Statement on Teaching — Daniel Patterson

Every student creates their own understanding of the material presented to them. My goal as a teacher is thus two-fold: first, to create an environment that has the support and structure to carry out that creation and second, to provide them with engaging and challenging material that they will use to build up that understanding. Running parallel to that learning process is something more opaque to the student: assessment of how consistent their understanding is with what I, as the teacher, hope for them to get out of the class.

Consider the first aspect: creating an environment for learning. At Northeastern University, I designed and taught an upper level course on verified compilers, which both taught the students how to use a proof assistant and then applied that skill to proving a series of small compiler transformations correct. I was very conscious that learning how to use a proof assistant is quite difficult, even for my advanced students, all of whom had been programming for several years and some for much longer. This is primarily because while there are various tools (tactics, automation, etc) to provide an easier veneer, what underlies these proof assistants are dependently typed core languages, and building up huge dependently typed terms (as any non-trivial proof will be) is enough of a difficulty bump above even ordinary statically typed functional programming that for most it is as if learning to program again.

Understanding the challenge that students would have and the possible frustration that could result informed various aspects of the course design: first, that the vast majority of class time would be spent working in pairs on proofs, with me floating around to get people unstuck. At the end of each class, one group would present their work, which helped other groups by showing them an alternative or a completed version if they hadn't completed the in-class exercise. This way, students would both be able to practice by doing but also see the answer, all while having expert assistance available. I relegated traditional lecture material to readings that accompanied problem sets, with traditional office hours available. The second intervention was that I encouraged splitting out parts of proofs and continuing on, both modeling that during in-class demonstrations and structuring grading to encourage it in homework assignments. While quantitative results from small classes are difficult, qualitative results—in the form of surveys— confirmed that the class work was strongly appreciated, with nearly all the students successfully tackling the difficult material.

My next role is providing students with engaging and challenging material, which they can use to build up their understanding. In the accelerated version of the introductory computer science class at Northeastern University, I did this by designing a semester long project in which students built first a single player tile-based game and then eventually connected their clients to a server, replicating all functionality for each connected player. The project was structured to mirror the students increasing familiarity with data structures throughout the single player version, and then the conversion to the multi-player version was an exercise in refactoring, where nearly all functionality could be re-used, with only small additions that distinguished between input from the keyboard and input from the network to identify which player was moving or acting. Features of the framework we used made debugging the second part difficult, leading to some frustration, but the reward of interacting with classmates in an open world game they had built was appreciated by many.

I also wrote and delivered a lecture for the same introductory class where, after having used an untyped language throughout the semester, we livecoded a simple system to enforce (at runtime) some of the types they had been writing in comments. The lecture was highly dynamic—all typing I did driven by their input—and the eventual payoff so rewarding that the students applauded.

In my course on verified compilers, providing engaging and challenging material mostly meant designing exercises, both expository as in class examples and for homework. These exercises were the substance that students used to build their capacity both in theorem proving and compiler verification. For the course, we were using a common proof technique within the research literature called simulations, for which there were many complex examples, but nothing simple. My first example, which did constant folding only in the leaves of syntax trees (e.g., $2 + (2 + 2)$ compiles to $2 + 4$, not 6), demonstrated the various parts of simulation proofs while remaining very concise, thus could be used as a mental model throughout the rest of the semester.

The final role that I have as a teacher is to assess students. Assessment is usually taken synonymously with exams and homework, but my philosophy yields a slightly different view: courses must have some common content that a successful participant in the course will be able to understand. The course may, and indeed most likely will, have plenty of additional content that students might learn, and each students understanding of that material will vary. The purpose of assessment must be to determine whether students have sufficiently built an understanding of *core content* – and the possibility of the assessments being imperfect at capturing that must be acknowledged! Clearly, we should use rubrics (in the introductory class at Northeastern, I wrote rubrics that a staff of seven undergraduates used, after which I and two others reviewed the grading for consistency), but we should be open to changing the very structure of our assessments.

For example, while timed exams may be necessary for the sake of pragmatism (they can generally cover simpler material and still get some sense of an individuals knowledge), there are reasons to doubt their quality: some students work poorly under time constraints, and they are generally done under very different contexts than any realistic (whether academic or industry) setting. For example, in the introductory programming classes at Northeastern, while assignments are done using an interactive development environment, in exams programs are written on paper. One reason for this is to avoid plagiarism, but if we take plagiarism as simply an input that assessment should be designed with awareness of, rather than something that should be allowed to override other design constraints, we may come up with better alternatives.

For example, consider adapting the strategy of "code walks" used in some courses at Northeastern to exams as follows: an exam would take place over a set period of time, say 24 or 48 hours. At the end of the period, there is a several hour block (the normal "exam block") during which a random selection of students (say, 10%), on submission of their exam, will be asked to come and explain ("walk through") their code in person, with the grade dependent not on what they submitted but rather on their *explanation* of the submission. The long period to work on the exam allows students to take more time and use the programming environment they are familiar with while they develop their solutions. The code walk, by being applied stochastically, requires a comparatively small amount of staff time. It also should be relatively easy for students who did the exam: they explain the work that they worked on within last day or two, but it provides a strong (psychological) check on academic code violations, as the randomness means that there is no way to predict if you will be in that pool. Finally, if a student is able to submit work they did not do and yet explain it with confidence, then they clearly understand the material, and thus the actual pedagogical loss to such plagiarism is minimal. One risk to this, of course, is that someone could get stressed out explaining their own code – this should be mitigated by having the "code walk" activity not be something that only shows up in the context of exams, as indeed, the process of explaining code is a critical skill that students should learn to do regardless. We explore this not only as a potential real idea, but as a demonstration of the type of systematic thinking that I bring to teaching.