Brandeis University

Computer Science (1)21b (Spring Term, 2019)

Structure and Interpretation of Computer Programs

Course instructor: Harry Mairson (mairson@brandeis.edu), Volen 257, phone (781) 736-2724. Office hours 11am–12pm Monday, Wednesday and Thursday, and most any time by arrangement. I especially encourage you to communicate with me via electronic mail, for fastest and most reliable responses to your questions.

Teaching assistants and their office hours: Joel Richter [Head TA] (jrichter@brandeis.edu), Luyi Bai (kwipnc@brandeis.edu), Nathaniel Dimick (natdimick@gmail.com), Brian Gao (bgao@brandeis.edu), Mark Hutchens (mhutchens@brandeis.edu). Kevin Zhou (kzhou@brandeis.edu). ***Office hours to be arranged.***

TA coverage, which will be in the Vertica lounge, should be pretty comprehensive. Email is a good way to reach the TAs also, especially to request another meeting time of mutual convenience.

Note: You are guaranteed a hearing during the specified hours; you are free to try dropping in otherwise or meeting by arrangement. If you've got questions, it's our job to get you the answers.

Time and place: Class lectures are Monday, Wednesday, and Thursday, 10am–10.50am, Olin-Sang 101.

PREREQUISITES

Completion of CS21a with a grade of B or better. At least one semester of calculus, with a grade of B or better. Also CS29a (Discrete Math) would be a good idea (with a decent grade).

Required Textbook

Harold Abelson and Gerald Jay Sussman.
Structure and Interpretation of Computer Programs.[1]
MIT Press, 1996 *(second edition)*.

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 distributed via Latte.

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.

One of my computer science colleagues said to me, *"If computer science is supposed to teach the cutting edge, and the cutting edge keeps changing, why are we teaching people stuff that almost immediately gets outdated? Wouldn't it be better to teach some core principles of what the subject is about?"* That's a goal of this course. By analogy, think about physics. Undergraduates have learned Newtonian mechanics as freshmen since time immemorial—a world of massless beams and frictionless pulleys, even though our knowledge of the physical world has become far more refined since the seventeenth century. But we continue to teach mechanics out of a sense that, even though there is more "fine print" to our knowledge of the external world, Newton's mechanics describes in an essential way what the world really is. This course is an attempt to follow in that intellectual tradition.

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

[1] Try reading some on-line reviews of the book, say at amazon.com. They are bimodal...

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.

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.

IMPORTANT! This course *starts quickly.* Don't make the mistake of *starting slowly.* If you want to relax, wait two weeks before doing so (when we're doing the "Data" section, which is a little easier). Immediate lethargy runs the risk of leaving you hopelessly lost, like the student who asked me in Week 10, "what is this *lambda* thing everyone keeps talking about?"—a great question for Week 1.

Finally, recognize that "how hard will this course be?" is subject to multiple interpretations. Conceptually, this course is challenging. But that does not mean that the *grading* is particularly unforgiving. You have to do your work. But the grade distributions are not particularly different from other computer science courses.

What work will be required?

There will be 6 problem sets, a *midterm examination* on **Thursday, March 7** in the evening (time TBA), covering material in Chapters 1–3, as well as a *final examination* on a date TBA: the Registrar has not released its schedule for finals yet. We should have this information by the end of the month.

I will not endorse incompletes for students unable to make exams due to last-minute travel or other logistics constraints. If you have a problem, let me know within the first two weeks of class. On the other hand, I'll do whatever your academic dean tells me to do, at any time.

The problem sets include *reading* of Scheme code for large systems, *programming* completions or extensions to these systems, as well as *textbook exercises.* Exam problems will be easier variants of those you see in the problem sets.

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 writing up and handing in individual problem sets: no "group copies" will be accepted. (See *Grading and Homework* below.)

What Scheme?

You should download Racket, a Scheme implementation from Northeastern University, which has an MIT Scheme (so-called because of the textbook, also known in Racket as R5RS, maybe with slight differences) on it. The homework you turn in should run on MIT Scheme so the grader can check it out by running your code. You may need to check that some small input/output conventions (where implementation differences typically appear) obey the MIT Scheme idioms.

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.

Laptop, cellphone, outdoor clothing, and slouching policies, rules, guidelines...

You may not have an open laptop in class, unless you have the instructor's permission.

You may not check text messages during class, unless you have the instructor's permission. You risk dismissal from the day's lecture for texting.

You may not wear your hat and coat in class, unless you have the instructor's permission.

You need to sit up straight in your seat, with your feet placed flat on the floor. (Actually, this is more of a guideline than a rule...)

Note: The *outdoor clothing policy* and *slouching rule* are designed to keep you from *falling asleep* during lecture...

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. You should not be shipping code to each other via email, for example. We will consider your handing in code that someone else wrote a violation of another code—the honor code—and will act appropriately. However, it is OK to include pieces of code that appear publicly on the class mailing list.

Homework will usually require textbook exercises as well as programming assignments.

Submission protocol: Through LATTE—details to follow.

Late homework: The point of homework is to *do it*—so we are likely to be *moderately benevolent* about accepting late homework... *until* it is graded. After that, your chances plummet.

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

Responsibility Clause: You are bound by the honor code even if you were not aware of its details.

Responsibility for Responsibility Clause: You are bound by the Responsibility Clause even if you were not aware of its details.

... and so on ...

Tentative syllabus (39 lectures overall)

Administrivia and Introduction *[1 lecture]*

January 16: Introduction and course administration.

Procedures *[Chapter 1; 7 lectures]*

January 17: Scheme basics: read-eval-print loop, primitives, combinations, abstraction, introduction to the substitution model. [Abelson and Sussman 1.1] *Problem Set 1 handed out.*

January 22: Square-root program. Block structure and lexical scoping. [Abelson and Sussman 1.1]

January 23: Logarithmic-, linear-, and exponential-time algorithms. Example: Fibonacci numbers. [Abelson and Sussman 1.2.1–1.2.3]

January 24: Iterative versus recursive procedures, tail recursion. Examples: fast exponents, greatest common divisor. Orders of growth: time and space asymptotics and the "big Oh" notation. Lambda (λ) and `define`. [Abelson and Sussman 1.2.1–1.2.3]

January 28: Higher-order procedures: procedures as arguments, and procedures as returned values. [Abelson and Sussman 1.3]

January 30: Higher-order procedures (cont.). Picture language. [Abelson and Sussman 2.2.4]

January 31: Picture language. [Abelson and Sussman 2.2.4]

Data *[Chapter 2; 7 lectures]*

February 4: Building abstractions with data. Pairing. Box and pointer diagrams. Constructing `cons` cells from `lambda`. Alternative implementations of `cons`. The meaning of meaning: the `cons-car-cdr` contract. [Abelson and Sussman 2.1] *Problem Set 1 due. Problem Set 2 handed out.*

February 6: Dual representation. Syntactic sugar and lists. Hierarchical data: lists, trees, etc. List and tree recursion, with examples. Inductively defined data. [Abelson and Sussman 2.2.1–2.2.3]

February 7: Higher order procedures on hierarchical data: `map` and `filter`. Implementing binary trees.

February 11: Priority queues: a data structure based on binary numbers.

February 13: Quotation. Symbolic differentiation I. [Abelson and Sussman 2.3.1–2.3.2]

February 14: Symbolic differentiation II: a pattern matcher. [Abelson and Sussman 2.3.1–2.3.2]

February 25: A programming language for Euclidean geometric constructions: points, lines, circles, intersections—a functional geometry with no math.

State and Assignment: the Environment Model *[Chapter 3; 4 lectures]*

February 27: Demise of the substitution model. State and assignment: `set!` Introduction to the environment model. [Abelson and Sussman 3.1] *Problem Set 2 due. Problem Set 3 handed out.*

February 28: Environment diagrams. [Abelson and Sussman 3.2]

March 4: Environment diagrams. Message passing and object-oriented programming.

March 6: Message passing and object-oriented programming.

Midterm Examination: **March 7** (evening)

March 7: Review for midterm.

March 11: Results of midterm.

Streams and Lazy Evaluation *[Chapter 3; 4 lectures]*

March 13: Programming with streams. Higher-order procedures. Implementation via delayed evaluation. [Abelson and Sussman 3.5.1] *Problem Set 3 due. Problem Set 4 handed out.*

March 14: Infinite streams. [Abelson and Sussman 3.5.2] Streams and state: lazy data structures. Memoized streams. [Abelson and Sussman 3.5.3–3.5.4]

March 18: Example: delayed trees. State versus streams. Joint bank accounts. Ghostbusters. Stream crossing.

March 20: What distinguishes programs from data? Computer science as the applied philosophy of mathematics.

Metacircular evaluator *[Chapter 4; 7 lectures]*

March 21: Defining Scheme in Scheme: metacircular definition of evaluation. Control structures and environment structures. [Abelson and Sussman 4.1.1–4.1.3] *Problem Set 4 due. Problem Set 5 handed out.*

March 25: Dynamic binding.

March 27: Normal-order evaluation. Streams as lazy lists. [Abelson and Sussman 4.2]

March 28: Syntactic sugar.

April 1: Separating syntactic analysis from evaluation. [Abelson and Sussman 4.1.7]

April 3: A tail-recursive evaluator: introduction to continuations as an implementation technique for building interpreters.

April 4: Tail-recursive evaluator (part 2).

Compiler *[Chapter 5 and Problem Set 6; 8 lectures]*

April 8: Lexical addressing and the compile time environment. *Problem Set 5 due. Problem Set 6 handed out.*

April 10: Lexical addressing (part 2)

April 11: Metacircular compiler: compiling Scheme into Scheme.

April 15: Metacircular compiler: compiling Scheme into Scheme (part 2).

April 17: Metacircular compiler: compiling Scheme into Scheme (part 3).

April 18: Variation: a register machine compiler.

April 29: Data types and type checking.

May 1: Type checking and type inference. *Problem Set 6 due.*