COMPUTER SCIENCE 520 (AUTUMN TERM, 2000)
Foundations of Programming Languages

Course instructor: Harry Mairson (`mairson@cs.bu.edu`), MCS 283, phone 353-8926. Office hours 3–4pm Wednesdays and Fridays, and by arrangement. I especially encourage you to communicate with me via electronic mail, for fastest and most reliable responses to your questions. I try to read e-mail every 5 minutes, 24 hours a day.

Teaching assistant: Peter Møller Neergaard (`turtle@cs.bu.edu`), MCS 212, phone 353-1498. Office hours to be announced.

Time and place: Monday, Wednesday and Friday, 2–3pm, MCS B31.

WWW: `http://www.cs.bu.edu/faculty/mairson/Courses/cs520.html`

Required Textbooks

Robin Milner.
Communicating and Mobile Systems: the π-Calculus.
Cambridge University Press, 1999.

Carl Gunter.
Semantics of Programming Languages: Structures and Techniques.
MIT Press, 1992.

Both of these texts have been ordered and are available at the BU Bookstore. However, I encourage you to check on-line for better deals: the last time I looked, Amazon had the Gunter book at about half-off.

What is this course about?

This question is not easy to answer, because I'm teaching the course for the first time, and a fair amount of the material I'm learning more or less at the same time you are! But safe and secure behind that veil of ignorance, let me try to explain...

This course is on the *theory* and *foundations* of programming languages: how to formalize and reason about some essential constructs of programming languages. The course comes in two parts: one about concurrent and mobile processes, the other about more traditional language semantics.

Models of computation exist, in part, to explain the salient, essential features of some kind of real computing. The reason freshman physics persists—a world of massless beams, frictionless pulleys, etc.—is because it explains essential features of the Newtonian physical universe. The reason interest in Turing Machines persists is that it describes essential features of machine architecture. The reason interest in λ-calculus persists is its similar relationship with programming languages.

In a similar vein, Robin Milner invented the π-calculus as a formal model of concurrent, distributed processes. What is process interaction? Mobility? Synchronization? Nondeterminism and choice? Given the interest in these topics brought on by dominant research interests in the department, to say nothing of the concerns of the "real world," I thought π-calculus would be a good thing to look at. This part of the course will be driven by lots of examples (how to program in this calculus), with attention shown as well to reasoning about programs—in what sense can we say that two programs are doing the same thing?

The second part of the course will be about more conventional programming languages, with an interest in *types*, *semantics*, and *operational equivalence*. The main focus of attention will be the *typed λ-calculus*—a primitive functional programming language—and PCF (Programming language for Computable Functions), a generalization of typed λ-calculus with a more powerful kind of recursion. Questions we will be interested in asking include: how do we attach "mathematical meaning" to programs in these languages? What kinds of axiom systems do we have for reasoning about program behavior, and how do we show that these rules are sound and complete? (Soundness means they don't lead us to conclusions that are false; completeness means that the truths we seek can be found using the rules.) How can we test the behavior of programs, and how do we show that the "operational equivalence" between two programs, inferred from testing, coincides with their mathematical equivalence as defined by the semantics? (Thinking about different ways of viewing finite automata puts these questions and concepts in a certain perspective.) The latter question is the essence of what's called *full abstraction*, and we'll have a look at this problem. Finally, time permitting, we'll consider how to build a mathematical semantics for the *untyped* λ-calculus: how can every expression have a value, when every value is also a function on values?

How hard will this course be? Prerequisites.

I do not think this course will be a lot of *work*—the problem sets will probably not be too long—but I think the material will be conceptually *difficult*. I would not advise taking this course unless you are committed to doing some hard thinking.

As prerequisites for the course, I would recommend a fair mathematical maturity—you should not run screaming from inductive definitions and proofs, some basic algebra and logic, or abstract arguments—and a strong grasp of functional programming, at about the level of CS320, the undergraduate programming languages course.

What work will be required?

There will be several problem sets, an in-class midterm, and a take-home final.

Tentative syllabus

In a course, I usually block out each lecture for every day at the beginning of the term. Because I am teaching this course for the first time, and I don't know what the student reaction will be (I do not want to lecture to myself), I'm only noting here the material I intend to cover from the two textbooks, which I intend to follow quite closely.

Communicating and Mobile Systems: The π-Calculus

1. Introduction
2. Behavior of Automata
3. Sequential Processes and Bisimulation
4. Concurrent Processes and Reaction
5. Transitions and Strong Equivalence
6. Observation Equivalence: Theory
7. Observation Equivalence: Examples
8. What is Mobility?
9. The π-calculus and Reaction
10. Applications of the π-calculus
    (programming with lists, persistant and mutable data)
11. Sorts, Objects, and Functions
    (object-oriented programming, functional programming as name-passing)

Semantics of Programming Languages

1. Introduction to Semantics
2. The Simply-Typed λ-Calculus
   Syntax. Rules (typing, equivalence). Models. Completeness.
3. Categorical Models of Simple Types
   Products and cartesian closure. λ-calculus with constants and products.
4. Recursive Definition of Functions
   PCF, a programming language for computable functions. Fixed points in complete partial orders. Fixed point semantics of PCF. Bounded recursion.
5. Relating Interpretations
   Full abstraction. Extensions of adequacy results.
6. Universal Domains
   Untyped λ-calculus. Domain equations.