ECE 573: Compilers and Translation Engineering

Spring 2011

Lectures: Tuesdays and Thursdays, 9:00–10:15, EE 224
Course web page: http://www.engineering.purdue.edu/~milind/ece573/2011spring

Instructor: Milind Kulkarni (milind@purdue.edu)
Office: EE 324A
Office Hours: Tuesdays and Thursdays, 10:30–11:30, or by appointment

Course Description: This course covers the tools and techniques required to build a compiler for computer programs. The basics of compiler front ends (which translate a source program into an internal representation) will be covered in the first third of the course. The remainder of the course will discuss "back-end" compiler techniques, such as register allocation, instruction scheduling and program analysis.

Prerequisites: A solid grounding in C++, Java or some other high level programming language. A working knowledge of data structures and assembly language. A grasp of the basic fundamentals of computer organization and architecture.

Textbook: Fisher and LeBlanc, *Crafting a Compiler in C*. While I highly encourage you to purchase the textbook (it will be especially useful for the project), it is not required. The course will primarily be taught from lecture notes that will be posted online.

Course Outcomes: At the end of the course, you will be able to:

1. Describe and explain the terminology, representation and use of formal languages and grammars;
2. Describe and explain the terminology and techniques of lexical analysis, parsing, semantic actions and code generation;
3. Design and implement a compiler for a small language based on their knowledge of the previous two points.

More specifically, at the end of the course, you will be able to:

- Explain the various passes of a compiler (scanners, parsers, semantic actions and code generation, register allocation and basic optimizations) and how they relate to the overall compilation process.
- Explain and implement the algorithms for each of these processes.
- Be able to implement each of these passes and integrate them into a full compiler.
- Explain program analysis techniques that are used for code optimization, such as dataflow analysis, def-use analysis, liveness analysis, etc.

- Describe basic code transformations and their application to program optimization.

Course Grading:

55% — Tests (2 midterms @15%, 1 final @25%)
40% — Project
5% — Class participation

The course will be curved, although the contributions of each portion of the course will be fixed, as per the above scale. My intention is that students receiving an A or higher can expect to pass the CE-2 QE question without much difficulty, while those receiving lower grades may need extra preparation.

Problem Sets: There will be 8–10 problem sets given over the course of the semester. These problem sets *will not be graded*, but will serve as useful practice for the types of problems that will appear on the midterms and final.

Tentative Course Topics: Below is the list of topics that will be covered in this course, and a rough estimate of how long we will spend on each. There is 1 week of slack in the schedule, and 1 week allocated for exams.

Topic	# of weeks	Reading
Structure of a compiler (introduction and overview)	0.5	Chapters 1 & 2
Scanning	0.5	Chapter 3
Parsing (recursive descent, overview of shift-reduce)	1	Chapters 4–6
Semantic routines (building a symbol table and AST)	1.5	Chapters 7–12
Semantic routines (for functions)	1	Chapter 13
Code generation (generating three-address code from AST, peephole optimizations, etc.)	1	Chapter 15
Instruction scheduling	0.5	Handouts, notes
Register allocation	0.5	Handouts, notes
Program optimizations (code motion, strength reduction, etc.)	1	Handouts, notes
Control flow analysis (building a CFG)	0.5	Handouts, notes

Topic	# of weeks	Reading
Dataflow analysis (lattice theory, specific DFAs)	2.5	Handouts, notes
Pointer/alias analysis	0.5	Handouts, notes
Parallelism (dependence analysis, optimistic parallelization)	2	Handouts, notes

Exams and midterms: Exams will be in class, open book and open notes. The tentative dates and topics for the exams are below.

Exam topics and dates:

- Midterm 1 — Scanning, parsing, semantic routines (Thursday, February 17th)
- Midterm 2 — Code generation, register allocation, instruction scheduling, peephole optimizations, loop optimzations (Thursday, March 31st)
- Final — Cumulative, with emphasis on dataflow analysis, pointer analysis and dependence analysis (TBA)

Project: The bulk of your grade will be determined by a course project. This project involves implementing a full-fledged optimizing compiler for a simple language. You may implement your project in any language, although using a high-level language such as C++ or Java will probably make your life easier.

The project consists of multiple steps, each of which will be graded separately. However, each step builds on the results of previous steps, so it behooves you to ensure that each step works properly. The bulk of your project grade (60%) is based on the performance of your final compiler on several predetermined test programs; the intermediate steps will, together, constitute 30% of your project grade; the final 10% will be based on your compiler's performance on several undisclosed test programs.

The project steps (and due dates) are as follows:

Step	Description	Due date
0	Verify that you can properly turn in project steps, choose project partner	Friday, Jan. 14th
1	Use Lex, other tools, or a hand written lexer to scan a program written in the language given on the course web page. The output of this step will be one line for each token encountered, which contains the token and its type (an integer value that you decide on).	Friday, Jan. 21st

Step	Description	Due date
2	Using YACC, or another parser generator, write a parser for the grammar for the project language. Lexical analysis will be done by project step one. The output of this step is a parse tree (one symbol per line, indented by the depth of the tree). Parser error recovery does not need to be implemented. However the parser must stop correctly upon a detected syntax error.	Friday, Feb. 4th
3	Implement the semantic actions associated with variable declaration. The symbol table entry object has an identifier name field and a type field. In particular, when an integer or float variable declaration is encountered, create an entry whose type field is integer (or “float”) and its return type to N/A. Functions declarations do not need to be handled at this time. The string corresponding to the identifier name can either be part of the identifier entry in the symbol table, or can be part of an external string table that is pointed to by the symbol table entry. When a new scope is encountered, a new symbol table should be created. Thus, when entering a function, or the body of an IF, ELSE, WHILE or FOR loop a new symbol table needs to be created, and the symbols declared in that scope added to the symbol table. The output of your compiler should be a listing of the type of scope the symbol table is for (an IF, ELSE, FOR, FUNCTION or PROGRAM), the name, if any of the scope, and the symbol table entries, with each line containing the variable name and its type.	Monday, Feb. 21st (later to compensate for exam the previous week)
4	Process assignment statement and expressions. For this step, expressions will only appear in assignment statements. In this step an internal representation (IR) of the program will be formed. Build the IR by first constructing an Abstract Syntax Tree (AST), then producing an IR from that tree. This IR consists of a list of nodes, one node per IR statement. The nodes will appear in the list in the order they are generated by the semantic routines. The semantic actions for each sub-expression will produce code of the form <lhs>=<1st operand> <operation> <2nd operand>. The node will contain this information and pointers to the operation that is immediately reachable after this node. Implement the **read** and **write** statements.	Friday, Mar. 4th

Step	Description	Due date
5	Implement semantic actions for **if** and **for** statements. This includes creating IR nodes for the statements and writing a pass that traverses the IR and generates code executable on the Tiny simulator. The output for this step will be the output from running your program on the Tiny simulator. I would recommend that you be able to handle if statements by the 18th, with only do while statements remaining for the following week.	Monday, Mar. 21st (later due to Spring Break)
6	Implement semantic actions for subroutine definitions and subroutine invocation. I would suggest creating a separate IR and symbol table for each subroutine. As with the previous step, the output from this step will the be Tiny simulator output for the program	Friday, Apr. 8th
7	Perform an intraprocedural liveness analysis. You will have to construct a control flow graph (CFG) according to the procedure discussed in class, then write a dataflow analysis to compute liveness. Perform register allocation using the results of this liveness analysis, using the graph coloring algorithm described in class. The output will be the Tiny simulator output. This version of the Tiny simulator provides a limited number of registers.	Monday, Apr. 25th
8	Turn in the final version of your project. This gives you a chance to correct any remaining bugs and test your compiler	Saturday, Apr. 30th

Group work policy: You can (optionally) work on the project in teams of 2. You must decide if you want to work on a group *prior* to turning in Step 0. If you choose to work in a group, turn in Step 0, indicating who is in your group. Once you choose a partner, you must continue to work with this partner for the remainder of the project.

Late submission policy: Except for medical and family emergencies (accompanied by verification), there will be *no extensions* granted for project submissions. Late submissions will be scaled according to lateness, docking 10% from your score per day late, up to a maximum of 50%. Submissions more than 5 days late will be assigned a score of 0.

Campus Interruptions: In the event of a major campus emergency, course requirements, deadlines and grading percentages are subject to changes that may be necessitated by a revised semester calendar or other circumstances. In such an event, information will be provided through the course website and email.

Academic Honesty: There are no group assignments in this course—you are expected to complete all assignments by yourself. However, you are allowed to discuss general issues with other students (programming techniques, clearing up confusion about requirements, etc.). You may discuss particular algorithmic issues on the newsgroup (but do not copy code!). *We will be using software designed to catch plagiarism in*

programming assignments, and all students found sharing solutions will be reported to the Dean of students.

Punishments for academic dishonesty are severe, including receiving an F in the course or being expelled from the University. By departmental rules, all instances of cheating will be reported to the Dean. On the first instance of cheating, students will receive a 0 on the assignment; the second instance of cheating will result in a failure of the course.