CS6110
Advanced Programming Languages

1 Goals and Main Topics

This is a graduate course in programming languages taken by many Computer Science doctoral students to meet PhD course requirements. The course stresses principles rather than examining a variety of specific languages. Many of the ideas taught emerged from the ACM conference, POPL, *Principles of Programming Languages* or PLDI, *Programming Language Design and Implementation.* An example of such a principle is the idea that programming languages can have a *precise operational semantics*, independent of the machine code produced by compilers to run on modern hardware.

Operational semantics precisely defines the response of programs to input as a sequence of *steps of evaluation* that follow a deterministic application of computation rules. These rules are given independently of the translation to machine language. Indeed, a compiler is *correct for the language* if and only if the machine code it produces gives the same result on inputs as do the rules that define the language. These rules are often stated informally in a manual for the language or in a document defining the language.

We are mainly interested in "real programming languages" – those that are implemented and used by more than a few people. These are human artifacts as well as mathematical objects. As with all things human, they are not formal and "do the darndest things." So in the actual definitions of languages, say C for instance, we see cases where the language definition essentially says "we don't say what happens in this case." That deviation from mathematical practice causes delight and dismay, and we will see a few examples of this as we go, reminding us that this is not a course in

mathematics although it is very mathematical in spirit and aspiration. [1]

Another principle of programming languages is that *types are valuable in specifying what a program is intended to accomplish*, e.g. stating the computing task to be solved by the program. A type specification is typically at a level of abstraction above that of the rules of computation. In the early days of programming, type systems were very rudimentary. For example, the type of the output only told users whether the value produced was a finite precision natural number (of type *fixed*) or an approximation to a real number (type *float*). It was assumed that all programs were intended to halt.

Type checking is a service provided by compilers and interpreters to help ensure that a program is meeting at least a minimal level of specification. The history of programming languages shows a steady increase in the expressive richness of type systems and a corresponding increase in the complexity of type checking algorithms. That will be a phenomenon we study in this course. It is also a topic of lively debate and exciting change in modern programming language development.

The combination of a precise operational semantics and precise typing rules allows us to treat some programs as abstract mathematical objects as well as an executable artifacts. This approach to programming languages is covered in the *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, edited by J. van Leeuwen. On that view of theoretical computer science, there are two major branches, Algorithms (Theory A) and Semantics (Theory B). This course covers many topics in a *theory of computing course of type B*. Indeed, at Cornell the course is in large part theoretical. Although there will be small programming assignments.

As noted, real programming languages are also human artifacts. They are used to execute the algorithms created in Theory A. Like a good algorithm, a good programming language should help people "get computational work done well." To accomplish this, a programming language must produce code that runs well in the operating systems and on the hardware platforms that can be built. So there are some systems and engineering elements of this course. The theoretical ideas must work in practice. These practical issues are covered in depth in a compiler course.

[1] Modern examples of the darndest things are not as alarming as in the "good old days" of the wild west when a programmer could redefine the constant 2 to be another value, say 5. This is frowned on now.

Some of the CS 6110 teachers are also experts in compilation and understand operating systems and modern hardware in detail. Some are not, and one can see from various course offering over the past two decades and more, the CS6110 courses are mostly invariant under that difference. That invariance is evidence that the Theory B approach to this course is grounded in a substantial body of pertinent computing theory. (Theory B includes conferences such as Logic in Computer Science (LICS) and Mathematical Foundations of Computer Science (MFCS) and several others.)

Indeed, theory B has been especially lively over the past forty years leading to a whole new class of programming languages with extremely rich type systems and supported by automated reasoning tools in the form of programming assistants (also called proof assistants). We will look at the increasing influence of these assistants and expose students to the research currents driving this line of work. This has been a very active area of research at Cornell and at many universities in the US and EU.