SI 413, Unit 1: Introduction

Daniel S. Roche (roche@usna.edu)

Fall 2018

1 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.

2 Programming Languages

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

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.

3 Overview of compilation

Readings for this section: PLP, Sections 1.6 pp. 25-35 and 1.4 pp.16-23

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 following stages:

1. **Scanning** (also called *lexical analysis*): Turning the original source code into a stream of chunks called *tokens* using simple regular expression matching. This is like separating a text document into a list of words and punctuation marks.

   Comments and whitespace (except in langauges where whitespace is meaningful!) are removed at this stage.

2. **Parsing**: Using *context-free-grammars* to organize a stream of tokens into a parse tree. This, along with the scanning phase, constitute the *syntax analysis* of compilation, based only on simple rules of how the source code looks, and not on any of the meaning behind what the program is doing.

3. **Semantic Analysis**: Now we're getting somewhere! This stage of compilation turns a parse tree into an *abstract syntax tree*, or AST for short.

   And what is an 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.

4. **Optimization**: This stage is technically optional for a compiler to work, but you wouldn't be very happy if it didn't happen. At this stage, the compiler makes changes to the AST (essentially, changing

the program that you wrote!) in order to make it run faster on the target architecture.

This stage is very important in practice, and for long-established languages such as C and Fortran there is still a great deal of work that goes into improving and updating the optimization routines in compilers for each new kind of processor that comes out. This is how the same old code take advantage of newer and newer hardware features, which is great if you're a programmer who doesn't like to rewrite their code every year!

(Unfortunately, this is also the stage of compilation that we'll have almost no time to spend on this semester.)

5. **Code Generation**: This is the last stage of compilation, which turns an (optimized) AST into the output language, usually assembly or machine code. It can be fairly tricky becaues usually the low-level language you are compiling to has far fewer features than the high-level language you started with!

   Note that this last stage is the only one that really *needs* to be re-done if you want your compiler to work for a new kind of machine, or to translate into a different programming language.

Because we have almost no time in this class to discuss all the optimizations that compilers perform, we'll sometimes leave out step 4 in thinking about the process. The Dragon book is a great resource if you want to learn more about this on your own.

4 Interpreted languages

Nowadays, many languages are not fully compiled but rather **interpreted** directly. Think about C++ vs Javascript. With C++, you start by writing your source code and then compiling into an executable file, maybe called `a.out`. Importantly, *this executable can be run on any machine with the same microarchitecture, even if that machine doesn't have any compiler*! Typically when you install or purchase software, you're just getting this executable code; the original source code has already been compiled for you.

Javascript works differently. When you load a webpage with Javascript, *the actual code is sent to your browser* which then executes it on the web site's behalf. Your browser in this case is acting as an *interpreter* for the Javascript language. The advantage here is that, rather than compile executables for every possible kind of computer that someone might use when visiting their site, a website author just needs to write Javascript source code once and send that directly, leaving it up to the browser to execute. Of course, there is a price to be paid for this flexibility however, mostly in terms of execution speed.

In an interpreted language, the last stage of compilation is skipped; the interpreter executes the (optimized) AST directly. In fact, the first language which we will learn in this class, Scheme, is an interpreted language, and the interpreter we are using is DrRacket.