ECE 468: Intro to Compilers and Translation Systems Engineering

Fall 2017

Lectures: Mondays, Wednesdays and Fridays, 1:30–2:20, WALC 3087
Course web page: https://engineering.purdue.edu/~milind/ece468/2017fall/
Piazza discussion page: https://piazza.com/purdue/fall2017/ece468/home

Instructor: Milind Kulkarni (milind@purdue.edu)
Office: EE 324A
Office Hours: Mondays 2:30–4pm; Thursdays 10–11:30am
TA: Chris Wright (wrigh338@purdue.edu)
TA Office Hours: Wednesdays 5–7pm; Fridays 3–5pm, EE 207

Course Description: The design and construction of compilers and other translators. Topics include compilation goals, organization of a translator, grammars and languages, symbol tables, lexical analysis, syntax analysis (parsing), error handling, intermediate and final code generation, assemblers, interpreters, and an introduction to optimization. Emphasis is on engineering a compiler or interpreter for a small programming language–typically a C or Pascal subset. Projects involve the stepwise implementation (and documentation) of such a system.

Prerequisites: ECE 36200, ECE 36800.

Textbook: Fisher and LeBlanc, *Crafting a Compiler in C*. Lectures and class notes will form the bulk of the material for the class. The textbook will primarily be useful for the project.

Course Outcomes: At the end of the course, a student who has successfully met the course objectives will be able to:
1. Describe and explain the terminology, representation and use of formal languages and grammars [a];
2. Describe and explain the terminology and techniques of lexical analysis, parsing, semantic actions and code generation [a, e];
3. Design and implement a compiler for a small language based on their knowledge of the previous two points [a, b, c, e, k].

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, reaching definitions analysis, liveness analysis, etc.
- Describe basic code transformations and their application to program optimization.

Course assessment: The achievement of course objectives will be assessed through a combination of tests (2 midterms and a final) and a substantial course project. The tests will assess students' achievement of the first two outcomes, while the project will assess students' achievement of the third outcome.

Course grading: Grades will be assigned as follows:
- 45% — Tests (2 midterms @10%, 1 comprehensive final @25%)
- 40% — Project
- 10% — Problem sets (6–8 total)
- 5% — Class participation

There may be a constant curve (*i.e.*, all grades will be increased by a fixed amount) for individual exams at the instructor's discretion. Your course grade will be determined using an absolute scale: 97–100: A+; 91–97: A; 88–91: A-; and continuing down.

Problem sets: There will be 6-8 problem sets, approximately one every two weeks. Each set will be posted online on Monday and will be due in class the next Monday (you may submit earlier via email). The problem sets will be graded on a 0-1 system: 1 point for turning in a serious attempt, 0 points for not turning in a set or not making an honest attempt at the problems. While the problem sets factor in to your grade, the primary benefit is for your own study; midterm and exam questions will often be in the same format as the questions on the problem sets.

Exams and midterms: We will have evening midterms. The midterms and final are *open book and open notes*.

Exam topics and dates:
- Midterm 1 — Scanning, parsing, semantic routines (October 5th, 6:30–7:30pm, FRNY G140)
- Midterm 2 — Code generation, register allocation, instruction scheduling, peephole optimizations, loop optimizations (November 9th, 6:30–7:30pm, FRNY G140)
- Final — Cumulative, with emphasis on dataflow analysis, pointer analysis and dependence analysis (TBD)

Course Discussion: This term we will be using Piazza for class discussion. If you have questions about the course or the project, I encourage you to post them on Piazza. It's a shared discussion forum, where your question can be answered by myself, the TA or your fellow students! Piazza will be the primary method for disseminating course announcements, so *you must have a Piazza account*.

Note that to get full class participation points, you must a) sign up for Piazza, and b) post at least one public question or answer pertaining to the class.

Find our class's Piazza page at: **https://piazza.com/purdue/fall2017/ece468/home**

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
Pointer/alias analysis	0.5	Handouts, notes
Parallelism (dependence analysis, optimistic parallelization)	2	Handouts, notes

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 majority 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. Due dates are subject to change at the instructor's discretion. *Steps are due at 11:59 PM* on the date specified:

Step	Description	Due date
0	Verify that you can properly turn in project steps (see instructions below), choose project partner	Friday, Aug. 25th
1	Use ANTLR, 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).	Wednesday, Sep. 6th
2	Using ANTLR, 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 "yes" or "no" answer, indicating if the input program matches the grammar. Parser error recovery does not need to be implemented. However the parser must stop correctly upon a detected syntax error. | Friday, Sep. 15th |

Step	Description	Due date
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, or WHILE 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.	Friday, Sep. 29th
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, Oct. 13th
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.	Wednesday, Nov. 1st
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, Nov. 17th

Step	Description	Due date
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 either bottom-up register allocation or graph coloring. The output will be the Tiny simulator output. You will use a new version of the Tiny simulator that provides a limited number of registers.	Monday, Dec. 4th
8	Turn in the final version of your project. This gives you a chance to correct any remaining bugs and test your compiler	Friday, Dec. 8th

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, you must indicate it in your course questionnaire. *Once you choose a partner, you must continue to work with this partner for the remainder of the project.*

Project submission: Because this project requires writing a large amount of code, and each step of the project will depend on previous steps of the project, we will use version control (git) and Github to manage projects and submissions (see the project webpage for more details):

1. You and your partner should each create a git account (if you don't have one already)
2. Click the following link to set up the repository for your project: **https://classroom.github.com/g/fQrSIYuO**
   a. This will be the repository you will use for the entire semester
   b. If you are the first member of your group to set up the repository, create a team for you and your partner
   c. If your partner has already created a team, join their team
3. Make sure you fill out the course questionnaire so we know your team name: **https://goo.gl/khKtaQ**
4. Use *tags* to tell us which versions of your code to grade (see the submission instructions on the website)

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 beyond the instructor's control. In such an event, information will be provided through the course website and email.

Academic Honesty: Unless expressly allowed, 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 Piazza (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.