Caroline Lemieux – Teaching Statement

I am interested and qualified to teach programming and software engineering topics from the introductory to advanced levels, as well as programming languages and compilers topics. In the classroom, I aim to create an inclusive environment that leaves students with solid theoretical foundations and confidence in their ability to transfer these foundations between different programming languages, frameworks, and application domains. While I came into my first computer science class without any prior coding experience, its focus on the fundamentals of program design made the subject accessible to me. As a math major, what stood out for me was the fascinating bridge computer science created between logic and the concrete. My goal is to help students find their own avenues of interest in computer science.

I have been a teaching assistant for several courses during my undergraduate and graduate studies: UBC's CPSC 110 and UC Berkeley's CS 61A, the introductory programming courses at each university, and UC Berkeley's CS 164, the intermediate compilers course. Each had different learning goals and levels of teaching infrastructure.

Teaching Experience

I have TA'd two introductory programming courses. For CPSC 110 at UBC, I was an in-class TA, meaning that I mostly helped students work through problems in a flipped-classroom setting. For CS 61A at Berkeley, I ran two discussions and two lab sections, as well as a series of extra lectures. Through these experiences, I learned how to ask questions to identify the fundamental misunderstanding that prevented a student from solving a particular problem. With this knowledge, I was able to discuss that fundamental misunderstanding, and build up a series of questions that enabled the student to understand the concept from their own explanations.

In CS 61A, beyond my regular TA tasks, I ran the course's extra lecture series: a 1-hour extra lecture every week for students interested in pursuing further material in a smaller section. While I had some resources in the form of code to guide the lectures, I largely prepared lectures from scratch. To keep lectures interactive and make sure students were engaged, I frequently asked the students small, low-stakes, questions. This generally made it easier for them to ask deeper questions later on. When students seemed reticent to ask or answer questions, I had them discuss issues with a partner. Removing the pressure from them as an individual—the fear of being perceived as wrong or stupid—made them more comfortable to ask deeper questions.

Finally, I TA'd CS 164, the compilers course at Berkeley. This class involves both core theory of compilers (lexing, parsing, code generation, optimization) and basic formal programming languages topics (operation semantics, typing rules). This class was of a much smaller scale, 90-100 students, and so had less infrastructure. In addition to running discussions and office hours, which were similar in style to CS 61A, I also created the discussion worksheets, and created and graded exam questions. Throughout the process of creating course materials, I learned to write problems which revealed key complexities of the topic being taught. Working through these problems naturally brought students to identify their misunderstandings and ask questions that allowed me to effectively address them.

Online learning and interaction. At UBC, I was a teaching assistant for the Coursera offering of CPSC 110, "Introduction to Systematic Program Design". I was initially brought on to this role to edit the video lectures for the course. Later, I was one of the two TAs who helped run the forums and create projects for the course. Thanks to this course, I have extensive experience with video editing and packaging materials for an asynchronous learning format.

The experience really opened my eyes to the sheer amount of motivation students need to have to get through a MOOC. That knowledge propelled me to be extra proactive in driving engagement in events virtualized because of COVID: visit days, social hours, the Programming Systems seminar.

The key to driving engagement online is to abandon the idea that students want to speak up in front of everyone in a large online call. This type of broadcast is intimidating for the vast majority of students. Depending on the event, I have added personalized 1-1 interactions, forced breakout rooms, and focused on non-video call activities (i.e., short games or polls).

Teaching Philosophy

I believe that introductory university-level computer science courses should provide the opportunity for students with a variety of backgrounds to succeed, and focus on core skills necessary to apply programming to a variety of domains of interest. To me, these core skills are programmatic thinking—i.e. control flow, recursion, statefulness—and the ability to break down problems in order to solve them programmatically. This involves allowing students to reason on their own from problem statements, to expected program behavior, to code. These core skills are useful to students of all backgrounds, providing the necessary foundation for CS majors and arming non-CS majors with the tools they need to tackle programmatic problems in their own fields.

Emphasizing these core skills over the ability to solve a large number of clever programming problems is also important in creating a level playing field for incoming students. Although my high school education involved good preparation in English, Mathematics, and Sciences, there was no class I could have taken there that would have provided me with any reasonable programming experience. However, since my first CS course emphasized problem solving over clever coding, I was able to get up to speed with students with more experience. From there, I was able to follow a succession of intermediate and advanced courses in programming, logic, and architecture, which required programming experience.

In short: we must strive to create curricula that provide clear stepping stones for students without programming experience to learn core programming skills, even if this means a "slower" first class for students with programming knowledge. Our intermediate and advanced courses can build towards more industry-specific skills, but our core goal should be to enable students to learn whatever frameworks and languages they will run into, rather than teaching them how to use specific ones.

Teaching Interests

I particularly enjoy teaching core programming and software engineering classes at the lower undergraduate level. First, as discussed above, I think learning the core principles of programmatic thinking and design is immensely valuable to both CS and non-CS majors. Additionally, I think it is incredibly important to have representation at the front of the classroom in these first classes. In my student evaluations of teaching for CS 61A, I got the feedback: *"Caroline Lemieux is so inspiring. She is my CS female role model and I enjoy going to her extra lectures"*.

At the upper undergraduate level, I am interested in teaching topics in programming languages, compilers, and software engineering. The compiler course I TA'd at Berkeley, on top of core compilers topics, also introduced some formal programming languages theory (formal semantics and type-checking). I think the most important change that could be made to this course is to reduce the emphasis on parser theory: just present recursive descent parsing, then focus on the basics necessary for using parser generators in practice. I would love to create an upper-level undergraduate programming course with a heavier focus on concepts in software testing, debugging, and comprehension.

At the graduate level, there are several seminar-based courses I would be interested in teaching. Most notably, a course in automated testing, debugging, and program analysis, touching on the most recent developments in fuzz testing as well as classic papers in randomized testing. I have also spent a substantial amount of time studying papers at the intersection of machine learning and programming

languages or software engineering. Both these topics are well-suited to a seminar format, focused on student presentations of each paper, along with classroom discussion of the paper.

Mentoring

I have mentored several graduate students, both in my advisor's direct group and in Programming Systems at Berkeley more broadly. In some cases, I was directly mentoring them on research and communication. That is, I attended weekly research meetings, first providing a bridge between the professors and the more junior student, and then strategically making space in the meetings where the student could grow their own communication skills. I am also good at spotting when students in my group get demotivated. When I notice this, I work with them to understand the issue and resolve the underlying cause to get them back on track.

Most of my junior graduate student mentorship has been more informal. I naturally took on the role of general mentor to my labmates. I am someone to go to for questions about the technical details about the graduate program, or just advice and validation in research and graduate student life. One of the students whom I interacted with in this manner was kind enough to nominate me for the **Dmitry Angelakos Memorial Achievement Award** for this informal mentorship work. While fellow graduate students provide a form of peer mentorship which is difficult to fully have once the professor title arrives, I hope to retain this type of accessibility as a professor.

I have also had the opportunity to mentor several undergraduate students while at Berkeley. Most of these undergraduates were directed to me by my advisor. I am fairly hands-on with the students, with weekly meetings where we discuss (1) their progress (2) what they are blocked on, and (3) where they should go next. Depending on the motivations and time-commitments of the student, I place different weekly goals. My goal is to give students exposure to research, but also a concrete end result. For a student who is just exploring research, my end-goal is generally to get them to write a technical report which summarizes their work. In a few cases, these technical reports have had promising enough results for us to push them into a full paper.

I try to give students problems to work on that allow them to begin exploring different algorithms to solve the problem early on in the project. I find this gives them ownership of their own project and self-motivation to move forward. One of my undergrad mentees, who had previously worked on research in another group at Berkeley, told me that this approach made him interested in pursuing research and motivated his graduate school applications.

I will use a very similar approach to get my junior graduate students bootstrapped into research, though I will be able to give them problems that require a little more technical background. A core difficulty for me during my graduate studies was learning to come up with my own compelling research problems. Because of this experience, I am well-suited to give my students tools to do this themselves (e.g., specific brainstorming techniques, suggesting "opener" problems which, when tackled, will open many exciting research avenues). On the flipside, I am a talented project "closer", thus I can bring grounding to my students' own ideas. All the while, I will expose them to the thought process that enables me to close projects, so they will also develop this rounded set of skills.