CS 334: Principles of Programming Languages

Spring 2009 Syllabus

Prof. Morgan McGuire

Office hours	TCL 308, TW 2:30 – 4:00 pm (or any time by request!)
Phone	413-597-4215
E-mail	morgan@cs.williams.edu
TA	M. Catalin Iordan, Marius.C.Iordan@williams.edu
TA hours	TCL 312, MT 8-10pm, W 8-11pm
Course page	http://cs.williams.edu/~morgan/cs334
Lecture	TCL 206, TR 9:55 – 11:10 am
Textbook	Krishnamurthi, *Programming Languages: Application and Interpretation*, ver. 2007-04-26, plus provided papers and essays.

A programming language balances expressiveness, detection of potential program errors, compilation time, and run-time performance. Different balances suit different application domains. As a result, there are hundreds of useful programming languages, and we can expect to see hundreds more within our lifetimes. This course describes a set of formal mathematical tools for defining and implementing the semantics of a language and demonstrates them in the context of important real-world programming languages, with emphasis on theoretical properties of type systems.

1. Goals

You know how to reason about software artifacts and the algorithms that they implement. In this class you're going to reason about the tools used to create software itself. I believe that when you look at languages through the eyes of a designer, it improves way that you think about algorithms and the ways that you write, debug, and maintain code. Guided by that philosophy, the goals of this class are for you to learn:

1. how compilers, interpreters, and virtual machines work,
2. how to apply features such as polymorphic types and exceptions,
3. how a program is shaped by the language used to express it, and
4. the limits of programming languages (and why they are there!)

so that you can better:

1. design programs and algorithms,
2. select an appropriate language for a project,
3. learn new programming languages, and
4. appreciate certain deep ideas about computation.

You will understand why so many people are interested in designing new programming language and compilers, and may find that you are one of those people.

You'll learn a bit about a lot of different languages during this course, but surveying all programming languages or gaining proficiency in even some them are *not* goals of the course. That's because once you learn the ideas underlying all languages, you'll be able to understand any one instantly and be able to master it with practice. Don't take my word for it. Famous programmer Joel Spolsky says[1], "I have never met anyone who can do Scheme, Haskell, and C pointers who can't pick up Java in two days, and create better Java code than people with five years of experience in Java."

By the end of this course you'll be able to evaluate a new language and its suitability to solving a class of problems. For example, you'll be able to explain why the successful server-side application language Java failed to displace C++ for client-side and OS development, why scripting languages are ascendant, and why parallel programming languages are likely to be the next major paradigm shift. You'll be able to infer how a compiler analyzes a piece of code and understand why the language is designed that way. Finally, you'll build a strong intuition for language features: you'll know to expect in the next versions of current languages and have a good idea of which features you won't ever see together.

You already know how to program in Java—so why study "programming languages" in general and the ones covered in this course in specific? Because Java is just a fad. Lisp, C, Cobol, and Fortran were each once the most popular programming language. It is likely that ten years from now, you will be programming in a language very much unlike Java. Therefore, scientists and engineers must separate their ideas about computer science and software design from any one particular language.

Even within the context of Java, that language neither static nor encompasses all important ideas. For example, it recently added a polymorphic mechanism called generics, and still lacks type inference, tail recursion, and macros.

2. Structure

CS 334 is a required core course for the Computer Science major. The only prerequisite for this course is CS 136. As a 300-level course, CS 334 assumes a certain mathematical maturity and familiarity with computer science. Students who have not taken one or more 200-level courses such as CS 237, CS 256, and Math 251 will therefore be at a disadvantage.

The average student will spend 2.5 hours in lecture, 3 hours reading, 4 hours working on assignments, and half an hour in office hours for this course each week. That's a total commitment of about 10 hours per week. If you are spending significantly more time than that on the course to maintain a B average, please make an appointment or come to office hours and discuss this with me right away—a short meeting could save you hours on the assignments, and I'm here to help you both in and out of lecture. If you're spending less than 10 hours per week in the class, keep doing whatever you're doing and start doing the "challenge" problems if you get bored.

There are two lectures per week, which involve discussion and in-class exercises. There are two in-class exams and one scheduled final exam. There are required and optional reading from the textbook and research papers. Homework assignments comprise brief essay questions, mathematical exercises, and programming challenges.

There are no formal programming labs, although you will use a computer to check

[1] http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html

your work on many homework problems. Keep in mind that CS 334 is structured more closely to a theory course (like CS 256) than to a systems course (like CS 136 or CS 237). You'll find yourself thinking a long time to write a 30-line program. I'll evaluate most programs solely by reading, not running them. Most of the points on a programming question are for understanding and applying ideas. Whether a program actually compiles is not so important, and you could actually complete most of the assignments without ever running your code. Although you are welcome to work in any computing environment you wish within the bounds of the honor code, I provide relevant tools on the department's UNIX lab computers. If you are not familiar with these, talk to me as soon as possible so I can bring you up to speed with what you need to know.

Grades will be determined roughly as follows: Class Exams: 15% each, Final Exam: 30%, Homework and programs: 30%, Participation (in-class discussion, attendance, supporting other students, 2-min presentation): 10%.

I'm asking you to submit your solutions in hardcopy at the beginning of the class when they are due. **Late work and e-mailed solutions will not be graded.** There are four reasons for this policy. First, the TA and I grade assignments as soon as possible after they are submitted to give you timely feedback. We can't do that if work doesn't come in at the right time. Second, every student should have the same amount of time to work on each assignment and it isn't fair if someone receives a few extra hours. Third, I want to answer your questions about the assignment right away in lecture and can't do that if someone hasn't submitted yet. Finally, I've found that if you're overloaded or having trouble on an assignment, an extension only prolongs the problem and lets it snowball. Extensions treat the symptom—come to office hours and we'll fix the underlying problem so you can stay on track. Because everyone experiences emergencies and makes mistakes, I'll drop your lowest homework grade. That means that you can afford to miss one assignment without affecting your final grade. If you know you won't be in class when an assignment is due, then complete it early and have a friend submit it (although you're still the one responsible) or submit it yourself ahead of time.

Prof. Morgan in Math asks students to make 1-minute presentations on topics of their choice throughout the semester. I think it is a neat idea and we're going to try it too. Choose a topic related to programming languages (maybe from the recommended reading list), prepare a 2-minute blackboard presentation on it, and sign up to give your presentation at the beginning of one lecture during the semester. Plan on spending at least two hours preparing your presentation. When you aren't presenting, please come to class on time so that you can see your colleagues' presentations.

3. Honor Code

Homework is to be the sole work of each student unless the assignment explicitly states otherwise. Students may collaborate or receive help from each other on an occasional basis as long as all parties contributing or assisting are explicitly credited for their contributions at the end of the assignment and the nature of the contribution is explained. In particular, I hope you will help each other in learning the mechanics of how to compile programs in new languages and their syntax. If in doubt as to what is appropriate, ask me. Uncredited collaborations will be considered a violation of the honor code and will be handled appropriately. When you receive assistance from the TA, another professor, or me, you must indicate that as well (this is also how I know if TA hours are effective). The complete computer science honor code may be read at http://www.cs.williams.edu/resources/usage.pdf.

4. Topics

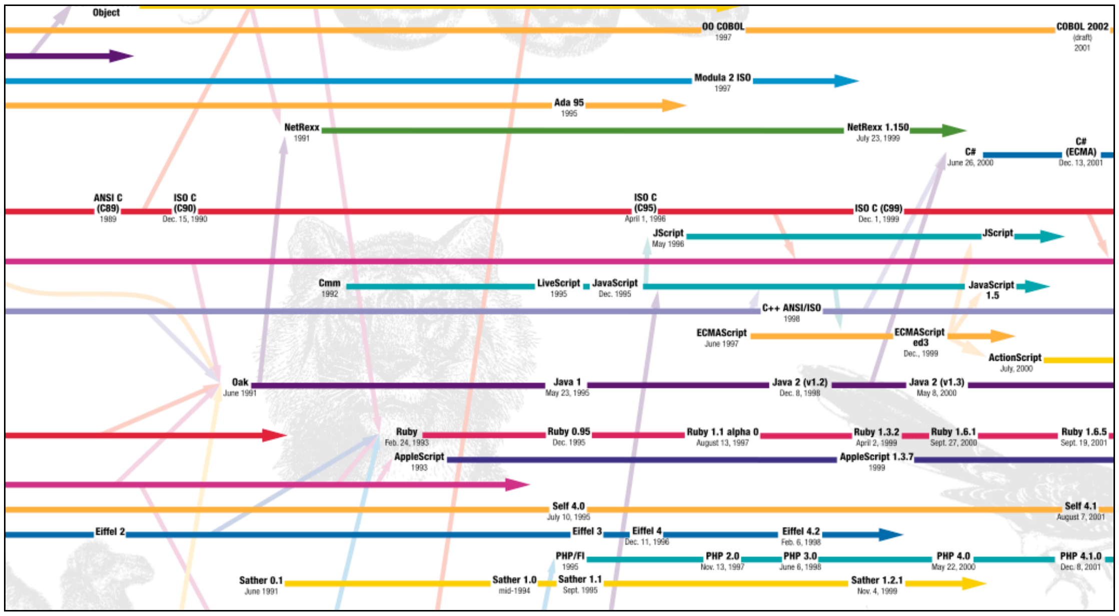
We'll cover all of the following topics, some as whole course units with lectures and assignments and others only in passing for context.

Big ideas:

- Syntax and semantics; formal specifications
- Type systems, safety, and correctness
- Features and expressiveness
- Static vs. dynamic properties and checking
- Imperative vs. functional programming
- The halting problem and computability

Language features and algorithms that implement them:

- Concurrency and synchronization ("threads")
- Exceptions
- Tail-recursion
- Continuations
- Static type-checking
- Polymorphism
- Closures (and scope)
- Automatic memory management
- Grammars and parsing algorithms
- Lazy evaluation
- Reflection
- Inheritance and subtyping
- The Hindley-Milner type inference ("unification") algorithm
- Macro languages and preprocessors
- Dynamic dispatch
- Protection mechanisms



Detail from "History of Programming Languages" published by O'Reilly