CS103 Daily Doggie Bag 1

The most important topics and details of the day.

January 17, 2006

**Topic 1. Welcome to CS103.** Be sure to familiarize yourself with the instructor and TA office hours, and the information available on the WebCT page, including grading and academic honesty policies.

**Topic 2. Central course topics.** In this course, underlying concepts will be at least as important as the particular languages being studied. These concepts cut across language forms, and include:

1. **Computability:** How do we understand computing power in a mathematically precise manner? What essential features must a programming language possess to realize this power?

2. **Syntax** (form) and **semantics** (meaning): What language constructs are desirable, feasible? How do we communicate and implement the meaning of these features?

3. **Static (compile-time) analysis:** Ensures safety of programs.

4. **Reasoning about programs:** well-designed languages allow careful thought and reasoning, not just hacking.

**Topic 3. Computable functions.** A numeric function, a mapping from numbers to numbers, is a basic mathematical concept. Intuitively, when we write \( f(x) \) to denote the application of a function \( f \) at a point \( x \), we always imagine that we get a single value. You may have your own ideas about what a function is, but mathematicians often define any numeric function \( f \) as a set of ordered pairs such that there exists no \( (a, b), (a, b') \in f \) for \( b' \neq b \) (informally, every input has a distinct output). We write \( f(x) \) to denote \( y \) such that \( (x, y) \in f \). In this view, a function is an infinite object.

Example: The “doubling” function is:

\[
\{(1,2), (2,4), (3,6), (4,8), \ldots\}
\]

The “tripling” function is:

\[
\{(1,3), (2,6), (3,9), (4,12), \ldots\}
\]

Definition: The domain of a function \( f \) is the set:

\[
\{x \mid (x, y) \in f\}
\]

The range of a function \( S \) is the set:

\[
\{y \mid (x, y) \in f\}
\]

This definition of function is very abstract and powerful. As programmers we often think of a function as some computational process that we define, provide an input and receive an output, but there is nothing in the above definition that necessitates that all functions be **computable**. Indeed, this very question, which numeric functions are computable, was central to the research of the first computer scientists, who were mathematicians and philosophers working at the beginning of the last century. What does it mean for a function to be computable? The idea is actually informal, and is generally accepted to be something like:

**Common notion:** a procedure is said to be computable for a given input if, and only if, it can be described in a finite manner as a set of definite actions, and provides an output in some finite amount of time.
**Topic 4. Turing machines.** Turing machines (TMs) were proposed by Alan Turing in the 1930s as a means to characterize and study the computable functions. Some important things to remember about TMs:

- TMs consist of an input/output tape of arbitrary size, divided into squares with 1 or 0 on each square, a read/write head that can shift left or right one square, or write 1 or 0 on 1 square, and an input card stack reader.
- Input cards allow programming in a very primitive, low level language.
- TMs are ideas, not things, even though the earliest computing devices were based on TMs. Note for example that the tape is arbitrarily long.
- TMs predate what we think of as "computers".
- Turing computable means definable as a TM, and a language is Turing complete iff it can define all Turing computable functions.
- Some TMs never halt, and there exists no TM that can always predict in a finite amount of time whether an arbitrary TM will halt on a given input; this is called the halting problem.

**Topic 5. Programming Languages are TMs.** The syntax of programming languages is analogous to TM input cards. The semantics of the language is analogous to the mechanics of the TM. Theoretically, programming languages are interesting because they provide models of computation. Practically, modern programming languages try to provide the right abstractions, or features, that make programming easy, safe, and productive.