ECE 573: Compilers and Translation Engineering

Fall 2009

Lectures: Tuesdays and Thursdays, 9:00–10:15, EE 115
Course web page: http://www.engineering.purdue.edu/~milind/ece573/2009fall
Course news group: purdue.class.ece573

Instructor: Milind Kulkarni (milind@purdue.edu)
Office: MSEE 344 (temporary)
Office Hours: Tuesdays and Thursdays, 10:30–11:30, or by appointment
TA: Lin Yuan (yuanl@purdue.edu)
TA Office Hours: TBD

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.

Textbook: Fisher and LeBlanc, *Crafting a Compiler in C*. Lectures and class notes will supplement the textbook.

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

- Explain the structure of a compiler including the various passes
- Explain and implement each of those passes
- Explain program analysis techniques as well as the optimizations they enable

Course Grading:

40% — Tests (2 midterms @10%, 1 final @20%)
50% — Project
10% — Class participation

Pandemic Preparedness: 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.

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
Dataflow analysis (lattice theory, specific DFAs)	2.5	Handouts, notes

Topic	# of weeks	Reading
Pointer/alias analysis	0.5	Handouts, notes
Parallelism (dependence analysis, optimistic parallelization)	2	Handouts, notes

Exams: All exams are in-class exams, and will be held on the dates given below, unless otherwise announced. Exams are open-book and open-notes. Note that while each exam mostly covers certain topics, there may be information from earlier in the course on each; especially, the final exam will be comprehensive. If you cannot make it to an exam for some reason, let me know as soon as possible so we can schedule a make up for you.

Exam topics and (tentative) dates:

- Midterm 1 — Scanning, parsing, semantic routines (September 29th)
- Midterm 2 — Code generation, register allocation, instruction scheduling, optimizations (November 3rd)
- Final — Program analysis, dependence analysis (TBD)

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
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, Sept. 4th

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.	Sunday, Sept. 20th
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, WHILE, 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.	Sunday, Sept. 27th
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. 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 node will have the following fields for assignment statements: successor edges; node type; left-hand side variable (a symbol index); operation type; first and second operands (symbol indices.) Eventually these nodes will be part of a flow graph – make allowances for lists pointers to successor and predecessor nodes. 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.	Sunday, Oct. 11th

Step	Description	Due date
5	Implement semantic actions for **if**, **while** 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.	Sunday, Oct. 25th
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	Sunday, Nov. 15th
7	Perform common subexpression elimination. This step consists of performing an intraprocedural available expression analysis and then CSE using the results of your available expression analysis. Perform register allocation. You may use the allocator in the book, or one of the other ones described in the handouts. The output will be the Tiny simulator output.	Sunday, Dec. 6th
8	Turn in the final version of your project. This gives you a chance to correct any remaining bugs and test your compiler	Saturday, Dec. 12th