Computer Science 320 (Spring Term, 2001)
Introduction to Programming Languages

Course instructor: Harry Mairson (`mairson@cs.bu.edu`), MCS 283, phone 353-8926. Office hours Tuesday and Thursday, 3.30–5pm, 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 fellow: Sebastian Burckhardt (`sburck@cs.bu.edu`), MCS 212, phone 353-1498. Office hours Monday 2–3pm, 4–5pm, Wednesday 2–3pm. There will also be a *recitation section* in B31 on Friday 10–11am.

Time and place: Tuesday and Thursday, 2–3.30pm, GCB 207.

Required Textbook

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

In addition, Scheme code for problem sets and some handouts will be available via the WWW home page for the course: `http://www.cs.bu.edu/faculty/mairson/Courses/cs320.html`. The course account is `cs320@cs.bu.edu`.

What is this course about?

This introduction to programming languages 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.

What Scheme?

The course will use MIT Scheme, which is running on `csa.bu.edu`. You are free to get your own Scheme, but the conventions (especially re input/output) may vary in some minor ways. The homework you turn in should run on MIT Scheme so the graders can check it out.

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 *in-class* midterm on material covered in Chapters 1–3, and a final examination. 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.

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 Tuesday, May 8, 2–4pm. If you have such constraints, discuss them with me early in the course. Please do not come to me at the end of term with a request for an incomplete due to travel or other logistics constraints.

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 `cs320` by midnight of the same day. Instructions how to do this will be distributed soon—we are working this out. Homework should be submitted electronically using `gsubmit`; details forthcoming.

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 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

28 lectures overall.

PROCEDURES *[Chapter 1; 4 lectures]*

January 16: Introduction and course administration. Scheme basics: read-eval-print loop, primitives, combinations, abstraction, introduction to the substitution model. Square-root program.

January 18: Substitution model. Block structure and lexical scoping. `Lambda` and `define`. Recursion. Iterative versus recursive procedures. Examples: fast exponents, greatest common divisor. [Abelson and Sussman 1.2.1] [Abelson and Sussman 1.1] *Problem Set 1 handed out.*

January 23: Orders of growth: asymptotics and the "big Oh" notation. Logarithmic-time, linear-time, and exponential-time algorithms. Example: Fibonacci numbers. [Abelson and Sussman 1.2.2–1.2.3]

January 25: Higher-order procedures: procedures as arguments. Procedures as returned values. [Abelson and Sussman 1.3]

Data *[Chapter 2; 4 lectures]*

January 30: Building abstractions with data. Pairing. Box and pointer diagrams. Constructing `cons` cells from `lambda`. Alternative implementations of `cons`. The `cons-car-cdr` contract. [Abelson and Sussman 2.1] *Problem Set 1 due. Problem Set 2 handed out.*

February 1: Hierarchical data: lists, trees, etc. List and tree recursion, with examples. [Abelson and Sussman 2.2.1–2.2.2]

February 6: Higher order procedures on hierarchical data: `map` and `filter`. Relation of *recursion* and *induction*: logic meets computation. [Abelson and Sussman 2.2.3]

February 8: Data abstraction. Quotation. Symbolic differentiation I: simple case. Symbolic differentiation II: a pattern matcher. [Abelson and Sussman 2.3.1–2.3.2]

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

February 13: Demise of the substitution model. State and assignment: `set!` Introduction to the environment model. Environment diagrams. [Abelson and Sussman 3.1, 3.2.] *Problem Set 2 due. Problem Set 3 handed out.*

February 15: Environment diagrams. [Abelson and Sussman 3.2]

February 20: Message passing and object-oriented programming. Dining philosophers.

February 22: Modelling with mutable data: queues, tables. [Abelson and Sussman 3.3]

Midterm *[1 lecture]*

March 1: $\boxed{\text{In-class examination.}}$ *Problem Set 3 due.*

Streams and Lazy Evaluation *[Chapter 3; 2 lectures]*

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

March 15: Infinite streams. Streams and state: lazy data structures. Example: delayed trees. [Abelson and Sussman 3.5.2–3.5.4]

Metacircular evaluator *[Chapter 4; 5 lectures]*

March 20: Defining Scheme in Scheme: metacircular definition of evaluation. Control structures. [Abelson and Sussman 4.1.1–4.1.2] *Problem Set 4 due. Problem Set 5 handed out.*

March 22: Environment structures. Dynamic binding. Introduction to normal-order evaluation. Streams as lazy lists. [Abelson and Sussman 4.1.3,4.2]

March 27: Separating syntactic analysis from evaluation. [Abelson and Sussman 4.1.7]

March 29: A tail-recursive evaluator: introduction to continuations as an implementation technique for building interpreters.

April 3: Nondeterministic computing: implementing backtracking. [Abelson and Sussman 4.3]

Compiler *[Chapter 5 and Problem Set 6; 4 lectures]*

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

April 10: Metacircular compiler: compiling Scheme into Scheme.

April 12: Variation: a register machine compiler.

April 17: Variation: generating the entire target code.

Types and type inference *[Problem Set 7 and readings; 4 lectures]*

April 19: Why types? Static type checking in a variant of Scheme. Weaknesses of monomorphic type systems. *Problem Set 6 due. Problem Set 7 handed out.*

April 24: Monomorphic type inference. First-order unification.

April 26: Introduction to polymorphic types.

May 1: Polymorphic type inference. *Problem Set 7 due.*