1 About these notes

The textbooks for this course are excellent, and also written at a very accessible and informative level. So expect these course notes to be a little more sparse than you might have seen in previous courses. Yes, you might actually have to read the textbook. Trust me, the author is a much better writer than your instructor!

In general, I advise that you complete the recommended readings before each relevant lecture. If you are struggling, this is one of the best tools to stay on top of the course content.

Think of these notes (along with the marked-up slides) as a shortened summary of what was discussed in lecture. They will usually be short, but sometimes will be more detailed, when we cover things that aren’t discussed well in the textbooks.

2 About this course

The main goal of this course is to help you become that “master programmer”. Simple enough, right? This is a hands-on course, so you will get to apply many of the ideas that we talk about in lecture. Over the next few months, you will:

- Learn to program in a functional language (Scheme)
- Write a complete interpreter from the ground up for a simple language
- Write a compiler for a virtual machine
- Learn a different programming language on your own
- Discover the wide variety of features offered by different programming language, and understand how languages are specified and described.

At Carnegie Mellon and most other respectable computer science departments, all this is split into about three different courses in the undergraduate curriculum. We will give a whirlwind overview in one semester. So hold on to your hats! And get ready to bring your programming skills to a new level.

The graded components of this class are Homeworks, Labs, Exams, and the semester-long Project. You will find that labs and lectures build on each other, but aren’t always overlapping. The time you will spend working on the labs is one of the most crucial components of the course. The Course Policy has many more details on all this and is worth your time to read over.

3 Programming Languages

Readings for this section: PLP, Sections 1.2 pp. 10-14 (required) and the rest of chapter 1 pp. 3-36 (recommended)

Programming languages are the medium by which we communicate our ideas and algorithms to computers. Just like human language, there are two competing concerns: how easy it is for us to write down our ideas (the expressivity of the language), and how easy it is for someone else (in this case, the computer) to understand them.
Higher-level languages are designed more with the programmer in mind, meant to make it easy to code up complicated programs. They are therefore more expressive but might be more difficult to compile. Lower-level languages, by contrast, are more focused on how the computer actually works. These languages trade expressivity for speed and simplicity.

There are a few major categories of programming languages, or paradigms. Different people disagree on the details here, but here’s what I would call the major programming language paradigms:

- **Imperative**: Straightforward languages where you write statements that roughly correspond to actual machine instructions. There might be some loops and functions, but generally not very many fancy features.
- **Functional**: Rather than a series of statements that are executed in order, programs are written as a combination of many small function calls. Recursion is usually preferred over loops. This style of programming actually predates computers!
- **Object Oriented**: Programs are designed around objects, which have both state (data) and behavior (methods). This is really a programming style that could be applied in many languages, but a few languages explicitly support and/or encourage this style of programming.
- **Scripting**: These languages are designed to let you solve small tasks and problems quickly. Often they are interpreted rather than compiled, and feature extensive library support and growing language features. The focus is on ease of use rather than speed.
- **Logic**: Not nearly as popular as the other types, logic languages equate the running of a program to the proving of a theorem. Sounds crazy (and is), but these languages can allow a very simple expression of very sophisticated programs.
- **Esoteric**: This class of languages is not really meant to be useful except as a diversion or proof of concept. You wouldn’t want to write an operating system in one of these languages, but they might be interesting for theoretical study, fun to learn, or revealing about the nature of computation.

You have probably already seen an imperative language (C), an object-oriented language (Java), and a scripting language (bash or Perl) in your coursework here. However, you probably haven’t seen any language from one of the most popular paradigms, functional programming languages. We will spend the beginning of this course learning Scheme, a simple yet powerful functional language. Some of you will also learn a logic programming language or an esoteric language in your course project.

One really important thing to remember is that programming languages can’t necessarily be compared simply in terms of “better” and “worse”. The best language to use depends on the task. Just like a master craftsman knows exactly which tool to use for any job, so the master programmer has a wide range of languages and knows how to choose an appropriate one for any problem.

## 4 Overview of compilation

**Readings for this section**: PLP, Sections 1.6 pp. 25-35 (required) and 1.4 pp.16-23 (recommended)

Compilation is the act of turning source code (in any programming language) into executable assembly code. More generally, it could be described as a type of code translation, translating from some programming language (such as C++) into another (yes, assembly is a programming language).

At a high level, compilation follows the stages you see in the slides and in figure 1.3 on page 26 of PLP. You should know and remember these stages, as we will spend some time talking about and implementing each one eventually!

Interpretation of a programming language differs from compilation in that executable assembly code is never produced. Instead, the abstract syntax tree is executed directly, and immediately. But you can see that most of the stages of compilation (scanning, parsing, semantic analysis) are the same for interpreted languages.

And what is an abstract syntax tree (AST), you ask? Again, we’ll cover this in much more detail later, but the important thing to remember is that it has nothing to do with the syntax of the language! The AST represents the meaning of the program in some useful way, regardless of the minor details about how the code was formatted or even what language it was written in! More on this later. In fact, it’s possible (in principle at least) for programs in two completely different languages to generate the same AST, if they have the same meaning. We’ll even see
examples of something like this in languages such as Clojure that are totally different from Java but compile to the same bytecode and run on the same virtual machine.