ECE573 Introduction to Compilers & Translators

Tentative Syllabus

Fall 2005 Tu/Th 9:00-10:15 AM, EE 115

Instructor	Prof. R. Eigenmann
Tel	49-41741
Email	eigenman@ecn
Office	EE334C
Office Hours	Tu 10:15-11:30 / Fr 8:30–10:00AM (may change)
Secretary:	Connie Boss, EE339
Course Web Page:	http://www.ece.purdue.edu/~eigenman/ECE573/

Prerequisite: ECE 468 (Compilers) or equivalent. If you have not taken an undergraduate, equivalent compiler course: yau may take this class, if you feel you are able to invest significant time into catching up with the topics of scanning, parsing, and semantic processing. Only an overview will be given on these topics. They will be tested in the midterm exam.

Prerequisite by Topic: (1) Basic understanding of computer organization, (2) Knowledge of compiler techniques, including parsing, semantic processing, and basic code generation, (3) Substantial implementation experience with a modern programming language.

Engineering Science: 2.0 credits

Engineering Design: 1.0 credit

Course Description: This course presents the concepts needed to design and implement compiler code generation and optimization techniques. The course focuses on program analysis and transformation techniques that enable common optimizations in state-of-the-art compilers for modern languages. In a class project, students will construct an optimizing compiler for a simple language.

Course Outcome: After completing this course, the student will have demonstrated:

- an understanding of the terminology and internal representations of compiler optimization passes (ABET categories 1,2,3,a,c)
- an understanding of the concepts and techniques for program analysis and optimization (ABET categories 1,2,3,a,c,e)
- an ability to design and implement a compiler for a small language based on their knowledge of (1) and (2) (ABET categories 1,3,4,a,b,c,e,g,k)

Course Format:
Class: Two 75-minute lectures per week.
Exams: A written in-class exam and a written final exam.
Project: Several project assignments to construct a compiler including several optimization techniques for a simple language. Computer Languages Used: Typically C, C++, Java

Text: Fischer and LeBlanc, Crafting a Compiler with C, Benjamin/Cummings, 1991, ISBN 0-8053-2166-7 (A new edition of this book is currently in preparation. Information is available at www.cs.wustl.edu/~cytron/cac/. Students are encouraged to evaluate this material. "Extra credit" is available for written evaluations of this edition.
Course notes and research papers will be used.

Additional References: (1) Cooper and Torczon, Engineering a Compiler, Morgan Kaufmann. (2) Muchnick, Advanced Compiler Design&Implementation, Morgan Kaufmann,ISBN 1-55860-320-4. (3) Aho, Sethi and Ullman, Compilers: Principles, Techniques and Tools, 1986.

Course Grading:

Tests	50%.	(20% midterm exam, 30% final exam)
Class participation	10%.	
Projects	40%	

Regrading policy: any information that you feel will affect your grade, including grade disputes and emergencies that prevent you from attending class, must be sent by email to the instructor. In general, the instructor will not respond to these notes or change your intermediate grades. However, the information will be factored into your final class grade. A regrade request will lead to the re-evaluation of the entire exam or project, which may lead to a higher or a lower grade.

You must check with the Instructor or TA before the end of the last week of classes that the grades on file agree with your records. If you don't, you may receive a grade other than the one you have earned.

Academic Honesty:
You are expected to do all programming and homework assignments on your own or within the designated group. You may not copy files or solutions from other students/groups or have friends do all or part of assignments for you.

You may discuss concepts and ideas with other students. You may answer general questions about programming language rules, help someone interpret an error message, or locate a bug. In general if the instructor or TA would provide a particular form of assistance, you may provide it too. When in doubt, check with the instructor or TA. It is your responsibility to protect your work from access by others. Your project code will be compared against other projects and past project submissions, using advanced software that checks for similarities. It can easily detect programs that differ merely by renamed program variables, reordered source lines, or edited comments.

Punishment for academic misconduct is severe, including receiving an F in the course, or even being expelled from school. All cases of cheating will be reported to the Dean's office.

Course schedule:

```
30 lectures (eff. 15 weeks) plus final exam

                                    week                        submit project step
Monday  Aug 22 Sem starts    T R    1                            Fridays @ 11:59PM
           29                T R    2                                    1
        Sep  5               T R    3                                    2
           12                T R    4                                    3
           19                T R    5
           26                T R    6                                    4
        Oct  3               T R    7
           10                . R    8       Tu: October Break            5
           17                T R    9       Oct 20: Midterm Exam
           24                T R    10                                   6
        Nov  7               T R    11
           14                T R    12                                   7
           21                T .    13      Th: Thanksgiving
           28                T R    14
        Dec  5               T R    15                                   8
           12                T R    16                                   9

        Dec 15      final exams week       Final exam date TBA
```

Tentative Course Outline:

Course Outline:

	Topic	#weeks	Reading
1.	Structure of a compiler	1	Chapters 1,2
2.	Overview of Scanning, Parsing, Semantic Processing	2	Chapters 3–13
3.	Overview of Program optimization	1	Handouts
4.	Local analysis and optimization	3	Chapter 15, handouts
5.	Global program analysis	3	Chapter 16, handouts
6.	Interprocedural analysis	2	Handouts, papers
7.	Global Optimizations	3	Handouts, papers
	Exams	1	

Reading: Reading the textbook sections corresponding to the presented lecture material is an essential part of the course. Note that Chapters 3–13 cover prerequisite material.

Exams: There will be a midterm exam during regular class hours. The final exam schedule will be known later in the semester and will be announced on the course web page. All exams are comprehensive. The first exam also covers prerequisite material (Overall compiler organization, scanning, parsing, semantic processing.) The midterm exams may include questions about the syllabus. All exams are "open textbook and notes". However it is strongly recommended that you page through the textbook as little as possible during the exam. It can slow you down significantly. So, use the textbook only in "emergencies".

Class Projects: In a group of two students you will implement a compiler that includes several optimization passes for a simple language. You will start with a grammar defined on the project web page. You can choose

the implementation language for your project. Groups will be formed in the first lecture and at the end of the second lecture. Students not attending these lectures will be assigned to groups. No TA is available (neither for course material not for the project). Students must work very independently on their projects.

Project submission guidelines will be posted on the course web page.

General implementation guidelines:

- The specification in the project steps below is intentionally kept brief. Examples and Q&A on the course web page will clarify details. If you think a project specification is unclear or ambiguous, ask for clarification *at the beginning of the semester*. No questions can be answered on the day of the project deadline. Make sure you check the course web page before the submission of each project step, so you can take advantage of the latest Q&As posted.
- Your compiler should be able to handle programs up to 1000 lines of code. The use of dynamic data structures is not mandatory. However, if you use C++ or Java as an implementation language, it is highly recommended that you use the available dynamic container types for symbol tables and other data structures.
- The project deadlines are latest, hard deadlines. You need to define your own schedule that fits best. No project extensions will be given. Keep in mind that ECN machines may be unavailable at times, and may be slow due to overloads before project deadlines. Machine downtimes of less than a day does not justify a late project submission.

Project grading policy: Although each project step will be graded, most weight will be given to your final compiler. The final compiler is expected to generate code that executes correctly on the TINY simulator. 60% of the project grade can be achieved if your final compiler executes the predefined test programs correctly. Additional 10% are given if your compiler correctly executes several undisclosed test programs. The remaining 30% of the project grade are assigned to the intermediate project steps.

Project Schedule:

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 defined by you.)
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.
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. When a scope is exited, the symbol table is destroyed. The program symbol table is destroyed at the end of the entire program. 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.
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. 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.
5	Implement semantic actions for subroutine definitions and subroutine invocation. create 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
6	Perform register allocation. You may use the allocator in the book, or one of the other ones described in the text. The output will be the Tiny simulator output.
7	Perform code scheduling before register allocation. Use the Cooper/Torczon Algorithm, preceded by virtual register allocation, if necessary. The output will be the Tiny simulator output.
8	Implement intraprocedural, global common subexpression elimination. This involves data flow analysis for available expressions, followed by local CSE.
9	Thoroughly test your compiler for submission of the final project.