ECE351 Course Outline, Winter 2018

Derek Rayside

January 11, 2018

[DRAFT until Jan 12]

drayside@uwaterloo.ca

1 Introduction

Welcome to ECE351! This is an introductory compilers course. The calendar description lists a number of topics:

> Programming paradigms, compilation, interpretation, virtual machines. Lexical analysis, regular expressions and finite automata. Parsing, context-free grammars and push-down automata. Semantic analysis, scope and name analysis, type checking. Intermediate representations. Control flow. Data types and storage management. Code generation.

1.1 Course Objective

The objectives of the course are two-fold: First, to teach students the theoretical aspects of compilers and language design, and second, to enable students to build a practical rudimentary compiler of their own. Monday lectures will usually discuss theory, and Friday lectures will usually discuss practice.

This course is very hands on, *i.e.*, you will write a circuit synthesizer and a circuit simulator for circuits described in a subset of VHDL. A circuit synthesizer is essentially a compiler for circuits. The lab notes describe these exercises in detail.

VHDL is a very complicated hardware description language to specify circuits. The language for which you will develop a synthesizer is a very small subset of VHDL.

Through the lab you will learn two important skills: First, you will understand how language design can be seen as a general form of problem solving. Second, you will learn how to "implement" a language by constructing an appropriate compiler.

Often language design and problem solving are seen as distinct activities. The point of the lab is to show that language design, in many ways, is the most general way to solve a problem. By most general solution I mean that your solution solves an entire class of problems, not merely one problem or an instance of a problem. For example, suppose that you need to multiply two matrices. The least general, and most direct, thing you could do is to actually multiply the matrices. More generally, you could write a program that multiplies two arbitrary matrices, and then use that program to multiply your two matrices of interest. Most generally, you could design a language that not only allows you to write programs to multiply matrices but also allows you to express programs to do more complex mathematical operations. Your language could be something similar to a suitable subset of Matlab, then write a program in that language that requests your two matrices be multiplied, then execute that program

Solve an entire class of problems by designing a language to describe those problems.

Good engineers do a cost/benefit analysis before designing and building. Good engineers also have have the skills to design a range of solutions.

with the interpreter/compiler you just developed. In order to solve problems at this level of abstraction you need to understand the ideas listed in the course description, and you will need to improve your programming skills.

We will use the term *transformer* or *translator* to describe a program that reads and writes structured text. As you will see, any sufficiently large transformer is actually comprised of many small transformers: the transformation is broken down into steps that typically pass through one or more intermediate forms before the final output is produced. These intermediate forms are typically simpler to process mechanically and less convenient for human use than the original input language.

A compiler is a particular kind of transformer that reads source code and produces assembly or machine code.

For example, the GNU Compiler Collection (GCC) has an intermediate form called RTL (Register Transfer Language). C/C++/Java/Fortran/*etc.* are first translated to RTL, then optimizations are performed on the RTL, and finally assembly code is generated for some particular chip.

2 *Coordinates*

Instructor:	Derek Rayside	DC-2597D	drayside@uwaterloo.ca https://ece.uwaterloo.ca/~drayside/cal/
Lab Instructor:	Not Assigned		
TAs:	Gabe Wong	in lab	gbwong@uwaterloo.ca
	Hari Krishnan	in lab	hgvedira@uwaterloo.ca
	Jakub Kuderski	in lab	jkudersk@uwaterloo.ca
Regular Lectures:	MF 10:00AM–11:20AM	DWE-1501	Theory Mondays, Lab Fridays (usually)
Extra Lectures:	Thu 4:30PM–5:20PM	DWE-1501	See schedule
Theory Tutorials:	F 12:30PM–1:20PM	EIT-1015	Even weeks (see schedule), if needed
Regular Labs:	TWTh 3PM–4:20PM	E2-2356A	http://sun5.vlsi.uwaterloo.ca/~ecepc/TimeTables/
Lab Tutorial:	Thu 4:30PM–5:20PM	E2-2356A	Weeks with a Lab due
Midterm:	Midterm Week	See schedule	https://ece.uwaterloo.ca/electrical-computer-engineering/current-undergraduate-students/academic-planning-and-support/mid-term-examinations/
Reports:	Thursdays 10PM	Git	every week (except midterm week)
Files:		Git	https://ecgit.uwaterloo.ca/
Discussion:		Piazza	https://piazza.com/uwaterloo.ca/winter2018/ece351

Files will be distributed and collected through Git. Each student will have their own private repository. We will have shared repositories for documents and skeleton code. Your Nexus credentials (username and password) should give you appropriate access to the repositories.

3 *Schedule — NEEDS UPDATING FOR W2018*

Lecture topic schedule is subject to change as the term progresses, depending on the class's needs and interests. The deadlines will not change.
'+' indicates date of a potential Friday tutorial

Week		Monday 10am (Theory)	Thursday 4:30pm	Friday 10am (Practice)	
1	Jan 01	*New Year's Day*		Course Intro	
2	Jan 08	Theory [$\mathcal{N}$0]	Finite Automata [$\mathcal{N}$1] Lab0 due 10PM	Lab1 intro (recursive descent) Tutorial 12:30PM (Automata)	 +
3	Jan 15	Program State [$\mathcal{N}$1]	Ambiguity [$\mathcal{N}$2.4] Lab1 due 10PM	Lab2 intro (transformations)	
4	Jan 22	Grammar complexity [$\mathcal{N}$2.0–2.3]	Lab2 due 10PM	Lab3 intro (OOP, equivalence) Tutorial 12:30PM	 +
5	Jan 29	LL(1) analysis [$\mathcal{N}$2.6]	Lab3 due 10PM	Lab4 intro (term rewriting)	
6	Feb 05	Grammar refactoring [$\mathcal{N}$2.5]	Lab4 due 10PM (no late penalty until after Sunday 10PM)	Lab5 intro (Parboiled) Lab6 intro (patterns) Midterm Review 12:30PM	 + +
7	Feb 12	*No class* `https://ece.uwaterloo.ca/electrical-computer-engineering/current-undergraduate-students/academic-planning-and-support/mid-term-examinations/`	*No lab*	*Midterm Week*	
–	Feb 19	*Reading Week*			
8	Feb 26	Midterm results Lab7 intro (visitor, hashing, *etc.*)	Code Review Lab 6 due 10PM	Optimization [$\mathcal{N}$4.0–4.2.2]	+
9	March 5	Optimization [$\mathcal{N}$4.2.3]	Lab7 due 10PM	Lab*X* intro	
10	March 12	Optimization [$\mathcal{N}$4.2.4 + 4.3]	Lab*X* due 10PM	Lab*Y* intro	+
11	March 19	Register Allocation [$\mathcal{N}$5.1]	Lab*Y* due 10PM	Lab*Z* intro	
12	March 26	Garbage Collection [$\mathcal{N}$5.2]	Lab*Z* due 10PM	*Good Friday*	+
13	April 2	Object Allocation [$\mathcal{N}$5.3]	*Wednesday, April 4 10am Logical Friday* Review	*Exam Period*	

Note: Labs *X*, *Y*, and *Z* will be chosen from the labs numbered 8+ in the Lab Manual.

4 *Marking Scheme*

Labs (incl. quizes)	40%
Midterm	10%
Final Exam	50%
Total	100%

Labs are weighted equally, except for LAB0, which has no marks.

Lab marks are based primarily on how your code performs on automated tests.

Lab marks might also be based partly on quizzes administered online. If there are quizzes associated with a lab, then they will be worth 25% of the grade for the lab.

a. You must pass the final exam in order to pass the course. If you do not pass the final exam, then your final mark will be the lower of your earned mark or 48%.

b. You must pass the labs in order to pass the course. If you do not pass the labs, then your final mark will be the lower of your earned mark or 48%.

Note that lab code is cumulative: there are earlier labs that you will need to have solutions for in order to complete later labs. Dependencies are discussed in the Lab Manual. If you are going to skip a lab, be strategic in your choices.

We will cut you some slack for one Git hiccup, subject to a personal interview with course staff.

We recommend that you commit+push frequently — at least at the end of every work session.

c. Late lab submissions are worth half.

- The labs are cumulative, so you need to keep up.
- You may commit+push your work as many times as you want.
- We mark the code on the *master* branch.

d. There might be bonus marks available for class participation: creating new tests, submitting patches, answering forum questions, mentoring other students, scribing lectures *etc.* Any positive contribution outside of the regular marking scheme is open for consideration.

5 *Reference Material*

The Course Notes and Lab Manual are intended to be fairly self-contained. Nevertheless, you may find it useful to read a text book some times. We will make an effort to cross-reference the following three books, which are available in the library:

- ***C**rafting a Compiler*[1]
- *Modern Compiler Implementation in Java*[2]
- *Programming Language Pragmatics*[3]

The tools that we will use are documented largely online.[4]

[1] Charles N. Fischer, Ron K. Cytron, and Richard J. LeBlanc, Jr. *Crafting a Compiler*. Addison-Wesley, 2010

[2] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge, 2004

[3] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 3 edition, 2009

[4] `parboiled.org`

6 Collaboration

Interaction is an essential part of learning for most people. We are going to try a novel structure in ECE351 to facilitate learning collaboratively and honestly. We here define different levels of collaboration that correspond to different maximum grades. Your grade for a lab will be the lower of your earned grade or the cap. For each lab you will declare your level of collaboration. The levels are:

Pre-lab material on the computing environment is excluded from these collaboration restrictions. By all means, please help each other out getting the computing environment working.

Skeleton code provided by course staff is excluded from these collaboration restrictions. You may look at the skeleton code on a computer collaboratively *before* you or your partner start writing your solutions and still declare *verbal* collaboration.

Individual: You discuss the labs with fellow students, perhaps making sketches on a blackboard/whiteboard. No written artifacts leave the discussion and get transferred to your code, i.e., your code is completely written by you.

cap 100%

Partner: You and a partner collaborate. You may look at each other's code. Each student is expected to do all of the typing on his or her computer. Partners are expected to have roughly equivalent skills.

cap 85%

Mentor: A mentor teaches a protégé. The mentor is expected to teach the protégé at the protégé's computer, and only the protégé operates the computer. The protégé does not look at the mentor's computer.

mentor cap 100%
protégé cap 80%
mentor is eligible for bonus points

At every level of collaboration, every student is expected to physically key in every character that he or she commits. Students are not permitted to share electronic files with each other, or with students who have taken this course in the past, except as facilitated by the instructors using the course version control system.

Course staff are excluded from the collaboration caps, *i.e.*, you may ask questions of the staff, they may look at your code, *etc.*, without that imposing a cap on your mark.

We will share test cases this way, for example.

6.1 Collaboration & Other Programming Languages

If you choose to implement the labs in another programming language, besides Java, it will be significantly more work, because you will have to build everything from scratch, without being able to use the skeleton code provided by the course staff.

As dispensation for this extra effort, you may collaborate at the *partner* level without the grade cap. You might also be eligible for bonus marks for tackling a more significant technical challenge.

7 *Classroom Conduct Policy*

New for 2018! Prof Patrick Lam (ECE459) and I are going to try this out. Let's see how it works. The goal is to balance the benefits of having access to computing devices and the internet against the distractions of same.

TIGERS! The human visual system has evolved to perceive saber-toothed tigers in the savannah. Fortunately, tigers are rare in Waterloo, Ontario. Unfortunately, your classmates are still human and hence their attention will be drawn to flashing lights (or Facebook, or movies, or video games) in their peripheral vision. We'd like to encourage everyone to be respectful of their classmates and to not distract them.

PRUDENCE. Wise use of computers and the Internet can be helpful for fully engaging in class. You might want to try out some syntax, or you might want to look up an API call, or you might want to verify your instructor's somewhat outrageous-sounding claim.

tl;dr: paper or text-oriented programs only in first 8 rows of class.

TEXT MODE. Devices in the first 8 rows of class should be operated in text-mode only: command prompt, text editor, IDE — in full screen mode. Paper is always good, of course.

Notifications should be turned off. Nothing moving on the screen. Web browsing should be done with a text-mode browser.

Using the terminal is a good technical skill that complements your learning in this course.

e.g., w3m, lynx, *etc.*

TUNING OUT. There are some old-fashioned ways of tuning out that are less distracting for your classmates than the flashing lights on your new-fangled gizmo:

- Doodling.[5]
- Do homework. On paper. Maybe for another class.
- Read a textbook. Or a novel.
- Knit. Crochet. Those of you feeling in an especially masculine mood can explore needlepoint[6] or macramé.[7]
- Pass notes. Write a joke to your friend. Pass it on.

EXCEPTIONS. You might have a good reason why you want to use GUI programs to support your learning and sit at the front of the class. Just let the instructor know.

ENFORCEMENT. This policy is part of our broader culture of trust and honesty. You are on your honour.

[5] Drawing uses the right side of your brain. Language and mathematics are done on the left side of your brain. So doodling gives your brain something to do, but still leaves the left side of it available to tune back in to class if something interesting happens.

[6] See Roosevelt Grier's book *Rosey Grier's Needlepoint For Men*. Grier was an all-star defensive tackle for the New York Giants and the Los Angeles Rams (both in the National Football League). He is 6'5" tall and weighed 284lbs.

[7] Macramé is what sailors used to do pass the time productively at sea and practice their knots. Make something fancy for your sweatheart.

8 *University Policies*

Intellectual Property: Students should be aware that this course contains the intellectual property of their instructor, TA, and/or the University of Waterloo. Intellectual property includes items such as: ***source code***, course notes, lab notes, questions or solution sets, lecture content (and any audio/video recording thereof), *etc.*

New for 2017! https://uwaterloo.ca/secretariat-general-counsel/faculty-staff-and-students-entering-relationships-
Don't post your code on GitHub!

Academic Integrity: In order to maintain a culture of academic integrity, members of the University of Waterloo community are expected to promote honesty, trust, fairness, respect and responsibility.

http://uwaterloo.ca/academicintegrity/

Code Clone Detection Software: ECE351 encourages a culture of trust and honesty, which the pedagogical literature has shown to be the most effective way to encourage learning and reduce cheating. While all labs in this course are to be done individually, the course collaboration policy provides a variety of options for legitimate engagement and learning with your peers.

The instructors might, at their discretion, use code clone detection software on your lab submissions, to cross-check compliance with the course collaboration policy. Please inform the instructors if you object to mechanical analysis of your code.

MOSS is the most common code clone detection tool for academic use, developed by compiler Prof Alex Aiken at Stanford: https://theory.stanford.edu/~aiken/moss/

AccessAbility: AccessAbility Services collaborates with all academic departments to arrange appropriate accommodations for students without compromising the academic integrity of the curriculum. If you require academic accommodations, please register with AccessAbility Services at the beginning of each academic term.

Needles Hall, Room 1401. https://uwaterloo.ca/accessability-services/

Grievance: A student who believes that a decision affecting some aspect of his/her university life has been unfair or unreasonable may have grounds for initiating a grievance. When in doubt please be certain to contact the department's administrative assistant who will provide further assistance.

Policy 70, Student Petitions and Grievances, §4, http://secretariat.uwaterloo.ca/Policies/policy70.htm

Discipline: A student is expected to know what constitutes academic integrity to avoid committing an academic offence, and to take responsibility for his/her actions. A student who is unsure whether an action constitutes an offence, or who needs help in learning how to avoid offences (*e.g.*, plagiarism, cheating) or about rules for group work/collaboration should seek guidance from the course instructor, academic advisor, or the undergraduate Associate Dean.

http://uwaterloo.ca/academicintegrity/
For information on categories of offences and types of penalties, students should refer to Policy 71, Student Discipline, http://secretariat.uwaterloo.ca/Policies/policy71.htm
For typical penalties check Guidelines for the Assessment of Penalties, http://secretariat.uwaterloo.ca/guidelines/penaltyguidelines.htm

Appeals: A decision made or penalty imposed under Policy 70 (Student Petitions and Grievances) (other than a petition) or Policy 71 (Student Discipline) may be appealed if there is a ground. A student who believes he/she has a ground for an appeal should refer to Policy 72 (Student Appeals).

http://secretariat.uwaterloo.ca/Policies/policy70.htm
http://secretariat.uwaterloo.ca/Policies/policy71.htm
http://secretariat.uwaterloo.ca/Policies/policy72.htm

References

[1] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge, 2004.

[2] Charles N. Fischer, Ron K. Cytron, and Richard J. LeBlanc, Jr. *Crafting a Compiler*. Addison-Wesley, 2010.

[3] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 3 edition, 2009.