L M U | C M S I 4 8 8

LANGUAGE TRANSLATION AND IMPLEMENTATION

Syllabus • Spring, 2014

Tuesdays and Thursdays
9:25 a.m. – 10:40 a.m.
Doolan 222
4 semester hours

Ray Toal
rtoal@lmu.edu
Doolan 110
310.338.2773

Learning Outcomes

Students will:

- Gain a working knowledge of the fundamental concepts of programming language design, specification, implementation and translation through the very practical exercise of designing, writing, documenting, and testing compilers and interpreters;
- Gain a familiarity with compiler theory and design techniques and the notion of virtual machines;
- Increase their software development expertise by building a complex system (a compiler) using a modern toolset including node.js, npm, and mocha, as an open source project hosted on a public repository; and
- Take a new look at C and assembly language, making it less likely that their skills in these areas will rust completely before graduation.

Prerequisites

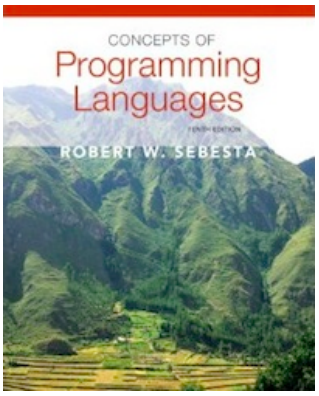
Fluency in a high-level programming language such as C# or Java. Courses in Programming Languages and Systems Programming are required; courses in Computer Science Theory and Computer Architecture are quite helpful.

Readings

Readings will be assigned from a variety of sources throughout the term. While no textbook will be explicitly required, many students would be well-served by picking up a textbook that covers programming language implementation, such as the most recent editions of Sebesta or Scott, and perhaps a good, solid introduction to Node.js development. Suggestions include:

- Robert W Sebesta, *Concepts of Programming Languages*, 10th Edition, Addison-Wesley, 2013.
- Michael L. Scott, *Programming Language Pragmatics*, Morgan Kauffman, 2009.
- Mike Cantelon, Marc Harter, T.J. Holowaychuk and Nathan Rajlich, *Node.js in Action*, Manning, 2013.
- Papers and articles referenced from Bryan Ford's Packrat Parsing and PEG page

- The Compiler Connection
- Intel 64 and IA-32 Software Developer's Manuals
- Sandpile
- Agner Fog's Software Optimization Resources
- The Linux Assembly



Additional papers and readings will be assigned throughout the course (including my own course notes, practice problems, and sample code). If you have projects or papers to work on, you'll have to find some additional readings on your own. Use judgment when researching on the web; a fair amount of information is often wrong, and much of the so-called sample code is especially atrocious. Regardless, you must take the time for effective self-study.

Assignments and Grading

You'll have several homework sets containing in-depth theoretical questions and non-trivial programming problems, and quizzes and a final exam with less difficult material. Of course you have to build a compiler in this course, but since you have only a semester to do it the compiler will be built off of an existing compiler for a simple language; you will extend it to a compiler for a more complex language. To keep you on track, the project will be turned in in pieces, specifically as part of the homework assignments for the semester. The assignments will also contain some theoretical problems for you to work out, too. You have to write a report documenting the architecture and design of the compiler; however, your writeup may appear on your wiki as opposed to a stodgy PDF. To help prepare you to meet industry expectations for college graduates, programming assignments will sometimes be required to be placed in version-controlled public repositories (such as Google Code or github). To reinforce the notion of separation of content and presentation in work, you *may* be asked to create homework solutions with the LaTeX document preparation system. Exams *will* cover material from lectures *not previously assigned* for homework: don't whine about this.

Generally, coursework may be done in groups of no more than *two* students; however, while only one solution set is turned in per group, both students are responsible for understanding *all* of its content and may be asked at any time for an oral explanation of any solution. Collaboration with other groups is fine but must be limited: you may share ideas and approaches but not solutions. You **must** acknowledge any help received. Academic dishonesty may result in expulsion; be certain your work meets the standards set forth in the LMU Honor Code.

Your final grade will be weighted as follows:

Homework sets	45 pts
Quiz 1	12 pts
Quiz 2	16 pts
Final Exam	27 pts

Letter grades are figured according to the usual scale: 90% or more of the total points gets you an **A**, 80% a **B**, 70% a **C**, and so on. These are minimal requirements; for example, if you get 82 points you are *guaranteed* a **B-** or better, though you might still get an **A** since 82 may be the top score.

Homework is due at the beginning of class; late assignments are docked 30% per class. Missing class just to get an assignment done on time will not be tolerated; the only good excuses for missing class are excellent surf conditions, family problems, sickness, and personal emergencies. *Skipping class just puts your fellow students at an advantage: we often spend class time going over things that will be "on the exam".*

Where assignments involve programming, the *quality* of your code, not just its *correctness*, will play a huge part in

determining your grade. I will not hesitate to assign **D**'s or **F**'s to working programs which are poorly structured, poorly commented, have poor identifier names and abbreviations, contain inappropriate hard-coded values, or are not easily maintainable for any reason. Appearance of the grading policy in this syllabus constitutes fair warning of the consequences of poorly written code.

Student Rights and Responsibilities

You have the right to:

- A syllabus with stated learning outcomes
- Have those learning outcomes addressed
- Student-instructor discussions when you need them
- Find the instructor during office hours for in-person chats
- Have questions answered via email (one-day turnaround max)
- Skype chats with the instructor, practically anytime
- Feedback on your work (and the right to challenge it or ask for more)
- A challenging experience (you should be working a little outside your comfort zone)
- Take photos of the board and record lectures
- An experience beyond what you would get from books, websites, online tutorials, and discussion forums
- Leave the course better skilled than when you came in

In return, you are expected to:

- Use the lab resources, TA time, and instructor time when you need them
- Perform work conscientiously, neatly, and in a timely manner
- Be considerate to your fellow students
- Not make lame excuses

Topics

- INTRODUCTION: Programs, interpreters, and translators; Analysis-synthesis model of translation; Examples of translators; Why study compilers; Issues in compiler design.
- PROGRAMMING LANGUAGE SPECIFICATION: Definitions of syntax, semantics and pragmatics; In-depth study of syntactic specifications; A brief look at semantics; Case studies: the definitions of Iki, Carlos, and Roflkode.
- COMPILER STRUCTURE: Logical and physical organization of a compiler project; Case study: a look at a compiler for the Iki language with three targets: JavaScript, C, and x86-64 assembly; Overview of development with Node.js.
- LEXICAL AND SYNTACTIC ANALYSIS: Foundations: Set theory and formal language theory; Syntax and microsyntax (Syntax diagrams, EBNF, PEGs); Regular expressions and context free grammars; Programming with regular expressions; Scanning and parsing fundamentals; Bottom-up vs.top-down parsing; Recursive descent parsing; Packrat parsing.
- SEMANTICS AND SEMANTIC ANALYSIS: Static vs. dynamic semantics; Names, scopes and bindings;

Object lifetimes; Run-time system organization (the stack, the heap, garbage collection); Classification of semantic objects for an intermediate representation; Symbol tables and their relationship to semantic objects; Symbol table implementation: binary trees vs. hashing; Attribute Grammars.

- IMPLEMENTING A SEMANTIC ANALYZER: Compilation techniques for control flow (expressions, assignment, sequencing, selection, iteration, recursion), declarations, basic types (arrays, records, strings, pointers) and type checking, subroutine calls and parameter passing; implementation of higher-order functions and closures; Heap management.
- INTERMEDIATE LANGAUGES AND VIRTUAL MACHINES: What is an intermediate language; Kinds of intermediate languages; Virtual machines; VM examples: The JVM, the CLR, and modern JavaScript engines, e.g., V8, JägerMonkey, Carakan; LLVM.
- THE x86 ARCHITECTURE AND ASSEMBLY LANGUAGES: Block level architecture; Machine language; Assembly languages; Assembly language programming techniques.
- CODE GENERATION: Back-end compiler architecture; Code generation issues (Basic blocks, Traces, Liveness Analysis, Register Allocation); Construction of executable code and libraries.
- CODE OPTIMIZATION: Overview of optimization; Data Flow Analysis; Peephole Optimizations; Constant Folding, Common Subexpression Elimination, Copy Propagation, and Strength Reduction. Global optimization: Loop optimizations; Induction Variable elimination, Optimizing procedure calls: Inline and closed procedures. Machine-Dependent Optimization: Pipelining and Scheduling; Processor specifics.
- MORE ON PARSING THEORY: LL(k) Parsing - Parse tables, properties of LL(1) grammars, transformations to make grammars LL(1) Shift-Reduce Paradigm; LR(k) Parsing: LR, SLR and LALR Implementations; Context free language hierarchy; More PEG theory.

University Information

All students will want to acquaint themselves with the useful information found in the following sources:

- LMU Academic Requirements and Policies (Pay particular attention to the sections on academic dishonesty)
- LMU Disability Support Services Office
- LMU Student Codes and Policies
- LMU Learning Resource Center

Ray Toal • *Computer Science Division* • *College of Science and Engineering* • *Loyola Marymount University*