*Contracting proofs to programs.*

John Mitchell, 
*Foundations for Programming Languages.*
http://theory.stanford.edu/~jcm/books/fpl-chap1.ps 
http://theory.stanford.edu/~jcm/books/fpl-chap2.ps

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