Instructor: Dr. Pedro C. Diniz, e-mail pedro@isi.edu
Lectures: Thursday, 8.00 – 10.50 AM, RTH 217, phone: (213) 740 4518
Office hours: Thursday, 11.00 to 12.00 AM (just after class), Office: SAL 213

Description: This course is intended to give the students a thorough knowledge of compiler
design techniques and tools for modern computer programming languages. This
course covers advanced topics such as data-flow analysis and control-flow
analysis, code generation and program analysis and optimization.

Prerequisites: Students should be familiar with the concepts in theory of computation (e.g.,
regular sets and context-free grammars); design and analysis of algorithms (e.g.,
asymptotic complexity, divide and conquer and dynamic-programming
techniques); and have strong programming skills using dynamic, pointer-based
data structures either in C or C++ programming languages. Students should also
be familiar with basic concepts of imperative programming languages such as
scoping rules, parameter passing disciplines and recursion.

Prerequisites include the USC courses listed below. If you do not meet these
requisites please contact the instructor to determine if you should take the class.

- CSCI 455x (Introduction to Programming Systems).
- CSCI301 - Theory of Computation or CSCI430 - Automata Theory
- CSCI420 - Concepts in Programming Languages

Assignments: We expect to have 5 individual homework assignments throughout the course in
addition to a first in-class exam (called mid-term) and a second in-class exam
(named final). Home works are due at the beginning of the class on the due date. Home works are organized and structured as preparation for the mid-term and
final exams, and are meant to be a studying material for both exams.

Late home works will not be accepted as the solutions are posted within hours of
the end of the class they are due. You need to turn in a signed hardcopy of your
home works (exception for remote students). In extreme circumstances please ask
a fellow student to turn in your homework in a sealed envelope.

There will also be 2 individual programming projects in this class. The goal of
these projects is to have the students experiment with very practical aspects of
compiler construction and program analysis. These projects will cover topics we
study in class and with the vast code basis available at the class web site you
should have no problem completing each of them in under a week.

In this class we are using some fairly old tools from the UNIX repertoire of tools,
namely a Lexical Analyzer generator (lex/flex) and a Syntactic Analyzer generator
(yacc/bison). I'm often asked why using such "dated" tools. The answer is simple.
Many of the more recent tools (with a few notable exceptions) have at their core
the same parsing and finite-state-machine (FSM) engines these older tool implement. If you learn and understand the basic technique exposed by these more "primitive" tools, you will be able not only to quickly use other more modern tool, but also understand better that lies "underneath the hood". Yet, and as the focus of the course is not on the classical lexical and syntactic analysis (we'll cover those in the first couple of lectures) we will be providing the students a complete implementation of the front-end of the prototype compile you will build upon. You do not need to learn Lex/Yacc but you need to understand what they do.

This year’s projects focus on the implementation of a very proper sub-set of a popular imperative programming language, the C language. You will be asked to develop a simple parser (lexical and syntactic analyzer) for this simple subset and generate an interpretable assembly code as described below. Tentatively, this year's project consists of:

1. Declaration Analysis, Type Checking and Storage Allocation
2. Code Generation with Register Allocation

For each project we will have a specific set of test programs some of which we make available to you before hand for testing. We use additional test programs for our own evaluation and whose output will be compared against your project’s outputs. At the end of each programming assignment we will make available an implementation that you might want to use for the following programming project in case there are dependences between projects.

For the last programming project you will have the ability to observe the operation of your generated code using an existing MIPS processor simulator (MARS v4.4) freely available and distributed as a Java jar package. We will cover in class the basic MIPS instruction set architecture and the use of this simulator, which you will be able to use to validate your generated code in the second project.

Grading:
We will assign 20% of this class grade to homeworks (some might have larger percentages due to difficulty), 30% for the programming projects (10% first, 25% second), 15% for the mid-term and 30% for the final exam. The Final exam is not comprehensive.

Important Note on Grading:  
It is absolutely against university policy for faculty to grant appeals for grade change for any reasons other than mistakes in grading. The student’s request is completely inappropriate and - even if faculty is sympathetic - it is not possible within the rules to grant the requested change.
The recommended textbook for this class is the second volume listed below. I’ll draw most of the material from this textbook but will occasionally draw some material from the other references. Occasionally, I’ll distribute supplemental materials in class to cover topics that are not easy to find elsewhere.

Title: Compilers: Principles, Techniques and Tools
Authors: A. Aho, M. Lam, R. Sethi and J. Ullman
Publisher: Addison-Wesley 2nd edition
Year: 2007

Title: Engineering a Compiler
Authors: Keith Cooper and Linda Torczon,
Publisher: Morgan-Kaufman Publishers, 2nd ed.
Year: 2010

Title: Advanced Compiler Design and Implementation
Author: S. Muchnick,
Publisher: Morgan-Kaufmann Publishers

Class Material: Please gain access to course-related material on-line at the class web site (http://www.isi.edu/~pedro/Teaching/CSCI565-Spring14/) or using the Distance Education Network (DEN) Blackboard facility at den.usc.edu. All students must register with DEN on website before gaining class web-page access.

Do not mail questions to the instructor. Instead, post all questions on DEN’s Blackboard. Your questions will be answered as promptly as possible, and typically within 24 hours after being posted.

Lectures: Below is a tentative description of the contents of each lecture excluding the midterm and final lectures (the first two lectures will be at a fairly fast pace as they are review of assumed known material). We may change the order to accommodate the materials you need for the projects. The first two lectures are meant to be review of material you have covered elsewhere. As such the pace is going to be higher than normal. Do not panic. We are here to help you.

Lecture 1: Introduction; Lexical Analysis, Regular Languages and Finite Automata
- Introduction to compilation and programming languages
- Typical compiler structure and its internal phases
- Lexical Analysis: Its goal and input/output
- Finite Automata Theory and Regular Expression.
- Automatic Recognition of REG (example of Lex/Flex)
- Limitations of Regular Expressions.

Lecture 2: Syntactic Analysis, Context-Free Languages
- Syntactic Analysis: Its goal and input/output
- Context Free Languages and PDA.
- Parsing Algorithms for LR Grammars; Shift-Reduce Techniques
- Automatic Recognition of CFL (example of Bison/Yacc)
• Ambiguity and how to avoid it.

Lecture 3: Semantic Analysis and Syntax-Directed Translation
• Limitations of Syntactic Analysis.
• Type Checking.
• S-Attributed Grammar and L-Attributed Definitions.
• Translation to Syntax Trees and DAGs.
• Evaluation Schemes.

Lecture 4: Intermediate Representation and Intermediate Code Generation
• Introduction to Intermediate Code Generation, High-level Overview
• Intermediate Representation: Three-Address Instructions
• Syntax-Directed Translation Schemes for Code Generation
• Code for Expressions, Assignment, and Arrays
• Boolean and Relational Operators, Conditionals, & Control flow
• Backpatching

Lecture 5: The Procedure Abstraction, Run-Time Environment and Storage Allocation
• Introduction, discussion of control abstraction, name spaces, and external interfaces
• Name Spaces & Symbol Tables.
• Symbol Tables & Storage Layout
• Allocating Storage and Establishing Addressability
• Run-time Structures for Object-oriented Languages
• Finishing up Run-time Structures for OOLs.

Lecture 6: Code Generation and Instruction Selection
• Introduction to Code Generation, High-level Overview
• Basic Blocks
• Code for Expressions, Assignment, and Arrays in DAGs and Basic Blocks
• Boolean and Relational Operators, Conditionals, & Control flow
• Procedure Calls & Dispatch, again

Lecture 7: Register Allocation
• The Importance of Register Allocation and Assignment
• Top-Down Simple Allocation Algorithm and Its Limitations
• Global Register Allocation via Graph Coloring

Lecture 8: Introduction to Optimization
• Control-Flow Analysis: Dominators and Post-Dominators
• Finding Loops and Loop Invariant Code
• Strength Reduction
• Constant Propagation and Constant Folding
• Basic Induction Variable Recognition

Lecture 9: Data-Flow Analysis
• What is Data-Flow Analysis?
• Reaching Definitions and Live Variables Analysis
• Forward and Backwards Problems
• Available Expressions
• Generalization of Data-Flow Analysis

Advanced Topics

Lecture 10: Instruction Scheduling
• Local Instruction Scheduling; List-scheduling algorithm
• Architectural complications
• Beyond basic-blocks

Lecture 11: Transformations for the Memory Hierarchy
• Introduction and Rationale: Registers versus Caches
• Loops and Locality: Temporal and Spatial Locality
• Scalar Replacement and Register Pressure
• Loop unrolling, Unroll-and-Jam and Tiling in Scientific Codes
• Case Study: Matrix-vector and Matrix-vector Multiplication