Computer Science 21b (Spring Term, 2002)

Structure and Interpretation of Computer Programs

Course instructor: Harry Mairson (mairson@cs.brandeis.edu), Volen 257, phone (781) 731-2236. Office hours 10am–noon Monday and 12–1pm Thursday, and by arrangement. I especially encourage you to communicate with me via electronic mail, for fastest and most reliable responses to your questions.

Head teaching fellow: John Rieffel (jrieffel@cs.brandeis.edu), Volen 111 (DEMO Lab), phone 736-3366; office hours Tuesday and Friday, 12.30–2pm.

Teaching fellows: Peter Møller Neergaard (turtle@cs.brandeis.edu), Volen 135, phone 736-2716; office hours Wednesday and Thursday 10.30am–12pm. Gregory Hornby (hornby@cs.brandeis.edu), Volen 111, phone 736-3366; office hours TBA.

Time and place: Monday, Wednesday and Thursday, 9–10am, Schwartz 106.

Required Textbook


The full text of the book is available on-line at http://mitpress.mit.edu/sicp/full-text/book/book.html. In addition, Scheme code for problem sets and some handouts will be available via the WWW home page for the course: http://www.cs.brandeis.edu/~cs21b. The course account is cs21b@cs.brandeis.edu. The class mailing list for sending and receiving information about class material, handouts, problem sets, etc., is cs21b@lists.brandeis.edu.

What is this course about?

“I always wanted to know what went into a ball-park hot dog. Now I know, but wish I hadn’t asked.” Thus began one of my favorite leads in a magazine article. This course has the same goal, only replacing hot dogs with programming languages.

This introduction to programming language structure and interpretation is based on the largely functional programming language Scheme, a simple variant of Lisp. We will look at what programs in this language mean, the underlying algorithms (also expressible in Scheme!) which are used to evaluate programs via interpretation and compilation, and how the language might have been defined differently. Along the way, we will talk about how large programs are structured, and related questions of programming pragmatics. An important goal of the course is to teach students to read programs as well as write them.

A key idea of the class is that inside many complex software systems we can find the essence of the features found in Scheme: namely, primitive expressions, means of combining such expressions, and a means of abstraction whereby such compound expressions can be named and manipulated as if they were primitive. Furthermore, the interpretation mechanism of Scheme—the dual use of evaluation of component parts, synchronous with functional application—has an equivalent universality.

While the major emphasis of the course is on functional programming, we will discuss other programming models as well: imperative programming, object-oriented programming, signal (stream) processing, and programming via rewrite rules and pattern matching. We will also discuss principles of data abstraction and abstract data types, and especially the control structures needed to interpret and compile the Scheme language itself. Finally, we will discuss data types, type checking, and the beginnings of type inference in the context of compilation. By the end of the course, you should have a pretty good idea how to design an interpreter or compiler for most any programming language or system.

---

1Try reading some on-line reviews of the book, say at amazon.com. They are bimodal...
How hard will this course be?

This course will be a fair amount of work and will be time-consuming, with particular emphasis on programming. Remember Hofstadter’s Law: it always takes longer than you think, even when you take Hofstadter’s Law into account. Because some of the ideas are a bit abstract (i.e., higher-order procedures), some mathematical maturity would be a good thing. On the other hand, no particular mathematical knowledge will be required. It is assumed that you have a knowledge of and sophistication that you would have gained from a good first course in programming.

What work will be required?

There will be 7 problem sets, an evening midterm on **Wednesday, March 13**, covering material in Chapters 1–3, and a final examination (see below). The problem sets include reading of Scheme code for large systems, programming completions or extensions to these systems, as well as textbook exercises.

In contrast to many other computer science courses, I encourage you to discuss the problem sets with your colleagues in the class, and to work collaboratively, particularly on difficult problems. Such collaboration must be mentioned explicitly in handed-in solutions, and each student is responsible for handing in individual problem sets: no “group copies” will be accepted.

What Scheme?

The course will use MIT Scheme, which is running on the department machines. You are free to get your own Scheme, but the conventions (especially re input/output) for other Scheme versions may vary in some minor ways, and it’ll be your job to sort it out. The homework you turn in should run on MIT Scheme so the graders can check it out by running your code.

Problem Sets

**Problem Set 1: Lunar Lander** A primitive video game designed using higher-order procedures.

**Problem Set 2: Doctor** A parody of psychiatry implemented using recursion on list structures.

**Problem Set 3: Stable Marriage** An American dream based on principles of object-oriented programming, implemented using procedures with encapsulated state.

**Problem Set 4: Streams** The procrastinator’s dream: computing using lazy evaluation.

**Problem Set 5: Metacircular Evaluator** What it’s all about: an interpreter that implements Scheme in Scheme. Sounds weird, but it isn’t: when it comes to interpreters, it’s turtles all the way down.

**Problem Set 6: Metacircular Compiler** Behind the smoke and mirrors: compiling Scheme into Scheme. Sounds weird, but it isn’t: we implement one-time syntax analysis, lexical addressing, and explicit control.

**Problem Set 7: Static Typechecking** An introduction to type systems for programming languages.

Final examination

The exam is scheduled by the university registrar to be on **Monday, May 13**, from 1.30–4.30pm. I will not allow incompletes for students unable to make the exam due to last-minute travel or other logistics constraints. If you have a problem, let me know within the first two weeks of class.
Grading and homework policy

In computing your grade, I expect to take the best of the following: either (1) final [100 percent]; (2) midterm [40 percent], final [60 percent]; or (3) homework [20 percent], midterm [35 percent], final [45 percent]. Everything is scaled: you are ranked in each category, and the ranks are weighed using the percentages. Recall the downside of scaled grades: when two friends are jumped by a bear, one friend says, “I don’t have to outrun the bear—I just have to outrun you!” So if you don’t want to do homework, it’s OK. And if you do badly on the midterm, but can recover on the final, it’s OK. But the only way you will learn enough to solve the exam problems is to do the homework!

I again encourage you to discuss homework problems with your colleagues in the class, and to work together on their solution. This way, you can help teach each other, and correct each other’s crazy ideas. However, you are to hand in your own solutions, and to explicitly state with whom you worked.

Homework will usually require textbook exercises as well as programming assignments. The expected method of submission will be either e-mail or direct copying of the file to the class directory cs21b by midnight of the same day. Instructions how to do this will be distributed soon—we are working this out.

Late homework: Assignments handed in \( n \) days late (weekends and holidays included) will have \( nt/10 \) points subtracted from the grade, where \( t \) is the total point score for the assignment. The only exceptions will be genuine personal difficulties and medical emergencies, as judged by the head teaching fellow.

Honor code: Cheating is a very serious business and will not be tolerated at all. The instructor and staff will make every attempt to be reasonable about assignments, due dates, etc., but infractions of the honor code will be dealt with severely.

Tentative syllabus

39 lectures overall.

ADMINISTRIVIA AND INTRODUCTION [1 lecture]

January 23: Introduction and course administration. Scheme basics: read-eval-print loop, primitives, combinations, abstraction, introduction to the substitution model.

PROCEDURES [Chapter 1; 7 lectures]


January 30: Orders of growth: asymptotics and the “big Oh” notation. Logarithmic-, linear-, and exponential-time algorithms. Example: Fibonacci numbers. [Abelson and Sussman 1.2.2–1.2.3]

January 31: Higher-order procedures: procedures as arguments. [Abelson and Sussman 1.3]

February 4: Higher-order procedures: procedures as returned values. [Abelson and Sussman 1.3]

February 6: Example: Picture language. [Abelson and Sussman 2.2.4]

February 7: Tail recursion and actor model. Problem Set 1 due.
DATA [Chapter 2; 6 lectures]


February 13: Constructing cons cells from lambda. Alternative implementations of cons. The cons-car-cdr contract. [Abelson and Sussman 2.1.3]

February 14: Hierarchical data: lists, trees, etc. List and tree recursion, with examples. [Abelson and Sussman 2.2.1–2.2.2]

February 25: Higher order procedures on hierarchical data: map and filter. Relation of recursion and induction: logic meets computation. [Abelson and Sussman 2.2.3]

February 27: Data abstraction. Quotation. Symbolic differentiation I. [Abelson and Sussman 2.3.1–2.3.2]

February 28: Symbolic differentiation II: a pattern matcher. Problem Set 2 due.

STATE AND ASSIGNMENT: THE ENVIRONMENT MODEL [Chapter 3; 5 lectures]


March 6: Environment diagrams. [Abelson and Sussman 3.2]

March 7: Environment diagrams. [Abelson and Sussman 3.2]

March 11: Message passing and object-oriented programming.

March 13: Modelling with mutable data: queues, tables. [Abelson and Sussman 3.3]

STREAMS AND LAZY EVALUATION [Chapter 3; 3 lectures]


March 18: Infinite streams. [Abelson and Sussman 3.5.2]

March 20: Streams and state: lazy data structures. Example: delayed trees. [Abelson and Sussman 3.5.3–3.5.4]

METACIRCULAR EVALUATOR [Chapter 4; 6 lectures]


March 25: Environment structures. [Abelson and Sussman 4.1.3]

March 26: Dynamic binding. Introduction to normal-order evaluation. Streams as lazy lists. [Abelson and Sussman 4.2]

April 8: Separating syntactic analysis from evaluation. [Abelson and Sussman 4.1.7]

April 10: A tail-recursive evaluator: introduction to continuations as an implementation technique for building interpreters.

April 11: Tail-recursive evaluator (part 2).
COMPILER [Chapter 5 and Problem Set 6; 5 lectures]

April 15: Lexical addressing and the compile time environment. Problem Set 5 due. Problem Set 6 handed out.

April 17: Metacircular compiler: compiling Scheme into Scheme.

April 18: Metacircular compiler: compiling Scheme into Scheme (part 2).

April 22: Variation: a register machine compiler.

April 24: Variation: generating the entire target code.

TYPES AND TYPE INFEERENCE [Problem Set 7 and readings; 5 lectures]


April 29: Monomorphic type inference. First-order unification.

May 1: Type inference and unification (part 2).

May 2: Introduction to polymorphic types.

May 6: Polymorphic type inference.

SUMMARY [1 lecture]

May 8: Summary and course review. Problem Set 7 due.