Lecture 01: Welcome

10:00 AM, Sep 6, 2017

Contents

To speak a second language is to have a second soul.

–Charlemagne, Emperor (A.D. 747 - A.D. 814)

Objectives

This first set of class notes is unusual: it *contains more than was said in class.* That's partly because some parts, like the details of the history, may interest you, and partly because it provides a place to look up things about the course organization and goals easily. Subsequent course notes will more accurately reflect what went on in class. –Spike

By the end of this class, you will know:

- the organization of CS17
- why you should study computer science
- the philosophy underlying CS 17/ 18
- what DrRacket looks like, and how to write a first (trivial) Racket program

1 Announcements

- Register on Courses At Brown ASAP, so that you will be able to get into a lab section and create an account on the CS department machines. This is very important, since it is the only way you can recieve our notifications.
- You should have gotten an e-mail about signing up for lab times, and another about piazza. If you didn't get either of these e-mails, please contact the HTAs at cs0170headtas@lists.brown.edu

2 Course Organization

Each week there are classes 3 times a week (Monday, Wednesday, Friday, 10 a.m.). There's a 2-hour lab, in which you'll build up strengths in some particular skill, sometimes applying things you've learned in class to understand something new. Think of these as the "wind sprints" of computer science. There's homework due every week, on Tuesdays. Over the course of the semester, there are four larger projects due as well. The final homework assignment is a take-home final exam. The syllabus, which is on the course website, gives the expected time commitments for these.

Many of you are also shopping CS15, and may say, "Wow, that sounds like a lot more work than CS15!" Not so. The number of hours spent per week by the average student is about the same. Furthermore, we have a terrific course staff here to help you succeed in CS17. We have four head TAs: Katie, Veda, Harriet, and Anna, and more than 20 UTAs who'll be running labs, holding TA hours, and generally doing everything they can to try to help you succeed in the course.

3 Course Design

CS17 is an integrated introduction to Computer Science.

"Integrated" means that we do both programming and more theoretical stuff together.

"Computer Science" is . . . well, what Computer Scientists do: programming, analysis, communications, systems design, machine learning, robotics, and a million other things.

The key word is "introduction." This course was designed for people who have never taken a computer science course, have never written a program, have never done anything more with a computer than send email.

There are a lot of people in this room who *have* had much more experience than that, and some of them may be bored at times. We're trying to avoid that by providing challenges to keep things

interesting for them, but any such challenge is just lagniappe — a little something extra. Any challenge problem on an assignment will not count towards the final grade.

Now let me talk to the lots-of-experience crowd about this course. I first encountered CS17 the year after it was first taught, i.e., the first time it was taught without major goofups. At that time I was an associate professor of computer science. I had been writing programs and developing algorithms since high-school, something like 25 years. I found the course — even the first couple of weeks — absolutely riveting. That's how I ended up teaching it. So while you may be bored on occasion, I hope you'll find it as exciting as I did.

3.1 Grading

Everyone: CS17 is graded on an absolute scale: if you learn enough and show it, you get an A, no matter how many other people get an A. Treat having more expert folks around as a chance to learn something from someone your own age rather than from an old fuddy duddy like Spike. And you more-expert folks: treat those moments when you're working with someone who knows less than you do as a chance to practice the kinds of skills that make you a valuable team member — communication, patience, clarity, the ability to listen well.

Each assignment will have some possible point total — it might be 100, it might be 43. But since each will have its scores adjusted, these numbers really don't matter. Furthermore, each assignment will count as some fraction of your final grade. Not every homework is the same difficulty as every other, and we'll be weighting the homeworks so that the harder ones count more.

For each assignment, I'll announce where the A/B, B/C, and C/NC dividing lines are. We'll then re-map the grades so that the A range corresponds to 90-100, the B range corresponds to 80-90, the C range is 70 to 80, and the NC range is 70 and below. We'll use these remapped grades in computing final averages.

That means that if your homework grade for homework 7 is a 75, and the A/B cutoff for homework 7 is 74, your grade will re-map to something like a 91 – a low A. Your friend who got a 97 on the homework will have the grade re-mapped to a 99, a high A. So there's still a real difference between you: you got a low A, your friend got a high A. And it's the *numbers* we'll be working with at the end of the semester, not the letter grades. So you could get As on every homework and project, but a B on the final, and end up with a B overall . . . by having really low As on everything before the final, and a really low B on the final. The point of announcing the cutoffs is not to coarsen the grade structure, but rather to help students have a sense of where they are: for many students, that first grade of 75 might seem like a disaster, until they realize that it's actually just into the "A" range, which seems pretty good!

For a worked sample of how the remapping works, take a look at the "Grading" section of the missive.

3.1.1 Partial grading

Homework is assigned for two distinct reasons:

- To help students develop skills through practice, or to learn small extensions of ideas discussed in class, and

- To provide an opportunity for evaluation, both so that we can assign grades, and so that students can know how well they're grasping the material.

Both are valuable, but there's no reason that they need to be completely linked. Because of this, we'll typically be grading some but not all of the problems on any particular assignment. If you want to game the system, you can try to guess which ones those are. You'll lose the benefit of actually learning the topics from the other problems, and there's a good chance that you cannot actually reliably guess what we'll grade, so you'll lose points, but it's your choice.

4 Other information

The remaining course policies are in the syllabus and course missive. You should read those; I won't disrespect you by regurgitating their contents here.

Much of the day-to-day information about the course will be disseminated via Piazza, which is a course-discussion tool you'll learn about.

5 Why Study Computer Science?

By now you have probably noticed that computing is applicable to almost all domains of knowledge. From sequencing the genome to figuring out how proteins fold; from encrypting sensitive financial information to predicting which stock prices will rise; from transforming page layouts to compressing media for quick transfer over the Internet... the list goes on and on. Regardless of whether your primary interest is computer science or some other field that makes heavy use of the discipline (like medicine, biology, cognitive science, economics, graphic design, or digital media, to name a few), we encourage you to study CS because of its fundamental practicality in today's Information Age.

Computer science is the new calculus.

At its heart, the field of computer science is about using computers to solve problems—specifically, problems that come to us in the form of a program specification, dictating that for some **input** some corresponding **output** is required. The solution to such problems is a (computer) **program**, which consumes input and produces output. Consider this well-known example: Google's search engine is a program that takes as input a keyword string. It also takes as input about 3 billion web pages. As output, it yields a subset of those pages that it deems most related to the keyword string—say, 17 million matches. Amazingly, it accomplishes this feat in a fraction of a second!

Computer Science = Problem Solving

This example highlights another compelling reason to study computer science. In our daily interactions with the world, intellectual challenges arise. Conquering them can require some ingenuity. The excitement of invention and the accompanying feeling of accomplishment are some of the additional rewards to be had by studying CS, above and beyond its obvious practicality.

6 About the CS 17/ 18 Sequence

In CS 17, we will teach you to write computer programs. However, more than that, we will help you hone your critical thinking skills, discover efficient and elegant solutions to challenging problems, clearly articulate your solutions, and collaborate with your fellow students to produce work of a higher quality than any one of you could have accomplished alone.

Multiple programming paradigms: CS 17 is founded on several principles: The first is that students should learn multiple programming paradigms. You can think of a programming paradigm as a style of writing computer programs. Three of the most important programming paradigms (i.e., styles) are **functional**, **imperative**, and **object-oriented**.

Although programming language theory continues to evolve, no one paradigm is perfect for all problems. Even if one is determined to use some particular paradigm, experience with multiple paradigms is essential to understanding the motivation for programming language features that support that paradigm.

To this end, we cover functional programming in CS 17, and imperative and object-oriented programming in CS 18. Historically, functional programming has not enjoyed as widespread use commercially as imperative and object-oriented programming. However, ideas from functional programming are slowly seeping into the mainstream, particularly as parallel programming increases in prevalence. For example, the MapReduce programming model, used internally at Google for parallel processing, incorporates key ideas from functional programming.

Multiple programming languages: A related principle is that students should learn multiple programming languages within each paradigm. One of the mottos during the initial design of CS 17/ 18 was "beyond syntax." This reflects our goal that students understand aspects of programming and programming language theory in a way that transcends the syntax of any particular programming language. To be able to see things from such a broad perspective, you need to learn several programming languages. Once you do, you will be in a better position to understand what is essential to expressing computation and what is idiosyncratic. You will also be in a better position to choose a programming language for a given task (you can do a compare and contrast) as no one programming language is perfect for all tasks. Finally, after learning a few languages, learning new ones becomes easier and easier (not to mention more and more fun!).

This year, in CS 17/ 18, you will be introduced to Racket, OCaml, Java, and Scala.

Racket as a first programming language: In CS 17, we start with a programming language called Racket. You have probably not heard of it. It is primarily a research and teaching language; it is not often used commercially.

Racket is a dialect of Scheme, which was developed in 1975 by Sussman and Steele.[1] It is a small, elegant, and powerful language. The *syntax* (grammar) and *semantics* (meaning) are clear and simple, making the language very easy to learn. Even so, there are parts of Racket that you won't learn, such as macros and mutative programming. The subset of Racket we do teach is even more manageable than the whole of Racket.

[1] Sussman contributed to the book on Scheme that is still used at MIT, and Steele later contributed to the development of Java.

For those of you with experience with other traditional programming languages, the workings of Racket may come as a shock. Programming in it may seem counterintuitive. You'll find, however, that the (few) pieces fit together just right. Racket is built not on an amalgam of "features" but on a few simple and powerful concepts.[2] This makes it easy to learn, so that after just a few lectures, you will be ready to harness its power.

Here's what we predict will happen: As you are first exposed to Racket, you'll say, "eeww; weird; too many parentheses." Once you get used to it, though, you will find it an extraordinary pleasure to use. Later, when you learn other programming languages, you will fondly recall Racket—and perhaps even wish you were using it. In any case, your knowledge of Racket will forever influence the way you program (in any language).

In a few weeks, we will turn our attention to another language, OCaml, a member of the ML family. OCaml starts with the semantics of Racket, but adds features like types and type-checking, pattern-matching, and a module system, which enables you to combine chunks of programs together in a clean way to create larger programs.

Where there is mastery, there is no mystery. Another of the themes of CS 17 is *mastery*. Our goal is to ensure that you completely understand your programs. Mastery requires that one fully understand one's tools. Because Racket is so small and elegant, you will be able to fully understand its semantics very quickly.

No magic. Many Brown CS courses provide "support code," that is, already-written programs for students to build on in doing their assignments. Certainly a student can obtain more impressive results if he or she builds on previously written programs. However, we feel it is both more satisfying and more educational for novice programmers to write programs from scratch. Our goal is to eliminate magic as well as mystery. This too is part of mastery.

Focus on problem-solving strategies. When you are faced with a long and complicated problem, how do you begin? One approach might be to step right up to a computer keyboard and start typing. That is *not* the approach taken here! First, you will want to think very hard about how to break that problem down into "digestable" parts—small parts, whose solution is readily obtained. And once you have solutions to all the small parts, you can start to piece those solutions together to construct a solution to the original long and complicated problem.

Part of learning to program effectively is learning to write small programs; confidently writing small programs is the first skill needed to write larger ones.

Mix learning of programming with learning analysis of algorithms: CS 17 is an introduction to computer science. Not surprisingly, we teach programming. Knowledge of programming is an essential part of computer science, *but it is only a part*. Unlike other introductory courses, this course is not just about programming. You will also learn a little of what makes computer science a *science*. You will begin your discovery of *algorithms* and the *analysis of algorithms*. An algorithm is a computational means of solving a problem. Analysis involves using mathematics to predict how long an algorithm will take to solve a problem.

[2] To quote from a book on its history, Sussman and Steele "realized that languages should not be developed by adding a lot of features, but by removing weaknesses and limitations that make additional features seem necessary."

Many traditional computer science curricula first teach programming for a full semester, and only afterwards teach algorithms and analysis. The designers of CS 17/ 18 think this is a bad idea. First, it conveys a false sense of what computer science is about. Programming is essential to some aspects of computer science, but not all. What is essential to all aspects of computer science is the abstract notion of computation, something you will learn to appreciate in CS 17.

Second, computational performance (how fast a program solves a problem, or how much memory it uses) is crucial to some applications, so it is worthwhile to keep computational performance in mind right from the start of your programming experience.

Multiple application areas: In CS 17/ 18, we draw our examples and assignments from a variety of subareas of computer science, including artificial intelligence, programming languages, software engineering, databases, financial computation, and graphical interfaces. But please note: we create no graphical user interfaces in CS 17; those come in CS 18. However, the concepts we study in CS 17 arise throughout computer science: e.g., in graphics and image-processing.

Though we avoid anything too flashy in this class, we do do interesting projects! CS 17 students discover the pleasure to be had from writing a program from scratch that can hold a (rudimentary) conversation with a person, a program that can play a passable game of Connect Four, and a program that interprets a small programming language. Most importantly, our students experience, again and again, the intellectual satisfaction of finding an elegant solution to a knotty computational problem. Our belief is that the development of that skill is the best foundation for further study in any area of computer science.

Pair programming: Studies show that students perform better in their CS courses the long-run when their introductory experience is collaborative. [3] Hence, we strongly encourage collaboration on *most* assignments in CS 17; more specifically, the final exam is expected to be done alone, but students are encouraged to collaborate on all other assignments.

Simply put, pair programming is "two people working together at a single computer" [4]. The practice has been popularized by a software development methodology called Extreme Programming (XP), and a number of researchers have studied the effects of incorporating pair programming into introductory and higher-level computer courses.

Why pair program? Again, simply put:

- You will produce better code [5]
- You will learn more, sharing your ideas with your peers and benefiting from their insights [6]
- You will become better at articulating your thoughts [7]

[3] In Support of Pair Programming in the Introductory Computer Science Course

[4] "Extreme Programming: A Gentle Introduction", http://www.extremeprogramming.org, (August 2007)

[5] Laurie A. Williams and Robert R. Kessler, "All I Really Need to Know About Pair Programming I Learned in Kindergarten", *Communications of the ACM*, (May 2000), and Laurie Williams and Eric Wiebe and Kai Yang and Miriam Ferzli and Carol Miller, "In Support of Pair Programming in the Introductory Computer Science Course", *Proceedings of the 15th Conference on Software Engineering Education and Training (CSEET'02)* (February 2002)

[6] Jennifer Bevan and Linda Werner and Charlie McDowell, "Guidelines for the Use of Pair Programming in a Freshman Programming Class", *Computer Science Education*, (2002)

[7] See footnote 5

- You will enjoy your work more and spend less time frustrated [8]
- You will be better prepared for more complicated software engineering tasks, both in school and beyond, where collaboration is imperative to success [9]

How do you pair program? Briefly, two partners sit down at a single computer, both situated so they can easily view the screen. There are two roles, **driver** and **navigator**, and about every fifteen minutes the two partners switch roles. The driver is responsible for computer input: typing, moving the mouse, scrolling, switching applications. The navigator is responsible for catching typos and thinking about the code at a higher level: e.g., is there a better way to implement this feature?; how will the code we are writing now fit in with the rest of our program?

Clark Cutler, one of the 17 head TAs in 2007, wrote a pair programming handout that includes some of the above information, and details how to pair program successfully. What we have not yet noted here is: *communication is crucial*, both to produce better code (e.g., speak up when you notice your partner's introduced an error), and to resolve any brewing conflicts before they escalate (e.g., if your partner consistently arrives late at your scheduled meetings). You can find Clark's handout on the course web pages, under Documentation. *Be sure to read this handout!*

Power, simplicity, and elegance: In CS 17, you will learn a clear methodology for deriving programs. That does not mean that programming does not require creativity, or that it is not difficult. Sometimes it will take an hour to discover a two-line program. But that two-line program will be a thing of beauty. Another principle of CS 17 is that we should strive for elegance. There is an elegant/beautiful solution to every problem you'll encounter in CS 17.

More generally, the kind of work you do in this class will be mostly problem-solving. You will, we hope, find the experience of taking this course both intellectually challenging and delightful. If you enjoy stretching your brain, you can expect to find this course deeply satisfying. At least once a week, you should have an "Aha!" experience, or perhaps a forehead-slapping one. It's a wonderful feeling, that feeling of insight, of understanding an elegant solution to a tricky problem, of the pieces coming together in your head. Ultimately, this is the very best thing about CS 17.

7 What you will learn this year

7.1 CS 17

Today, you should know basic algebra. By the end of this semester, you will know, among other things:

- the fundamental concepts of functions;
- how to program in (the main parts of) two languages, Racket and OCaml, and how to debug programs in those languages;
- the meaning and purpose of various logical programming constructs which are universal to all languages;

[8] See footnote 5
[9] See footnote 6

- how to program in a functional style;
- how to define data structurally and in a way that *proves* that your program can't have certain types of errors;
- various algorithmic solutions to sorting, a fundamental problem in computer science; and
- how to analyze a program's run time.

7.2 CS 18

If you take both CS 17 and CS 18, then by the end of the year, you will have learned, among other things:

- how to write and debug programs in Java, a popular object-oriented programming language;
- how to write programs that use networks, like a Web browser and Web server;
- how to write programs that use graphical interfaces, or GUIs;
- about **parsing**, the process of turning lines of text into a form that computers can understand;
- how to use mutation and **dynamic programming** to design clever algorithms;
- how to use **amortized analysis** to understand why those algorithms are faster;
- important data structures like **arrays**, **linked lists**, and **hash tables**; and
- how to put it all together writing programs in Scala!

8 Racket demo

Prof. Hughes (Spike) started up Dr. Racket, showed off the interactions and definitions window, and typed a very simple program:

```
> 17

17
```

The > is the prompt from Dr.Racket indicating that you should type something. The 17 is a tiny program, and the number 17 below it is the result of executing that program: DrRacket prints out the number 17.

Next, Spike wrote another simple program:

```
> (+ 3 2)

5
```

DrRacket printed out the number 5. You can probably figure out that the computer added together 3 and 2, and printed out the result, 5. You may be wondering, though, why Spike didn't write $3 + 5$, the way you were taught in math class. The answer has to do with something called **syntax**, which is the programming equivalent of grammar. Syntax is a set of language-specific rules that define what's okay to write as a program.

We'll spend the next several weeks enlarging our ideas of what a program can be, and developing a model for what's happening during execution of a program so that as you write programs, you can know exactly what they'll do; that's the sign of real mastery.

9 Sets

Informally, a set is a collection of things. Some sets that we'll be using a lot in this course are:

- $\mathbb{Z}$: integers: ..., -2, -1, 0, 1, 2, 3, ...
- $\mathbb{N}$: natural numbers: 0, 1, 2, 3, ... (some books start with 1; we won't)
- $\mathbb{R}$: real numbers: 0, 2.663, -19, π, ...
- $\mathbb{R}_+$: the set of positive real numbers

To define your own sets, there are several tools at your disposal. The first, and perhaps the most intuitive, is to use any spoken language of your choice: "Let C be the set of all students CS17."

Next up is *enumeration*. Enumeration is a way of indicating sets with braces. For instance, $S = \{1, 5, 7\}$. This means that $1 \in S, 5 \in S, 7 \in S$ are all true, but $x \in S$ is false unless $x = 1, 5, 7$. For clarification: the symbol $\in$ is read as "in," and is a way of saying that "element x is in set Y" ($x \in Y$).

The last way to define sets that Spike covered was *restriction*: $U = \{x \in S \mid x < 6\}$. In words, this means that U is the set of all elements in S that are less than 6. Or, that any element in S is in U, so long as the elements are less than 6. So, the elements in U are 1 and 5.

The formula for describing a set using restriction is as follows:

1. A name for an element, and a set that it comes from
2. A vertical bar
3. A condition that the element must satisfy to admit it to membership in the set

$\mathbb{R}_+$, the set of all positive real numbers, can be written using restriction. It would look like: $\mathbb{R}_+ = \{z \in \mathbb{R} \mid z > 0\}$.

10 Summary

- Be sure to register for CS17 to be able to signup for a lab, receive our emails, and get a CS account.

- CS17 will contain weekly homework assignments, weekly two-hour labs, four larger projects, and one take-home final exam.
- No prior experience is required to take this course.
- Grading is absolute; you're not competing with others.
- The CS17/18 sequence will teach you multiple programming paradigms (functional, imperative, and object-oriented) as well as four programming languages - Racket and OCaml in CS17 and Java and Scala in CS18.
- Many programs you write in this class will be from scratch - you will focus on breaking down a problem into parts, coding it from start to finish, and debugging it. Besides programming, CS17 focuses on algorithm analysis - you will learn various algorithmic solutions such as sorting, and how to predict how long an algorithm will take to solve a problem.
- We'll gradually develop a model for Racket program execution; for some very simple programs, it's very simple.
- $\mathbb{N}$ denotes the natural numbers $0, 1, 2, \ldots$; $\mathbb{R}$ denotes the real numbers (everything on the number line). $\mathbb{Z}$ denotes the integers, $\ldots, -2, -1, 0, 1, 2, \ldots$.
- Sets are collections of things. They can be described using several tools, including enumeration, restriction, and plain old words.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17document by filling out the anonymous feedback form: `http://cs.brown.edu/courses/cs017/feedback`.