Technological and Pedagogical Innovations for Teaching Introductory Discrete Mathematics to Computer Science Students

Adam Blank

CMU-CS-14-114

August 2014

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Randy Bryant, Chair
Klaus Sutner
Marsha Lovett

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

This research was sponsored by the National Science Foundation under grant numbers IIS-0910453, IIS-0968487, IIS-1054630, and DUE-1245735, the Department of Education under grant number R305B090023, and The David and Lucille Packard Foundation under grant number 200934720.

Keywords: discrete mathematics, introductory course, compiler, runtime analysis, assessment, peer review

For my father.

Abstract

Students studying computer science usually encounter an introductory discrete math course in their first or second year of undergraduate education. Before this course, the only computer science they have seen is often programming, and as a result they have a very narrow definition of our field. The realization that math—in particular, math that is *very different* than what they've seen in high school and calculus—is a large part of computer science can be very jarring and unwelcome. Moreover, these courses often simultaneously introduce the concept of a proof and formalism, numerous new symbols and terminology, have difficult content, and have slow grading.

We present several technological and pedagogical solutions to these problems wrapped together in a course implementation, piloted at Carnegie Mellon University as 15-151. 15-151 deliberately addresses and embraces these concerns rather than trying to avoid them. In particular, it invites students to explore content applications by programming, write and discuss proofs together in groups, use a "computational environment" to explore syntax and semantics, and read and evaluate each other's proofs to understand their errors and speed up grading.

Acknowledgments

I owe a great debt to a number of people who have made 15-151 and this thesis possible.

First, I want to thank everyone who has assisted me an an advisory role over the years: Randy Bryant, Klaus Sutner, Marsha Lovett, Emma Brunskill, Luis von Ahn, and Mark Stehlik. Their patience with me and constant advice have pushed me to do my best work and guided me through difficult choices, and I cannot express my gratitude enough.

Randy has been infinitely patient as I have gone on teaching tangents and changed research directions. Throughout changing perspectives and goals, Randy has been the glue that has held everything together. His advice and insight have been invaluable to me, in my reseach, in writing this document, and in accomplishing my goals. Without Randy's faith in me, 15-151 and this thesis would not exist–and my future would be very different.

Klaus has given me advice and guided me, both in my undergraduate research and as I continued on to graduate school. He was always willing to meet with me when I frequently dropped in unannounced, and he has constantly solved problems for me as they have arisen. When I wanted to teach 15-151, Klaus gave up teaching CDM for a year to co-teach with me, so it could happen. Klaus has consistently given me the freedom–both logistically and pedagogically–to try out new ideas; many of these ideas ultimately made it into this thesis, but Klaus helped me to explore even those ideas that didn't.

Marsha has consistently been available to supplement and foster the experimental teaching I was doing with her vast knowledge of educational research and her enthusiasm. We have talked through a variety of concerns and experimental designs as they have come up, and she has helped solve numerous problems.

Luis has offered advice and assistance in a wide variety of capacities; he helped me develop and let me try out my initial grading ideas, mentored me when I interned at Duolingo, and showed me that courses can be very different. Emma has pushed me out of my comfort zone and broadened the variety of approaches and results I have aimed for. Mark helped me persue my unique approach to my undergraduate degree (which prepared me for what came next), helped me make 15-151 happen, and has given an unreasonable amount of his time to me. Tom Cortina has helped me navigate various policy and student concerns, and he has always been available to lend an ear.

Second, I also want to thank everyone else who has been there for me personally and professionally throughout. Josh Zimmerman has constantly been there for me to make me smile. He has listened to numerous rants about teaching and helped me improve my writing, my courses, and my mood. Max Illfelder has inspired confidence when things weren't going well and a reality check when they were. Dmitriy Chernyak has suggested homework problems, listened to (and critiqued) ideas, and reviewed this document. Brendan Sullivan tested out various technologies and pedagogical ideas, and he has inspired a variety of new approaches to teaching material. Rosie Battenfelder has copied numerous handouts for 15-151, and always been happy to have a conversation. Deb Cavlovich has helped me figure out how to navigate the various issues that have come up over the past few years. Catherine Fichtner has always been positive, welcoming, and available to listen.

Third, I thank every one of my teaching assistants for all their hard work and ideas. None of this would have been possible without you.

Finally, I thank all my students. There is no point in improving education if there is nobody to benefit from it. It has been a pleasure to teach all of you. Many of your suggestions have significantly impacted my work, and I am very grateful for them.

Contents

List of Figures

Chapter 1

Introduction

Mathematical Foundations for Computer Science are generally considered to be a core part of undergraduate computer science curricula [Ral05, Dev01, Fra06, Hen01]. In fact, the joint ACM and IEEE task force on computing curricula suggest in their recent 2013 curriculum guidelines that 37 of the 165 "Tier-1 Core" (absolutely essential) course hours be spent on these preliminary topics [oCC13]. Importantly, as the technical report notes, the boundaries between these preliminaries and various applications and extensions are not always clear, and content decisions can lead to drastically different course sequence implementations. Even once content is chosen, there are various practical barriers that arise in the courses teaching it.

First, these courses are *difficult*! They cover a wide variety of material which often feels completely unmotivated to students. Since most of these courses cover logic and sets, they introduce an entire set of symbols, grammar, and writing style that is as comprehensive as learning a new language.

Second, enrollment is large and increasing, and the students have varied backgrounds and interests. In recent years, a wide variety of universities have seen exploding enrollment in introductory courses and large increases in the number of intended majors [LR14], and it does not look like this is likely to stop. As the student population increases, students' interests and backgrounds diverge even more significantly. Some students come into the computer science major with no experience while others believe computer science is entirely about programming. Some of our students dislike math and others have just taken a calculus sequence. These facts cause the design of prerequisite foundations courses to seriously consider both the incoming and outgoing perceptions of the students. In particular, we can very easily overwhelm, discourage, or under-prepare students.

Finally, grading proofs is time consuming and resource intensive but necessary to teach students how to write valid arguments. Even if there are enough graders, they are often unqualified or do not prioritize feedback. This can result in feedback that is either incorrect or significantly delayed which makes it very difficult for students to learn.

Given that these concerns face all foundations sequences, any implementation should be able answer all of the following in the affirmative:

- Does our *course design* cover all essential topics without overwhelming students?

- Does our *course design* address all important "meta issues" (with respect to student attitudes and prior knowledge)?
- Does our *course design* include interesting, motivational applications without overloading students with too much material?
- Does our *instruction* foster deep learning, so that students are ready for later courses that build on this material?
- Do our *assessments* help students explore the new language (i.e. logic, sets, proofs) we are teaching without increasing student effort significantly?
- Is our *feedback* consistent, timely, and of good quality?
- Do students utilize and learn from the *feedback* they are given?

In practice, each of these points individually is a tall order, and there is likely not a single implementation that actually solves all of them. In this thesis, we propose solutions and our prototype implementation of our solutions for each probing question.

We begin by discussing common choices of "meta content" and content. We provide recommendations on "clusters" of this content and how to enforce the "meta content" outcomes. Then, we review a variety of existing foundations for CS course sequences. Additionally, we present the unique situation at Carnegie Mellon and our re-implementation of their first course (Chapter 2). To complement choices in content, courses must choose instructional strategies. We discuss various strategies that have a basis in educational research and can increase student understanding and enjoyment in foundations courses, and our particular implementation of these strategies at CMU (Chapter 3).

To simultaneously improve assessment and feedback, we propose a new programming language that acts as a "computational environment" for exploring the mathematical language taught in these courses and our results in using its prototype (Chapter 4). We detail a symbolic complexity analysis assignment that cleanly integrates this language and many of the topics students learn as preliminaries in foundations courses into a "cool application" of these topics (Chapter 5).

Finally, we discuss a new way of structuring grading called *Judgement-based grading*, and two implementations of it that increase consistency, speed, quality, and coordination of grading (Chapter 6). Furthermore, we explain how Judgement-based grading can be used to back a form of peer-review of proofs by novice students that we leverage to decrease necessary grading resources and increase student learning (Chapter 7). We discuss promising preliminary results for both Judgement-based grading and peer-review that uses it.

Part I

Developing a Discrete Mathematics Course for Freshmen Computer Science Students

Chapter 2

Discrete Mathematics for CS Majors

We begin this chapter with an in-depth dissection of the "meta content" that must be addressed, either implicitly or explicitly, when teaching computer science students proofs and discrete foundations for the first time. In particular, we review students' backgrounds and preconceptions coming into the course.

Then, we separate the possible course content into "clusters", explore the ways these clusters are usually implemented in a curriculum, and discuss drawbacks and benefits to several common choices.

We review various implementations of splitting up domain content over a curriculum and various motivational computer science applications of the material. Finally, as a case study, we consider the curriculum at Carnegie Mellon and our implementation of such a course: "15-151: Mathematical Foundations of Computer Science."

2.1 Meta Content

Given the unique position that introductory discrete math courses have in CS curricula as both a *transition* from programming, *prerequisite* fulfillment for a large variety of courses, and an introduction to formal reasoning and proofs, it is unsurprising that they are difficult and often distressing for students. Instructors who teach these courses *must* be keenly aware of the *meta content* associated with these courses. The *meta content* is, in many ways, more important than the domain content being taught in the course.

2.1.1 Calculus, Computer Science, and You

Students taking an introductory discrete math course have usually learned two approaches to handle what they call "math":

(1) "watch-and-repeat", in which the teacher gives them a pattern or algorithm to follow, and they do so without thinking about it

(2) "plug-and-chug", in which they arbitrarily plug values into an equation to "find the answer"

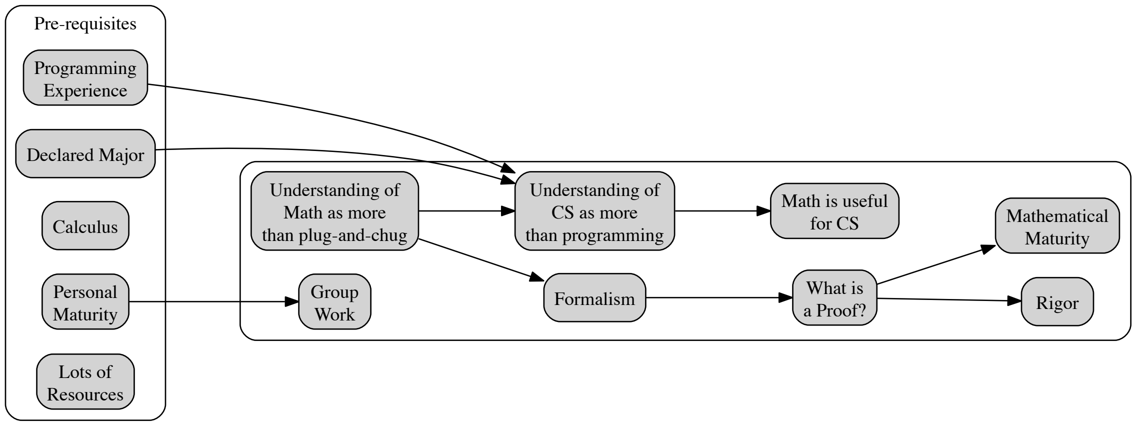


Figure 2.1: A pre-requisite map for discrete math related meta topics

This insanity [Loc], of course, inspires many of them to hate "math" and many of the others to think they're bad at it. So, it is no surprise that a course that alleges to teach them more math is met with groans and worry. If this were not painful enough, many of them "learned induction" and "learned combinatorics" in high school; so, they expect it to not be difficult. These expectations compounded with an insistence for them to actually *think*, rather than repeat or chug, can compound their fears that they're bad at math. Those fears then can convert to worries that computer science is not for them. So, what do we do? We recommend the following:

- **Discuss the difference between "math" (what they know) and *math* (what you're teaching)**. Do this clearly and at the beginning of the course. Compare and contrast what problems, solutions, and expectations look like in both domains. Repeated differentiation between their prior knowledge and expectations for the course can preemptively prepare them for when their prior mathematical knowledge will be insufficient to support the "new math" they are learning. Enforcing this difference can help them understand that *it is okay* that they cannot directly build on what they already know [ABD^{+}10].
- **Get them interested and invested**. Use cool applications and interesting problems. Give them opportunities to explore their interests further. Student perceived value of discrete mathematics is often very low when coming into the course; it is the job of the instructor to help students see *intrinsic* or *instrumental* value of these topics. Claiming that it "will be obvious later" rarely works, and can often back-fire and cause students to distrust the instructor. Furthermore, "fake" value (applications that really are not; prerequisites that also have low intrinsic value, etc.) often does not add very much [ABD^{+}10]. We discuss how to foster intrinsic motivation and provide some possible choices for "cool applications" later in this chapter.
- **Make them feel safe**. Provide an environment known for second chances and

extra help. Make policies that provide second chances. Because students often do not feel confident either in the new college setting or in a math setting, prior experiences and stereotype (and other) threats can predispose students to give up easily. Counteracting this with policy and course environment as much as possible can somewhat mitigate this.

- **Make your expectations clear**. This is particularly important when it comes to the question "what is a proof?", because the topic is inherently poorly defined. Student perception that expectations or instructions are arbitrary can increase negative feelings and lower barriers to giving up.
- **Talk to them about the "growth mindset"**. Many students believe that they were not born "math people", and, thus, they have no control over how they perform on mathematical tasks. This sort of thinking is called the "fixed mindset" by Dweck (a psychologist). It is particularly a problem in a foundations course, because it motivates students to give up on difficult problems rather than working on them.

  The "growth mindset" is effectively the opposite: these students believe that their intelligence is flexible and that by working hard they can make new pathways in their brains. Students with a "fixed mindset" do significantly worse than those with a "growth mindset" over time, and, furthermore, students who are encouraged to think with a "growth mindset" can actually completely turn around their performance [Dwe08].

  The instructor's mindset has a notable effect on the students. Instructors who believe in the "fixed mindset" generally (unconsciously) give less productive feedback and (unconsciously) encourage students to think with a "fixed mindset" as well.

  While the research is not *entirely* conclusive on which mindset is actually the biologically correct one, many researchers believe in significant "neural plasticity" (which backs up the "growth mindset"). Furthermore, making an analogy between math/programming (which people believe not everyone can do) and reading (which we believe everyone should be able to do) helps as well [Wil09].

  The original research has been discussed for computer science [MT08], as well as applied in a programming context [CCD$^+$10, SHM08]. Explaining that this course is intended to help them train their brains to think in a new way, and backing it up with this research can inspire confidence–particularly in at-risk groups [ACM09].

2.1.2 Programming, Computer Science, and You

Very commonly, when students choose to pursue the computer science major, they are basing their decision on how much they enjoy programming. While programming ability and enjoyment is a great signal for computer science aptitude, if it is the only signal students are using (and it often is), the first non-programming course students take in the major can create a very large internal struggle as students' initial perceptions meet reality.

Furthermore, this problem is not trivially solvable by trying to describe that computer science "involves a lot of math", because this has the possibility of over-correcting in either direction. Since students have the perception that math is just "watch-and-repeat" and "plug-and-chug", or, better yet "math is calculus", pushing them to recognize that math is needed in computer science can deter students for whom it should not. On the other end of the spectrum is a student who saw basic induction/combinatorics/probability in high school in algebra or pre-calculus.

In this situation, the student might vastly underestimate how difficult these topics can actually be, and more importantly he or she can believe they are *ahead* of the other students. Various, more complicated, strategies can be (and are) attempted to explain what a computer science major entails to incoming students, but the point we intend to make here is that in the status quo many students enter CS programs with an overly narrow view of computer science as programming.

While this is not necessarily a *problem*, it should certainly factor into the design of a foundations course. In particular, we recommend that foundations of math for cs courses *embrace* this student preconception rather than either downplaying it (e.g. exclaiming "There is more to computer science than just programming!") or ignoring it (e.g. no programming, coding, or connections to programming as examples, assignments, explanations) We recommend the following:

- **Make connections to programming as a *theme* and through examples.** When teaching new concepts, try *as often as possible* to point out the similarities to programming. A few examples are induction/recursion, scope of quantifiers/scope of functions, and poorly-formed expressions/invalid syntax. It is important to use this as a *theme* throughout the course rather than just appealing to it a few times, because it makes the course feel less foreign and intimidating.
- **Use programming assignments to keep their attention and force them to work through details.** Computer science students generally (but not always) prefer demonstrating understanding via programming (because it is familiar) to writing proofs. As such, mixing programming and proofs on the homework and showing how the relate and *complement* each other can be very useful to help students understand the material. Additionally, students can check the results of their programs on examples very easily, but checking their proofs is a significantly harder skill to learn. Finally, another advantage of programming assignments over proofs assignments is that they are significantly easier to grade! In the past, we've given assignments like "implement a relations library and use it to compute certain interesting things" or "here is a combinatorial problem, implement a program that counts…".
- **Give them an environment with immediate feedback to explore sets, logic, etc. and test their assumptions.** We discuss our preliminary implementation of such an environment in Chapter 4.
- **Be careful not to abuse their programming ego, if they have one.** One of the downsides of associating programming with discrete math is that it can undermine their confidence in *programming*, or, worse, activate certain prior struggles. Be-

cause this is possible, courses have to be careful to make sure that the programming part of the assignment is not too difficult. Choosing an expressive programming language can help with this. Also, surveying students to make sure they enjoy the programming assignments can make a difference.

- **Make sure to emphasize the relation to *different kinds* of programmers.** To over-simplify, some students are primarily interested in *building things* (web applications, programs with cool results, mobile applications, etc.), and other students are primarily interested in *understanding how things work* (what things are behind the compiler, parallelism, how does the machine work). These students have very different outlooks on the foundations course. The first group wants to know why the math will be useful to them in writing programs, and the second group wants to know how it will help them understand programs, the machine, etc. It is important to make sure motivations include perspectives for *both* groups.

2.1.3 Intrinsic Motivation

While computer science students taking a foundations course are often self-motivated for *programming* tasks, they are very rarely intrinsically motivated for mathematical tasks. As we would like to convince students that math is interesting and useful in its own right, this is a large deficit we are starting at. So, it is very important to understand how to increase (or accidentally) decrease student intrinsic motivation.

Deci, Koestner, and Ryan have done several meta-analyses of studies to understand what types of rewards increase and decrease intrinsic motivation [DKR01, DKR99, ECP99]. We summarize the most relevant results here.

A *tangible reward* is a non-verbal reward that has significance to the receiver (a grade, money, a certificate, cupcakes during recitation). The meta-analysis showed that *unexpected tangible rewards* do not have a significant effect on college students, but *expected tangible rewards* decrease intrinsic motivation with an effect size of $d = -0.36$. Furthermore, Butler and Nisan compared feedback, numerical scores, and no response, and found that feedback (without a score) led to increased intrinsic motivation and future performance [BN86]. These conclusions indicate that, whenever possible, we should avoid (or possibly delay) grades in favor of feedback.

A *verbal reward* is positive feedback from a respected figure (such as the instructor). If the verbal reward is perceived as "controlling", meaning that the student believes that the rewarder is trying to get the student to do something, it can have a negative effect, but if the reward is perceived as "informational" (e.g. telling the students they are performing better than other classes), it can have a positive effect size of $d = 0.43$. Our recommendation here is to emphasize how well the class is doing, and possibly explicitly ask TAs to give positive feedback to struggling students to keep them motivated.

Helping students care about our material intrinsically is particularly difficult, because of the perceptions they have coming in that we discussed in the last two sub-sections. But, these perceptions make intrinsic motivation all the more important if we want students to continue with computer science!

2.1.4 Proofs and Rigor

The goal in teaching students about proofs is to help them write in a clear, deliberate, and precise way. Unfortunately, it is not exactly agreed upon just how formal is the *right* amount of formal. Moreover, as students progress through courses, the standards for "rigor" nearly always decrease. In essence, is because we begin *trusting* that they understand the details, and, so, we no longer require them. However, lowering the standards too early can result in students skipping steps that they really did not realize need to exist.

On a high level, there are three "standards" of proof which are common among mathematicians and computer scientists: machine-checkable proofs, formal proofs, and human-readable proofs.

A machine-checkable proof is formally specified and intended for a *computer* to interpret using languages like Cog or Agda. Since these proofs are specified for computers, they often contain details that a human reader might find trivial or, conversely, miss details a human evaluator might not understand as can be seen in Figure 2.2 [Pie13]. This mismatch between the details and the audience, along with the irrelevant programming knowledge that must be acquired to use it, make this type of proof undesirable for teaching budding computer scientists and mathematicians their first type of proof.

```
Theorem double_injective' : forall n m, double n = double m
    —> x = y.
Proof.
  intros n. induction n as [| n'].
  Case "n = O". simpl. intros m eq. destruct m as [| m'].
    SCase "m = O". reflexivity.
    SCase "m = S m'". inversion eq.
  Case "n = S n'".
    intros m eq.
    destruct m as [| m'].
    SCase "m = O". inversion eq.
    SCase "m = S m'".
      assert (n' = m') as H.
      SSCase "Proof of assertion". apply IHn'.
        inversion eq. reflexivity.
      rewrite —> H. reflexivity. Qed.
```

Figure 2.2: A machine-checkable proof that $f(x) = f(y) \Rightarrow x = y$ where $f(x) = 2x$

A detailed formal proof works from a small set of known truths (axioms) and applies a series of formal rules to make new truths (inference rules). Only a small subset of mathematicians who are concerned about the "foundations of mathematics" generally think about these proofs. Due to their extremely high level of detail, this type of proof is not appropriate for students who are attempting to prove (eventually) complex results. Figure 2.3 on page 10 shows a detailed axiomatic proof of a very simple proposition.

Finally, human-readable proofs are written for *human* mathematicians and/or computer scientists. These proofs emphasize the important, novel parts of the argument.

1.	$p \Rightarrow q$	Premise
2.	$q \Rightarrow r$	Premise
3.	$(q \Rightarrow r) \Rightarrow (p \Rightarrow (q \Rightarrow r))$	$\Rightarrow$ Intro.
4.	$p \Rightarrow (q \Rightarrow r)$	MP: 3, 2
5.	$(p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r))$	$\Rightarrow$ Distrib.
6.	$(p \Rightarrow q) \Rightarrow (p \Rightarrow r)$	MP: 5, 4
7.	$p \Rightarrow r$	MP: 6, 1

Figure 2.3: A detailed formal proof that $p \Rightarrow q, q \Rightarrow r \vdash p \Rightarrow r$

They act as templates which provide enough information for an interested person to formalize it as a detailed formal proof or a machine-checkable one. Figure 2.4 on page 10 shows a human-readable proof of the same claim as Figure 2.2.

Suppose $f(x) = f(y)$. Then, $2x = 2y$. Dividing both sides by 2, we see that $x = y$ which is what we were trying to prove.

Figure 2.4: A human-readable proof that $f(x) = f(y) \Rightarrow x = y$ where $f(x) = 2x$

Some courses begin by asking students to write formal proofs (or *nearly* formal proofs, in the sense that the justifications are not required to be theorem names and numbers). Many courses simultaneously let students write (*near*) formal proofs and human-readable proofs from the beginning, allowing steps to be skipped or lack justification eventually. It is important, however, to force students to write human-readable proofs eventually, because, like essays and code, there is a *stylistic* element to writing proofs. We have found that calling it "proof design" rather than "proof style" communicates the idea more clearly to students, but nonetheless, it is incredibly important to grade proofs not only on their correctness, but on their formatting and clarity as well.

However, some problems arise when proof design becomes an important criterion for correctness. First, because it is likely no two instructors will expect exactly the same standard, it is important to communicate to students exactly the standard expected; so, the students do not feel like they are being graded arbitrarily. It is crucial that this standard is reflected in the solutions you hand out.

One issue that will likely come up in grading proof design is students who have already learned some things about proofs will often object to your particular standard, particularly if it is overly rigorous at the beginning, because they see no need for it or something different was expected in a different course they took. The only real solution is to explain that when grading, everyone has to be held to the same standard, and leaving out steps is either an indication of not understanding something–or already understanding it. Unfortunately, because the majority of students leave out steps due to not understanding why they are important, the grading has to err on the side of rigor.

Another concern is that students will think they should memorize words "that you want them to say"; to deal with this, you can either provide multiple examples of phrasings or evaluate them conceptually (e.g. give them a question with multiple wordings written out and ask them which ones work/do not work) and why.

In general, no matter what type of proof students are expected to write, they will often care much more about the answer than the proof itself; generally, this is because high school has trained them to only care about the answer (and not really the work). Dealing with this by deemphasizing the answer (making it worth 1 point) is often not well received, but it is important. Instead, overemphasizing *justification* will give students something to look toward.

2.1.5 Mathematical Problem Solving

Many foundations courses intend for students to improve at problem solving via the problems they discuss and solve throughout the course. "Problem solving" includes understanding problem statements, using heuristics to approach problems, choosing good strategies, working toward a solution (possibly struggling), dealing with dead-ends and frustration, and ultimately piecing together a solution that actually works. First, we outline two existing approaches to teaching problem solving.

Shoenfeld has a massive set of literature on helping students develop strategies for most of these things in an elective course with fewer than 10 students [AKMS98]. In particular, his uses rare lectures, student presentations, small-group work, whole-class discussions, and individual work (homework) to help students work on these skills. He spends a significant amount of effort in this course creating a "mathematical community" in which students critique and help each other, explore interesting problems, and investigate areas of mathematics in whichever direction the community finds interesting. The problems Shoenfeld uses are intended to be not trivial, but accessible to a wide audience, ideally solvable in multiple ways, demonstrate either content or methods, and not need "tricks" to solve.

CS304 at Stanford, originated by Polya and taught by Knuth many times is another approach for a different audience. The instructor begins the course by choosing around five *unsolved* problems. Like Shoenfeld's course, the number of students is very small (less than 10-15). CS304 differs from Shoenfeld's course in several important ways. First, students are expected to combine math and programming to solve the problems. Second, in Knuth's words "[t]his is a class on how to do research"; so, the students come in prepared to solve more simple problems [RK89].

The big differences between these two courses and a foundations sequence are (1) foundations courses are usually much larger, and (2) the course is not optional. Additionally, we would ideally like to incorporate the ideas about teaching students who are new to problem solving (from Shoenfeld) with using programming and computer science applications as the targets (from Knuth).

Scale eliminates our ability to do presentations and makes building a "mathematical community" significantly more difficult. But we can still ask students to do group work, and choose problems that fit Shoenfeld's criteria. Additionally, later in the sequence, we can choose problems that are similar to the ones in CS304 with the exception that we simplify them, make them less open ended, or do not grade on completion. Our implementation (Chapter 3) attempts to incorporate many of these ideas, and we talk about how one might go about implementing them more explicitly there.

2.1.6 Collaborative Learning

Because foundations courses can present difficult problems, they naturally create fantastic opportunities for students to help each other and communicate over problems. We believe this is a fundamental skill for computer science students (if not everybody) to learn, and fostering it explicitly, and guiding students on *how* to work together can make a large difference. We discuss ways to do this, in depth, in Chapter 3, as well.

2.2 Domain Content

There is far too much potential discrete mathematics content to fit into a single course (or even multiple courses); so, it is natural that various curricula pick and choose which topics to include to various degrees. In this section, we break down the *domain content* into *clusters* that nearly always appear together due to applications and dependencies. We provide a high-level discussion of the considerations involved in choosing a particular *cluster* of content and possible unintended results of particular choices. We intend this clustering to allow course sequence developers and instructors to more easily see the wealth of potential content and to give a local picture of possible topics to cover or remove to fit the meta content.

2.2.1 Essential Preliminary Content

Logic, sets, functions, induction, and proof strategies are *so essential* that any first discrete math course missing one or more of these topics is incomplete. Every non-preliminary topic relies on at least one of these, and most of them rely on several. Furthermore, many topics not even directly related to this material (algorithms, AI, machine learning, programming languages) all rely on many of these topics or, at the very least, the understanding of what a proof is.

Since one of the major outcomes of foundations courses is to communicate the concept of proofs, it can be advantageous to introduce proof strategies as early as possible. A natural impediment to introducing proofs is the lack of formal systems *to prove things about*! Sets, logic, summations, and recurrences serve very well as this basis. As a result, it is very common for courses to begin with one of these schema:

- Logic $\rightarrow$ Sets $\rightarrow$ Proofs
- Sets $\rightarrow$ Proofs
- Logic $\rightarrow$ Proofs

It is important to distinguish between teaching students *proof strategies* and teaching students how to *select a proof strategy*. These are separate outcomes, and it is very common to teach the first, but not the second. Understanding *how to approach* an

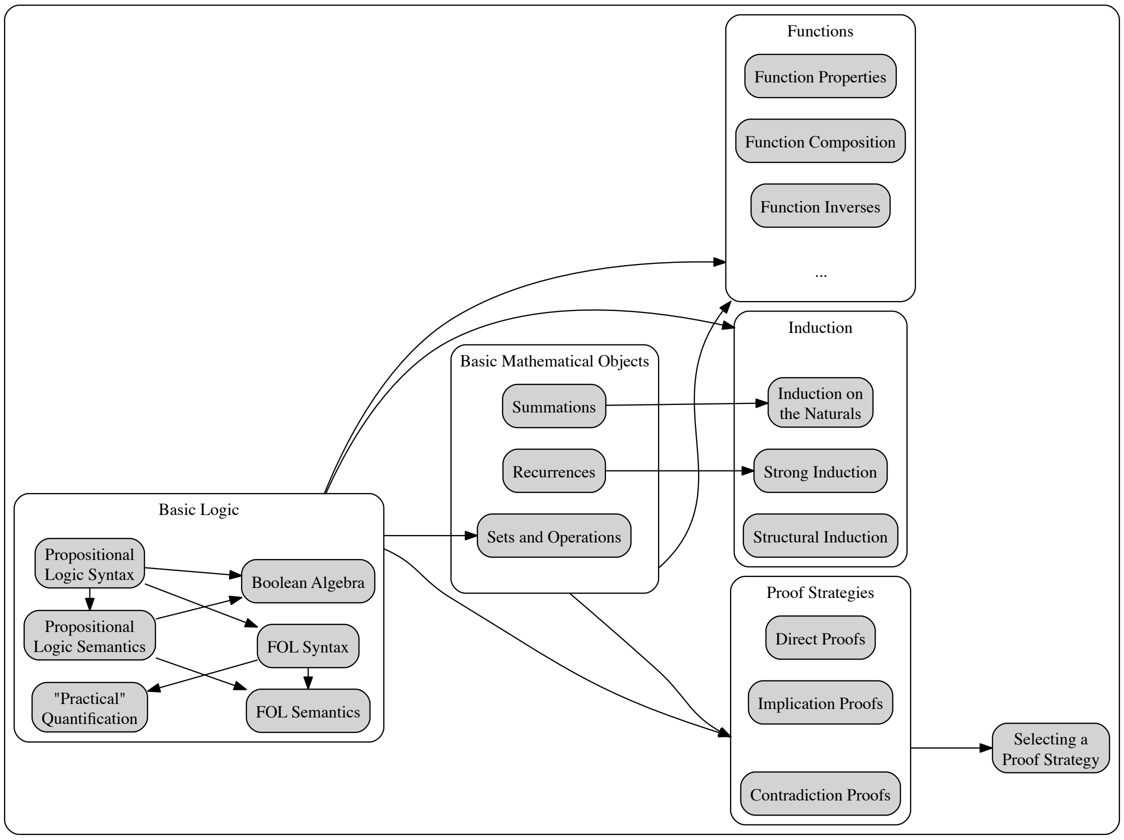


Figure 2.5: A prerequisite map for the essential preliminary discrete math topics

open-ended question is significantly more difficult to learn, and it is easy to avoid forcing students to learn it by providing hints that indicate how to proceed.

Unfortunately, many of these preliminary topics are dry or not immediately applicable to computer science–particularly to students who still think computer science is programming. For this reason, while courses cover all of these topics, they often delay many of them until the need arises for an application. For example, discussing functions (e.g. bijections, inverses, compositions, etc.) is often delayed until they can be compared to relations, become necessary to discuss infinite cardinalities, or are needed to explain RSA.

In the case that a curriculum has a required functional programming course, it is common to teach structural induction in that course instead (and require the foundations course as a prerequisite to ensure students have mathematical maturity).

2.2.2 Graphs and Relations

Graphs and binary relations are natural, primitive structures that are both used to formalize many concepts in computer science. There is also a very natural connection to make between connected components and equivalence closures. As a result, when courses cover graphs as *formal* objects, they are almost always covered in sequence with relations. When graphs are covered, there are several different sub-clusters of material that can be covered. If graphs are treated as a data structure, which is often left to a data structures course, then various implementations of graphs as adjacency lists and adjacency matrices (and powers of the adjacency matrix) can be covered.

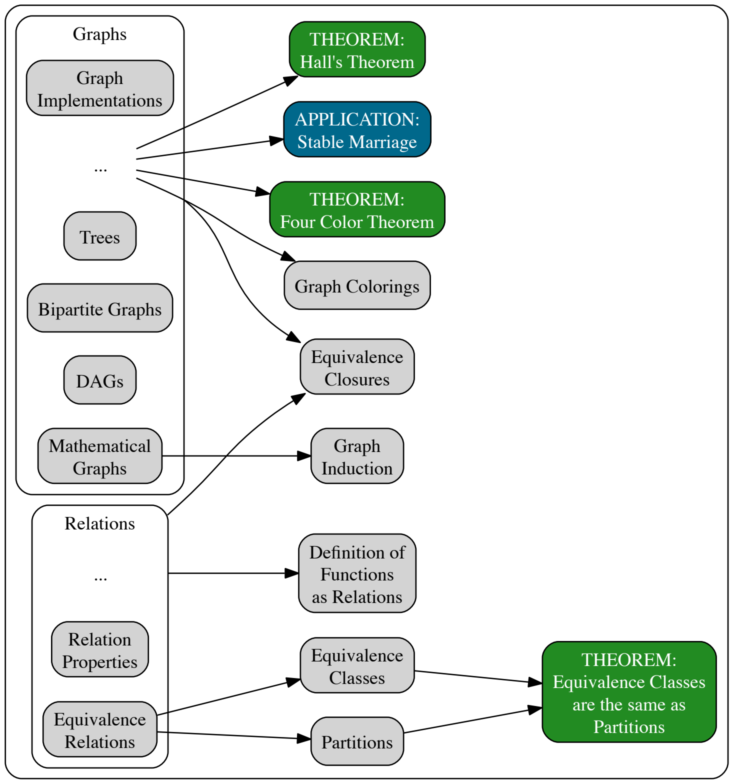


Figure 2.6: A prerequisite map for graphs and relations

More commonly though, graphs are treated as formal objects and trees, colorings, matchings, and/or networks are covered. It is surprisingly rare for courses that cover formal graphs to ask students to explore applications (beyond *theorems* like the four color theorem or *algorithms* like Gale-Shapley).

2.2.3 Number Theory, and Algebra

Number theory and, if it is covered, algebra, are basically *always* covered as an excuse to discuss RSA. This, by itself, is not a large problem, but it should be noted that the particular machinery needed to understand why RSA works (i.e., Euler's Theorem via Lagrange's Theorem or Lagrange's Theorem for the special case of $\mathbb{Z}_n^*$) is particularly unintuitive, and having seen the proof does not really provide much insight; rather, it can make RSA feel *more* like magic! This is particularly true if students are uncomfortable with the *concept* of making calculations in $(\mathbb{Z}_n^*, +, \times)$.

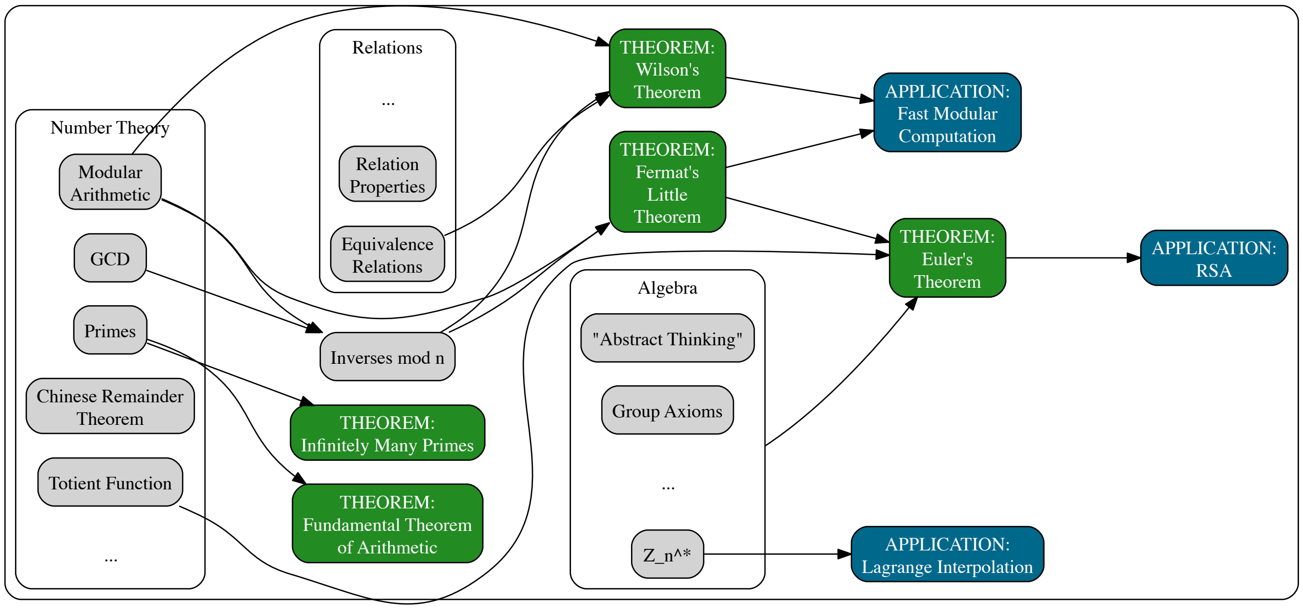

Number Theory
Modular Arithmetic
GCD
Primes
Chinese Remainder Theorem
Totient Function
...
Relations
...
Relation Properties
Equivalence Relations
Inverses mod n
THEOREM: Infinitely Many Primes
THEOREM: Fundamental Theorem of Arithmetic
THEOREM: Wilson's Theorem
THEOREM: Fermat's Little Theorem
Algebra
"Abstract Thinking"
Group Axioms
...
Z_n^*
APPLICATION: Fast Modular Computation
THEOREM: Euler's Theorem
APPLICATION: RSA
APPLICATION: Lagrange Interpolation

Figure 2.7: A prerequisite map toward RSA

Abstract properties of operations like associativity or existence of inverses are *completely* foreign to many students; so, not only do they not understand them applied to $\mathbb{Z}_n^*$, specifically, but there is a disconnect as to *why* we even switch to this field over the one they've been using for their entire lives. None of these problems is insurmountable, but the treatment of number theory and algebra need to be very carefully nuanced to get these points across. As a result, we recommend that, if possible, introductory foundations courses avoid RSA. Courses that insist on covering RSA should devote a significant amount of time to the extra meta goal of helping students become comfortable with abstract algebra. Some work has been done on using exploratory environments for abstract algebra which could prove useful [CP97, Bos97, She97, Kal97].

One very common approach to number theory is to prove a theorem grab bag (fundamental theorem of arithmetic, infinitely many primes, Wilson's Theorem, Fermat's Little Theorem/Euler's Theorem, existence and uniqueness of inverses mod n, etc.). While many of these theorems are individually interesting and deep, proving them in lecture results in students feeling like number theory is a grab bag of impossible tricks that they would have never come up with on their own.

It is our belief that many of these theorems are better suited to being guided exercises, and, furthermore, that rather than presenting them all together as "number theory", it is less overwhelming for the students if they are presented *with the proof strategies*. So, infinitely many primes would be an early example of a contradiction proof, and existence and uniqueness of inverses mod n would go with uniqueness of representation.

2.2.4 Combinatorics and Probability

For evident reasons, combinatorics usually directly precedes probability and statistics. Combinatorics is a golden opportunity to help students become more comfortable with recurrences as *combinatorial objects*. While examples like poker hands are very useful as *practice questions*, these sorts of examples make combinatorics seem even more out of place than it already does. In particular, if combinatorics is being taught in (or right after) a course that heavily stresses proofs and formal arguments, *very* great care has to be taken to motivate why we are allowed to make vague counting arguments and claim two sets have the same cardinality.

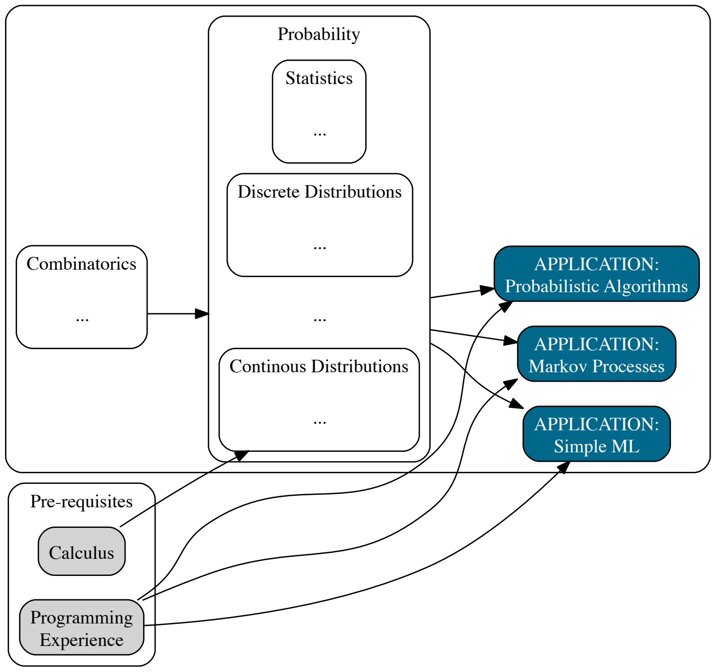


Figure 2.8: A prerequisite map for combinatorics and probability topics

Probability can be entirely motivated as a formalism to analyze "events of *probabilistic code*" rather than to analyze "events in a sample space." This is particularly useful as a way to (1) force students to be precise, and (2) keep computer science in students' minds without overloading them early on with real applications. Some courses only cover discrete distributions the first time and leave continuous ones for later. Regardless of when continuous distributions are covered, there is the distinct concern that students likely have not used calculus in at least a year.

In particular, it is unreasonable to expect that students (at least initially) have done integrals of anything other than polynomials in quite some time. This is an artifact of the tendencies for calculus to be an early freshman course, computer science to cover discrete math first, and probability to be left until later. There are reasonable ways to deal with this potential issue, but ignoring it is likely to lose a non-trivial percentage of the students who understand but need to look up how to do calculations.

2.2.5 Theory of Computation

Theory of Computation is probably the *most varied* set of topics in *introductory* discrete math courses. On one end of the range, some curricula require students to have heard of the halting problem and/or DFAs. Other curricula introduce students to the whole Chomsky Hierarchy in their first course in discrete math. Some courses are very rigorous about the topics they cover in this sequence, and others require students to create DFAs/regular expressions/etc but not prove things about them. These variety of goals cause the courses to be fundamentally different. In the version where only the halting problem is discussed, it is usually tacked onto diagonalization or induction. In the proof-based course that covers large portions of the Chomsky Hierarchy, working toward understanding what types of formal languages exist is the primary goal of the course.

Our recommendation is to carefully consider the downstream courses before choosing which direction to go in. For some students, the halting problem, DFAs, regular expressions, etc. can be incredibly good motivators for why math is important in Computer Science which can make an introductory discrete math course much less dry. Additionally, giving students an idea of what theoretical computer science looks like can have a large affect on what electives students choose (both positive and negative!).

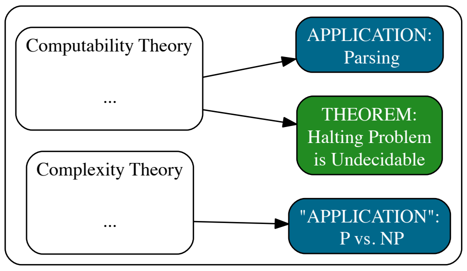


Figure 2.9: A prerequisite map for computability and complexity topics

2.3 Application Content

Applications and "cool" programs and theorems can be used to help students get more invested in the course material. We discuss some applications to investigate and programs to write that might help decrease the feeling that math is not useful in Computer Science and get students more invested in the domain content.

We do not go into major detail here; instead, we provide an overview of some possible choices for applications. In later chapters of this thesis (Chapter 4, Chapter 5), we go into significant detail for the first two applications here. We have used the other two applications several times in previous courses and seen them succeed, as well.

This list is not intended to be exhaustive, rather, we hope to inspire further discussion on what constitutes a good introductory application. For instance, all of our examples are small programming projects intended to connect with students' understanding of programming, but we believe that non-programming must exist as well with the same positive properties.

2.3.1 Basic Logic

Logic is unbelievably dry, but it can be used to solve amazing problems. Giving students a framework in which they use a SAT solver and a logical description of a well-known problem (e.g. Sudoku) to solve a problem could be a really interesting example or introduction. We propose using a modification of one of the tools we've developed (see Section 4.5) to tightly integrate this. In fact, the language {Setʇy} which we introduce in Chapter 4 is exactly an attempt to make logic more interesting.

2.3.2 Induction, Asymptotic Analysis, Relations, and Functions

Most curricula have a course which is devoted to helping students understand program correctness and analysis. We often ask students to manually trace through code to compute complexity bounds–implicitly asking them to understand an algorithm that usually works. We can (and did!) provide a framework for students to do this analysis symbolically using a toy language. This assignment uses real math to solve a real problem—and then, it becomes applicable *again* later, when we get to computability, as it is undecidable in general. See Chapter 5 for a more complete discussion of this idea.

2.3.3 Number Theory and Group Theory

Students are very often fascinated with cryptography, secret sharing, and redundancy. This is exactly what makes RSA such a killer application. We point out that another one, Lagrange Interpolation, is *much easier* to understand and implement. It has two very clear applications: one for correcting erasures and the other Shamir's Secret Sharing Algorithm (where k parties are needed to unlock a secret). Furthermore, it has the absolutely amazing quality of showing the usefulness of group theory, in practice, because it is identical to the Chinese Remainder Theorem in a different field.

2.3.4 Computability Theory

For courses that go very heavily into computability theory, if you're already covering CFGs, DFAs, and regular expressions, a great assignment would be to ask students to *use* all of these things to *implement* `grep`!

2.3.5 Probability

If the course covers continuous distributions and statistics, these can be pooled to ask the students to do very simple machine learning on real data.

2.4 Course Sequencing

Fundamentally, there are three common ways to split up the material that might appear in this sequence of introductory courses:

(1) It can be split up into two courses: a course on preliminaries, proofs, and some content; a second course on a selection of the remaining content. This choice seems popular for schools that are on the *quarter* system.

(2) Preliminaries, proofs, and a majority of content (except computability) are all taught in one very difficult course.

(3) One course on preliminaries, proofs, and some of the content; required "higher level" electives to fill in the remaining content.

Each of these strategies is used by at least one top CS university, and they each have their own set of advantages and disadvantages. We make the common ways of sequencing mathematical foundations for CS explicit, because our implementation, at CMU, follows none of them. In fact, the previous implementation at CMU did took another, distinct approach not represented in any of the other universities as well.

2.4.1 Summary of Various Universities

We provide an approximation of the implementations various universities have for their foundations course sequences. We deliberately leave out CMU, which we discuss in the next section in greater depth. This summary is *an approximation*, as we are sure the content changes from time to time, and we have not looked at every lecture of every course over every year.

University	Prelim	Graph	Relation	Numb. Thy	Algbr.	∞	Computability	Complexity	Combi.	Prob.
MIT (s)	✓	+	✓	✓	X	✓	HP	X	✓	D + S
Berkeley (s)	✓	✓	✓	✓	X	✓	HP	X	✓	D + C + S
Stanford (q)	✓	✓	✓	X	X	✓	+	✓	⟵second course⟶	
UW (q)	+	X	✓	✓	X	✓	+	⟵second course⟶		
Cornell (s)	✓	✓	✓	✓	?	✓	DFAs	X	⟵second course⟶	
UIUC (s)	✓	✓	✓	✓	X	✓	⟵second course⟶		⟵optional course⟶	

Table Key: ✓means covered; X means not covered; + means *a lot* is covered;? means sometimes covered;
HP is halting problem; D is discrete; C is continuous; S is statistics.

For reference, we include the numbers for the courses we based this information on:

MIT:	6.042
Berkeley:	EECS 70
Stanford:	CS 103, CS 109
UW:	CSE 311, CSE 312
Cornell:	CS 2800, various options
UIUC:	CS 173, MATH 461/463

A quick glance at this table reveals some notable commonalities and differences between these approaches. First, *no two of these approaches is the same*! Number theory (with an eye toward RSA) is extremely common, while algebra (Groups/Rings/Fields) is extremely uncommon. Nearly everyone covers the halting problem and gives a basic overview of the mathematical definition of a graph. Every university we looked at covers discrete probability, but many ignore continuous probability and/or basic statistics.

2.5 Our Implementation at CMU: 15-151

Because we attempt to take our own advice, many of the principles we've described about various types of meta content are apparent in 15-151. In this section, we situate our course in the CS introductory curriculum at Carnegie Mellon, discuss our primary goals, and briefly review how well we met these goals.

2.5.1 Overview of Carnegie Mellon Introductory CS Courses

By the beginning of freshman year, CMU students have already declared their majors. Of the CS students, approximately 70% have had prior programming experience, and the others all take the first introductory programming course, 15-112, in the fall semester. Regardless of prior experience, CS students take 15-151 (or, previously, 21-127).

We briefly discuss the interaction each of the CS introductory courses has with 15-151:

15-112: Fundamentals of Programming. This course does not get to recursion until week seven; so, we cannot give very many programming assignments until after then. Perhaps more importantly, this course is very time consuming for students; so, they are constantly fighting a battle between spending time on 112 and spending time on 151.

15-122: Principles of Imperative Computation. This course very heavily promotes contracts and invariants, but it only covers sketches of proofs related to them, because 15-151 is a co-requisite–not a prerequisite. 15-122 also covers asymptotics from an empirical perspective. All of the students who had previous programming experience take 15-122 simultaneously with 15-151.

15-150: Principles of Functional Programming. This course covers the fundamentals of functional programming in SML. As a result, structural induction, recursive data types, and recursive programs are its bread-and-butter. Because students usually take this course the semester after 15-151, we help foreshadow its contents, but, for the most part, we treat these topics as "off limits."

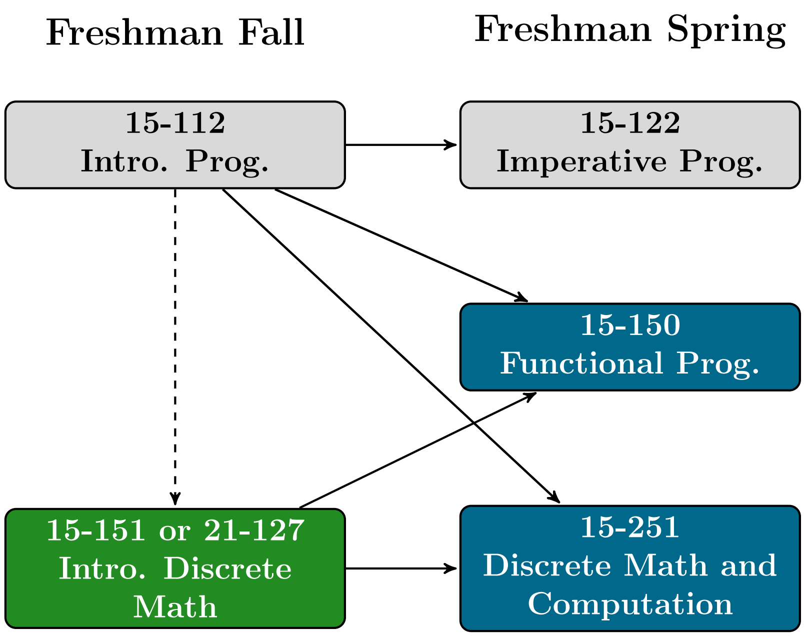


Figure 2.10: Carnegie Mellon CS introductory sequence

15-251: Great Theoretical Ideas in Computer Science This course covers effectively *a superset of the material we cover*. It is required for the major, and almost all CS freshmen take it the semester after they take 15-151. This may sound strange, but it is intentional, and we discuss it below.

The first three courses on this list (15-112, 15-122, and 15-150) are relatively new, as of a re-design of CMU's curriculum several years ago [BSS10]. After this re-design, 21-127 (the math department's version of 15-151) and 15-251 remained the same. Although the instructors of 15-251 have been aware of the changes to 15-151, it still remained relatively constant.

2.5.2 Implementation Goals and Details

Content

Course	Prelim	Graph	Relation	Numb. Thy	Algbr.	∞	Computability	Complexity	Combi.	Prob.
21-127	✓	X	✓	✓	X	✓	X	X	✓	X
15-151	✓	✓	✓	X	X	✓	X	X	✓	D
15-251	✓	+	?	✓	✓	✓	+	✓	✓	D

Preparation for Future Courses (15-251, 15-150)

15-251 is a *difficult* course. The lectures act like a survey course, in which, each week the course progresses to a new topic (a new course), and covers the first few lectures of that course. Then, the homework is (usually) chosen to be relatively difficult problems from the material covered. 15-251's drop rate is very high, and the homework routinely takes students upwards of 30 hours a week. 15-251's primary "meta goal" is to teach students how to approach new, interesting, difficult problems, and finishing the course report that they learned a lot (but not very much *domain* content). This course is taken by **second semester freshmen**. 30% of these students **programmed for the first time fifteen weeks before beginning 15-251**. 15-151, then, has the remarkably obvious meta goal of ensuring that students who are about to take 15-251 are ready to do so. It is significantly preferable to force students to take an easier introductory discrete math course for a second time to allowing them to move on and get crushed because they did not have an appropriate background going into 15-251.

The way 15-151 accomplishes this is by spending twice as much time covering the material 15-251 covers in the first half of the semester. This controlled and monitored overlap allows students to ease, relatively slowly, into an understanding what the math for computer science looks like. Then, when they finally get to 15-251, they ease into very difficult problems by working with content they are extra prepared for.

Formalism and Proof

Although 15-151 covers content, the primary *content* goal is actually one of the meta goals in a standard course: *proof design* (or "What is a proof and how do I write one?"). We want students to fundamentally understand basic proof strategies, the mathematical language that is used to communicate ideas, and how to put together a coherent argument.

Transition Students from "Programming" to "Computer Science"

As 70% of the students in 15-151 *have* programmed before, 15-151 is also tasked with the job of convincing those students that while programming has a place in computer science; so, do other things. We do this using the applications and programming assignments that we've already discussed in this chapter.

Separate Problem Solving from Formalism

Although problem solving is a meta goal of 15-151, it was not a *mastery goal*; in other words, we wanted to expose students to problem solving so that it would not be surprising later, but we tried to avoid testing particularly difficult problems which would require a mastery of problem solving on exams. We made this decision, because 15-251 is almost entirely a course about problem solving; so, our goal was to make sure students had minimal exposure and were comfortable with it before moving on.

Not just another math course!

One of the biggest differences between 21-127 (the course students previously took) and 15-151 is that 15-151 *is in the CS department*, and we know our students are interested in Computer Science. So, we took full advantage of this. We provided CS applications and motivations; we engaged the students in programming exercises; we made explicit connections to other courses in the core of the CS major. Interestingly, CMU was the only school (in the chart above) that asked the math department to teach the first foundations course.

2.5.3 Postmortem: 15-151 Structure

We (anecdotally) succeeded at many of these goals, but there were some places for improvement as well. We detail these successes and improvements in this sub-section.

From student accounts, they were very well **prepared for 15-251**. For the first part of the course (that repeats some of the content in 15-151), despite the problems being more difficult in 15-251, many people reported it being very easy. Additionally, their performance backs up this feeling.

While our students definitely succeeded at **learning formalism**, we did not do a good enough job explaining exactly what we were expecting. Our transition to letting them be less formal was also not as complete as it should have been. Despite this, we believe that students really did understand the benefits of being somewhat formal, even if they did not agree with the extent which we required it.

Our **transition to non-calculus math** succeeded. Students eventually understood that the answer is a small piece of the result. A particularly effective motivator was an explicit comparison to "high-school math" in the first lecture.

Although our **separation between formalism and problem solving** seems to have helped prepare students for 15-251, we believe it made us less able to help students develop intrinsic motivation for the material. We did not teach very many natively interesting parts of computer science (because they are taught in 15-251), and formalism is hard to get excited about without these.

"15-151 is harder than 21-127"

A common student complaint during (and after) the course is that 15-151 was harder than it's predecessor. While it is true that some of the questions in 15-151 were intentionally more difficult than their 21-127 analogs, we did our best to offset this in multiple ways. First, those difficult questions were almost exclusively on the homework. Second, we eliminated time pressure on exams by designing exams to be completable in a significantly shorter time period than the students had. Third, we gave a significantly more generous curve than 21-127 gives. And finally, we gave significant, extra support to the students via the course model and office hours. We believe that, ultimately, students learned significantly more, and were happy they had taking 15-151 when they got to 15-251.

While the (final) grades students got in 15-151 were not always analogous to those they might have received in 21-127, it is important to remember that one of the explicit goals we took on was to pass only students who could succeed at 15-251. Unfortunately, 21-127 has a history of being very inconsistent in this respect.

Our strategy allows students who were not ready for 15-251 at the end of the semester the opportunity (though, they admittedly often did not see it that way) to take 21-127 afterwards and see the material again before moving on to 15-251. In the past, concepts did not provide a signal in this respect: students would often arrive in 15-251 with no hope of passing. And, unfortunately, they would get stuck in 15-251: a course where the problems are incredibly difficult and the basic concepts are assumed when what they really needed was a review of those basic concepts. Furthermore, traditionally, 15-251 is somewhat more reasonable in the Fall, as opposed to the Spring, putting students who needed a second semester to get ready in a prime position to succeed the first time in 15-251.

In an ideal curriculum, the need for such a solution would not be necessary, but 15-251 is traditionally taken by students in the Spring of their Freshman year. Given our data that students who take 15-112 (introductory programming) in their Freshman Fall are significantly more likely to be in the group that needs the extra time, and that independent research ([HM00, WS01]) supports this, we recommend one of the following:

- Make 15-112 a *pre*requisite for 15-151 instead of a *co*-requisite (or recommend that students take 21-127 after they have completed 15-112)
- Make 15-122 a prerequisite for 15-251

These changes either force or increase incentives for students who are less sure what computer science involves to delay their foundations course until they at least have a grasp of programming.

Chapter 3

Pedagogy in the Discrete Math Classroom

In this chapter, we describe a way to provide the students with multiple opportunities for exposure (lecture, group work, homework), extra opportunities for practice (group work, homework, recitation), and an aligned form of summative assessment (untimed exams). This model allows students to get a deep understanding of discrete math and proofs, because we *expose* it in a variety of ways, let them *actively work* on problems, and help them form bridges between various levels of *abstraction*. Our model of a discrete math classroom uses collaborative learning as both a means to practicality of implementation *and* a desired outcome of the course. At MIT, the main foundations course, 6.042, uses some very similar ideas, and we describe how our implementation differs throughout.

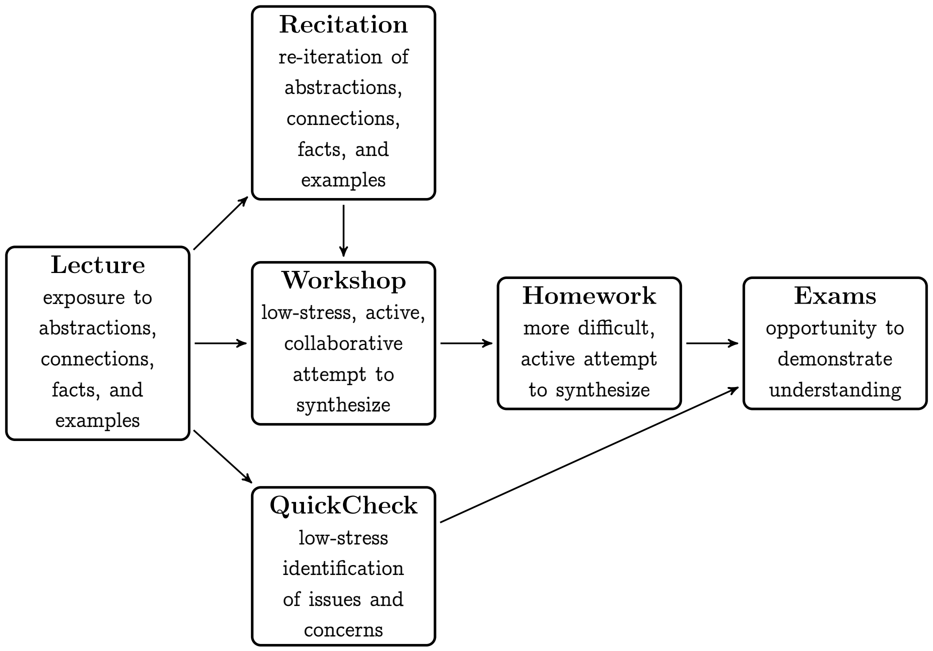

Recitation
re-iteration of abstractions, connections, facts, and examples
Lecture
exposure to abstractions, connections, facts, and examples
Workshop
low-stress, active, collaborative attempt to synthesize
Homework
more difficult, active attempt to synthesize
Exams
opportunity to demonstrate understanding
QuickCheck
low-stress identification of issues and concerns

Figure 3.1: Flow of learning from instruction to assessment in our model

3.1 Transition from High School

As most instructors who have taught freshmen know, there is a substantial difference between first-semester freshmen, second-semester freshmen, and other undergraduates. For many students, it is their first extended period away from home, and it is their first time with the freedom to make all of their own choices. In particular, time management becomes a very important skill to learn and master. For better or worse, this heavily influences the design of the course as well as its policies. We made as much of an effort as possible to be firm in policies, but forgiving in individual instances.

We suspect some of the most major differences between our implementation and 6.042 stem from the fact that the majority of their students are sophomores, not first-semester freshmen.

Computer Science students at Carnegie Mellon have almost universally been at the top of their class up until this point. This creates the very real possibility that the introductory discrete math course is the first time they will ever be given a question they legitimately wo not be able to solve which, in turn, creates the first time students ever need to ask for help. This can be very demoralizing to students, and it too influences policies and design of the course. We kept very careful track of students who were struggling and did our best to seek them out to give them extra help. Because of the impact this issue has on enjoyment, we paid special attention to it and attempted several solutions throughout the iterations of 15-151. In particular, our most important goal was to make students comfortable coming to office hours as soon as possible.

3.2 Course Meetings

Unlike most other undergraduate courses, 15-151 meets *five* days a week: three days of “lecture” and two days of “recitation”. The “lecture” days are split between more *traditional lectures*, in which we present new material, and *workshops*, in which students solve problems and write proofs in small groups. In total, the second time 15-151 ran, there were 19 lecture days and 22 workshop days; so, they were split about evenly. We further break down the goals of each course meeting as follows:

- 16 days on proof writing
- 29 days on learning discrete mathematics
- 7 days on problem solving
- 3 days on avoiding mistakes
- 6 days on exploring a particular extension of a topic

3.2.1 Lecture

Students are expected to actively participate in lecture. Most lectures involve large “check your understanding” components, in which students talk to each other to solve short, conceptual problems. This is a form of “peer instruction”, piloted by Eric Mazur, for usage in teaching introductory physics classes [Maz09], and such techniques have become increasingly popular since. Peer instruction traditionally uses clickers and multiple choice questions, but our questions vary from discussing how to write part of a proof to checking if a function is injective. In a divergence from most courses of this type, we *do not* have a flipped classroom. The students are not expected to do reading or watch lectures outside of class time. For example, 6.042 *does* have a completely flipped classroom. As a result, they are able to spend *all* of their class time is what we call workshops.

The biggest sacrifice we make for *not* flipping the class room is that we prove very few "fundamental" results during lecture. In 15-151, these sorts of results generally appear toward the end of the list of the workshop questions. The reasoning for this is simple: if students are unable to do more elementary proofs or exercises with a topic, they will likely not understand a proof of a complicated result, and their time is better spent trying to understand more basic material, anyway.

Fostering an "Active Classroom"

We found that the most significant piece of creating an "active classroom" was to establish *new expectations* for the students. This amounts to convincing the students to participate (productively or not) early in the course; then, as the course progresses, they naturally continue to participate and attempt every posed question.

In the first run of 15-151, we wrote a simple web application called "CourseTime" which allowed students to enter answers (often proofs; sometimes results) to posed exercises during lecture. "CourseTime" allowed the instructor to simultaneously push the question to all the student groups, monitor the results and time, and choose solutions to show to the class. Ultimately, we found the system to waste time, because the entry format for symbols (i.e. $\LaTeX$) is far too slow for introductory students. However, aspects of the system worked very well:

- Because there was something collected, students almost universally participated
- Because we displayed solutions *anonymously*, students were able to claim them when they were proud and remain anonymous otherwise
- Because we knew how many solutions were submitted, we had a good idea of how many students had finished solving the problem

Perhaps the most interesting thing is that the attitude of participation continued once we stopped using the system (because of the time concerns) and asked students to solve problems in groups anyway. A variety of students even volunteered to answer questions without the system.

In the second run of 15-151, we never used "CourseTime" at all and attempted to eyeball how many students had finished. In many cases, students did not attempt problems, and fewer students answered questions.

Postmortem: Active Lectures

In general, students responded positively to the active lectures; most of them usually at least attempted the questions posed. Attendance at lectures was nearly full– despite explicitly telling the students that it was not required!

Our approach certainly makes it take longer to cover domain content, but, in our case, this was something we could afford! Even at our slower speed, we covered at least as much material as the traditional 21-127 course. We believe that most students felt more productive and comfortable in our form of lecture, as opposed to traditional ones.

3.2.2 Recitation

Recitation time is mostly spent on more simple examples intended to drive home the most important points from lecture and workshop. Days where there is no new topic to discuss are often spent discussing common mistakes or reviewing a particularly difficult topic.

We anticipate that some might claim that recitation for this course (in particular, two of them a week) is overkill. It is worthy of note that the first time 15-151 ran, we attempted to use this time for entirely supplementary material, and we found that the extra support that spending recitation on review gave was absolutely worth the time spent on it.

Again, we suspect that this is influenced substantially by the fact that students in 15-151 are all first-semester freshmen. This is supported by the fact that 6.042 has no recitations at all.

3.2.3 Workshops

Workshops are a type of course meeting held in some of the lecture time slots. Students work in small groups of roughly 3 people to practice what they learned in lecture by solving problems, and writing up their solutions on a whiteboard. Importantly, students get feedback on their proof writing immediately–without any negative consequences to their grade. It also provides them with an opportunity to learn to work better in groups to solve problems.

Workshops intentionally have more problems than we expect students to solve. In fact, only two groups ever completely finished a workshop. The last problem is usually substantially more difficult than the others to give students who are faster a chance to solve more interesting problems.

We do not score participation, beyond students who do not attempt to solve anything at all. We labeled some homework questions as "checkpoint" questions; for these questions, we expected students to bring some indication of effort to the workshop. These "checkpoint" questions were an attempt to force students to review basic definitions before the workshop, but we did factor them into credit for the workshop. 6.042 has short quizzes at the beginning of class, called preparation checks, if there is no other quiz scheduled for that day that serve a similar purpose.

In order to keep students on task, we assign groups randomly, and we ensure no student will be assigned a group-mate multiple times. We change the group assignments every 2-3 workshops.

Postmortem: Workshops

Workshops are significantly more "active" and interesting for students. Many students looked forward to workshops to help solidify their understanding of topics they had learned recently. This extra practice led to misconceptions being cleared up earlier than normal (i.e. in the workshop formative assessments rather than the exam summative assessments), because TAs and instructors checked their work at most steps. Although students were still not particularly interested in correcting their work, the format of workshops allows TAs and instructors to convince them that corrections can be learning experiences as well.

Students work in groups on their homework in 15-151, but they have little previous experience with teamwork. Workshops gave us a chance to help students develop best teamwork practices and monitor that development.

Perhaps the biggest downside to workshops is the increase in resources (TAs, preparation time). Depending on how many TAs a course has, their availability, and their understanding of material, it can be a tall order to pay and maintain the extra hours. For reference, during workshops in 15-151, one teaching assistant was assigned to between 9 and 13 students; given that we only had 8 teaching assistants, we solved this mathematical problem by having three smaller lectures with fewer students in each one.

Two outstanding research problems that we have begun considering but do not have any results on yet are:

- Students (in our case *freshmen*) who come in under-prepared can both make their group go considerably slower, *and* be completely left behind by a group rushing to finish.
- Although there is a lot of research on grouping younger children by ability and interest, to our knowledge there is very little about college-age students–particularly in a problem-solving environment.

The questions in each workshop varied from beginning with a basic review of definitions to difficult, homework-level problems placed last. We–intentionally–put no requirements on *how many* problems students had to solve, but we did insist they go in order. This had a natural effect of groups speeding through problems they considered easy, and eventually getting stuck (for some time) at problems at their *proximal zone of development*. In other words, most workshops, we challenged nearly everybody and helped them solve problems they would not have been able to before class.

For almost the entire semester, lectures and workshops were in a 1:1 ratio. We introduced a new topic in lecture, students tried to use it two days later in workshop, and solidified their understanding on a homework assignment. Perhaps the only exception to this is the part of the course about "proof strategies", in which we explained the proof format quickly at the beginning of the workshop, and let students dig in. This was particularly convenient, because going through an entire lecture explaining a proof strategy without being able to ask questions rarely results in full understanding.

Ultimately, we consider workshops to have been very successful. Without them, we would not have been able to simultaneously challenge everyone, help students learn to work in groups, or provide such varied content.

3.3 Exams and Quizzes

In the 15 week semester, we have *three* exams, each of which is cumulative, but focused on the recent material, and a final. If students do not miss more than three workshops, we *completely drop* their lowest exam score and replace it with their final exam score. The intention here is to give students a second chance.

3.3.1 Exams

In the spirit of testing understanding and not how quickly students can solve problems, we give students *three hours* to complete an exam designed to take between $\frac{1}{3}$ and $\frac{1}{2}$ of that time period. We also allow them cheat sheets on most of the exams, because we are far more interested in conceptual understanding than memorization. Even with cheat sheets, one of the questions on the exam is often a problem that they've seen somewhere else in the course.

Unfortunately, some students still felt time-pressured; ideally, we would have given take-home exams, but given the level of cheating that occurred, we were uncomfortable with that choice.

We found that many students who did poorly on the first exam justified their scores by noting that they could just drop that exam. Unfortunately, this allowed them to convince themselves that they would magically do better on the second exam. So, they did not attempt to correct their study habits and understanding of material. Perhaps only allowing them to drop the second or third exam would fix this problem.

3.3.2 QuickChecks: Formative Assessment

QuickChecks are quizzes that are proctored in recitation intended to be exactly like a very short exam (usually one question). Depending on the QuickCheck, students get between 5 and 10 minutes to complete it. The major way that QuickChecks differ from quizzes is that *they do not factor into students' grades*. They serve an entirely formative role; since they mirror exam questions, not being able to solve one is an indication to students that they would bomb that part of the exam. Additionally, they give students low-stakes practice with time pressure. QuickChecks *are* collected and corrected; so, students get feedback and have an incentive to actually try. In recitation, after the QuickCheck is collected, TAs *immediately* go over the solution and any questions or concerns students have. In this way, the question is still fresh in their mind.

6.042 has "miniquizzes" which are factored into students' grades.

In our opinion, QuickChecks were one of the best ideas from the course. They give students a reason to attempt at least one problem every recitation, and with a little extra effort, they get significantly more *formative* feedback. By reviewing the solution *immediately after* the QuickCheck, while their ideas are still fresh, we help students correct misconceptions before they get too far.

3.4 Results and Conclusions

Ultimately, we set out to create a course that supported the many meta-goals of a discrete mathematics for computer science students course. We believe that we supported our meta-goals as follows.

- We supported **formalism** providing extra attempts for students to master the various subtleties of a "formal proof" in workshop and recitation. All of the handouts and slides had *complete* proofs (sometimes annotated). We often discussed examples of proofs that were not formal enough and why in lecture.
- We successfully **used programming to make them like math** by creating programming assignments to tie together math and programming, some of them were not particularly engaging to students, but others were among the most liked assignments in the course.
- We provided a significant number of opportunities for students to practice **working in a group** during lecture, workshops, and homeworks.
- While we were not particularly interested in **mathematical problem solving** as a meta-goal of our course, we did want a weaker form: "mathematical maturity". In our case, we wanted them to mathematically mature enough to succeed in 15-251. To accomplish this, we provided a variety of difficulty of questions, but continually exposed them to a small amount of problem solving on each summative assessment. Students who continued on to 15-251, generally said that it was significantly easier than for their counterparts who took 21-127, at least for the first half of the course.
- We used time during lecture to **convince students that math is important to computer science**. Any time we moved on to a new topic, before discussing the base material, we gave an overview of the variety of applications of the topic.

We believe our strategies in 15-151 were successful (despite a few techniques that did not quite work), because students were engaged, interested, challenged, and successfully prepared for the rest of the Computer Science major.

Part II

Using Technology to Teach Discrete Mathematics

Chapter 4

{Setʇy}: A Programming Language for Teaching Discrete Mathematics

Introductory discrete mathematics courses are almost always responsible for introducing students to the symbols, phrases, and definitions that are used in proofs. This can be very difficult for students, because the syntax is usually completely foreign, and the semantics of written mathematics can be very confusing. For example, classical implication is very counter-intuitive, because students often initially believe that it is related to *causality*, but it is not.

Learning the "language of mathematics" is just as difficult as learning a new programming language or a foreign language, and, because of this, it is very important that these courses help students in whatever way possible. {Setʇy} gives students a computational environment in which they can explore mathematical language. Instead of guessing if a statement they wrote on paper is true, they can check their understanding by instantiating it in various ways using {Setʇy}. Because students must learn the syntax of the "language of mathematics" as part of the course normally, {Setʇy} provides very little overhead.

We begin this chapter by discussing, in more detail, why a language like {Setʇy} might be useful in this setting (and particularly useful for computer science students). Then, we give a description of the syntax and interesting semantic choices for the language. Finally, we provide some sample exercises that show how students might use {Setʇy}, both to learn simple things like how to negate a logical statement, and, also, to test hypotheses while solving problems.

4.1 Motivation

Generally, when computer science students take foundations of mathematics courses, they (1) *are familiar or comfortable* with programming, and (2) *unfamiliar or uncomfortable* with logic, sets, and formalism. We designed our programming language, {Setʇy}, to have syntax and semantics as close as possible to the logic, sets, and formalism that are *new* to students. This allows us minimum overhead (in teaching syntax) and maximum leverage (in teaching semantics) in teaching students the "language of mathematics".

Depending on the course, the "language of mathematics" can be slightly different, but the general idea is nearly always identical. The differences are usually on the token level (symbols and keywords), and they can be changed trivially. The "language" we are teaching is *actually* as complicated as any real-world language like English. Part of teaching proofs is helping students understand that there is a structure to them in the same way that there is a structure to essays. While we do not pretend that they will always do the equivalent of a "five paragraph essay", we start there, because it is easiest to emulate and explain. Consider the following (annotated) example of a proof that might show up in a foundations course:

Claim: $A \setminus C \subseteq (A \setminus B) \cup (B \setminus C)$
Proof:

Introduction $\left\{\begin{array}{l}\text{We want to show that for any } x,\ x \in A \setminus C \Rightarrow (A \setminus B) \cup (B \setminus C), \\ \text{because this is the definition of subset.}\end{array}\right.$

Declarations $\left\{\begin{array}{l}\text{Let } A, B, C \text{ be sets. Suppose } x \in A \setminus C. \text{ Then, by} \\ \text{definition of set difference, } x \in A \wedge x \notin C.\end{array}\right.$

Sub-strategy $\left\{\begin{array}{l}\text{We continue by cases: } x \notin B \text{ or } x \in B. \\ \text{Case 1} \left\{\begin{array}{l}\text{Suppose } x \notin B. \text{ Then, because we also know that} \\ x \in A, \text{ by definition of set difference, } x \in A \setminus B.\end{array}\right. \\ \text{Case 2} \left\{\begin{array}{l}\text{Now, suppose } x \in B. \text{ Then, because we also know that} \\ x \notin C, \text{ by definition of set difference, } x \in B \setminus C.\end{array}\right.\end{array}\right.$

Conclusion $\left\{\begin{array}{l}\text{Finally, since } x \in A \setminus C \Rightarrow (A \setminus B) \cup (B \setminus C), \\ \text{the claim follows by definition.}\end{array}\right.$

In this example, we've annotated specific parts of the "language of mathematics" to indicate special words and vocabulary. Specifically, the red phrases are "connecting phrases"–in a way, they are similar to "transitions" in an essay. Clearly, these sorts of phrases are exchangeable, but their presence is important. The blue symbols are vocabulary that students have to learn. The purple words are overloaded English words that have a more specific meaning in the "language of mathematics", and these pieces of vocabulary can be particularly difficult for students, because they are written the same way as the informal ones.

This example also demonstrates that this language has sentence fragments (e.g. $A \setminus C$ is a fragment, but $A \setminus C \subseteq B$ is a sentence). We've also carefully demonstrated that the proof can be broken up into "sections" which mirror those of an essay.

The important conclusion here is that the language they are learning is complicated, and we expect them to go from nothing to writing essays in one course. When we teach logic and sets, students need to learn vocabulary, syntax, sentence structure When we teach proofs, they need all of this *and* how to put the structure together.

{Setʇy} is intended to help students learn many of these pieces in the following ways:

(1) Use their understanding of programming to help them learn the new syntax. Students often do not realize that there is an actual syntax, they think it is okay to get symbols "sort of right". They do understand the idea of "invalid syntax" in a program though! So, by asking them to write some programs using the mathematical language, we get this for free.

(2) Use their understanding of programming to help them learn the new semantics. A student who sees the claim above ($A \setminus C \subseteq (A \setminus B) \cup (B \setminus C)$) might not understand what it even *means*. As the vocabulary of the new language increases, students get more confused. Using {Setʇy}, a student might write the following program:

```
A := {1, 3, 5}
B := {2, 4, 6}
C := {4, 9, 2}
print A \ C, (A \ B) ∪ (B \ C)
```

Importantly, there is *no reading/appealing to more definitions*! While learning to read and understand definitions is absolutely an important skill, some students will take longer to be able to do it. This allows them to verify that their understanding is (or is not!) correct.

(3) Use their understanding of programming to give them more intuition on the things they are trying to prove. In other words, if the question were asking instead: "Prove or disprove: $A \setminus C \subseteq (A \setminus B) \cup (B \setminus C)$", a student could write a program to try to figure this out. For example:

```
for A ∈ 𝒫([3]):
   for B ∈ 𝒫([3]):
      for C ∈ 𝒫([3]):
         if not subset(A \ C, (A \ B) ∪ (B \ C)):
            print "Counter−example:", A, B, C
print "Done with all tried examples."
```

Again, the point to stress here is that they are using a for loop and an if statement. Everything else here is just the "mathematical language" that they're trying to learn.

4.2 {Setʇy} Language Description

{Setʇy}'s syntax is modeled on a combination of Python and an unambiguous version of the part of the "language of mathematics" typically taught in an introductory discrete mathematics course. Whenever possible, it matches the symbols and words that students are expected to use in proofs to communicate mathematically. As it takes after Python, whitespace is significant. In an attempt to match notation as much as possible, {Setʇy} allows programmers to interchange Unicode symbols for keywords. For instance, the symbol "∀" is interchangeable with the keyword `forall`.

4.2.1 Primitive Types

In {Setʇy}, all values are sets. For convenience, if {Setʇy} can infer that the user intends to treat a value as one of the three primitive types, it will pretty print it. But users need to understand the underlying set representation, because occasionally sets are coincidentally another type. This is an intentional design decision intended to help students understand how to build other types out of sets.

It is perhaps useful to think of this set-based representation as analogous to the two's complement representation of integers, as it applies to C. That is, writing the expression `2147483647 + 1` in C will overflow, and unless the user understands the underlying bit representation, the result will seem incorrect.

Sets

In {Setʇy}, there are no urelements[1]. The `set` type is the set of *hereditarily finite sets*. Formally, let $V_0 := \varnothing$ and $V_k := \mathcal{P}(V_{k-1})$ for $k \in \mathbb{N} \setminus \{0\}$. Then, S is a value in {Setʇy} if and only if $S \in V_\omega := \bigcup\limits_{k=0}^{\infty} V_k$.

The easiest way to instantiate a set is as a set literal. The empty set is written as `@`, $\varnothing$, or `{}`. If $e_1, e_2, \ldots, e_k$ are expressions, then $\{\mathrm{e_1}, \mathrm{e_2}, \ldots, \mathrm{e_k}\}$ is a valid expression as well. In other words, the standard set notation is valid {Setʇy} code.

Natural Numbers

{Setʇy} allows the programmer to use natural numbers as shorthand for the sets they actually represent. Theoretically, {Setʇy} could use any reasonable definition, but it uses the *von Neumann ordinals*, as they have a number of useful properties. The *von Neumann ordinals* are defined as follows:

- $0 := \varnothing$
- $n := (n-1) \cup \{n-1\}$

Unrolling the definition, you can see that $n = \{0, 1, \ldots, n-1\}$. Since this is a definition, the numerals and corresponding sets may be used interchangeably in code.

Booleans

{Setʇy} allows the programmer to use `T` and `F` as shorthand for true and false, respectively. When a set is used as an operand to a boolean operation (such as and, or, etc.), the empty set evaluates to `F` and all other sets evaluate to `T`. This naturally corresponds to the question $\exists x . x \in S$ for any set S. Only primitive boolean operations built into the language return `T` or `F`, and this limitation was intentional; so, that it can be used as an introductory boolean algebra exercise.

[1]A *urelement* is an "atomic" non-set element that may be a member of a set.

4.2.2 Primitive Operations

Set Algebra

{Setʇy} supports basic set operations like union, intersection, and difference. In {Setʇy} code, union is written either as $\mathtt{e}_1 \cup \mathtt{e}_2$ or $\mathtt{e}_1$ union $\mathtt{e}_2$. Similarly, intersection is $\mathtt{e}_1 \cap \mathtt{e}_2$ or $\mathtt{e}_1$ intersect $\mathtt{e}_2$, and difference is $\mathtt{e}_1 \setminus \mathtt{e}_2$ or $\mathtt{e}_1$ minus $\mathtt{e}_2$. {Setʇy} has no concept of a "universe"; so, attempting to calculate values like $\cap\varnothing$ raise a run-time error.

Boolean Algebra

{Setʇy} has native support for boolean algebra as well. Conjunction, disjunction, and negation are built-in (as `and`, `or`, `not`) respectively. Implication (`implies`) and bi-implication (`iff`) are mapped to the user-defined functions with those names. In other words, once the user implements them, the syntax is as expected.

Boolean Operators

Since all values are sets and numbers are interpreted as *von Neumann ordinals*, the only boolean operator necessary is set containment. In {Setʇy}, this is just $\mathtt{e}_1 \in \mathtt{e}_2$, or $\mathtt{e}_1$ in $\mathtt{e}_2$. Like with implication, $<$, $>$, $\leq$, $\geq$, $\subseteq$, and $=$ may be implemented by the user. This is useful as an early exercise to help verify students understand von Neumann ordinals and basic set operations. The "special function names" to define the behavior of these operations are as follows: `lt`, `gt`, `geq`, `leq`, and `subset`.

In practice, {Setʇy} implements $<$ as a very complicated ordering that respects natural numbers, pairs, nesting, and cardinality. This became necessary to ensure design decisions (see below) were implemented correctly. Additionally, in the interest of speed, equality testing is also built into {Setʇy} directly.

Additionally, the user may specify a `powerset` function which will allow the compiler to treat the `subset` function as a partial order wherever necessary.

4.2.3 Variables and Statements

Variables in {Setʇy} can be local or global, and they always have type set. Automatic type promotion from "boolean" to "natural number" to "set" is done by the compiler, as required. Variables are declared using $\mathtt{v}_1 := \mathtt{e}_1$, where v_1 is a variable identifier and e_1 is an expression. {Setʇy} also allows several "definition statements" in which users can specify representation of more complicated structures. The most important is `pair`. To define the internal representation of the expression $(\mathtt{v}_1, \mathtt{v}_2)$, the user may write $(\mathtt{v}_1, \mathtt{v}_2) := \mathtt{e}_1$, where v_1 and v_2 are free in e_1. The intended representation is the Kuratowski definition of pair, namely: $(\mathtt{v}_1, \mathtt{v}_2) := \{\{\mathtt{v}_1\}, \{\mathtt{v}_1, \mathtt{v}_2\}\}$. The standard library provides such a definition. To make pairs useful, the user must also define projection functions, π_1 and π_2, which can be implemented in {Setʇy} as `pi1` and `pi2`. This makes a nice early exercise for sets. {Setʇy} also allows the user to define $[n]$ by writing $[\mathtt{v}_1] := \mathtt{e}_1$ for a variable $\mathtt{v}_1$ and an expression $\mathtt{e}_1$.

4.2.4 Functions

All functions in {Set∤y} take a single argument of type set and return a set. As a convention, if a function is intended to be nullary, it should always be given the argument $\varnothing$. Note that function names and variable names are distinct; so, a user may specify a function called f and a variable called f in the same scope. The BNF for function definitions and function calls is as follows:

$\langle$*function-call*$\rangle$ ::= **name** ($\langle$*expr*$\rangle$)

$\langle$*function-definition-stmt*$\rangle$ ::= $\langle$*func-def-inline*$\rangle$ | $\langle$*func-def-suite*$\rangle$

Functions must be declared before use. {Set∤y} supports three notations for functions

One-line "Mathematical" Functions

For simple manipulations, {Set∤y} allows users to write single line functions. Notationally, these are intended to mirror function definitions like $f(x) = x^2$ or $f(S) = S \cup \{5\}$. As such, the BNF is
$\langle$*func-def-inline*$\rangle$ ::= $\langle$*function-name*$\rangle$ ($\langle$*argument-name*$\rangle$) := $\langle$*expr*$\rangle$ \n

Standard "Programming" Functions

Multi-line functions mirror more traditional programmatic function definitions. They are indent-delimeted, and they *do* require the usage of the `return` keyword. A "function" that does not end all paths in a return $\mathtt{e}_1$ statement is considered invalid. Otherwise, the notation is identical to a one-line function:
$\langle$*suite*$\rangle$ ::= \n **indent** $\langle$*stmt-list*$\rangle$ **dedent**

$\langle$*func-def-suite*$\rangle$ ::= $\langle$*function-name*$\rangle$ ($\langle$*argument-name*$\rangle$) := $\langle$*suite*$\rangle$

Piecewise Functions

{Set∤y} allows the user to create multiple clauses of a function. For instance, `Fibonacci` in {Set∤y} might be written:

```
f(0) := 0
f(1) := 1
f(n) := f(n−1) + f(n−2)
```

Each of these clauses may be either a one-line or multi-line function, themselves. If there are redundant clauses, {Set∤y} throws a compiler error. But functions do not necessarily have to be exhaustive. For instance, one could write the negation function in {Set∤y} as:

```
neg(T) := F
neg(F) := T
```

In this case, evaluating the expression `neg(5)` would cause a run-time error. This distinction is necessary to preserve the user experience that there is only one type.

4.2.5 Input and Output

{Setʇy} deliberately does not allow access to files (or file descriptors). {Setʇy} has two functions that allow the user to "print": `print` and `printx`. These two functions are identical, except that `print` always appends a newline, whereas `printx` does not. `print` statements are defined with the following BNF:

$\langle$*print-stmt*$\rangle$ ::= `print` $\langle$*print-list*$\rangle$

$\langle$*print-list*$\rangle$ ::= $\langle$*expr*$\rangle$
| **string**
| $\langle$*print-list*$\rangle$, $\langle$*expr*$\rangle$
| $\langle$*print-list*$\rangle$, $\langle$*string*$\rangle$

That is, users may print expressions and "strings". For instance,

```
print "this is {1,2} union {3,4}", {1,2} union {3,4}.
```

4.2.6 Conditionals and Loops

Conditionals and loops in {Setʇy} derive their syntax from Python. They are whitespace delimited, and the initial statement is terminated with a colon. There is no "else if" construct. There are two types of for loops: one that iterates over numbers and another which iterates over the elements of a set.

Since sets are unordered structures, there are no guarantees on the order of execution of the loops that iterate over sets. Each element is guaranteed to be iterated over, however. All numerical loops start at *zero* and loop *until* the upper bound. In other words, if the upper bound of the loop is n, then the for loop iterates exactly n times.

Upon each iteration, any changes to the upper bound/set being iterated over and the variable of iteration will be reset to their proper values. In this way, unless the user *returns* inside a loop, they will always execute each iteration exactly once. The BNF for conditionals and loops is as follows:

$\langle$*conditional-stmt*$\rangle$ ::= `if` $\langle$*expr*$\rangle$ `:` $\langle$*suite*$\rangle$
| `if` $\langle$*expr*$\rangle$ `:` $\langle$*suite*$\rangle$ `else :` $\langle$*suite*$\rangle$

$\langle$*loop-stmt*$\rangle$ ::= `for` **name** `in` $\langle$*expr*$\rangle$ `:` $\langle$*suite*$\rangle$
| `for` **name** `to` $\langle$*expr*$\rangle$ `:` $\langle$*suite*$\rangle$

4.2.7 Set Comprehensions

Many languages (like Python and Haskell) provide "list comprehensions" which (in Python syntax) look like:

```
[f(x) for x in L if cond]
```

A similar, but more general, notation is used to specify sets in mathematics. {Setʇy} attempts to emulate this generalized notation and allows users to enter sets like:

```
{(x,y) | x in [5] and x < y}
{x | subset(x, Y)}
{x | y in [2] and z in [3] and x = y*z}
```

To accomplish this, {Set⌉y} automatically infers free variables in set comprehensions wherever possible. Without any intervention, {Set⌉y} handles all built-in functions and relations. Additionally, users can specify how their functions should be interpreted in set comprehensions (as partial orders) and {Set⌉y} will automatically start understanding them. For instance, if a user specifies that a (decision function) $A(x, y)$ is a partial order on $\mathbb{N}$, then {Set⌉y} can start inferring that $x \in \mathbb{N}$. The {Set⌉y} compiler attempts to infer free variables by using the built-in and user specified rules recursively. If there are multiple restrictions, {Set⌉y} will take the most restrictive universe that it can. Additionally, the compiler uses a topological sort to ensure that the dependencies are actually well-defined. Examples (in real code) of how this is useful are plentiful:

Here, the compiler can automatically infer the fact that the domain of x is the set $N := [10]$, without this being explicitly specified.

```
N := [10]
print {x | y in N minus {0,1} and z in N minus {0,1} and x = y*z}
```

In this code, the compiler can infer the domain of y, and, so it is okay that y's universe is not specified.

```
equivalence_classes(p) :=
   S    := pi1(p)
   dom  := pi1(pi2(p))
   codom := pi2(pi2(p))
   return { {y | (x,y) in S} | x in dom}
```

The grammar for set comprehensions is very simple. {Set⌉y} supports explicitly defining the universe for a variable and also letting {Set⌉y} try to infer it:

$\langle$*set-comprehension*$\rangle$ ::= { **name** in $\langle$*set-expression*$\rangle$ | $\langle$*expr*$\rangle$ }
  | { $\langle$*expr*$\rangle$ | $\langle$*expr*$\rangle$ }

4.2.8 Quantifier Expressions

{Set⌉y} supports first-order logic expressions as a primitive. It insists on a notation that is not quite standard to force students to understand that notation in math, like in programming, must be consistent. In particular, quantified statements in {Set⌉y} look like $\forall(x \in N).P(x)$. It does, additionally, support quantifying over multiple variables simultaneously, as is common in mathematics, as well. Here is the BNF:

$\langle$*quantifier*$\rangle$ ::= ∀ | `forall` | ∃ | `exists`

$\langle$*var-pair-expression*$\rangle$ ::= (**name** , **name**)

$\langle$*quantified-expression*$\rangle$ ::= $\langle$*quantifier*$\rangle$ (**name in** $\langle$*set-expression*$\rangle$) . $\langle$*expr*$\rangle$
  | $\langle$*quantifier*$\rangle$ ($\langle$*var-pair-expression*$\rangle$ **in** $\langle$*set-expression*$\rangle$) . $\langle$*expr*$\rangle$

4.2.9 Decorators

Another idea that {Setʇy} borrows from Python (or Java) is the idea of a function decorator (or annotation). As of the current implementation, specific decorators are built into the language, but a future version should allow users to define their own. {Setʇy} provides three very important decorators:

Requires Decorator. Invariants and contracts are a key concept taught in 15-122, Carnegie Mellon's CS2 course. Since many of the students in 15-122 also take the introductory discrete math course, we added a decorator that is identically named to the on in 15-122's language C_0 (a safe "subset" of C). The `@requires` annotation takes an expression and evaluates it at runtime, throwing an assertion error if the expression evaluates to `0`.

Recurse Decorator. This will be discussed in significantly more detail later in this chapter, but the recurse decorator takes a function of a function's arguments and throws a runtime error if a recursive call does not evaluate to a strictly smaller value.

No-Memoization Decorator. By default, {Setʇy} memoizes every function. This decorator (basically useful for "print" functions) tells {Setʇy} that it should not use a memoized value.

The BNF just allows users to prepend one or more decorators to a function definition:

⟨*requires-stmt*⟩ ::= `@requires(` ⟨*expr*⟩ , **string** `)`

⟨*recurse-stmt*⟩ ::= `@recurse(` ⟨*expr*⟩ `)`

⟨*nomemo-stmt*⟩ ::= `@nomemo`

⟨*func-with-decorators*⟩ ::= ⟨*function-definition-stmt*⟩
| ⟨*requires-stmt*⟩ `\n` ⟨*func-with-decorators*⟩
| ⟨*recurse-stmt*⟩ `\n` ⟨*func-with-decorators*⟩
| ⟨*nomemo-stmt*⟩ `\n` ⟨*func-with-decorators*⟩

4.3 Language Design

{Setʇy} has several design decisions which deliberately differ from most standard languages. Each of these was a deliberate decision that influences how {Setʇy} can be used as a pedagogy tool. We discuss the specific implementation and goals for these features in this section.

4.3.1 Type Coercion and Internal Representation

From the user's perspective, every variable has the same type. This allowed students to get familiar with the idea that everything necessary for computation can be built out of sets. Unfortunately, just because something is possible does not mean that one would want to do it in practice. In particular, because the *von Neumann* definition of the natural numbers grows exponentially in the number of empty sets, if the compiler *actually* used it, {Setʇy} would be unable to print out the value of $15 - \{\varnothing\}$, for instance.

{Setʇy} is as lazy as possible about representing sets. All sets are internally represented as *balanced trees*, and the {Setʇy} compiler makes function calls to a (modified) version of the pblSet library.

All sets contain the following fields in the internal representation. These are updated whenever sets are mutated.

- `bool num_nonnumber` – This field records the number of sets that cannot be interpreted as numbers that are in the set.
- `long max_number` – This field records the maximum *number* stored in the set. It is -1 if there are no numbers in the set.
- `long nesting_number` – This field records the *depth* of the set; that is, the deepest number of { in the set. Numbers are considered to have a nesting number of 0.
- `bool is_lazy` – A lazy set is one which represents a number, but does not *actually* have the recursive elements in it. This is useful to avoid storing the set representation if it is never seen by the user.
- `long size` – This is the cardinality of the set.

We say a set, s, can be considered a number iff `s.num_nonnumber == 0` and $|s| - 1$ `== s.max_number`. That is, s only contains other numbers and the maximum number it contains is one smaller than its value. Then, the numerical value of s is just its size.

4.3.2 Total Ordering on Sets

{Setʇy} provides a *total ordering*, $\prec$, on sets that extends several partial orderings. In particular, the following properties are true of the total ordering:

- If $S_1 < S_2$, $S_1 \prec S_2$.
- If $S_1 \subset S_2$, $S_1 \prec S_2$
- $\mathsf{F} \not\prec \mathsf{0}$, $\mathsf{0} \not\prec \mathsf{F}$
- If $S_1 \in \mathbb{N}$ and $S_2 \notin \mathbb{N}$, $S_1 \prec S_2$
- $(A_1, B_1) \prec (A_2, B_2) \Longrightarrow (A_1 \prec A_2 \vee B_1 \prec B_2)$

The total ordering is computed recursively as is shown in Figure 4.1.

4.3.3 Guaranteed Termination

{Setʇy} is deliberately not Turing-complete. In order to facilitate reasoning about termination, {Setʇy} does not have while loops, requires certain restrictions on recursion, and function definition. In this way, students are forced to make their programs correct at the stage of *writing*–rather than delaying to the proof stage. This is discussed more in the particular examples from the classroom. Here we are more concerned with *how* {Setʇy} forces termination of programs.

Since all sets in {Setʇy} are finite, the only possible sources of non-termination occur in loops and recursion. We discuss the design decisions that disallow non-termination in these cases as well. Formally, one could prove that all {Setʇy} programs terminate using

```
int set_elem_compare(a, b) {
   if (is_set(a)) { a = coerce_to(a, SET); }
   if (is_set(b)) { b = coerce_to(b, SET); }
   if (type(a) == type(b) {
      if (sets_equal(a, b)) { return 0; }
      if (type(a) == NUMBER) {
         return coerce_to(a, NUMBER) - coerce_to(b, NUMBER);
      }
      else if (|a| != |b|) { return |a| - |b|; }
      else if (nesting_number(a) != nesting_number(b)) {
         return nesting_number(a) - nesting_number(b);
      }
      else {
         /* Find the largest element in one set but not in the other */
         iterator iterA = largest_to_smallest(a);
         iterator iterB = largest_to_smallest(b);
         while (na=iterA.next() && nb=iterB.next() && !set_elem_compare(na, nb));
         return set_elem_compare(na, nb);
      }
   }
   return type(a) - type(b);
}
```

Figure 4.1: {Set⁊y} total ordering of hereditarily finite sets

a structural induction proof on the formal semantics of the grammar, but we omit this in favor of a more high-level explanation.

Finite Loops. As described in the grammar, {Set⁊y} only has two types of loops: `for-to` and `for-in`. Both of these loops *lock* the iterator and upper bound in user-inaccessible variables that get reset every iteration. Additionally, since all sets are finite, all `for-in` loops must end, and all upper bounds are natural numbers. Most notably missing is a form of *unbounded looping* which would allow non-termination.

Strictly-Smaller Recursion. Every time a recursive call is made in {Set⁊y}, a dynamic check is performed on the total ordering of sets, $\prec$, between the initial argument and the new one. If this ordering does not say the new argument is smaller, the program halts with a run-time error.

No Co-Recursion. Some languages allow functions to be used before definition or stubs of functions that will be defined later. {Set⁊y} disallows these features to avoid co-recursive functions that occur in cycles.

It is worthy of note that these precautions to avoid non-termination really do make some functions uncomputable in {Set⁊y}. For example, the Ackermann function:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

is not computable in {Set⅂y} under these rules, as one might expect. {Set⅂y} provides a mechanism to explicitly specify a function of the arguments that is decreasing on the total ordering (rather than just the whole argument). This mechanism is the `@recurse(f(arg))` decorator which replaces the check for $a' \prec a$ with $f(a') \prec f(a)$. For instance, using the `@recurse` decorator, the gcd function *can* be written in {Set⅂y} as can be seen in Figure 4.3.3 on page 44.

```
@recurse(pi1(arg) + pi2(arg))
@requires(x >= 0, "err2")
gcd((x,y)) :=
   if x = 0:
      return y
   if y = 0:
      return x

   if x < y:
      return gcd((x, y−x))
   else:
      return gcd((x−y,y))
```

Figure 4.2: {Set⅂y} code that computes the `gcd` function

4.3.4 Automatic Memoization

By default, {Set⅂y} memoizes every function. Since the only real side-effect is a print, almost all the functions that students write *should* be memoized, and understanding when memoizing is helpful is not a relevant detail for many of the use cases of {Set⅂y}. Ideally, {Set⅂y} would have included a purity checker that automatically memoizes if and only if there were no side-effects, but there was not enough time to add this feature. Instead, {Set⅂y} provides a `@nomemo` decorator that tells {Set⅂y} to not memoize that function.

The implementation of auto-memoization is fairly straight-forward. The compiler creates a global table for each function that stores inputs and their respective outputs, as they are called. Then, a clause to check if the input is already in the table is injected at the beginning of the function.

4.3.5 Libraries and Definition Files

{Set⅂y} provides a way of "using" libraries, other files, etc. like most languages. {Set⅂y} uses "definition" files, in the spirit of C header files, to define (1) which libraries are used by the library (**uses**), (2) what functions are provided by the library (**provides**), and (3) the definitions provided by the library (**defines**). {Set⅂y} expects every program to have a corresponding `.def` file which may include none or all of these sections, depending on the program. The standard {Set⅂y} library provides a variety of primitive definitions, functions, and "types". The compiler automatically resolves all dependencies between libraries by searching on the user's `SETTY_PATH`. If there is a circular dependency, it alerts the user.

4.4 Standard Libraries in {Setʇy}

Although everything in {Setʇy} is a set, users may define their own "types" built on top of sets. This is particularly useful when teaching a variety of mathematical structures. Students can start by building them out of sets and defining functions on top of them.

4.4.1 Standard Type Libraries

By convention, every user-defined "type", `type` should define three functions:

- `is_type` – Although one cannot tell if a set was *intended* to be of a particular type, it is often useful to ensure that it has the right structure.
- `type_print` – `print` statements treat everything as sets, but each type has a "canonical" way of printing out in standard mathematics.
- `type_printx` – This function should always just be the same as `type_print`, except there should be no trailing newline.

The standard library implements booleans, pairs, lists (as cons cells), rational numbers (as canonicalized pairs), and relations (as sets of pairs). Depending on the exercise and/or course, these standard types can be left out or enhanced.

4.4.2 Standard Function Libraries

The rest of the standard library implements the "special" functions (like equals, iff, lt), and other very primitive functions that are necessary in many programs. We found it very useful to implement most of the standard library *with* the students in an early recitation to make sure they understood everything. The standard library consists of three modules:

Logic Module. The logic module defines the `implies(p, q)` and `iff(p, q)` functions.

"Preliminaries" Module. The preliminaries module defines important set and numerical operations on top of the built-ins. In particular, it provides:

- `equals(x, y)` which returns T iff $x = y$.
- `subset(x, y)` which returns T iff $x \subseteq y$.
- `cardinality(x)` which returns $|x|$.
- `lt`, `leq`, gt, and geq which define $<, \leq, >, \geq$ on numbers, respectively.
- `powerset(x)` which returns $\mathcal{P}(x) = \{y \mid y \subseteq x\}$.

Arithmetic Module. The arithmetic module provides basic arithmetic operations. In particular, it defines:

- `min(S)` returns the minimum element in S; it raises a runtime error if $S = \varnothing$.
- `max(S)` returns the maximum element in S; it raises a runtime error if $S = \varnothing$.
- `sum(S)` returns $\sum_{x \in S} x$.
- `even(n)` returns T iff n is even.
- `odd(n)` returns T iff n is odd.
- `power(x, y)` returns x^y.
- `gcd(x, y)` returns $\gcd(x, y)$.

The specific implementations of these functions can be found in Appendix ??.

4.5 Teaching Discrete Mathematics with {Set†y}

The design and implementation decisions for {Set†y} are intended to facilitate the goal of providing students with a *computational environment* for various concepts in basic mathematics. {Set†y} can be used to automate student checking and experimentation in foreign topics. Additionally, in various settings, {Set†y} can even be used to automate *grading* of exercises, and, as a result, students can get more practice. In this section, we outline a variety of exercises over a variety of topics that use {Set†y}.

4.5.1 Logic

When students encounter propositional and first-order logic for the first time, they are often confused by both the syntax and semantics. Implementing logical predicates and examples can help students get a better understanding.

Teaching Semantics of Propositional and First-order Logic

Classical implication and bi-implication can be very confusing for students when they first see them. Students can write definitions for these (and other) logical connectives in {Set†y} and write code to print out the truth tables to check their correctness. For example, we gave students these two predicates:

```
part_a(p) := ¬p ∨ p
part_b((p, q)) := ((¬p ∨ q) ⇒ (p ⇒ q)) ∧ ((p ⇒ q) ⇒ (¬p ∨ q))
```

And an example of printing out the truth table for part_a in {Set†y}:

```
# Print the header
print "p | ¬p ∨ p"
print "--------"

# Print the rows of the truth table
for p ∈ {T, F}:
   print p, "|", part_a(p) # Part A!
   print "------"
```

Then, we ask them to write the code to print out the truth table for `part_b`:

Exercise 1. Print the rows of the truth table. Use the example above to help!

```
print "p | q | ((¬p ∨ q) ⇒ (p ⇒ q)) ∧ ((p ⇒ q) ⇒ (¬p ∨ q))"
print "————————————————————————————————————————"
print "do something here!"
```

This exercise has several benefits over the standard "here is a formula, write a truth table for it" exercise:

- It can be auto-graded, and students can get multiple attempts as a result.
- It turns the tedium of writing out all the cells into writing a program (which they're all somewhat familiar with). This is particularly important, because it can help them see the parallel between what they've already seen (programming) and logic.
- Students write nested for loops which helps them *see* why there are 2^n rows.

Because {Setʇy} coerces types automatically, the code the students initially write prints out 0's and 1's instead of T's and F's!. We told the students that this is a syntactic problem and ask them to fix it.

Exercise 2. In your truth tables above, {Setʇy} printed out 0's and 1's instead of T's and F's. Think of some boolean expression such that p ?? ?? $\Leftrightarrow p$, where each ?? is a placeholder for a single operation or value. Go back and fix the truth tables to use this expression so that {Setʇy} prints with T's and F's.

To avoid overhead with UNIX and infrastructure this early in the course, we gave students a web interface pictured in Figure 4.3 to fill in these problems. As mentioned before, this allowed us to auto-grade the question and give immediate feedback.

Grammar and Syntax of Logic

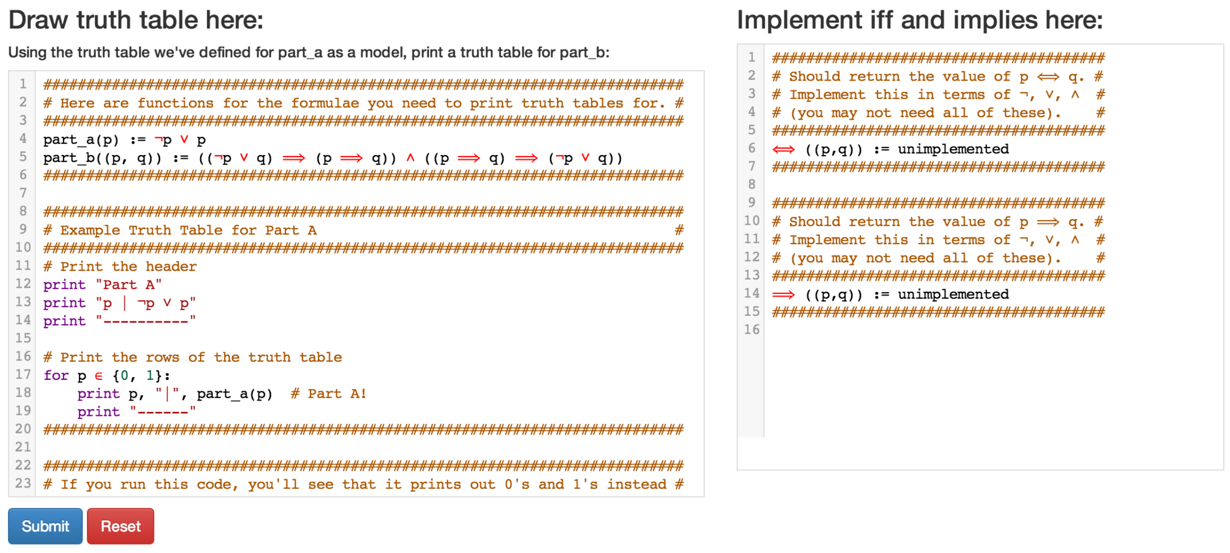
Because it is a programming language, {Setʇy} is very strict about grammar. For example, to express "Every natural number is greater than or equal to 0", we could write

```
∀(n ∈ ℕ). n ≥ 0
```

This has the benefit of forcing students to think of the sentence formally. In particular, the parentheses and the period are part of the syntax of the language. Leaving them out would be like leaving out commas and semi-colons in English—grammatically incorrect. Pointing this out to students can help them think of the "mathematical language" they're learning more formally.

Translating Between English and Logic

Another very common exercise early in discrete math courses is to ask students to translate back and forth between english and symbolic logic. {Setʇy} makes one direction


Draw truth table here:
Using the truth table we've defined for part_a as a model, print a truth table for part_b:

```
###########################################################################
# Here are functions for the formulae you need to print truth tables for. #
###########################################################################
part_a(p) := ¬p ∨ p
part_b((p, q)) := ((¬p ∨ q) ⟹ (p ⟹ q)) ∧ ((p ⟹ q) ⟹ (¬p ∨ q))
###########################################################################

###########################################################################
# Example Truth Table for Part A                                          #
###########################################################################
# Print the header
print "Part A"
print "p | ¬p ∨ p"
print "----------"

# Print the rows of the truth table
for p ∈ {0, 1}:
    print p, "|", part_a(p)  # Part A!
    print "------"
###########################################################################

###########################################################################
# If you run this code, you'll see that it prints out 0's and 1's instead #
```

Submit
Reset
Implement iff and implies here:

```
#######################################
# Should return the value of p ⟺ q. #
# Implement this in terms of ¬, ∨, ∧  #
# (you may not need all of these).    #
#######################################
⟺ ((p,q)) := unimplemented
#######################################

#######################################
# Should return the value of p ⟹ q. #
# Implement this in terms of ¬, ∨, ∧  #
# (you may not need all of these).    #
#######################################
⟹ ((p,q)) := unimplemented
#######################################
```

Figure 4.3: Web interface for {Set⌉y} truth table exercise.

(from informal to formal) very easy, and, again, autogradable. For example, we can give students the sentence

There is an x in A such that for all b in B, $b > x$.

and ask them to fill in `formal` with a formal definition of that sentence:

```
formal(_) := unimplemented
```

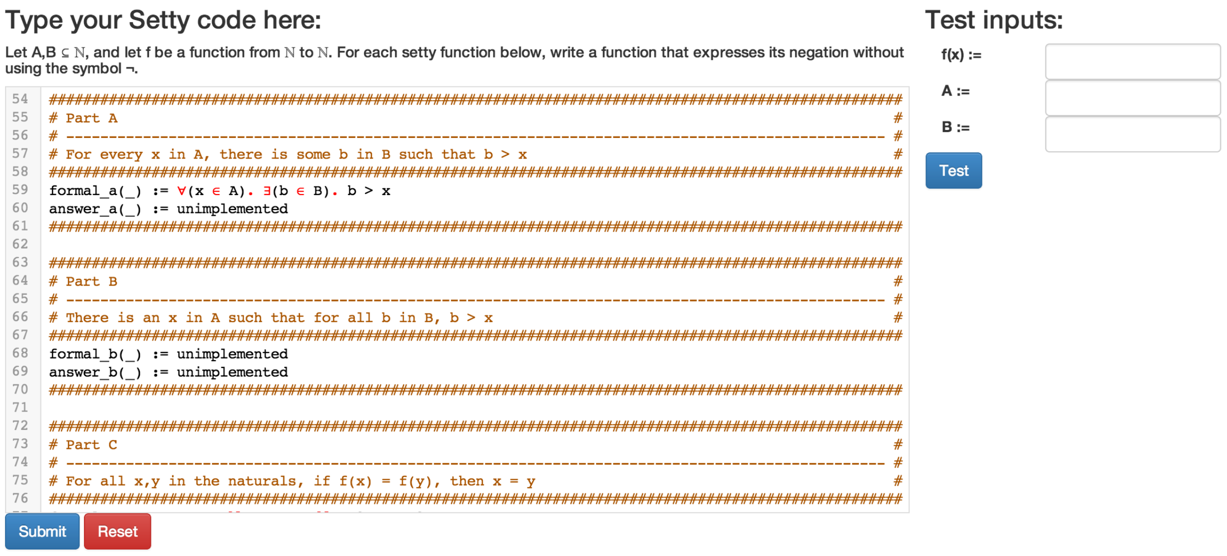
The other direction is harder but can be done indirectly. Generally, the intended learning outcome of asking students to translate formal logical statements into English is to make sure students understand what the symbols mean. By asking students to define predicates used in a sentence to make it true and false, we can test their understanding without asking them to explicitly give the English sentence. For example, we can give students the following exercise:

Exercise 3. Define `f_true` below such that if $f =$ `f_true`, then `formal` $=$ T. Similarly, define a function `f_false` such that if $f =$ `f_false`, then `formal` $=$ F.

```
formal(_) := ∀(x ∈ ℕ). ∀(y ∈ ℕ). f(x) = f(y) ⇒ x = y
f_true(x) := unimplemented
f_false(x) := unimplemented
```

Negating Statements

As an extension of the previous exercise, we can also ask students to negate their statements (pushing in negation symbols as far as possible). This requires a post-processing step to check that the negations are really pushed all the way in, but it can be done with a simple sed script. Again, for this exercise, we gave students a web interface, pictured in Figure 4.4, to limit technical overhead.



Let A,B ⊆ ℕ, and let f be a function from ℕ to ℕ. For each setty function below, write a function that expresses its negation without using the symbol ¬.

```
####################################################################################################
# Part A                                                                                           #
# ------------------------------------------------------------------------------------------------ #
# For every x in A, there is some b in B such that b > x                                           #
####################################################################################################
formal_a(_) := ∀(x ∈ A). ∃(b ∈ B). b > x
answer_a(_) := unimplemented
####################################################################################################

####################################################################################################
# Part B                                                                                           #
# ------------------------------------------------------------------------------------------------ #
# There is an x in A such that for all b in B, b > x                                               #
####################################################################################################
formal_b(_) := unimplemented
answer_b(_) := unimplemented
####################################################################################################

####################################################################################################
# Part C                                                                                           #
# ------------------------------------------------------------------------------------------------ #
# For all x,y in the naturals, if f(x) = f(y), then x = y                                          #
####################################################################################################
```

Figure 4.4: Web interface for {Setʇy} negation exercise.

To allow students to test their negations and sentences, we gave them a box where they could define the function f and the sets A and B that are used throughout the problem. Then, {Setʇy} can tell them if their sentences work on those values or not. Additionally, to get practice writing simple {Setʇy} functions, we can ask students to define the absolute difference $|x - y|$, where x and y are natural numbers. It was used in the later parts of the problem. We can ask a variety of questions using different English phrasings to test this outcome. We give some examples below.
For each part of **Exercise 4**, *consider the informal logical sentence provided. For (a) and (e), we have translated it into symbols for you. For (b), (c), and (d), translate this sentence into logical symbols, and define formal_part as your answer. Negate formal_part, and use logical equivalences to get rid of any negation symbols. Then, define neg_part as your answer.*

Exercise 4(a). "For every x in A, there is some b in B such that $b > x$."

```
formal_a(_) := ∀(x ∈ A). ∃(b ∈ B). b > x
neg_a(_) := unimplemented
```

For this question, we provide an example of a student's attempts and {Setʇy}'s responses to them.

```
neg_a(_) := not forall x. exists b. b > x
```

Notice that the student has used the english versions of the keywords here–{Setʇy} has no problem with this. It will, however, give a syntax error, as quantifiers must specify the domain of individuals; the error message should actually explain *why* this is important in addition to what has happened. Additionally, the grader would complain about the usage of "not", as the problem forbids it.

```
neg_a(_) := ¬ forall x in A. exists b in B. b > x
```

The compiler will still complain about the quantifiers, but this time it is a matter of syntax. {Set⅄y} requires parentheses around the variables and their types for quantifiers, because it reinforces the idea that logical statements are part of a *formal language*. The grader will still complain about the ¬, because it is still logically a "not".

```
neg_a(_) := exists (x in A). forall (b in B). b <= x
```

This time, the compiler and grader will accept the student submission.

Exercise 4(b). "There is an x in A such that for all b in B, $b > x$."

```
formal_b(_) := unimplemented
neg_b(_) := unimplemented
```

Exercise 4(c). "For all x, y in the naturals, if $f(x) = f(y)$, then $x = y$."

```
formal_c(_) := ∀(x ∈ ℕ). ∀(y ∈ ℕ). f(x) = f(y) ⇒ x = y
neg_c(_) := unimplemented
```

Exercise 4(d). "For each $b \in \mathbb{N}$, there is a natural number x where $f(x) = b$."

```
formal_d(_) := unimplemented
neg_d(_) := unimplemented
```

Exercise 4(e). "For all $x, y \in \mathbb{N}$, for all natural numbers e, there is one called d such that if `abs_diff`$((x, y))$ is less than d, then `abs_diff`$((f(x), f(y)))$ is less than e."

```
formal_e(_) := ∀(x ∈ ℕ). ∀(y ∈ ℕ). ∀(e ∈ ℕ). ∃(d ∈ ℕ). abs_diff((x,y)) < d ⇒
    abs_diff((f(x),f(y))) < e
neg_e(_) := unimplemented
```

Exercise 4(f). "For all natural numbers e, there is a $d \in \mathbb{N}$ s.t.for all x, y in the naturals, if `abs_diff`$((x, y))$ is less than d, then `abs_diff`$((f(x), f(y)))$ is less than e."

```
formal_f(_) := unimplemented
neg_f(_) := unimplemented
```

Exchanging Quantifiers

We use the issue of *exchanging quantifiers* as an example of how {Set⅄y} can help students investigate logical formulae.

*For each part of **Exercise 5**, consider the following definitions of ef, and fe:*

```
ef(_) := ∃(x ∈ℕ). ∀(y ∈ℕ). g((x, y))
fe(_) := ∀(y ∈ℕ). ∃(x ∈ℕ). g((x, y))
```

*Many of the parts of this question do not have unique solutions. Your goal is to find *a* solution that works.*

Exercise 5(a). Define $g(x, y)$ such that `ef` and `fe` are both true.

```
g_always_true((x,y)) := unimplemented
print ef(@) ∧ fe(@)
```

Exercise 5(b). Define $g(x, y)$ such that `ef` and `fe` are both false.

```
g_always_false((x, y)) := unimplemented
print ¬ef(@) ∧ ¬fe(@)
```

Exercise 5(c). Define $g(x, y)$ such that exactly one of `ef` and `fe` is true.

```
g_exactly_one((x, y)) := unimplemented
print (ef(@) ∨ fe(@)) ∧ ¬(ef(@) ∧ fe(@))
```

Exercise 5(d). Is it possible to define $g(x, y)$ such that the truth values of `ef` and `fe` from (c) are swapped? Use your answer, define `relate` such that (1) `relate` is always true, and (2) `relate` relates the truth values of `ef` and `fe`.

```
relate(_) := unimplemented
print relate(@) = T
```

Again, the interactive process of *trying to find* a solution and *incrementally editing it* until the compiler says it works allows students much better feedback than just a normal paper exercise.

4.5.2 Sets

Just like with logical formulae, {Setʇy} forces a particular grammar for sets and set comprehensions and allows students to explore various concepts by asking {Setʇy}.

Playing with Sets

Many of the earliest misconceptions that students have with sets center around the empty set and various set operations. {Setʇy} can help students clear this up. We provide an example of a series of probing questions.

In {Setʇy}, you can define variables by writing `x := y`. *For instance, if we wanted a name for the empty set:*

```
empty1 := ∅
empty2 := @
if empty1 = empty2:
   print "the empty sets are equal!"
```

Here is a function to create a set containing a single element:

```
singleton(x) := {x}
```

Exercise 6(a). Write a function `doubleton` that creates a set containing `x` and `y`.

```
doubleton((x,y)) := unimplemented
```

Exercise 6(b). Can you choose `d_x` and `d_y` to make the following if statement true? We pass `@` into `d_x` and `d_y` as "dummy arguments"–they are not actually used.

```
d_x(_) := unimplemented
d_y(_) := unimplemented
if singleton(d_x(@)) = doubleton((d_x(@), d_y(@))):
   print "Yay!"
```

Exercise 6(c). Define a functions `is_empty` that returns T iff the set *is* the empty set.

```
is_empty(S) := unimplemented
```

Exercise 6(d). Define a function `has_empty` that returns T iff the set *contains* the empty set.

```
has_empty(S) := unimplemented
```

Exercise 6(e). Consider the following {Setʇy} code. Guess its output. Then, run it.

```
if is_empty(@):
   print "@ is empty."
if has_empty(@):
   print "@ has empty."
if is_empty({@}):
   print "{@} is empty."
if has_empty({@}):
   print "{@} has empty."
```

Exercise 6(f). Consider the following {Setʇy} code. Guess its output. Then, run it.

```
A := {1,2,3}
B := {2,3,4}
C := {{1,2}, {1,3}}
print A ∪ B
print A \ B
print A ∩ C
print ∩ C
print ∪ C
print ∪ @
print ∩ @
```

Cardinality

We can ask another line of questions for cardinality.

Exercise 7(a). Recall that $[n] = \{1, 2, \ldots, n\}$. Consider the following:

```
card_union((n,m)) := |[n] ∪ [m]|
print card_union((10000000, 1000000))
```

This definition will give the correct answer...if it ever finishes. Unfortunately, we can make n and m arbitrarily large, and the normal definition of cardinality in {Setʇy} has $\mathcal{O}(|S|)$ runtime. Can you define `card_S` differently so that it is not linear any more?

```
card_S((n, m)) := unimplemented
```

(The answer to this one is to take the max of the two inputs; the cardinality questions are particularly interesting, because the original implementations actually work–so, they can still be used as a sanity check.)

Exercise 7(b). Consider another one with the same problem:

```
card_intersection((n, m)) := |[n] ∩ [m]|
print card_intersection((10000000, 1000000))
```

Try defining `card_S2` as well.

```
card_S2((n, m)) := unimplemented
```

Exercise 7(c). We defined subset in class. Give a definition of it in {Sety}.

```
subset((a,b)) := unimplemented
```

Exercise 7(d). Consider the following {Setℸy} code. Guess its output. Then, run it.

```
print 𝒫({1})
print 𝒫(∅)
```

Exercise 7(e). What is $|\mathcal{P}(\varnothing)|$? Try using {Setℸy} to check your answer.

Exercise 7(f). What is $|\mathcal{P}([n])|$? Try defining your answer in {Setℸy}.

Set Comprehensions

When set comprehensions are introduced, students are often confused by both the syntax (where are the variables being bound?) and the semantics (what exactly goes in the set?). {Setℸy} allows students to check their syntax; so, that they know they are turning in a valid expression. It also allows them to print out what their set has in it. This can be useful to check if they've missed a silly case (e.g. their definition of primes includes 0 or 1). As usual, we provide some probing questions.
*For each part of **Exercise 8**, we will ask you to write a set comprehension to define the given set. As an example to start you off, we define evens, the set of even natural numbers, and evens_less_than_ten, the set of evens that are less than 10:*

```
is_even(n) := n − ((n / 2) ∗ 2) = 0
evens(_) := {x ∈ ℕ | is_even(x)}
evens_less_than_ten(_) := {x | x < 10 and x ∈ evens(_)}
```

Exercise 8(a). Now, define odds(S) (the set of members of S that are odd) and odds_less_than_x((S, x)) (the odds in S that are less than 10). Make sure that you do not use x to mean two different things!

```
odds(S) := unimplemented
odds_less_than_x((S, x)) := unimplemented
```

Exercise 8(b). Define the subset of S of non-negative multiples of 151.

```
multiples_of_151(S) := {}
```

A Pythagorean triple is (a, b, c) such that $a^2 + b^2 = c^2$ and a, b, c are positive integers. {Setℸy} does not have syntax for triples; so, you should make them out of pairs instead. To represent (a, b, c) in {Setℸy}, use ((a,b),c).

Exercise 8(c). Define the set of Pythagorean triples using only elements of S.

```
pythagorean_triples(S) := {}
```

Exercise 8(d). Define the collection of all sets of elements of S with at least two members.

```
sets_of_S_with_2(S) := {}
```

Exercise 8(e). Define the set of prime numbers in S.

```
primes(S) := {}
```

Exercise 8(f). Go back and test your answers using various definitions. We've provided up_to_500 as a test case. Can you think of any others?

```
up_to_500(_) := {0} ∪ [500]
```

4.5.3 Foundations of Mathematics

Because {Set‡y} defines every type using sets, it provides the opportunity to teach foundations of mathematics. In particular, we can guide students through the definitions and their motivations:

(a) If we define what "zero" is and what "the successor of x" is, we define all of the natural numbers. Consider the following definition:

```
zero(_) := @
succ(n) := {n}
```

(b) This definition seems "reasonable", and we can do a lot with it, but the most important thing we would like to be able to do is define $x < y$ using only set operations. There is no natural way to do this with this definition. Consider this definition instead:

```
zero(_) := @
succ(n) := n ∪ {n}
```

(c) Can you define nth, which returns succ applied to 0 n times, in terms of $[n]$?

```
nth(n) := unimplemented
```

(d) How can we define $x < y$ with this definition?

```
lt((x, y)) := unimplemented
```

(e) Consider this definition of pairs:

```
(x, y) := {{x},{x,y}}
```

(f) A definition of pairing (putting two elements together) is useless without a definition of "unpairing" (taking the elements back out). Can you think of a definition of pi1 ("extract the first element of the pair") using only set operations?

```
pi1(p) := unimplemented
```

(g) We'll give you a working definition for `pi2` ("extract the second element"). But here, we'll give you a *bad* definition (`pi2bad`) that does not work. Can you find an input that `pi2bad` fails on?

```
pi2bad(p) := (∪ p) \ {pi1(p)}
```

4.5.4 Examples, Patterns, and Direct Proofs

{Set⁊y} (like many other programming languages) can be used to help students do large *computations* [Sut05, Mar10]. Unlike most other languages, {Set⁊y} is ideal for most questions asked in an introductory discrete mathematics course, because of its simple, "math" syntax

Examples

One very useful strategy that most courses try to communicate to students is to try small examples. In the case of an existential quantifier, *all we need for the proof is an example.* Students can use {Set⁊y} to help them work out small examples and test solutions. We provide two examples of how translating into {Set⁊y} makes this easier.

Consider the following homework question we gave in 15-151:

Let $n \in \mathbb{N}$ such that $n > 0$, and $S = [n] \times [n]$. Let T be the set

$$T = \{(x, y) \in \mathbb{Z} \times \mathbb{Z} \mid 0 \leq nx + y - (n + 1) \leq n^2 - 1\}$$

Prove that $S \subseteq T$. Also, determine and prove whether or not $S = T$.

The question of interest here is really $T \subseteq S$. Part of the problem is for the student to figure this out. It could help to translate this into {Set⁊y}. Here is a first attempt:

```
Z := {0} ∪ [500]
cross(S) := {(x, y) | x ∈ S ∧ y ∈ S}
ZxZ := cross(Z)
Ss := cross([n])
Ts := {(x,y) ∈ ZxZ | 0 ≤ nx + y − (n+1) ∧ nx + y − (n+1) ≤ n∗n − 1}

print Ss \ Ts
```

Aha! This throws an error that n is undefined, which, of course makes sense. The reason this is relevant is that a common mistake on this problem is to not realize that these sets are parametrized in n. Writing them out formally forces the student to realize this. In a "fixed" version, of course, we are still slightly incorrect, because {Set⁊y} only supports positive numbers. We can add a loop at the end to figure out what is going on:

```
Z := {0} ∪ [500]
ZxZ := {(x,y) | x ∈ Z ∧ y ∈ Z}
```

```
S(n) := [n] × [n]
T(n) := {(x,y) ∈ ZxZ | 0 ≤ nx + y − (n+1) ∧ nx + y − (n+1) ≤ n∗n − 1}

for n ∈ Z:
 print S(n) \ T(n)
```

And another homework question:

Let X and x be sets. Prove or disprove each direction of the equality.

$$(X \setminus \{x\}) \cup \{x\} = X$$

The idea is similar here. Let us take a wide variety of sets and check if the two sides are equal:

```
PX := 𝒫({1,2,3})
for X in PX:
   for x in {1,2,3}:
     if (X \ {x}) ∪ {x} ≠ X:
        print X, "and", x, "are not equal"
```

Patterns

Another useful strategy that we try to communicate to students is *pattern recognition*. {Setℸy} can help with this by doing the small examples for students. All they have to do is write code.

Consider another 15-151 homework question:

Write down the positive integers, delete every second, and form the partial sums of those remaining:

1	$\cancel{2}$	3	$\cancel{4}$	5	$\cancel{6}$	7	$\cancel{8}$	9	$\cancel{10}$	11
1		4		9		16		25		36

Claim: The nth item in the sequence is n^2.

To solve this problem, first, we translate the algorithm into code:

```
LIMIT := {0} ∪ [1000]
sum := 0
n := 1
for i in {2x + 1 | x in LIMIT}:
   sum := sum + i
   if sum ≠ n∗n:
      print "The partial sum for", n, "is not", n∗n, ". It is", sum
   n++
print "The partial sums are correct up to", max(LIMIT)
```

If we find a counter-example, we are done! If we do not find a counter-example, we should start looking for a proof of correctness.

Here is one more example:

Write down the odd numbers starting with 43. Circle 43, delete one number, circle 47, delete two numbers, circle 53, delete three numbers, circle 61, and so on:

$\textcircled{43}$	45	$\textcircled{47}$	49	51	$\textcircled{53}$	55	57	59	$\textcircled{61}$	63	65	67	69	$\textcircled{71}$
73	75	77	79	81	$\textcircled{83}$	85	87	89	91	93	95	$\textcircled{97}$	99	101
103	105	107	109	111	113	115	117	119	121	123	125	127	129	

Claim: The circled numbers in this sequence are all prime.

Again, we can translate this into {Setʇy} code:

```
is_prime(n) := <implemented in a previous exercise>
LIMIT := {0} ∪ [1000]
to_skip := 0
skipped := 0
for i in {2x + 1 | x in LIMIT ∧ x > 42}:
   if to_skip > skipped:
      skipped++
   else:
      to_skip++
      skipped := 0
      if not is_prime(i):
         printx "Found a counter-example:", i, ". This was the"
         printx to_skip
         print "ith number circled."
```

Direct Proofs

These direct proofs are just more of the same.

(1) Let SidesOf(x, y) be "the number of sides of x is y".

$$\forall(x \in \{0,1\}).\exists(y \in \{\triangle, \square\}).\text{SidesOf}(y, x+4) \Longleftrightarrow (p \vee \neg p)$$

For this one, we have to do a bit of modelling. There is no $\triangle$ in {Setʇy}. So, for a shape, let us just use $[n]$ to mean "a shape with n sides". Then, `SidesOf(shape, sides)` is just `|shape| = sides`:

```
triangle := [3]
square := [4]
SidesOf((shape, sides)) := |shape| = sides
```

Now, to solve the problem:

```
for p in {T, F}:
   print ∃(x ∈ {0, 1}).∃(y ∈ {triangle, square}). SidesOf((y, x+4)) ⇔ (p∨¬p)
```

(2)

$$\exists(x \in \{\varnothing, \{0\}\}).[\forall(y \in \varnothing).|y| = 0 \wedge |y| \neq 0] \Longleftrightarrow [|\{0\}| = |\mathcal{P}(x)|]$$

This one is directly translatable:

```
print ∃(x ∈ {@, {0}}). [∀(y ∈ @). |y| = 0 ∧ |y| ≠ 0] ⇔ [|{0}| = |𝒫(x)|]
```

4.5.5 Recurrences and Combinatorics

The Fibonacci Numbers pop up a lot in seemingly strange situations. They are defined in terms of a *recurrence*. We will study recurrences later in the course, but for now, all you have to know is that the later terms are defined using the previous ones. The Fibonacci numbers are defined as follows:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

So, for instance, to generate f_3, we see that $f_3 = f_2 + f_1 = (f_1 + f_0) + f_1 = (1 + 0) + 1 = 2$.
Prove that there is no largest even Fibonacci number.

Because {Set𝔮y} memoizes functions, we can easily calculate things like Fibonacci numbers. Furthermore, the "translation" into {Set𝔮y} is basically the same as the math. That is:

```
f(0) := 0
f(1) := 1
f(n) := f(n−1) + f(n−2)
```

The benefit is clear. {Set𝔮y} allows us to write the mathematical definition, but it makes it execute quickly. We can explore recurrences and various properties via this. You can imagine another place where {Set𝔮y} shines is in enumeration problems. A common question is to find partitions of a set. This can be done idiomatically in {Set𝔮y} as follows:

```
partitions(n) := (@, @)
partitions(n) := {(X ∪ {n}, Y) | (X, Y) ∈ partitions(n−1)} ∪
                 {(X, Y ∪ {n}) | (X, Y) ∈ partitions(n−1)}
```

4.5.6 Pilot Results

In the second run of 15-151, we used {Set𝔮y} for all the programming assignments, and we used many of the exercises outlined above. Unfortunately, the overhead of UNIX and some initial bugs proved to outweigh potential benefits. Additionally, the slower feedback cycle of a compiled language proved to be a non-trivial deterrent; this is especially true, because it forced them to drop into the uncomfortable UNIX environment even more frequently. Furthermore, we failed to stress the idea that {Set𝔮y} should be used as a computational environment as much as we should have.

One of the original goals of {Set𝔮y} was to make it more clear how a language could be built out of mathematical primitives. The features of the language designed around this goal (everything is a set, `print {0,1}` $\to$ 2, all functions take one argument, etc.) ended up making these difficult concepts the *only* thing students associated with the language. In other words, the goal to teach foundations of mathematics overshadowed *all the other goals* to the point where they were not even seen as goals.

4.6 Future Work

We believe the issues we had with the first run are mostly solvable now that we know about them. We plan to write a new version of {Setʇy} over the summer with different design considerations. The new version of the language will be interpreted, not compiled. We plan to use `emscripten` or a tool like it to compile the new interpreter to javascript. This would allow students to access the language from a friendly browser environment based on repl.it.

Perhaps most importantly, we will be dropping reliance on set theory. Sets will still be a primitive type, but so will naturals, products, lists, and functions.

4.6.1 Backed by Z3

The new version of {Setʇy} will be backed by Z3, a theorem prover from Microsoft Research. This will allow us to have two very important new features. The first new feature is the support of arbitrary, potententially infinite, sets. We can store known constraints on sets and when it becomes necessary to know what is in the set, we can ask Z3. The second new feature is support for a form of logic programming in which the programmer lists constraints on a predicate, and a satisfying predicate is found. We provide two examples of where this second feature might be useful.

Sudoku

Consider the following code in which we suppose the existence of a `puzzle_solution` for the Sudoku puzzle `puzzle`:

We can then use Z3 to actually solve the puzzle (if there is a solution). Approximate code (using the Python bindings for Z3) can be seen in Figure 4.7, below.

Halting Problem

The second application is as a demonstration of computability. Using the same `suppose` and `insist` keywords from before, we can demonstrate that the halting problem is undecidable. The interesting part is that the new version of {Setʇy} should be able to *determine this just from the code*!

```
puzzle = [[1, 2, 0, 0, ...], ...]
VALS := {0} ∪ [8]
suppose exists (puzzle_solution : VALS ∗ VALS ∗ [9] −> bool) such that:
   insist ∀(i,j ∈ VALS). puzzle[i][j] != 0 ⇒ puzzle_solution(i, j, puzzle[i][j])
   insist ∀(i,j ∈ VALS). ∃(n ∈ [9]). puzzle_solution(i, j, n)
   insist ∀(n ∈ [9]). ∀(i, j ∈ VALS). ∀(n2 ∈ [9] \ {n}). puzzle_solution(i, j, n)
       ⇒ ¬puzzle_solution(i, j, n2)
   insist ∀(n ∈ [9]). ∀(i ∈ VALS). ∃(j ∈ VALS). puzzle_solution(i, j, n)
   insist ∀(n ∈ [9]). ∀(j ∈ VALS). ∃(i ∈ VALS). puzzle_solution(i, j, n)

   BOXES := {{(i + 3∗bi, j + 3∗bj) | i,j ∈ {0, 1, 2}} | bi, bj ∈ {0, 1, 2}}

   insist ∀(n ∈ [9]). ∃((i, j) ∈ BOXES). puzzle_solution(i, j, n)
as puzzle_solution
```

Figure 4.5: Future {Setʇy} code to solve Sudoku puzzles

```
suppose exists (halts string −> bool) such that:
   CONFUSE() :=
      me := get_my_source()
      if halts(me):
         while True: pass
         insist not halts(me)
      else:
         return True
         insist halts(me)
as halts
```

Figure 4.6: Future {Setʇy} code that exibits the undecidability of the halting problem

```
from z3 import *
solution = Function('solution', IntSort(), IntSort(), IntSort(), BoolSort())

constraints = []

for i in range(len(puzzle)):
   for j in range(len(puzzle)):
      if puzzle[i][j] != 0:
         # The result must match what we already know.
         constraints.append(solution(i, j, puzzle[i][j]))

      # Every cell must have some value
      constraints.append(Or([solution(i, j, n) for n in range(1, 10)]))

      #...but only one value
      constraints.append(And([
        Implies(
         solution(i, j, n),
         And([Not(solution(i, j, np)) for np in range(1, 10) if np != n])
        )
      for n in range(1, 10)]))


# Every row must have some value
constraints.append(And([
 And([
  Or([solution(i, j, n) for j in range(9)])
 for i in range(9)])
for n in range(1, 10)]))

# Every column must have some value
constraints.append(And([
 And([
  Or([solution(i, j, n) for i in range(9)])
 for j in range(9)])
for n in range(1, 10)]))

# Every box must have some value
constraints.append(And([
   And([
    Or([solution(i + boxi*3, j + boxj*3, n) for i in range(3) for j in range(3)])
   for boxi in range(0, 2) for boxj in range(0, 2)])
for n in range(1, 10)]))

s = Solver()
s.add(constraints)

if s.check() == sat:
   m = s.model()
   r = [[ [n for n in range(1, 10) if is_true(m.evaluate(solution(i, j, n)))][0]
       for j in range(9)] for i in range(9)]
   print_matrix(r)
else:
   print "failed to solve"
```

Figure 4.7: Translation of the code in Figure 4.5 to Z3 bindings in Python

Chapter 5

Symbolic Algorithm Complexity Analysis

In this chapter, we describe an assignment where students write several static analyzers that, when given the source code of a program (in a subset of the Setty language), generate estimated asymptotic bounds of various features (output, steps of computation) of the program. Instead of running the program on various inputs, students decompose the program into *structures* (loops, if statements, function calls, etc.) and feed pieces of the result to a symbolic algebra library.

The assignment is intended as a teaching exercise in combining mathematics (iteration, induction, asymptotics, computability) and programming. As a result, we take certain liberties and restrictions for granted, but the algorithm gives reasonable results on many inputs.

In our case study, we made the mistake of asking students to use a mostly object-oriented style (using Python) which ultimately did more harm than good, because most of the functions that the students wrote were recursive, and there was a lot of "magic code" where student-written functions were called by staff-written functions. As a result, we present a slightly tweaked version of the assignment we gave in 15-151, in the hope that it solves several small problems we encountered.

It is crucial to distinguish between the programming languages *pedagogical presentation* and the *formalism* throughout this chapter, because the ultimate consumers are intended to be students with a relatively weak math background. For instance, a formal definition of the semantics of Sub{Setty} is irrelevant to the learning outcomes, but crucial to the design of the language and how easy the algorithms will ultimately be to implement. Furthermore, depending on the students, presenting this as inference rules *could* be reasonable, but it likely would not be. We describe the parts that the students implement using a more *pedagogical presentation* and relegate much of the formalism to Appendix A.

5.1 Overview

We begin with an informal discussion of the four complexity analyses the students are asked to write. Because the target language can make these analyses vary drastically, we describe the specific one we recommend, a “subset of {Set⁊y}”, called Sub{Set⁊y}, that focuses on language features we care about in Section 5.3.

- `time-upper`$(F) : \mathbb{N} \to \mathbb{N}$, a symbolic upper bound on the runtime of F.
- `output-upper`$(F) : \mathbb{N} \to \mathbb{N}$, a symbolic upper bound on the return value of F.
- `time-lower`$(F) : \mathbb{N} \to \mathbb{N}$, a symbolic lower bound on the runtime of F.
- `output-lower`$(F) : \mathbb{N} \to \mathbb{N}$, a symbolic lower bound on the return value of F.

For example, consider the following Sub{Set⁊y} function, F:

```
F(n) :=
   out := 1
   for i → n:
      out := out * n
   return out
```

We can see that the time complexity is $\mathcal{O}(n)$, because the loop always runs n iterations, and the output complexity is $\mathcal{O}(n^n)$, because F always *returns* out which started as 1 and was multiplied by n, n times. To compute these bounds for an arbitrary Sub{Set⁊y} program, we run it through the pipeline shown in Figure 5.3.

We don’t discuss parsing the language, because it is irrelevant to the assignment; though a parser and AST should obviously be provided to students.

5.2 Pedagogical Outcomes

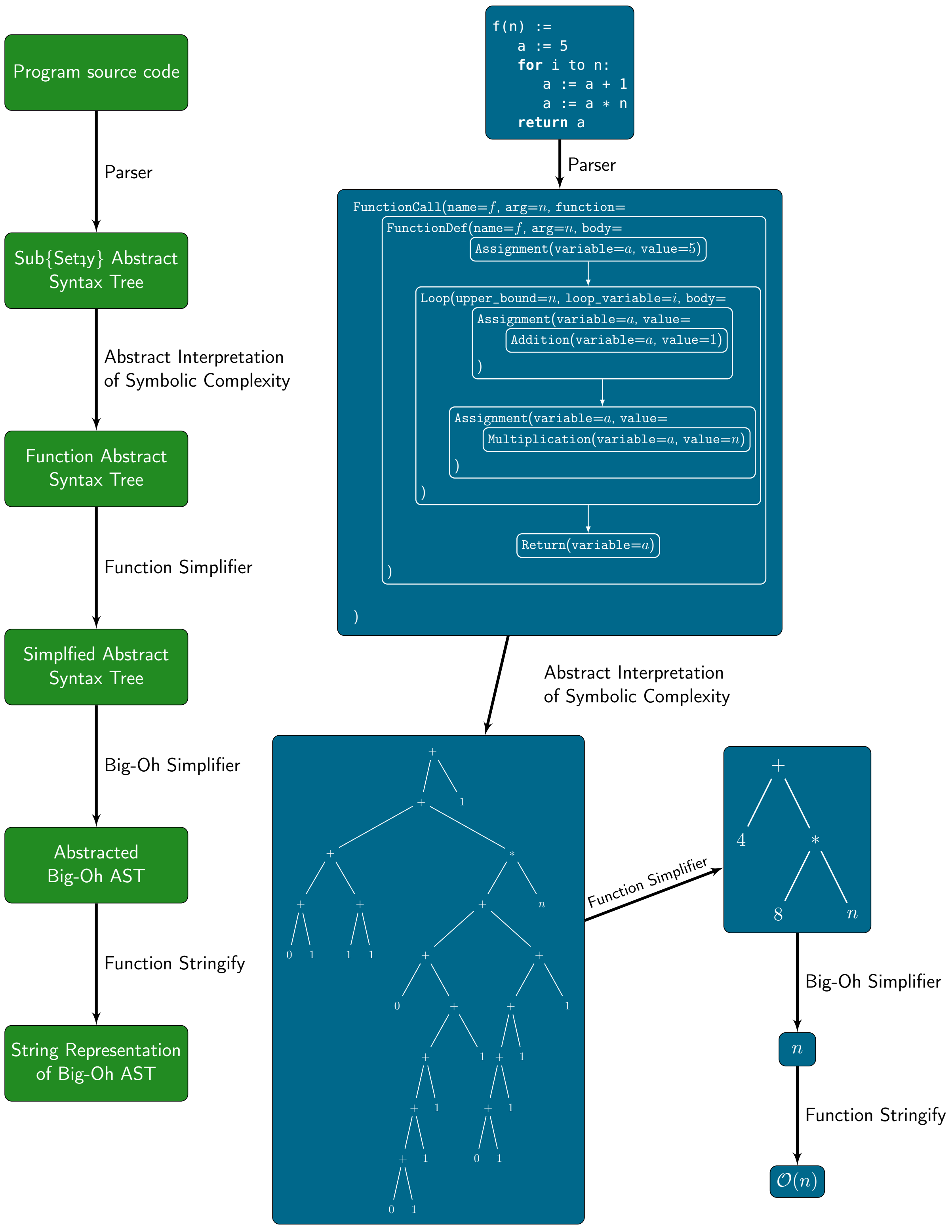
Before describing the specifics of the assignment, we give a broad overview of the intended outcomes, other possible assignments and approaches in the same space, and what students might implement for the assignment.

This assignment is intended to help with the instruction of several topics. In particular, we are concerned with Big-Oh Analyses, Recurrences and Closed Forms, Confusion Between Analyses.

Learning Big-Oh

Nearly every Computer Science curriculum discusses “big-oh analysis” at some point. For better or worse, the topics covered under this umbrella vary drastically. Topics that might or might not be covered include:

- An informal description of how the analysis “ignores details”
- A description of “common Big-Oh classes”: linear, logarithmic, quadratic, exponential, etc.
- A “procedure” for determining what class a function is in


```
f(n) :=
    a := 5
    for i to n:
        a := a + 1
        a := a * n
    return a
```

```
FunctionCall(name=f, arg=n, function=
    FunctionDef(name=f, arg=n, body=
        Assignment(variable=a, value=5)
        Loop(upper_bound=n, loop_variable=i, body=
            Assignment(variable=a, value=
                Addition(variable=a, value=1)
            )
            Assignment(variable=a, value=
                Multiplication(variable=a, value=n)
            )
        )
        Return(variable=a)
    )
)
```

Program source code
Parser
Sub{Setɿy} Abstract Syntax Tree
Abstract Interpretation of Symbolic Complexity
Function Abstract Syntax Tree
Function Simplifier
Simplfied Abstract Syntax Tree
Big-Oh Simplifier
Abstracted Big-Oh AST
Function Stringify
String Representation of Big-Oh AST
Parser
Abstract Interpretation of Symbolic Complexity
Function Simplifier
+
4
*
8
n
Big-Oh Simplifier
n
Function Stringify
$\mathcal{O}(n)$

Figure 5.1: The complexity analysis pipeline.

- A formal definition of $\mathcal{O}(-)$
- Proofs (and disproofs) that functions are in various classes
- Different analyses (output, time, space, etc.)
- The limit definition of $\mathcal{O}(-)$
- $\Omega(-), \Theta(-), o(-), \ldots$

None of these is good or bad in isolation, but there is often a non-trivial bridge between the first few and the rest. As always, we are interested in helping students who are comfortable with programming and helping them get more comfortable with math.

Transfer from Simple Recursion to Recurrences

There is often an additional boundary between writing simple recursive functions and understanding how recurrences "work". By asking students to programmatically convert procedures to recurrences (in particular, *different ones* like time and output) allows them to experiment and try to understand what the recurrence actually means.

Additionally, we prefer that students have a deep understanding of *why* they can analyze functions the way they do–and, we claim asking them to implement what amounts to their procedure bridges exactly this gap. Furthermore, they see real-world uses for closed forms of summations and recurrences which are a lot more interesting than "here is a function, write a recurrence for it".

Confusion Between Analyses

Because most students only see "runtime" analysis first (and reasonably so!), when they are eventually told about other analyses like output, space, etc., they often see them as unimportant, or worse *confuse them with runtime analysis*! By asking students to analyze programs in multiple ways, we reinforce the differences. In fact, in the loop time analysis, they actually *use their output analysis*!

5.2.1 Standard Approaches

It should be noted that we *are not* suggesting that the standard approaches we are about to discuss should not be used. In fact, we believe they should *both* be discussed *before* this assignment. The main point of this section is to demonstrate some common misunderstandings that occur through these methods and how our approach (following them up) can help students remove these misconceptions.

Walking Through Code

One of the standard approaches is to break down the code into pieces and evaluate explaining that loops are "$\mathcal{O}(n)$ times the inside", or another similar wording. (This is clearly a vast over-simplification of this approach, but it is often characterized by this sort of wording.)

This approach is particularly good for intuition, because students are forced to break down the structure of a program, but because they do not think of the syntax of a program as a recursive structure, this can make it feel like guess-work. There is also a strong disconnect between this approach and the formal definition of Big-Oh. Furthermore, some students treat each program as a distinct problem and are not directly able to create a generic approach from the individual examples. Finally, these examples tend to be very small programs which can lead to the impression that the approach is not useful or uncommon in the "real world".

Timing Graphs and Regressions

Another common approach is to time a program on multiple inputs, and create a best-fit curve that matches the points. First, we point out that because all computers are different and the timings will never be ideal, this approach is not particularly reliable and there are very frequently anomalous points in the curve, but we do not view this as a particularly important concern. Like the previous approach, this one is not particularly helpful for helping students work with the formal definitions–and unlike the previous approach, the times do not help develop students' intuition.

Another concern is that because some programs (particularly those with exponential runtimes) take a prohibitively long amount of time to run, students might not run them on enough inputs. We ran a timing analysis on a (non-memoized, exponential) implementation of the Fibonacci function, and these were our results:

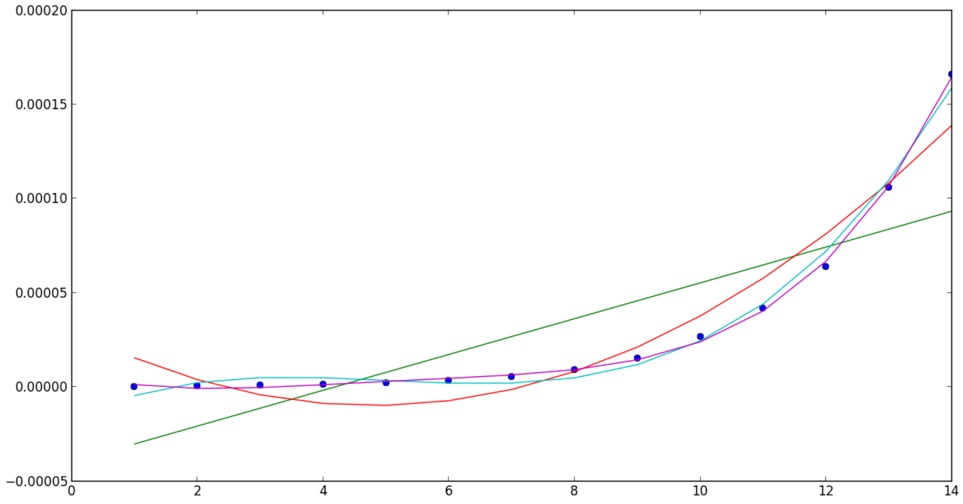


Figure 5.2: Runtimes for inputs to Fibonacci of increasing sizes

The trend-lines are polynomials with degree 1, 2, 3, 4, and 5 that are best fits; a naive student might be convinced by this graph that Fibonacci is $\mathcal{O}(n^3)$.

A different function (Timsort) exposes two more possible concerns. First, the linear and quadratic trend-lines look basically identical which could be very confusing. Second, some functions behave very differently on various ranges of inputs (Timsort is technically $\mathcal{O}(n^2)$ on small inputs and expected $\mathcal{O}(n\lg(n))$ on larger inputs.) So, the range the student chooses might be very misleading as well.

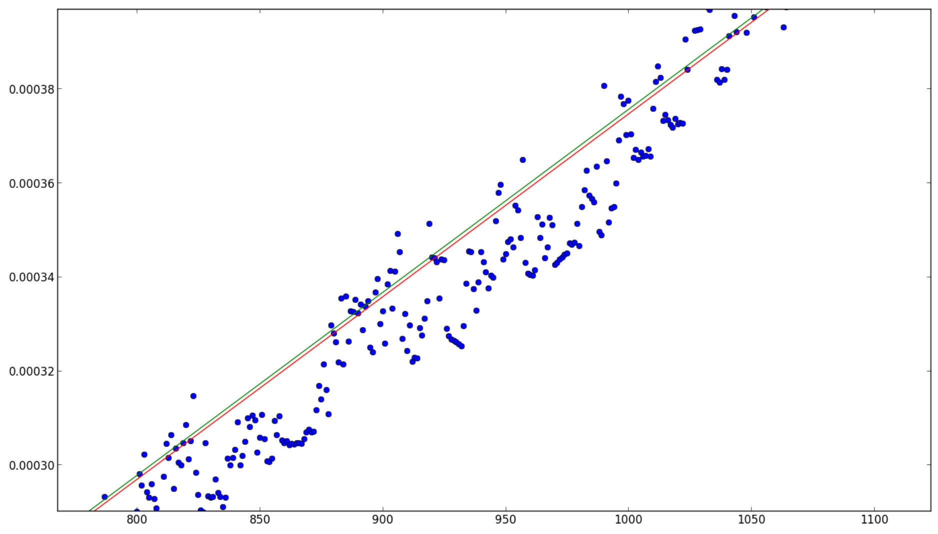


Figure 5.3: Runtimes for inputs to Timsort of increasing sizes (zoomed)

5.3 Sub{Set⅂y}

Because this assignment will likely be the first time students will be doing an analysis of *programs*, there are some important distinctions to make before even describing the assignment. The most important one is the distinction between the *target language* of the analysis and the programming language they will be *writing the analysis in.* The target language is incredibly important, because the pieces of the analysis vary drastically based on it; the only real requirements of the language the analysis is written in are: (1) a symbolic algebra library is available, and (2) students are already familiar with it. If students already know a functional language, we recommend they use that; otherwise, we recommend Python or Java.

The requirements of the target language are significantly more stringent. It should look like psuedocode, and the semantics should seem familiar to students. The target language must not have an unbounded loop construct or co-recursion, because this construct is significantly more difficult to deal with than others. It should have as few types and built-ins as possible. It is no coincidence that {Set⅂y}'s design satisfies many of these constraints; the target language we suggest is effectively a *subset* of {Set⅂y}, which we call Sub{Set⅂y}.

5.3.1 Syntax

Here we give a BNF of the Sub{Setʇy} syntax. We give formal names to each construct, so that the analysis will be more clear later. On the left is the formal name, in the middle is the syntax, and on the right is the intended meaning.

Prog	P	::=	`define`$(X; x; m; P)$	$X(x)$ `:=` m P	defines a procedure
			e	e	expression
Cmd	m	::=	`ret`(e)	`return` e	return
			`seq`$(m_1; m_2)$	m_1 m_2	sequencing, right-associative
			ε		terminal
			`assn`$(x; e)$	x `:=` e	sets an assignable's value
			`if`$(b; m_1; m_2)$	`if` b: m_1 `else`: m_2	conditional
			`for`$(x; 0; e; m)$	`for` x `to` e: m	bounded, indexed iteration
Exp	e	::=	X	X	function variable
			true	T	truth
			false	F	falsity
			`eq`$(e_1; e_2)$	$e_1 = e_2$	equality
			`lt`$(e_1; e_2)$	$e_1 < e_2$	less than
			`and`$(b_1; b_2)$	b_1 `and` b_2	conjunction
			`or`$(b_1; b_2)$	b_1 `or` b_2	disjunction
			`not`(b)	`not` b	negation
			x	x	nat variable
			`nat`$[n]$	n	numeral
			`plus`$(e_1; e_2)$	$e_1 + e_2$	addition
			`times`$(e_1; e_2)$	$e_1 * e_2$	multiplication
			`divide`$(e_1; e_2)$	e_1/e_2	integer division
			`minus`$(e_1; e_2)$	$e_1 - e_2$	subtraction capped at 0
			`power`$(e_1; e_2)$	$e_1\hat{}e_2$	exponentiation
			`call`$(X; e)$	$X(e)$	application

5.3.2 Semantics

The intended semantics (which we describe in full detail later) should be clear from context. The assignment should provide students with an interpreter for the language implementing these semantics; so, students can figure out the semantics from examples and running code. Additionally, since they are theoretically familiar with {Setʇy} already, they should already have an intuitive understanding of these semantics from when they used it previously.

5.4 Our Approach

It is best to view this project as multiple parts which should be completed in order. In this section, we give a brief overview of each piece and what we describe to the students followed by an in depth discussion of background, formalism, and algorithms.

5.4.1 The Standard Interpreter

Students are given code that implements the standard Sub{Setʇy} semantics; it already records the total number of steps to evaluate programs. If they have not already seen the language, we describe it and give a few example programs. We walk students through running a few functions and inputs and discuss that we want to output complexities instead of direct answers. We also describe the format of the programs we will be analyzing (e.g. function definitions followed by a function call).

Sub{Setʇy} Concrete Semantics

We provide the formal semantics of Sub{Setʇy}, which would be implemented in code for students, in Appendix A.

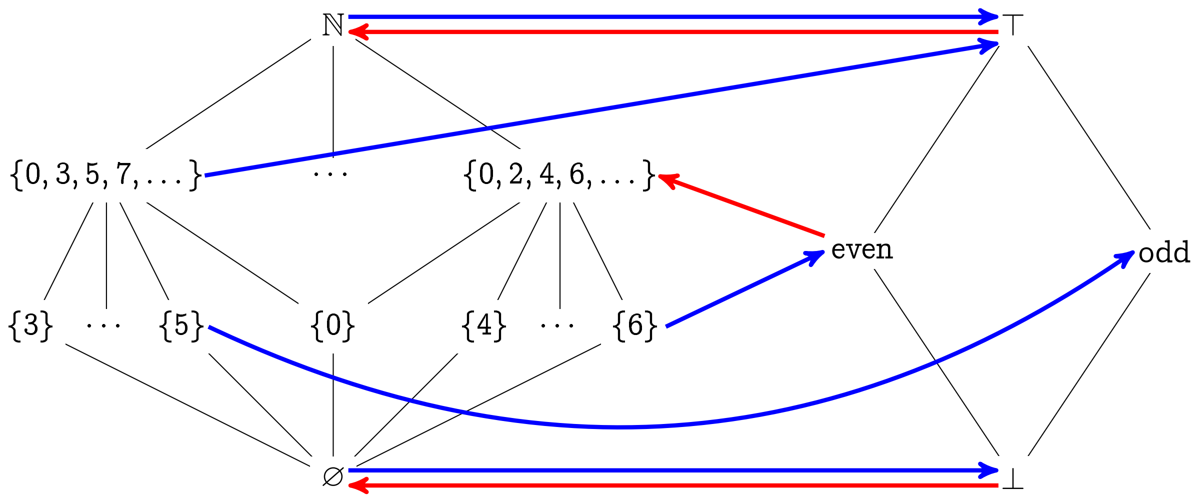
5.4.2 Abstract Interpretation and Symbolic Algebra Library

We introduce students to the idea of a symbolic algebra library and explain all the simple functions (`add`, `mul`, `div`, ...), as well as `canonical`, `big-oh`, etc. functions. Then, we ask students to replace all the expression steps with symbolic algebra functions. We make the point that the program they are writing is not computing a precise value, but trying to approximate it with a symbolic function in terms of the input, instead.

Abstract Interpretation

Formally, the concept that makes these analyses work is *abstract interpretation*. We do not recommend that any of the formalism (or, really even the name) be explained to students, but the general idea is important for them to understand so they can do the assignment. We give a quick overview of abstract interpretation for completeness.

Abstract Interpretation [Cou05] is a type of static analysis that can give *approximate* answers to questions that are often undecidable. One considers a “concrete semantics” which is an exact answer to the analysis being considered, as well as an “abstract semantics” which approximately answers the analysis by grouping together various answers that the concrete semantics might give. For instance, below we have a concrete semantics made up of subsets of the natural numbers which is approximated by the lattice $\{\top, \text{even}, \text{odd}, \bot\}$. The *blue* arrows represent an “abstraction function” which takes concrete values and represents them with an abstract value; the *red* arrows represent a “concretezation function” which represents abstract values with the canonical concrete value that they represent.



Symbolic Algebra Requirements

In the standard concrete semantics of Sub{Setɿy}, programs evaluate to *numbers*, but the analyses we are interested in work over a lattice of *symbolic functions*. We are not particularly interested in the actual symbolic library used, but we specify the syntax we expect in the next section. Importantly, we assume our symbolic library has the following functions:

- $\texttt{canonical}(f)$ – returns the canonical representation of of any symbolic function f; for example, we would expect

$$\texttt{canonical}(\texttt{add}(\texttt{add}(\texttt{const}[1], \texttt{const}[2]), \texttt{var}[n]))) = \texttt{add}(\texttt{var}[n]), \texttt{const}[3])$$

- $\texttt{big-oh}(f)$ – returns the canonical function for which $\mathcal{O}(f) = \texttt{big-oh}(f)$
- $\texttt{big-omega}(f)$ – returns the canonical function for which $\Omega(f) = \texttt{big-omega}(f)$
- $\texttt{big-theta}(f)$ – returns the canonical function for which $\Theta(f) = \texttt{big-theta}(f)$
- $\texttt{equals}(f, g)$ – returns true if the canonical representations of f and g are the same, false if they can easily be determined to be different, and ? otherwise

We also assume expressions in our library use the following syntax:

Exp e ::=	`const`$[c]$	c	constant function
	`infinity`	`infty`	maps all inputs to ∞
	`var`$[n]$	n	variable function
	`add`$(e_1; e_2)$	$e_1 + e_2$	addition
	`mul`$(e_1; e_2)$	$e_1 * e_2$	multiplication
	`div`$(e_1; e_2)$	e_1 / e_2	division
	`sub`$(e_1; e_2)$	$e_1 - e_2$	subtraction
	`pow`$(e_1; e_2)$	e_1`^`e_2	exponentiation
	`max`$(e_1; e_2)$	`max`(e_1, e_2)	max
	`min`$(e_1; e_2)$	`min`(e_1, e_2)	min
	`log`$(e_1; e_2)$	`log_`$e_1(e_2)$	log
	`replace`$(e_1; $`var`$[n]; e_2)$	$[e_2/n]e_1$	replace variable
	`sum`$(e_1; $`var`$[n]; e_3)$	`sum` e_1 `from` n`=0 to` e_3	summation
	`iter`$(e_1; e_2; $`var`$[n_1]; e_3; $`var`$[n_2])$	`(init` n_1 `as` e_3 `for (`n_2 `to` e_2`){` `run` e_1 `}` n_1	indexed iteration
	`recurrence`$(e_1; e_2; e_3; $`var`$[n])$	$T(n) = e_1 T(e_2(n)) + e_3(n)$	recurrence

5.4.3 Overview of Complexity Analyses

Because this assignment is intended to mesh the mathematics students have learned with the programming, we deliberately give a math-dense presentation of the analyses to the students.

For simplicity, we only analyze Sub{Set⌉y} programs of the form

$$P = \texttt{define}(X_1; x_1; m_1; \texttt{define}(X_2; x_2; m_2; \texttt{define}(\cdots; \texttt{call}(X, x))))$$

When analyzing such a program P, we say we are analyzing the Sub{Set⌉y}function X(x), where x is a symbolic variable.

For each of output and time complexity, we provide students with the *concrete* semantics (e.g., the semantics we're trying to approximate), and the *abstract* semantics (e.g., the semantics they will implement). Their job is to change the interpreter to actually implement these analyses.

5.4.4 Output Complexity

We discuss in more depth what we mean by "output complexity". We tell students that we are going to replace the standard number input to the function with a *variable that could represent any number. Then, we discuss an example that* should *output a logarithmic answer, but instead will (eventually) output $\mathcal{O}(n)$. We either explain or ask why that is still* a correct answer. *We provide each of the* `if`*,* `for`*, and* `recurrence` *algorithms. Note that any of the commands can be omitted from the language if the assignment goals are supposed to be different.* In this analysis, we are interested in approximating the output of a Sub{Set⁊y} function $F(x)$.

Concrete Semantics

Given a Sub{Set⁊y} function $\texttt{F}(x)$, consider the mathematical function $f = \texttt{F}$. Then, consider the least upper bound in the symbolic functions lattice, U, such that $\texttt{big-oh}(U) > \texttt{big-oh}(X)$ for all X in the lattice where $f(x) \in \mathcal{O}(U)$. That is, U is the tightest Big-Oh class in the lattice that is representative of the output of F.

The symbolic function U is the concrete semantics of our analysis. This, however, is incredibly difficult (or impossible) to compute.

Similarly, we are concerned with a largest lower bound, L for the Big-Omega analysis. These actually must be considered together for the abstract analysis, because, for instance,

$$\texttt{big-oh}(f/g) = \texttt{big-oh}(f)/\texttt{big-omega}(g)$$

So, we will assume they are computed simultaneously.

Abstract Semantics

Our abstract semantics over-approximates U by modifying the original concrete semantics in various ways. We describe the changes to the concrete semantics of Sub{Set⁊y} that are necessary to complete this analysis.

First, we consider symbolic functions of the variable x to be values instead of just functions of constants. All of the values originally in the analysis should be turned into their analogs in the symbolic function syntax. For instance, the rule that steps $\texttt{plus}(e_1; e_2)$ to n, where $n = e_1 + e_2$ should instead step it to $\texttt{add}(e_1; e_2)$, since in this case e_1 and e_2 should already be "values" which means they are symbolic functions of x. Note that division and subtraction must feed through the second value from the lower bound analysis, because of the reason discussed in the concrete semantics section.

Second, we must approximate each of the individual commands: if statements, for loops, and recursion. Instead of formally describing the semantics, we provide a high-level description of the algorithms, because it is more faithful to the description we provided to the students.

If Statements. Since we are trying to approximate *all of the inputs* at the same time, it does not make sense to ask "which branch does the program take?". We told students that they could attempt to determine if one branch is *always* taken as extra credit, but we did not require they even attempt it. Instead, we simultaneously evaluate both branches separately and *merge* the variable assignments. For example, consider the following code:

```
a := 4
b := 5
if a < b:
   c := n*n
   d := 5
else:
   c := n
   e := 10
```

Running the true branch results in $\{a : 4, b : 5, c : n^2, d : 5\}$, and running the false branch results in $\{a : 4, b : 5, c : n, e : 10\}$. Then, since we are interested in the Big Oh result, we take the pairwise maximums of the assignments resulting in $\{a : 4, b : 5, c : n^2, d : 5, e : 10\}$.

Complications arise if either or both branches of the if statement have a return. In that case, we continue two simultaneous evaluations until both step to a return. In other words, instead of stepping to a single command, we step to two commands which are both stepped simultaneously.

For Loops. Our symbolic algebra syntax provides

$$\texttt{iter}(e_1; e_2; \texttt{var}[n_1]; e_3; \texttt{var}[n_2])$$

which represents an indexed iterated function. In particular, it represents the value of output *after* code like the following:

```
output := initial_value
for i → UPPERBOUND(...):
   output := LOOPBODY(i, output)
```

This is easy to add to our symbolic algebra library as a variety of cases; for instance, if the loop body is `output := output * i`, then at the end of the loop, `output` should be

$$\texttt{initial\_value} \prod_{i=0}^{\texttt{UPPERBOUND}(\dots)} i = \texttt{initial\_value} \cdot \texttt{UPPERBOUND}(\dots)!$$

Most common cases are as simple as this. So, then, the task becomes turning loop bodies that rely on *multiple input variables* into loop bodies that only rely on one. The algorithm we used was as follows:

- Abstractly interpret the entire loop once; then, the variables mapping contains symbolic functions in terms of various variables.
- For each of the variable mappings, check if the loop changed them.
- For each variable mapping that changed, determine which variables the computed symbolic function is in terms of.
- Find any variable that is dependent on at most itself and the iteration variable. Then, convert all references to this variable to `iter` symbolic functions, and repeat the check for dependencies.
- If we ever get stuck updating variables, then set the output of everything to `infty` to make sure our analysis is sound.

For example,

```
j := 5
for i to n: # i not dependent
   i := j   # because resets on iteration
   j := i+1 # j only dependent on self
   k := j   # k dependent on j
```

Since j is only dependent on the iteration variable and itself, it is of the form:

```
j := 5
for i to n:
   j := j+1
```

And we can replace its value with $\texttt{iter}(j+1; n; j; 5; i)$.

Recursion. A generic recurrence (for a program) takes on the following form:

$$f(n) = \sum_{i=0}^{q} a_i \cdot f(g_i(n)) + c(n)$$

That is, it is a sum of recursive calls where each recursive call is a function of its arguments and an additional amount of work that is non-recursive.

Our algorithm handles recursion by constructing the recurrence from the program and feeding this recurrence to the symbolic algebra analysis using the recurrence constructor.

Depending on what one wants to teach the students about, we can ask students to implement simple summation closed forms and/or closed forms for recurrences (either by the Master Theorem or directly).

In particular, we found that limiting our attention to programs with recurrences of this form

$$f(n) = a \cdot f(g(n)) + c(n)$$

worked well.

In this representation, $g(n)$ represents some function on n such that $n > g(n)$ for all $n \in \mathbb{N}$. a is the number of recursive calls, and $c(n)$ is the amount of non-recursive work we do.

Recall that the goal is turn the function call into a symbolic function that represents it. So, whenever we want to step a recursive call, we can accurately step it to some $\texttt{recurrence}(-)$ (doing every other construct normally). This will yield a correct, but unsimplified output.

For example, consider the following recursive function:

```
f(x) :=
   if x > 5:
      y := 5
      z := f(x−1) + x
      return y ∗ f(x−1) + 2 + z
   else:
      return 2 ∗ f(x−1) + 2
```

If we are stepping $\texttt{call}(f; x)$, we would get:

$$\texttt{max}(5*\texttt{recurrence}(1; x-1; 0; x)+2+\texttt{recurrence}(1; x-1; 0; x)+x, 2*\texttt{recurrence}(1; x-1; 0; x)+2))$$

The max is the result of our if algorithm; we choose those particular parameters, because they represent the recursive call (with no extra overhead). Now, what remains is to "simplify" this expression; so, that it is in a usable format. In other words, we would like to combine the overhead and various recurrences together. This can be accomplished with a recursive function that combines by cases on which type of symbolic function it is given.

In particular, the result for this recurrence should be

$$\texttt{recurrence}(6; x - 1; 2 + z; x)$$

, which can be easily analyzed by the symbolic algebra library.

All of the other parts of the interpretation can be done exactly, and we do not need to change them. It is important to note that all of the changes we made over-approximate the output; so, our abstract semantics will always be sound.

5.4.5 Time Complexity

We discuss the difference between output complexity and time complexity. We tell students that our time complexity analysis might use our output analysis, but that we do not really care about the values of the variables any more; so, they should create a new copy of the original interpreter and work from there for the time complexity analysis. We tell students to change the step counter to use the symbolic algebra function. Then, we describe each of the if, for, and recurrence algorithms. In this analysis, we are interested in approximating the *number of steps* of a Sub{Setɿy} function $F(x)$.

Concrete Semantics

Consider the $\mapsto$ relation (defined as the standard semantics) of Sub{Setɿy}. Every Sub{Setɿy} function, F, takes some number, $\mathcal{S}(F, x)$, of steps when given the input x until it is a value. The concrete semantics of this analysis is the analogous to the concrete semantics for the output analysis, except of $\mathcal{S}$ instead of F directly. That is, U is the least upper bound in the symbolic functions lattice, U, such that $\texttt{big-oh}(U) > \texttt{big-oh}(X)$ for all X in the lattice where $\mathcal{S}(F, x) \in \mathcal{O}(U)$. Similarly, we are concerned with a largest lower bound, L for the Big-Omega analysis.

Abstract Semantics

Our abstract semantics over-approximates U by modifying the original concrete semantics in various ways. We describe the changes to the concrete semantics of Sub{Setɿy} that are necessary to complete this analysis.

Just like before, we consider symbolic functions of the variable x to be values instead of just functions of constants. Unlike before, we do not need to keep track of any of the variables. Instead we keep track of a symbolic function representing how many steps we have taken; it should be incremented every time we use a rule. Also just like with output, we must approximate each of the individual commands.

If Statements. The idea is identical to the output analysis; we continue two analyses (one for the true branch, one for the false branch) until they both step to a return, and then we take the maximum of the two step functions.

For Loops. We step through the loop body once, yielding a step count S; then, our step count is just `output_complexity(upper_bound)` $\times$ S.

Recursion. Here, we can calculate all of the necessary pieces individually by running through the code a single time. The non-recursive work is just the step count of running through the function; the number of recursive calls can be counted directly; we should take the `max` of all the recursive call arguments.

5.4.6 Closed Forms

If it is a learning outcome, we can ask students to implement closed forms of summations and/or recurrences. In the version we gave, we asked students to implement the Master Method, because of the mileage that can be gotten out of it.

5.5 Pedagogical Outcomes of Our Approach

We believe that *writing down* the algorithms they have been using to compute complexities can be very illuminating. This is particularly true when they realize that the reason they had to do output complexity first is because they actually use it when they are computing time complexity.

This assignment definitely satisfies our original goal of giving a "cool" demonstration of how math can be useful in Computer Science, and while the assignment is somewhat involved the result feels like a very big accomplishment for what is actually not that much work (assuming some of the complexities of how the symbolic algebra library works are hidden).

The analysis *does not always work perfectly* which provides an opportunity to discuss (1) why that is, and (2) that Big-Oh is just an upper bound.

The assignment neatly complements limitations of computation, as the language deliberately does not have unbounded recursion. Depending on the course, this (and other phenomena like it) could be completely motivated talking points.

Finally, depending on the level and interests of the students, pieces can be included or removed very easily which makes for a very customizable set of outcomes and prerequisites.

It should be noted that there are several drawbacks to the assignment. First, it necessitates introducing interpretation and the idea of "programs about programs", but this is necessary anyway when one talks about the halting problem. Second, it requires a minimal understanding of iterated functions, but again, this can be a positive, if the course discusses functions/induction/etc.

5.6 Case Study in 15-151

The second time we taught 15-151, we used this assignment, and we believe it was a success overall, but there are some things we would do differently next time.

Many of the problems that arose in our implementation of the assignment were due to the immature code base, and bad decisions about what should actually be in the assignment description and starter code. In particular, we gave the students a "shell" of an interpreter, rather than the concrete interpreter for the language; we believe providing the entire interpreter would have reduced the overhead in understanding *interpretation*. Our organization of code in the object oriented style (which students were largely not familiar with) also caused confusion, because students were unable to reason about when or how the analysis was getting called recursively. Finally, the assignment description wasn't entirely robust, and we should have been more explicit about implementations of several of the algorithms.

These issues aside, students seemed to enjoy the assignment. We believe it reinforced many of the pedagogical outcomes as intended, and it definitely showed students how the mathematics we were teaching fits in a programming context.

Part III

Coping with Consistency, Timeliness, and Quality of Feedback in Large Courses

Chapter 6

Judgement-based Grading

In this chapter, we discuss grading content in domains with student work that is written in a fuzzy medium (e.g. a short essay or mathematical proof) that contain very little subjective content. In particular, our focus has been on the *proof domain*, but almost all of the discussion applies equally to short answer questions about literature, science, and many other domains.

We begin by explicitly providing a high-level model of grading (who are the actors, what does the output look like). Then, we discuss a variety of criteria that can be used to determine the quality of grading. We discuss common schemes for grading and provide an overview of our new grading method "Judgement-based Grading". Throughout the discussion of each of these systems, we "grade the grading"–each of the methods emphasizes a different subset of the quality criteria.

6.1 What is Grading?

We begin by defining some terminology that will be useful throughout our discussion.

- A *prompt* is a piece of prose that is given to students in an attempt to assess their understanding of a concept/idea/fact/etc.
- A *submission* is a student artifact that attempts to respond to a *prompt*.
- We say a prompt is *subjective* precisely when multiple reasonable experts in the domain might generate multiple distinct submissions; that is, if there are multiple correct responses that cannot be reconciled with each other.
- A submission is *fuzzy* if it is not relatively simple to programmatically determine its correctness. So, for instance, a multiple-choice prompt would result in a submission that is not fuzzy, but an essay prompt would result in a fuzzy submission.
- A *score* is a result drawn from a very limited pool of possible scores (e.g. a number, percentage, GPA, letter grade) that is intended to indicate the level of match between the set of "gold standard" submissions and a particular submission.
- *Feedback*, as defined by Hattie [HT07] is "information about the content and/or understanding of the constructions that students have made from the [prompt and

submission].” Hattie goes on to describe the three major questions good feedback *must* answer (paraphrased):

- What are my goals?
- What progress am I making?
- What do I do next?

6.1.1 Defining Grading

Finally, we can define *grading* as the combination of two mappings:

- A *grader-driven* mapping from (prompt, grading agents, submission) to (score, feedback)
- A *student-driven* mapping from (score, feedback, student) to (student adjustment, instructor adjustment)

Or, to put it another way, we consider the adjustments students and instructors make (which can be evaluated using future summative assessments) to be *part of the grading process*; the moment when graders “release grades” is only part-way through the process. Because we cannot *change* the second mapping, our goal in grading must be to make the first mapping compatible with the second one.

6.1.2 Grading Agents

It is important to understand that the set of “grading agents” is not uniform. We say a grading agent is *trained* if they are aware of the instructor’s preferred *grading best practices* **and** *domain content*—not just one or the other. This definition is deliberately restrictive; we differentiate trained agents from *qualified* ones who might not have ever graded or be aware of how to grade, but understand the domain content enough that given (hopefully minimal training), they can be trained agents. A typical set of grading agents contains a combination of the following types of people, of varied quantity and allocated effort:

- An *expert grader* (e.g., the instructor, a domain authority) is a respected grader who, given enough time and effort, would generate the “gold standard” of grading.
- A *journeyman grader* (e.g., an experienced undergraduate or graduate TA) is a qualified grader who, given enough time and effort, would generate a reasonable approximation of the “gold standard” of grading *if they were trained*.
- A *novice grader* (e.g., an unexperienced undergraduate TA) is a grader who is grading or TAing for the first time. With enough resources, they are sometimes paired with a journeyman or expert grader to review their work. They often make up the majority of the graders, and are rarely trusted to review submissions without some guidelines given by a journeyman or expert grader.
- An *unqualified grader* (e.g., an undergraduate TA used to “fill the last TA spot”) is a grader who is not competent in the domain content. They are often indis-

tinguishable from novice graders at first glance, but they will generate very faulty output in the grading process.

6.2 Grading. . . Grading

In this section, we attempt to outline various criteria that we believe determine the quality of the grader-driven grading mapping. These criteria provide a way to compare and evaluate various implementations of the grading mapping. They span a wide variety of parts of the grading process. As before, we concern ourselves not only with criteria that effect the correctness and speed of generation of the feedback and score, but also how compatible the feedback and score are with *student/instructor consumption*.

Throughout these criteria, we assume the prompt and grading pool are fixed. We make no claim that every criterion is equally important, but a grading system that accounts for *strictly more* criteria is better. Furthermore, if a grading system ignores an entire category of criteria, it has a severe deficiency.

6.2.1 Criteria for Scoring

Timeliness. Because scores offer a signal (albeit a poor one) to students about what they understand and what they do not, it is imperative to provide students with the scores in a timely manner. Similarly, instructors need averages/medians to possibly re-target or review material if they differ significantly from expectations.

Correctness. Since scores are usually used to compute student final grades, correctness is obviously important. It should be noted that edge cases on correctness can often err on the *low* side, as long as the grading system allows for students to request *re-grades*.

Consistency. Whatever scale/criteria/features are used must be *consistent* throughout the grading and over all students. This is particularly important in courses which use non-standard cutoffs of some kind to generate final grades in the course, but even in other courses, it is important to promote *fairness* across grading.

6.2.2 Criteria for Feedback

Timeliness. Feedback, particularly good feedback, offers students a very strong signal about what they can improve and how to do so; as feedback is delayed, students become significantly less likely to read and integrate it into future assessments. Furthermore, slow feedback becomes a huge source of *resentment* toward the graders (who are often the course staff) and instructor(s). According to Hattie [HT07], "the effect sizes from delayed feedback were -0.06 for easy items, 0.35 for midrange items, and 1.17 for difficult items."

Answers Hattie's Questions. If the feedback provided to students answers all three of Hattie's questions, it is significantly more useful and actionable for students.

Tone. Extrinsic rewards and feedback perceived as "controlling" generally have a very negative effect size. Arguably, *no feedback* is a better alternative to telling students no actionable information in the feedback. Furthermore, if the feedback is condescending, does not explain *why* the errors were errors (particularly on *stylistic errors*), students respond very negatively and will often argue until the end of the semester, refusing to change.

Personalization and Style. Over the course of our prototype systems, we have tried multiple styles of feedback. Throughout, we have found that *perceived* personalization–that is, that the student believes the feedback was written explicitly for their submission even if it was not–has a large effect on how important students view feedback. Consider the following styles of feedback for the same error (for demonstration purposes, assume the actual submission uses k, and that x was intended by the grader to "stand in" for an arbitrary variable):

- The proof misuses the induction predicate in a way that assumes the conclusion.
- Your submission accidentally assumes the conclusion in the IH.
- Your submission assumes $P(x) \Longrightarrow P(x+1)$ in the induction hypothesis.
- **Your submission assumes $P(k) \Longrightarrow P(k+1)$ in the induction hypothesis.** Then, in your induction step, you proceed to prove $P(k) \Longrightarrow P(k+1)$; you probably meant to assume $P(k)$ instead. You actually used that assumption instead!

There are a number of interesting progressions and features throughout these examples, and based on our experience talking with students, their improvement, and how often they reviewed their feedback, we believe they are significant.

(1) **"The proof" vs. "Your submission"**
"The proof" sounds "robotic" and students view the feedback as more general and less applicable to their own situation. "Your submission" sounds more friendly, even though the content is no different.

(2) **"Induction Hypothesis" vs. "IH"**
Although this is a typical abbreviation, jargon should be avoided as much as possible in feedback, it makes it harder to understand, which means it is more likely for a student to stop trying to understand it.

(3) **"P(x)" vs. "P(k)"**
Despite the clear usage of a "stand in" variable, students are more likely to believe the feedback is actually an error, rather than *their* error if it does not match exactly with their submission.

(4) **General vs. Specific**
To answer Hattie's questions, feedback must do more than provide what the error is. In the last incarnation of the feedback, it goes into more detail, describing the exact situation that happened in the submission, how to fix it, and attempting to convince the student that the feedback is not an error. It

should be noted that despite all of this, the feedback is still general enough to apply to a large percentage of submissions!

Location Information. Feedback is often about an attribute of the submission that is localized to a very particular spot (or multiple spots). If the feedback is not associated directly with that spot, it can be difficult for students to find it. Furthermore, even if the location is marked, if the feedback is not *visible* at exactly that location, there is still a disconnect.

Feedback is Prioritized. For better or worse, many students are driven by the *score* and pay little attention to the feedback itself. A grading system should try to combat this in any way possible.

6.2.3 Criteria for Grading Pool Compatibility

Many courses use what we call a *grading pool* (a pool of varied graders) rather than a single grader. Grading pools become particuarly important in large courses where it is infeasible to use a single grader. A grading system must take extra considerations into account if it is to be compatible with varied grading pools.

Supports A Grading Pool. Many courses ask multiple graders to grade in a variety of fashions. Some common allocations are "one grader to one prompt", "many graders to one prompt", and "one grader to one *course* section". Each of these have their own challenges. "One grader to one prompt" challenges timeliness and correctness, because each grader is not balanced and checked, and each individual grader must grade a large number of submissions; however, this style makes consistency relatively easy. "Many graders to one prompt" makes timeliness easier, but consistency (graders must work together) significantly more difficult; furthermore, because this style generally requires *more* graders, the average quality of grading can go down. Finally, allocating graders *over course section* instead of prompt causes a significant consistency loss, because every section is graded consistently *internally*, but *over the whole course*, each section might be extremely different. Any grading system intended to be used with a grading pool *must* consider these concerns.

Minimizes Necessary Training. Because the quality of the agents in the grading pool is not uniform, any unqualified or novice agents will need training. If a grading system requires significant training from an expert grader, that can be very costly.

Grows with Scale. If the grading strategy is very resource intensive, this could present a major problem. An ideal grading system will not treat every grading agent equally, because they are not! It can be very useful to only ask experts to do "expert tasks", journeymen to do "journeymen tasks", etc.

Minimizes Damage of Known Grading Phenomena. There are a number of known phenomena when grading that reduce impartiality (e.g. looking at student names, grading submissions that diverge drastically as "special cases", and following guidelines syntactically rather than semantically). A good grading system will minimize these concerns.

6.2.4 Criteria for Grader Efficiency

Maximizes Speed per Submission. All student submissions are not created equally; some submissions are particularly difficult to grade and others are very easy. Median and average speed per submission are reasonable indicators of grader efficiency.

Minimizes Repetitive Work. When graders are forced to repeat the same work (e.g. writing the same piece of feedback) repeatedly, it (1) decreases quality as they grade more submissions, and (2) makes graders more hesitant to continue grading.

Maximizes Grading Malleability. Fairly frequently, as grading progresses graders realize they have made errors. In particular, several distinct concerns are

- A mistake was much more frequent/less frequent than expected, and consequently, the grader would like to retroactively change the scoring of that mistake on all submissions.
- A piece of feedback was poorly worded, or a grader better understood the mistake a group of students made when reading a different wording of it. Consequently, the grader would like to clarify the feedback for all of the students who made that mistake.
- The grades students received turned out to be far too high or too low, at the instructor's discretion; ideally, this would be fixable by finding and fixing the *actual deductions* that did or did not happen and why. (Generally, instructors will just curve the exam instead, which does not actually fix the problem; it just masks it.)

Uses Technology Appropriately. Some (or all) parts of grading might be better done by a computer. A reasonable grading system will leverage technology whenever possible. This is particularly true for repetitive or automatable parts of grading.

Guides Graders through Grading. To minimize required training, the grading process should be *difficult to do incorrectly*; that is, the grading system should force graders to avoid making errors as much as possible.

Provides Escalation Opportunity. Occasionally, graders need to "escalate" submissions. This happens in grading pools when novice graders need more help; sometimes, graders want another pair of eyes to verify that their analysis of the submission is correct. Another (hopefully occasional) reason for escalation is the finding of a cheating case.

Availability. Ideally, grading should be able to occur anywhere; there are positives and negatives to "grading parties" (where the entire grading staff grades in the same room, at the same time), but sometimes graders cannot be present.

6.2.5 Criteria for Instructor Usage

Easily Monitorable. Ideally, an instructor would be able to assess the percentage completion of grading. This is both for accountability (is one grader not doing his work?), for planning purposes (announcing to the class when grading will be finished), and for re-allocation purposes (getting help to a grader who was allocated too much work accidentally).

Easily Summarizable. An instructor should be able to easily understand what misunderstandings the students have. In particular, an instructor should be able to modify future lessons to deal with obvious concerns based on assessment performance.

6.3 Restricting Domain Content and Prompts

Different types of domain content and prompts require different grading procedures and systems. In part, this is because the types of criteria that graders should be concerned with can differ drastically, but more importantly, a grading system that works very well in one domain might be ill-suited or just *worse* for another domain. We make two distinctions that we believe characterize the space well: fuzzy vs. non-fuzzy submissions and objective vs. non-objective domains.

Because we were primarily concerned with *informal proofs*, the combination we are most concerned with throughout the remainder of this chapter is *fuzzy* and *objective*. In this section, however, we briefly overview examples of domains/prompts that fall under all four quadrants, as well as grading strategies for the three quadrants we will not focus on.

6.3.1 Fuzzy-Objective

Fuzzy, objective domains/prompts are particularly difficult to automate, because state-of-the-art natural language processing, while amazing, can miss nuances and subtleties that are very important to the output of grading. Examples that fit into these quadrants are actually surprisingly diverse: programming "style", short answer questions in many domains (science and literary, to name two), math calculations that request students "show their work", grammar correction, and, of course, informal mathematical proofs are all common and difficult grading problems.

6.3.2 Non-fuzzy-Objective

One of the major reasons we do not focus on this quadrant is that there is nearly always a better way to grade these items than strategies used in the fuzzy-objective quadrant. The examples in this quadrant are typical: multiple choice questions, programming correctness, and formal proofs. These can all be graded with programmatic solutions at a higher speed and accuracy than would ever be possible for human graders.

6.3.3 Non-objective Prompts

Any non-objective domain is either (1) not intended to scale, or (2) very difficult or impossible to quantify. A prototypical example of these quadrants is the SAT essay.

Another example is creative writing. These quadrants tend to either avoid feedback entirely, grade on effort, grade on "insight", or some other amorphous quantity.

6.4 Standard Grading Systems

In this section, we review the types of grading systems typically used in fuzzy-objective domains. We discuss the criteria they excel at–and those that they fail to deal with.

6.4.1 "It feels like a. . . " Grading

"It feels like a. . . " grading is a very poorly defined process in which a score is generated for the student submission at the whim of the grader. Occasionally, it is supplemented with "overall" feedback that might give a few points to fix on, but the feedback is rarely anything specific. Given the lack of emphasis on feedback as an important item at all, we will not spend a significant amount of effort on this strategy.

6.4.2 Rubric Grading

Rubric grading (not to be confused with "scoring guidelines", discussed below) is a standard technique to facilitate grading is a scoring rubric [ML00]. A rubric is defined by

- a set of "dimensions" of grading (grammar, thesis, evidence)
- a set of "levels" indicating how good the work is
- a set of descriptors for each dimension × level

Rubrics have many advantages as compared to "holistically grading" submissions by assigning them subjective values based on the instructor's perceived quality of submission. We review our criteria for grading systems, starting with the positive features and then reviewing the negative ones. We note up front that there is an explicit tension in this grading system between training and existing agents in the pool. Unfortunately, grading agents need to be experts to successfully use this grading system effectively.

Positives

Assuming we have expert graders, correctness and consistency of scores and feedback are very high with rubrics. Because the system emphasizes feedback over scores, nearly all of the criteria for feedback are also of very high quality. Because the instructor tends to grade a large majority of submissions using this system, the instructor usage criteria are also largely perfect.

Negatives

The criteria that rubric grading fails on are grading pool compatibility and grader efficiency. Even if the graders try to use technology (e.g. Microsoft Word's feedback features), there are not many places for technology to penetrate the procedure. Grading is not at all malleable, the grader is fully in control of the procedure, and grading is very time consuming. Fundamentally, this grading system is not at all scalable.

6.4.3 Scoring Guidelines

Scoring guidelines are a list of attributes that are written *before* grading occurs by an *expert* grader. These attributes are generally *all positive* (submissions should do $x, y, z, \dots$) or *all negative* (submissions should not do $a, b, c, \dots$). This is a very common strategy for courses with large groups of teaching assistants and AP exam grading. While this system, overall, is more suitable to fuzzy, objective domains, it *still* has many drawbacks.

Positives

This grading system handles the criteria for scoring *very well*; it allows journeymen and novice graders to grade quickly and consistently; for the most part, scores are within an allowable delta of correct, as well. This form of grading supports nearly every type of grading pool, as long as the grading guidelines are written before grading is supposed to begin. Depending on the quality of the feedback provided, graders can be incredibly efficient with this strategy; if the guidelines are positive ones, a common strategy is to put check marks next to locations that satisfy the guidelines, allowing grading to go extremely quickly. Generally, this strategy allows the instructor to choose *either* availability *or* escalation opportunities.

Negatives

Unfortunately, scoring guidelines fail on nearly every feedback criterion. If significant feedback is provided, it is usually written down–and it is very repetitive to write the same piece of feedback on every submission with that mistake. Graders are rarely trained to write down more than "key words" or the guideline itself as feedback, and because they are in-experienced and going quickly, tone can be very negative and personalization is lacking. Because scores are sometimes put online (via blackboard or another system) and feedback usually must be picked up in person, the scores are prioritized over the feedback, as well. In our opinion, the negatives on the feedback criteria are the *most serious* for this grading system.

Scoring guidelines can minimize training, but at the expense of poor feedback and much damage due to known grading phenomena. Experienced graders can help novices and journeymen by working with them to explicitly guide them through grading, but this creates a serious deficit to efficiency. Furthermore, this system is *not at all malleable*; it fails on all possible reasons feedback/scores would need to be changed. To change anything, *all submission that were already graded* must be re-graded. Guidelines also fail to produce useful signals to the instructor for monitoring grading or summarizing it.

6.4.4 Machine Grading

Machine grading (usually supported by machine learning) replaces human graders with an algorithm that determines a score and occasionally outputs feedback. Feedback on these systems tends to miss nearly all of the criteria, unless the system has been explicitly designed for the prompt in question. These scoring machines also tend to be completely opaque, and because they work off of data, they may be within an acceptable range of correctness for scoring, but instructors will likely not match exactly with their determination of correctness. It is worthy of note that for non-objective domains or grading that does not require feedback, machine grading is a significantly more reasonable strategy.

6.5 Our Approach: Judgement-based Grading

We propose a modified version of scoring guidelines called judgement-based grading. We call a single grading criterion (a correct or incorrect attribute of the submission) a *judgement*. With judgement-based grading, the judgements are a by-product of the grading process. And the point values are not even considered until *all the submissions have been reviewed*. Graders review submissions by attributing any applicable judgements to each submission they see, adding new ones as they see fit–this is very similar to grading with scoring guidelines except it uses our web interface and the grader does not see or consider point values during this process. The simplest version of the interface, which we indeed implemented in our first prototype, would provide graders with a list of check boxes (one for each judgement) to click.

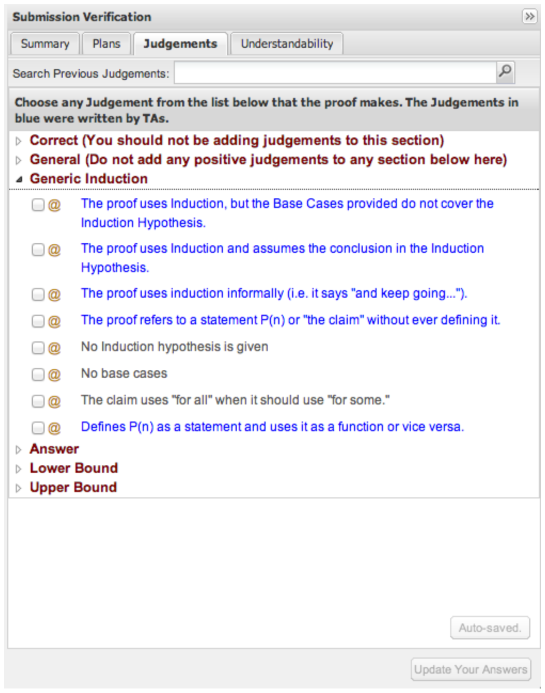
6.5.1 List Judgement-based Grading

Our first implementation of Judgement-based Grading centered around a semi-static list with check-boxes. The submission appeared on the left, and the grader had a categorized list of judgements to select from. A grader added a new judgement by clicking below the list. After reviewing all the submissions, a grader would "allocate" points to judgements, and the scores would be automatically generated.

The major benefit of this implementation was that it increased grading speed; however, it did not address many of the important criteria we evaluate grading systems on. For example, students saw their feedback as a list of judgements next to their proof, and they could click on an @ to see the location they were attached to. We created a second, more complete implementation which we discuss below.

6.5.2 Templated Judgement-based Grading

The general idea in the new system is similar to the old one, but it had more features and a more complicated type of judgements that allow it to be more useful. We begin by briefly describing the procedure a grader would go through on our newer prototype.



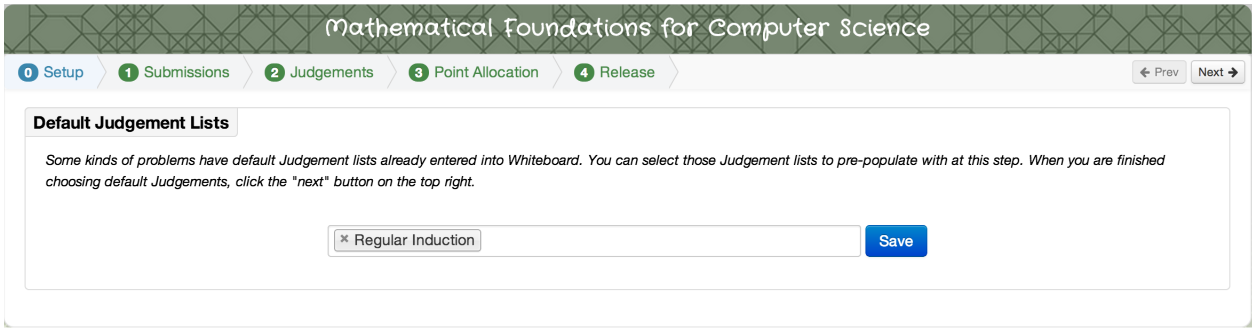
Choose any Judgement from the list below that the proof makes. The Judgements in blue were written by TAs.

- Correct (You should not be adding judgements to this section)
- General (Do not add any positive judgements to any section below here)
- Generic Induction
  - The proof uses Induction, but the Base Cases provided do not cover the Induction Hypothesis.
  - The proof uses Induction and assumes the conclusion in the Induction Hypothesis.
  - The proof uses induction informally (i.e. it says "and keep going...").
  - The proof refers to a statement P(n) or "the claim" without ever defining it.
  - No Induction hypothesis is given
  - No base cases
  - The claim uses "for all" when it should use "for some."
  - Defines P(n) as a statement and uses it as a function or vice versa.
- Answer
- Lower Bound
- Upper Bound

Figure 6.1: The judgement list in our original prototype system

Setup Step

In the first step, "Setup", a grader could partially pre-create the judgements by drawing on "common pools of judgements" or previous incarnations of the same prompt.



Default Judgement Lists

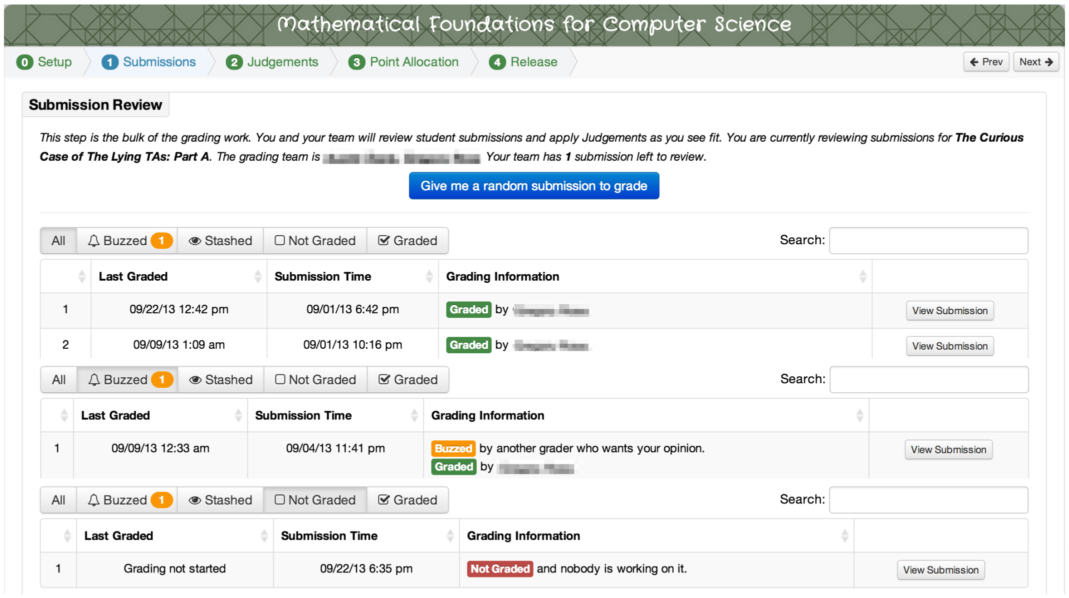
Some kinds of problems have default Judgement lists already entered into Whiteboard. You can select those Judgement lists to pre-populate with at this step. When you are finished choosing default Judgements, click the "next" button on the top right.

Figure 6.2: A grader choosing to begin with a pre-populated set of judgements

Submission Review Step

Then, the system would provide them with a view of the state of each submission (which graders have worked on it, when the grading was last edited, etc.), or allow them to start reviewing a random submission. The "random submissions" were not exactly random; the system uses several features of submissions to determine how to group and order them. For instance, it is relatively easy to determine if a submission will be frustrating to grade by its length. If students were allowed to work together to solve problems in groups, another useful ordering is to put an entire group's submissions in order. This allows less cognitive load for the remaining members of the group after the first person.

In the image below, we superimpose three different "states" in the same image. The top two rows show a partial screenshot of a listing of all the submissions; the middle shows what a "buzzed" submission looks like (we discuss this later), and the bottom filters to only show ungraded submissions.



Submission Review

This step is the bulk of the grading work. You and your team will review student submissions and apply Judgements as you see fit. You are currently reviewing submissions for ***The Curious Case of The Lying TAs: Part A****. The grading team is* [redacted] *Your team has* ***1*** *submission left to review.*

	Last Graded	Submission Time	Grading Information	
1	09/22/13 12:42 pm	09/01/13 6:42 pm	Graded by [redacted]	View Submission
2	09/09/13 1:09 am	09/01/13 10:16 pm	Graded by [redacted]	View Submission

	Last Graded	Submission Time	Grading Information	
1	09/09/13 12:33 am	09/04/13 11:41 pm	Buzzed by another grader who wants your opinion. Graded by [redacted]	View Submission

	Last Graded	Submission Time	Grading Information	
1	Grading not started	09/22/13 6:35 pm	Not Graded and nobody is working on it.	View Submission

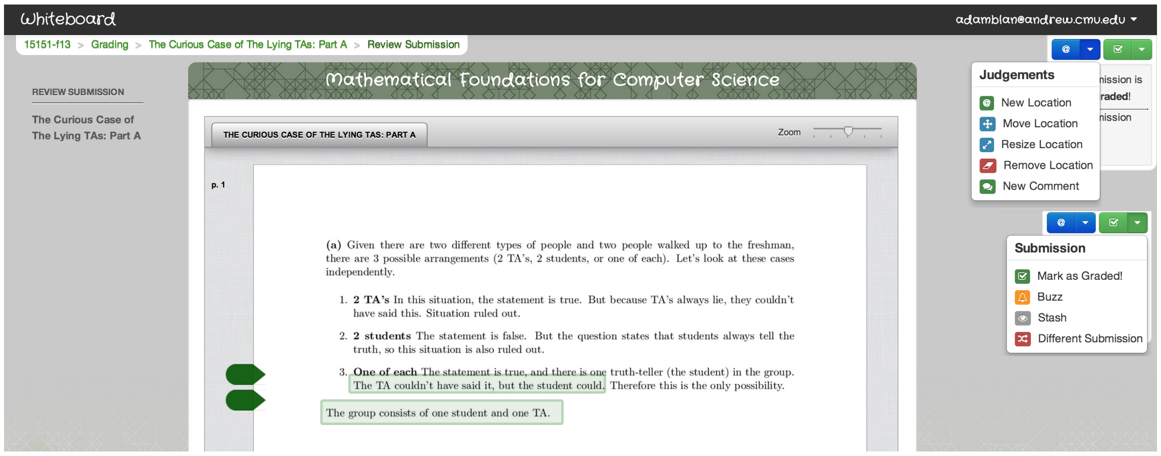
Figure 6.3: A grader reviewing the states of submissions

Upon choosing a submission, the grader is provided with two menus ("Judgements" and "Submissions") that allow them to mark up a particular submission. When using List Judgement-based grading, the judgements are chosen and locations are attached to the judgements; the newer system asks the grader to choose locations (or location agnostic "comments"), and judgements can be applied on top of those locations. The Judgements menu allows graders to create and modify locations; the submission menu allows graders to change the state of the submission.

We bring special attention to two features: stashing and buzzing. Stashing is the equivalent of "putting the submission on the bottom of the pile"; it keeps the grader in the state of grading the submission, but when asked, the system will provide any other submission first. Buzzing asks the grader to write a short message and choose another grader; then it sends an email to the other grader with the message and a link to the submission.

Judgement Attribution: Choosing a Judgement

Once the grader chooses a location with an attribute worth commenting, she moves on to choosing or creating a judgement to match her comment.


Whiteboard
adamblan@andrew.cmu.edu
15151-f13 > Grading > The Curious Case of The Lying TAs: Part A > Review Submission
REVIEW SUBMISSION
The Curious Case of The Lying TAs: Part A
Mathematical Foundations for Computer Science
THE CURIOUS CASE OF THE LYING TAS: PART A
Zoom
p. 1
Judgements
New Location
Move Location
Resize Location
Remove Location
New Comment
Submission
Mark as Graded!
Buzz
Stash
Different Submission

(a) Given there are two different types of people and two people walked up to the freshman, there are 3 possible arrangements (2 TA's, 2 students, or one of each). Let's look at these cases independently.

1. **2 TA's** In this situation, the statement is true. But because TA's always lie, they couldn't have said this. Situation ruled out.
2. **2 students** The statement is false. But the question states that students always tell the truth, so this situation is also ruled out.
3. **One of each** The statement is true, and there is one truth-teller (the student) in the group. The TA couldn't have said it, but the student could. Therefore this is the only possibility.

The group consists of one student and one TA.

Figure 6.4: A grader in the middle of grading a submission

In List Judgement-based Grading, judgements are a string of text describing the error and/or feedback. Our newer system has a slightly more complicated model. A judgement is made up of a "judgement schema" and a set of "explanations"; each schema and explanation is a "templated sentence". Intuitively, the schema is intended to briefly give an overview of the mistake or attribute being commented on, and explanations are *sometimes* associated with that schema. In other words, because a particular mistake can present in different ways (with possible different solutions or analogies useful in each situation), the explanations are intended to drill down to the correct instance.

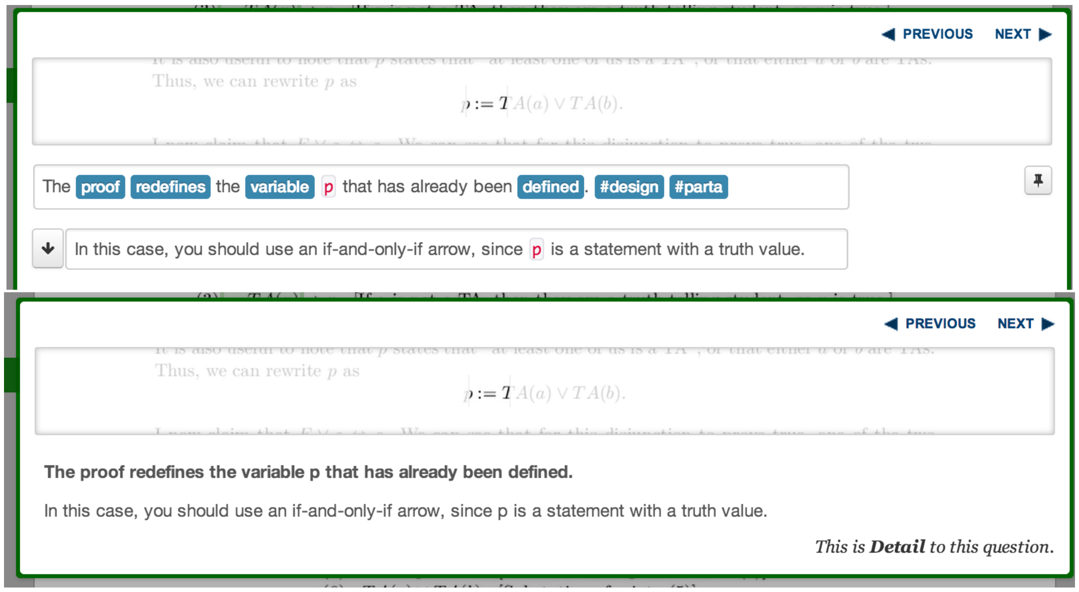
A templated sentence is a string of text that might have *tags* and *templates* in it. Tags are hash tags–just like on Twitter! They are used for categorization of judgements; tags with a double hash at the beginning do not show up in the sentence. A template is a variable or phrase that is a stand-in for a real variable or instance in a submission; the template can be instantiated differently at every location it occurs, or it can retain the original template, instead.

A student reviewing their feedback can click on a location and see the context and the selection; the schema shows up without templates or tags in bold, and the explanations show up as a paragraph.

Judgement Attribution: Customizing the Judgement

When a grader is trying to find a judgement, she may filter by one or many tags. Once she has found the judgement she is looking for, she pins it.

Then, she can choose explanations or add new ones, and instantiate templates. If the same template appears in multiple templated sentences in the same judgement (e.g. multiple explanations or an explanation and the schema) or multiple times in one templated sentence, instantiating it once replaces it in all instances.


◀ PREVIOUS NEXT ▶

Thus, we can rewrite p as

$$p := TA(a) \vee TA(b).$$

The proof redefines the variable p that has already been defined. #design #parta

In this case, you should use an if-and-only-if arrow, since p is a statement with a truth value.

◀ PREVIOUS NEXT ▶

Thus, we can rewrite p as

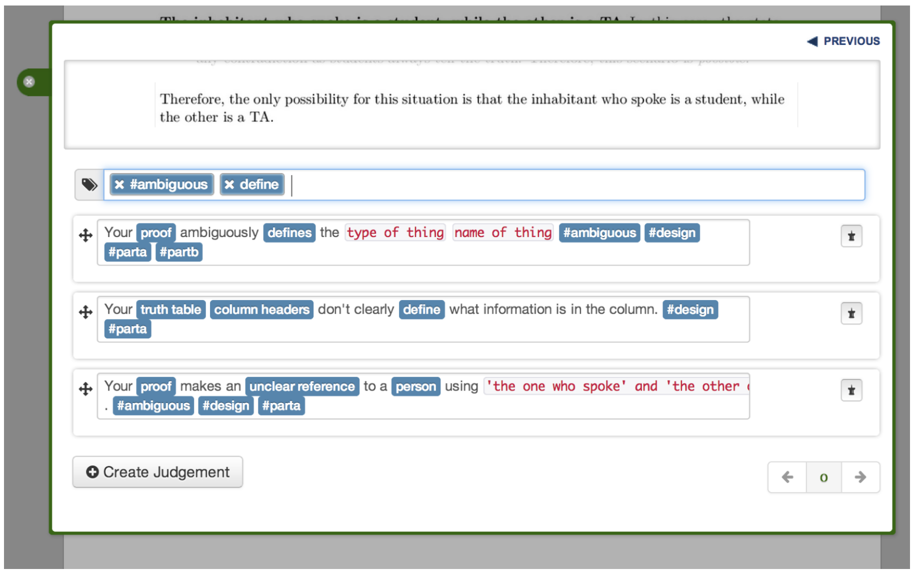
$$p := TA(a) \vee TA(b).$$

The proof redefines the variable p that has already been defined.

In this case, you should use an if-and-only-if arrow, since p is a statement with a truth value.

This is ***Detail*** *to this question.*

Figure 6.5: Top: A grader choosing a judgement; Bottom: A student reviewing feedback on a submission


◀ PREVIOUS

Therefore, the only possibility for this situation is that the inhabitant who spoke is a student, while the other is a TA.

× #ambiguous × define

Your proof ambiguously defines the type of thing name of thing #ambiguous #design #parta #partb

Your truth table column headers don't clearly define what information is in the column. #design #parta

Your proof makes an unclear reference to a person using 'the one who spoke' and 'the other . #ambiguous #design #parta

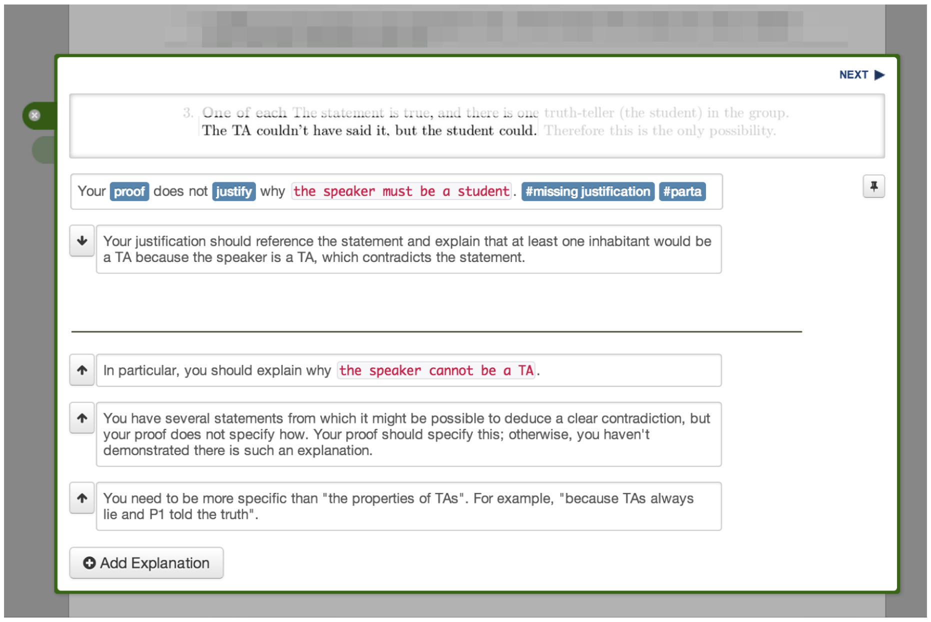
Create Judgement

0

Figure 6.6: A grader searching through judgements

Judgement Review

In all instances, the grader cannot give feedback without using an explanation or judgement. During Judgement Review, a list of judgements, their explanations, and their frequencies is provided and they can be edited, re-assigned, etc. The next step in grading is point allocation–the first time in the whole procedure where the graders think about point values.



3. ~~One of each~~ The statement is true, and there is ~~one~~ truth-teller (the student) in the group. The TA couldn't have said it, but the student could. Therefore this is the only possibility.

Your proof does not justify why the speaker must be a student. #missing justification #parta

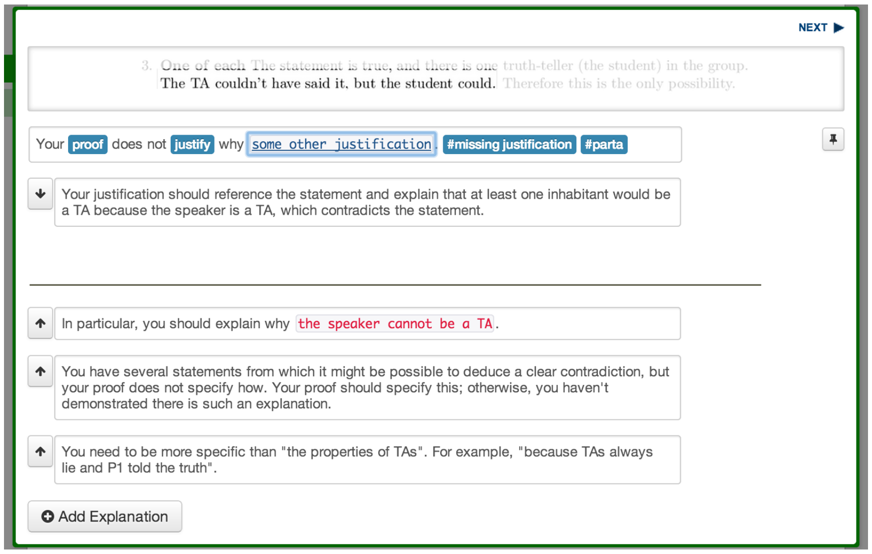
Your justification should reference the statement and explain that at least one inhabitant would be a TA because the speaker is a TA, which contradicts the statement.

In particular, you should explain why the speaker cannot be a TA.

You have several statements from which it might be possible to deduce a clear contradiction, but your proof does not specify how. Your proof should specify this; otherwise, you haven't demonstrated there is such an explanation.

You need to be more specific than "the properties of TAs". For example, "because TAs always lie and P1 told the truth".

Figure 6.7: A grader choosing the best fitting explanation (or adding a new one)



3. ~~One of each~~ The statement is true, and there is ~~one~~ truth-teller (the student) in the group. The TA couldn't have said it, but the student could. Therefore this is the only possibility.

Your proof does not justify why some other justification. #missing justification #parta

Your justification should reference the statement and explain that at least one inhabitant would be a TA because the speaker is a TA, which contradicts the statement.

In particular, you should explain why the speaker cannot be a TA.

You have several statements from which it might be possible to deduce a clear contradiction, but your proof does not specify how. Your proof should specify this; otherwise, you haven't demonstrated there is such an explanation.

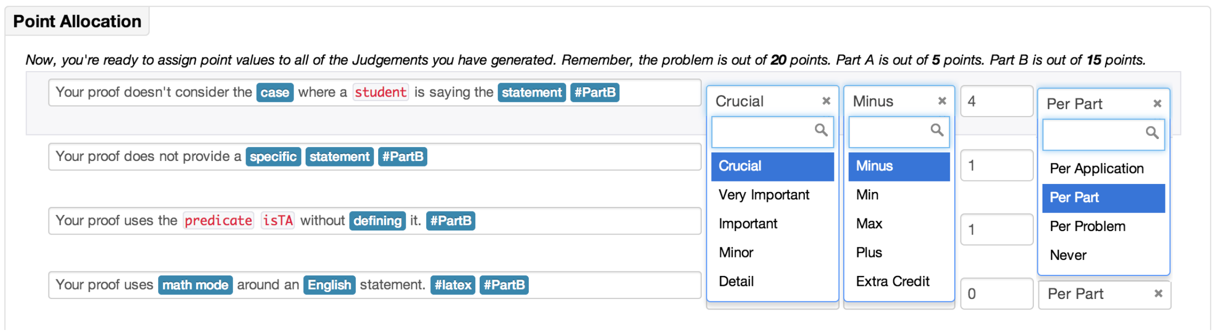
You need to be more specific than "the properties of TAs". For example, "because TAs always lie and P1 told the truth".

Figure 6.8: A grader filling in a template for a particular student

Point Allocation

Point allocation is the process during which judgements turn into a rubric. For each judgement, the grader chooses four options: (1) how important is the judgement, (2) how should the judgement affect the score, (3) how much should the judgement affect the score by, and (4) in what way should the judgement be applied.

When students see their feedback, they get a score for the problem, but they do *not ever see* the numerical value the judgement changed their score by; instead they see "This judgement is `<very important>`.", or an equivalent.


Point Allocation

*Now, you're ready to assign point values to all of the Judgements you have generated. Remember, the problem is out of **20** points. Part A is out of **5** points. Part B is out of **15** points.*

Your proof doesn't consider the case where a student is saying the statement #PartB — Crucial — Minus — 4 — Per Part

Your proof does not provide a specific statement #PartB — 1

Your proof uses the predicate isTA without defining it. #PartB — 1

Your proof uses math mode around an English statement. #latex #PartB — 0 — Per Part

Figure 6.9: A grader allocating points to judgements

"Minus" and "Plus" affect the score the same way standard additive and subtractive grading guidelines do; "Min" and "Max" indicate that any submission receiving that judgement should receive a minimum or maximum of that many points; "Extra Credit" is applied on top of the calculated score.

The "Per" judgements indicate how many times the judgement should be applied. For instance, at the beginning of the term, an instructor might not want to penalize for a particular judgement because they want to treat it as a formative experience; in such a case, the "Per" option would be never. Alternatively, a student who makes the same stylistic mistake all throughout a multi-part problem should probably not be penalized every time it is made; in such a case, "Per Problem" would make it only affect the score once.

6.6 Evaluating Judgement-based Grading

Finally, we evaluate our system on our own criteria.

The system ensures that scoring is identical throughout by calculating submission scores via a deterministic algorithm. The correctness of scoring is *at least as good* as with scoring guidelines, because ultimately, an expert will likely to the point allocation. Arguably, the (positive) difference provided by our system is that the point allocation occurs *after* the submission review; so, the allocator is significantly more informed about student understanding.

For a similar reason, our system is reasonably good at supporting a grading pool. Submissions can be sliced in any way, and there is no dependency on paper to limit timing. Because the system guides graders through the phases of grading, training is as minimal as possible. The system is very good at allocating types of graders to tasks; it is backed by a permissions model in which the instructor(s) can limit which graders can complete each phase; in particular, instructors can request k trusted graders to complete judgement reviews. The system, at no time, shows graders any names, does not allow a grader to edit a score as a "special case", and the templates make it clear when a part of the judgement is semantic instead of syntactic; so, we argue that Judgement-based grading does a good job of minimizing damage from known grading phenomena.

Judgement-based grading tries to get as much grader efficiency as possible. Instead of writing every piece of feedback many times, the system is a write-once-use-forever system; as a result, grading malleability is obviously handled well. Graders have multiple points at which they can easily change entire sections of the grading everywhere. Judgement-based grading attempts to use technology to handle any repetitive or error-prone parts of the grading process (e.g. writing judgements, finding judgements, adding up scores). If the system fails to guide a grader correctly, and a bad judgement is created, it should be discovered in review or by a fellow grader. "Buzzing" is an explicit answer to allow asynchronous escalation, and the lack of paper and easy communication makes the grading very available.

Judgement-based grading provides the instructor with an easily monitorable view of what grading is done and who is responsible for what is left. Additionally, it summarizes the most common judgements, score distribution (and how it changes as point allocation changes), and we believe the system can be easily extended to display any other summaries that might be useful.

Our system attempts to require good feedback in several ways. The two-tier feedback is an attempt to get graders to provide actionable items (in the explanations) and answer Hattie's questions. Graders have no medium for providing extrinsic rewards, and judgements worded in a "controlling tone" can be edited by the judgement reviewer. We believe personalization is where our system really shines. Because templates and explanations allow multiple levels of reuse, the comments–although very general, and not particularly customized–make students feel like they were written explicitly for them. Throughout using our second prototype system, many students were not even aware how much re-use there was in the feedback they received. The system forces locations to be chosen before the judgements, and the judgements are displayed in-line. Finally, by never showing the students the actual effects of the individual judgements on their score, they are forced to try to understand the feedback itself rather than try to "get back three points".

In addition to fitting the majority of our criteria, judgement-based grading allows us to expose only parts of the judgement attribution step to the students for peer evaluation which we discuss in 7.

There are some potential concerns with Judgement-based grading as a grading system. First, because there is a new application for both the instructor and the graders to learn, which increases training costs and requirements. Second, because all grading is done on the computer, it is easier to get distracted by the Internet or a game; this concern sounds trivial, but it is surprisingly damaging, because of how distasteful grading can be to complete. Finally, and most crucially, because the system require that *every* concern be explained before point values are effected, it is possible that grading time is increased; in our experience, the decrease in grading time due to lack of repetition far outweighs it.

Chapter 7

Verifications

Student learning relies very heavily on timely and reliable feedback in introductory proof courses, because students have a limited number of chances to improve their content understanding and writing style. The current method we use to provide this feedback–having a teaching assistant or instructor grade all the assignments manually–is fundamentally unscalable. Since this method of providing feedback is already problematic for large classes (with enrollments of more than 100 students), there is no hope of scaling it a MOOC (Massively Open Online Course) with 10,000 or more students.

Furthermore, writing proofs intimidates students, because it requires them to not only understand content but to also write in a clear, deliberate, and precise way. As a result, students often have tremendous difficulty with the first course(s) that requires them to write proofs.

We aim to alleviate both of these problems by introducing a *peer evaluation* system which helps students learn and also helps teaching assistants and graders to grade more quickly and accurately. We approach this problem as one of *human computation*, where we break up the task into small pieces that non-experts (the students) are capable of doing.

Our approach attempts to simultaneously give students more opportunities to learn and give graders some assistance. In order to help the students learn more deeply, we ask them to do *peer evaluations* of several proofs on most homework assignments. We aggregate and combine these peer evaluations together to help the grader provide fair assessment and detailed feedback while expending less effort and time than normal.

Though peer evaluation systems are fairly common, grading proofs presents a novel challenge, because they are simultaneously *fuzzy* and *objective*. Because mistakes in the proof-domain are almost entirely objective in nature, it is possible to disassociate the *scoring* of submissions from the *attributing of mistakes* to submissions. Another challenge is that even individual errors in proofs can be complex and difficult to describe and understand.

So, our system must guide non-experts (or *unqualified graders*, in the terminology from Chapter 6) through the evaluation of the proofs if graders are to get any useful information from it. We approach the problem as one of *human computation*, in which the understanding of non-experts is collected and combined to produce a correct and useful solution.

Our system has the potential to increase student learning in several unique ways because of our approach. Typically, proofs courses include some form of proof review, but these questions tend to be infrequent and contrived. By asking students to regularly do proof evaluations, we hone their proof review skill on examples that are often close to the proofs they might write [Bal12]. Training the students to be good proof reviewers is often a meta-goal of introductory proofs courses.

When using our system to evaluate proofs, students must examine and select from a list of possible mistakes. We believe that by studying this list, they will better understand the mistakes they could make themselves. This understanding can also generalize to the mistakes they might make on future problems.

Finally, by giving students more examples of proofs to evaluate, we give them more opportunities to see proofs throughout the course. By reviewing several submissions for the same question, students get a chance to try to understand several different approaches beyond the one they used. This can help students learn more tricks and patterns relevant to each proof topic. In a similar vein, by showing students proofs written by several authors, we give them the experience of being the audience for whom they are writing. This can help them adopt good stylistic features which they see and stop using styles that make the proof harder to read.

We have implemented a prototype of our system and begun testing it in several math and computer science courses at Carnegie Mellon University. Through the prototype, we have seen that the student reviews very often provide the grader the entire list of mistakes necessary to grade the proof. Additionally, we ran a small-scale controlled experiment to test if using our approach resulted in students improving their skills at reviewing and/or writing proofs in comparison to just solving more proof questions. We found that our approach does not increase students' reviewing skill but *does* increase their problem solving/writing skill.

7.1 Related Work

There is a significant body of related work on theorem provers and peer review, but this work tends to explore different types of proofs or topic domains.

7.1.1 Theorem Provers

There are several existing environments that facilitate writing formal ([SS94]) and machine-checkable (coq, agda) proofs, and systems such as Sieg's ProofTutor have been successfully used in formal logic courses. One possible solution to our research problem might be to use one of these well-established systems in the classroom. Unfortunately, writing detailed axiomatic proofs is tedious and does not transfer well to writing human-readable proofs.

7.1.2 Peer Grading of Essays

Prior peer review systems have been developed in other domains. PeerScholar [PJ08] has explored some related problems, but their results are all in the domain of scientific essays in which students are graded on holistic content and grammatical features of their writing. Our domain of proof differs from the essay domain that PeerScholar targets because of their large objective component. There are only a limited number of mistakes students make for any particular proof topic. In comparison, there are infinitely many grammatical mistakes that students can make, and because of all the irregularities in natural language, naturally occurring examples that can help other students fix their mistakes are rare.

7.1.3 Peer Grading of Code

Other work [Tan11, Con11] has been done to analyze peer grading of code style. Although code style review is much closer to proof review, it still has several important differences. Most of the style features which peer reviewers attempt to find can be easily described and tend to be the same or very similar over all programs. Conversely, the errors which students make in proofs are often specialized to topic (and a few are even further specialized to problem-specific).

Furthermore, grading coding style does not require domain-expertise. The features typically assessed ("the code has mixed tabs and spaces") are low-level, and reviewers do not need to understand the code to evaluate if they apply or not. In contrast, proofs can be arbitrarily complicated, there can be multiple solutions to a single problem, and there is more than one correct way to describe results.

7.2 Our Approach

Our approach uses peer grading as a form of *human computation*. Intuitively, although the students in the course are only just learning how to write proofs, we scaffold the peer grading in such a way that they can contribute useful work and learn as well.

We leverage the fact that Judgement-based Grading (Chapter 6) is already friendly to unqualified graders by augmenting it with an explicit division of labor between qualified graders and the students, and a process that is even more guided for students. In particular, to help students review proofs, the system provides them with a list of *judgements* (correct or incorrect attributes of a submission) entered by a qualified grader in the grading pool prior to the peer reviewing. To determine what the judgements should be, the qualified grader reviews ten student submissions, grading them normally. In our experience, the generated list contains nearly all of the errors a student could make–and it is easily augmented later if students discover missing judgements. We have found over the course of using our prototype system that the average number of distinct errors for any given question is approximately 25. This means that in most cases, the number of judgements which students must understand to helpfully evaluate proofs is relatively small.

We decompose the problem of reviewing proofs mainly by separating point allocation and mistake attribution (Judgement-based Grading). To accommodate this, our approach uses a modified form of rubric grading that can be used to facilitate both normal grading and our version of peer evaluation.

We call the process of reviewing a single proof a *verification*. Adding student verification to an assignment necessitates several changes to the normal homework assignment process.

When using verifications, students submit their homework using our website. If at all possible, students typeset their homework using a system like LaTeX, but we deployed a system to scan in student submissions in batch form to remove this restriction last semester. When the deadline for the homework occurs, the graders for the verified questions grade 10-15 submissions to populate the judgements. In practice, this small number of submissions generates the majority of judgements the question will need. If the question was used previously or there is another source of judgements, they are used either in addition to or in place of this step. When this step is finished, the course staff releases the verification assignment to the students.

Students are given a quota of verifications to complete. In general, they receive no more than two different questions with between three and four verifications for each. Once the verification due date arrives, a grader for a verification problem gets aggregate information from all the students who looked at each proof. Our interface collates this information for them to make it more useful. In practice, all of this extra contextual information speeds up the grading process significantly. Our system automatically provides feedback to the students about the accuracy of their grading. If the grader would like to change the grades the students receive for their accuracy, the interface provides a way to do this.

7.3 Research Questions

We are interested in the educational effects of introducing our peer evaluation system into introductory proofs classes. In particular, the research questions we would like to answer are:

- Does using verifications reduce the amount of effort and time graders spend on the proofs?
- Will reviewing other students' proofs (both correct and incorrect) [ZZ11, TMM$^+$09, GR07] help students learn how to write proofs more quickly and correctly?
- Is the ability to review proofs correlated to skill at writing them?
- Can reviewing other students' proofs give some benefits which just writing proofs alone cannot?

7.4 Preliminary Results

7.4.1 Grading Effort

We have deployed our system in several courses at Carnegie Mellon as case studies, but, here, we present the results from the most recent course, Mathematical Foundations for Computer Scientists (15-151). Based on several of the questions students were asked to verify in 15-151, we found that asking students to complete between three and five verifications provided graders with a superset of the actual judgement set 83% of the time. This means that a grader almost always only has to look at around 10 judgements instead more than 20. In practice, almost all of the teaching assistants who used verifications told us that they preferred to grade a question where students had done verifications to grading a normal question and that it was significantly faster.

7.4.2 Student Ability

To determine if verifications effect students' ability to *write proofs* or *do verifications*, we ran a controlled experiment.

Participants

Twenty-seven undergraduate students ages 18-24 years old participated in this study. All students were currently enrolled in an introductory discrete mathematics course and had not completed any verifications previously. Participants were given $40 as compensation for their involvement.

Procedure

This study was approved by the institutional review board at Carnegie Mellon University. All participants gave informed consent. Before they began they were randomly put in the control group ("do more proofs") or the experimental group ("do more verifications").

The main procedure of the experiment was as follows:

(1) All participants completed a pre-test in which they reviewed one Mathematical Induction proof and answered one proof-based induction question. Participants were given 20 minutes to complete this pre-test.

(2) **Control Group.** Participants were given 30 minutes to solve proof-based induction questions. We prepared three questions which was more than any participant finished.

Experimental Group. Participants were given 30 minutes to review as many proofs as possible. We prepared seven proofs to be reviewed which was more than any participant finished.

Both Groups. In addition to the proofs to review or questions to solve, both groups were given a reference solution to the induction proof they solved on the pre-test.

(3) All participants were given a 10 minute break while the experiment proctor (who was a qualified instructor or teaching assistant of an introductory proofs course) graded (1) the question participants solved on the pre-test and (2) the practice the participant did.

(4) All participants were given 5 minutes to review the graded proofs and reviews.

(5) All participants completed a post-test in which they reviewed one Mathematical Induction proof and answered one proof-based induction question. Participants were given 20 minutes for to complete this post-test.

Materials

We generated all study materials from questions given in previous instances of Carnegie Mellon introductory proofs courses. Before running the study, we asked several prior teaching assistants to do each of the questions to review ambiguities and help us generate time periods.

The list of judgements for induction proofs was generated by teaching assistants using our verification system in previous semesters. The list used to grade student proofs was identical to the one used in the pre-test and post-test in the verification.

The pre-test and post-test can be found in Appendix B.

Results

We aimed to measure two distinct outcomes: (1) accuracy of verifying, and (2) correctness of induction hypotheses. For both outcomes, we measured individual (post-test) - (pre-test) scores.

To measure accuracy of verifying, we computed a "verification score" for each subject for both their pre-test and post-test by counting the number of false positives and false negatives. We found no statistical difference between the experimental and control groups in their verification difference score.

To measure improvement on induction hypothesis details, we computed an "IH correctness score" for each subject for both their pre-test and post-test by counting the number of false positives and false negatives *only including items that were about correctness of the induction hypothesis.* Table 1 shows the means and standard deviations for both control and experimental groups for the difference in post-test and pre-test IH correctness scores. Note that these scores represent "how many fewer IH mistakes" a subject made; so, higher is better. We found there *was* a statistical difference ($d = 0.39, p < 0.029$) between the control and experimental groups. We removed two participants (one from the control group and one from the experimental group) from this analysis, because they used a non-strong induction hypothesis in their post-test.

Intervention	M	SD
Proofs (control)	1.0	2.0
Verifications (experimental)	2.0	3.0

Table 7.1: Mean and Standard Deviation for IH correctness difference for control and experimental groups

Discussion

We discuss this more in future work, but a likely reason students do not improve on verification accuracy is that their attention was split between a long list of judgements and a long proof. This results in students being unable to apply what they know even if they understand the incorrect points.

The significant effect on student performance suggests that verifications do a better job at helping students to formulate their induction hypotheses than solving more proofs questions does. We believe this result is related to the "worked examples" phenomenon that has been heavily studied in several content areas, and adds to the literature on faulty worked examples [TMM+09].

7.5 Future Work

We created the current version of our system as a prototype to evaluate if our approach was feasible. Our initial results have been positive, and we have already begun the next step of creating a more complete system with additional features we have found necessary.

Our results thus far indicate that asking students to review a several page proof and select judgements from a pool of 20-30 overwhelms them and detracts from the effect verifications could have. As an attempt to fix this problem, we are deploying a new, more decomposed version of our verifications this fall. We split the reviewing of each individual submission into three stages completed by multiple people. In addition to our previous decomposition, we split up errors into their *locations* and *descriptions*.

The first stage of our new system is called Location Finding. In this task, a student reviews a single proof and selects sections of text where he or she believes the submitter made an error and provides a sentence explaining the error in their own words. Additionally, the student can describe "global" errors in the proof that do not occur in a single location. Students are encouraged to select any location they believe might represent an error so as to get a superset of the errors of the proof. Since multiple students find the location errors for each proof, the locations with true errors are likely to be chosen many times.

The second stage is called Judgement Matching. In this task, a student reviews a series of locations (which were found by somebody else in the first stage) where the submitter likely made an error and he or she either claims there is no error or matches it to one of the pre-populated judgements. This task helps standardize the feedback so the grader can actually use it. Additionally, it forces students to think about what the judgements actually mean.

The third stage is called Judgement Refinement. In this task, a student receives several pairs of locations and judgements (generated by stages one and two) and provides an alternate wording of the judgement.

We have found that many of the possible benefits of verifications only manifest themselves far in the future. This makes it difficult to motivate verifications for students. Along with deploying the new decomposition, in the next version of verifications, we will ask the students to do a review of *their own proof* after they review others. Furthermore, they will receive back some percentage of the points they lost for finding their own errors. We believe this will be enough of an incentive to get the students to try hard on their verifications.

Once the base system consistently works well, there are many enhancements to refine it. Ideally, we would not need the grader to aggregate the feedback from the students; perhaps, with a large enough data set, we can use machine learning to aggregate the feedback automatically. The verification experience would be enhanced if we used a better distribution method that matched tasks to students based on accuracy and/or the perceived learning benefit they might get out of it. Another helpful improvement might be to tell students who are doing verifications which judgements are likely to apply to the proof. We might be able to accomplish this, again, using machine learning.

Verifications are intended to help deal with scale in grading. A clear application of this technology is to MOOCs where the student-to-teacher ratio is absurdly high. A reasonably accurate version of our system could provide in depth feedback to students at little cost to instructors.

Finally, we hope to generalize our technique to other domains that are not proof-based. In theory, many of the same pieces would work in any domain with objective truths in the submissions. We hope to eventually explore other domains like code and essays and see if our approach adds anything to the existing techniques in those domains.

Part IV

Conclusion

Chapter 8

Conclusion

Designing, teaching, and grading an introductory mathematical foundations for computer science course is a tall order. Existing implementations vary drastically and have a wide variety of success. We designed "Mathematical Foundations for Computer Science", a new course in Carnegie Mellon's School of Computer Science intended to try new pedagogical strategies and focus on mathematics as a tool for computer science students–who are often hostile to it. We taught this course twice, tweaking our strategies somewhat the second time. We believe parts of the construction and implementation of this course could be useful to instructors interested in creating a similar course sequence elsewhere.

At the beginning of this thesis, we claimed that any introductory mathematical foundations for computer science course should solve several difficult problems. This thesis provides information and strategies necessary to attack all avenues of these problems.

References for Math for CS Courses Designers

When *designing a course*, the references we provide in Chapter 2 of domain clusters, student attitudes and preconceptions, meta-content, and sample course sequences at a variety of universities should help instructors cover all essential topics without overwhelming students, and address all important "meta issues" (with respect to student attitudes and prior knowledge). We discuss a variety of interesting, motivational applications, and a novel assignment (Chapter 5) in great depth.

Novel Pedagogy and Systems for Math for CS Courses

When *teaching* these courses, we aim for deep learning, and we attempt to set up a foundation for later courses. We discussed some interesting pedagogy we used in our implementation, and possible fixes and extensions thereof. The novel programming language, {Setʇy}, we developed to help students learn the "language of mathematics" and the semantics of set theory and logic, leveraging their existing comfort with programming, contributes to deep learning as well. {Setʇy} also provides an environment to allow students multiple attempts at assessments on these topics–converting them from summative to formative.

Judgement-based Grading provides a grading system and implementation that forces better quality and useful feedback while decreasing effort and time necessary to provide them. Additionally, we believe the structure and quality of the feedback makes students more likely to take advantage of it and use it to learn. Finally, verifications help students get better at writing proofs, as well as getting them more involved in the grading process.

Future Work

The solutions we've developed are varied and diverse, and many of them need work to be usable at a larger scale. However, while not completely mature, we believe our solutions *can* be extended and modified to solve the fundamental problems in teaching an introductory mathematics for computer science course.

Appendices

Appendix A

Formal Sub{Setʇy} Semantics

A.1 Static Semantics

We omit the static semantics of Sub{Setʇy}, but they consist primarily of the judgement type: "m well-formed", which checks that every path of every command ends in a `ret`.

A.2 Dynamic Semantics

Program states in Sub{Setʇy} must keep track of the following:
(1) The set of defined *function* identifiers and their current values
(2) The set of defined *value* identifiers and their current values
(3) A "call stack" of executing functions

Let **FunIde** be the set of valid function identifiers in Sub{Setʇy}, and **VarIde** be the set of valid `nat` and `bool` identifiers in Sub{Setʇy}. Formally, we define these concepts inductively, as follows.

$$\begin{array}{rcll}
\mathsf{ProgramState} & P & ::= & (F, K) \\
\mathsf{CallStack} & K & ::= & \varnothing \quad | \quad (V, K) \\
\mathsf{VarMapping} & V & ::= & \mathbf{VarIde} \rightharpoonup_{\text{fin}} \mathbb{N} \\
\mathsf{FunMapping} & F & ::= & \mathbf{FunIde} \rightharpoonup_{\text{fin}} (\mathbf{Cmd} \times \mathbf{VarIde})
\end{array}$$

To define the dynamic semantics of Sub{Setʇy}, we inductively define the following judgements:

$e \ \mathsf{val}$	"e is a value$''$
$F, K : \ e \ \mapsto \ e'$	"e steps to e'"
$F : \ m \mid K \ \mapsto \ m' \mid K'$	"m steps to m', updating K to K'"
$P \mid F, K \ \mapsto \ P' \mid F', K'$	"P steps to P', updating F to F' and K to K'"

A.2.1 Stepping of Expressions

Primitives are already values, and variables step to their current value:

$$\frac{}{\mathsf{T}\ \mathsf{val}} \qquad \frac{}{\mathsf{F}\ \mathsf{val}} \qquad \frac{}{\mathtt{nat}[n]\ \mathsf{val}} \qquad \frac{V(x) = n}{F, (V, K) :\ x\ \mapsto\ n}$$

To evaluate equality (and less than), first evaluate the left operand, then the second, then compare them:

$$\frac{F, K :\ e_1\ \mapsto\ e_1'}{F, K :\ \mathsf{eq}(e_1; e_2)\ \mapsto\ \mathsf{eq}(e_1'; e_2)} \qquad \frac{b_1\ \mathsf{val} \qquad F, K :\ e_2\ \mapsto\ e_2'}{F, K :\ \mathsf{eq}(b_1; e_2)\ \mapsto\ \mathsf{eq}(e_1; e_2')}$$

$$\frac{b_1\ \mathsf{val} \qquad b_2\ \mathsf{val}}{F, K :\ \mathsf{eq}(b_1; b_2)\ \mapsto\ b_1 = b_2}$$

We omit `lt`, as it is nearly identical to `eq`. To evaluate conjunction (and disjunction), short circuit if possible, or evaluate if necessary:

$$\frac{F, K :\ b_1\ \mapsto\ b_1'}{F, K :\ \mathsf{and}(b_1; b_2)\ \mapsto\ \mathsf{and}(b_1'; b_2)} \qquad \frac{b_1\ \mathsf{val} \qquad b_1 = \mathsf{F}}{F, K :\ \mathsf{and}(b_1; b_2)\ \mapsto\ \mathsf{F}}$$

$$\frac{b_1\ \mathsf{val} \qquad b_1 = \mathsf{T} \qquad b_2 \wr S_1 \mapsto b_2' \wr S_2}{F, K :\ \mathsf{and}(b_1; b_2)\ \mapsto\ \mathsf{and}(b_1; b_2')} \qquad \frac{b_1\ \mathsf{val} \qquad b_2\ \mathsf{val}}{F, K :\ \mathsf{and}(b_1; b_2)\ \mapsto\ b_2}$$

We omit `or`, as it is very similar to `and`. Negation is similarly simple:

$$\frac{F, K :\ b\ \mapsto\ b'}{F, K :\ \mathtt{not}(b)\ \mapsto\ \mathtt{not}(b')} \qquad \frac{b\ \mathsf{val}}{F, K :\ \mathtt{not}(b)\ \mapsto\ \neg b}$$

To evaluate addition, we evaluate the first argument, then the second argument, then add them:

$$\frac{F, K :\ e_1\ \mapsto\ e_1'}{F, K :\ \mathtt{plus}(e_1; e_2)\ \mapsto\ \mathtt{plus}(e_1'; e_2)} \qquad \frac{e_1\ \mathsf{val} \qquad F, K :\ e_2\ \mapsto\ e_2'}{F, K :\ \mathtt{plus}(e_1; e_2)\ \mapsto\ \mathtt{plus}(e_1; e_2')}$$

$$\frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val} \qquad e_1 + e_2 = n}{F, K :\ \mathtt{plus}(e_1; e_2)\ \mapsto\ n}$$

We omit `times`, `divide`, `minus`, and `power` as they are nearly identical to `plus`.

A.2.2 Stepping of Commands

Sequencing of commands is stepped from left to right. We step the left command until it is a terminal. Then, if it is a `ret`, we are done; otherwise, we step the right command until it is a terminal.

$$\frac{F:\ m_1 \mid K \ \mapsto\ m_1' \mid K'}{F:\ \texttt{seq}(m_1; m_2) \mid K \ \mapsto\ \texttt{seq}(m_1'; m_2) \mid K'} \qquad \frac{F:\ m_2 \mid K \ \mapsto\ m_2' \mid K'}{F:\ \texttt{seq}(\varepsilon; m_2) \mid K \ \mapsto\ \texttt{seq}(\varepsilon; m_2') \mid K'}$$

$$\frac{}{F:\ \texttt{seq}(\texttt{ret}(e); m_2) \mid K \ \mapsto\ \texttt{ret}(e) \mid K'} \qquad \frac{}{F:\ \texttt{seq}(\varepsilon; \texttt{ret}(e)) \mid K \ \mapsto\ \texttt{ret}(e) \mid K'}$$

To step assignment, we first evaluate the expression to be assigned, and then *replace* the value of the assignable with it.

$$\frac{F, K:\ e \ \mapsto\ e'}{F:\ \texttt{assn}(x; e) \mid K \ \mapsto\ \texttt{assn}(x; e') \mid K}$$

$$\frac{e \ \mathsf{val}}{F:\ \texttt{assn}(x; e) \mid (V, K) \ \mapsto\ \varepsilon \mid (V \setminus \{(x, \_)\} \cup \{(x, e)\}, K)}$$

To step a conditional, we evaluate the condition, and step to the corresponding command.

$$\frac{F, K:\ b \ \mapsto\ b'}{F:\ \texttt{if}(b; m_1; m_2) \mid K \ \mapsto\ \texttt{if}(b'; m_1; m_2) \mid K}$$

$$\frac{b \ \mathsf{val} \qquad b = \mathsf{T}}{F:\ \texttt{if}(b; m_1; m_2) \mid K \ \mapsto\ m_1 \mid K} \qquad \frac{b \ \mathsf{val} \qquad b = \mathsf{F}}{F:\ \texttt{if}(b; m_1; m_2) \mid K \ \mapsto\ m_2 \mid K}$$

To step a for loop, we evaluate the upper bound, set the iterator to 0 in the state, and repeatedly sequence the body of the for loop until the iterator hits the upper bound.

$$\frac{F, K:\ e \ \mapsto\ e'}{F:\ \texttt{for}(x; n; e; m) \mid K \ \mapsto\ \texttt{for}(x; n; e'; m) \mid K}$$

$$\frac{e \ \mathsf{val}}{F:\ \texttt{for}(x; 0; e; m) \mid (V, K) \ \mapsto\ \texttt{for}(x; 1; e; m) \mid ((V \setminus \{(x, \_)\} \cup \{(x, 0)\}, K)}$$

$$\frac{e \ \mathsf{val} \qquad i < e}{F:\ \texttt{for}(x; 0; e; m) \mid (V, K) \ \mapsto\ \texttt{for}(x; 1; e; m) \mid (V \setminus \{(x, \_)\} \cup \{(x, i+1)\}, K)}$$

$$\frac{e \ \mathsf{val} \qquad i = e}{F:\ \texttt{for}(x; i; e; m) \mid K \ \mapsto\ \varepsilon \mid K}$$

A.2.3 Stepping of Programs

Defining a function adds the necessary information to the state and steps to the remaining program.

$$\frac{(X, (\_, \_)) \notin F}{\texttt{define}(X; x; m; P) \mid F, K \ \mapsto\ P \mid F \cup \{(X, (m, x))\}, K}$$

To evaluate a function call, we first fully evaluate the argument, then we get the command from our state, add a frame to the call stack, and evaluate to the command stored for that function.

$$\frac{(X, (\_, \_)) \in F \qquad F, K : e \mapsto e'}{F, K : \texttt{call}(X; e) \mapsto \texttt{call}(X; e')}$$

$$\frac{(X, (m, x)) \in F \qquad e \ \mathsf{val}}{\texttt{call}(X; e) \mid F, K \mapsto \texttt{eval}(X; m) \mid F, (X, \{(x, e\}), K)}$$

$$\frac{F : m \mid K \mapsto m' \mid K'}{\texttt{eval}(X; m) \mid F, K \mapsto \texttt{eval}(X; m) \mid F, K'}$$

Finally, we must consider `ret`. We only step `ret` when we are completely done evaluating a function. If this is the case, we pop the top element of the call-stack when we return.

$$\frac{F, K : e \mapsto e'}{F : \texttt{ret}(e) \mid K \mapsto \texttt{ret}(e') \mid K} \qquad \frac{e \ \mathsf{val}}{\texttt{eval}(\texttt{ret}(e)) \mid F, ((X, V), K) \mapsto e \mid F, K}$$

Appendix B

Verification Study: Pre-test and Post-test

Pre-Test

Participant ID:

INSTRUCTIONS:

- There are two sections on this Pre-Test.
- The first section will ask you to find errors in Mathematical Proofs.
- The second section will ask you to write a proof using Mathematical Induction.
- Please read all of the instructions carefully and write down all of your work.

Part I: Reviewing Mathematical Proofs

In this part, you will review a proof. Assess the proof by first reviewing the base case(s) and reviewing any options related to them. Once you've done the base cases, move on the induction hypothesis, and then finally move on to the induction step. As you select things wrong with the proof, you should assume that all errors you've selected judgements for are *corrected* for the remainder of the proof.

1. Induction

Consider the following problem:

Prove $\exists(c > 0).\ \exists(n_0 \geq 0).\ \forall(n \geq n_0).\ A(n) \leq c2^n$, where $A(n)$ is defined as follows:

$$A(0) = 0$$
$$A(1) = 0$$
$$A(n) = 1 + A(n-1) + A(n-2) \text{ for } n \geq 2$$

Consider the following proof:

Let $f(n) = A(n)$. We want to prove $\exists(c > 0)\ \exists(n_0 \geq 0).\ \forall(n \geq n_0).\ f(n) \leq c2^n$.
Let $c = 1$, $n_0 = 2$. We go by induction on n.

Base Case $(n = 2)$.

$$f(2) = 1 \text{ because } f(0) = 0 = f(1)$$
$$2^2 = 4$$
$$f(2) \leq 2^2$$

Induction Hypothesis. Suppose $f(k) \leq 2^k$ is true for some $k \in \mathbb{N}$, $k \geq n_0$, $k \leq n-1$.

Induction Step. Note that $f(n) = 1 + f(n-1) + f(n-2)$, by our recurrence. So, $f(n) \leq 1 + 2^{n-1} + 2^{n-2}$ by the IH. We see that $2^n = 2^{n-2} + 2^{n-2} + 2^{n-1}$ by algebra. Since $1 \leq 2^{n-2}$, $f(n) \leq 2^n$.

So, $\exists(c > 0)\ \exists(n_0 \geq 0).\ \forall(n \geq n_0).\ f(n) \leq c2^n$ is true.

The actual question is on the next page. Feel free to flip between this page and the next one while answering this question.

Directions:

- For each of the statements below, write T if it applies to the proof and F otherwise.
- For each of the statements below, indicate supporting locations in the proof by using the statement number.

---

1 ( ) Uses at least one variable without declaration or quantification

2 ( ) Quantifies at least one variable multiple times

3 ( ) Several consequtive steps of the proof are unrelated or unconnected.

4 ( ) Starts from the conclusion and arrives at something which is true without explicitly stating the work is reversible

5 ( ) Uses $P(n)$ or refers to "the claim" without defining it

6 ( ) Uses a function as a statement or the reverse (e.g. proves "$P(n) = n + 1$" by induction on n)

7 ( ) States "assume $P(k)$" without defining k (either before or after)

8 ( ) Too few (but at least one) base case(s) are given to establish the induction hypothesis.

9 ( ) The proof includes an unnecessary base case.

10 ( ) The induction hypothesis assumes the conclusion of the proof.

11 ( ) The induction hypothesis does not say "assume," "suppose," or something similar.

12 ( ) The induction hypothesis assumes $P(k) \implies P(k+1)$ instead of just $P(k)$.

13 ( ) The induction hypothesis assumes the statement holds for "some $k \leq L$, $k \geq 0$" instead of "all $k \leq L, k \geq 0$" where L is the variable inducted on in the strong induction hypothesis.

14 ( ) States in the IH "assume for some L that $P(k)$ is true for all $0 \leq k \leq L$", but then goes on in the Induction Step to show that $P(k+1)$ is true.

15 ( ) The induction step assumes the conclusion (independently of the induction hypothesis).

16 ( ) The proof does not explicitly invoke the induction hypothesis.

17 ( ) The induction step invokes the induction hypothesis where it does not apply.

Part II: Writing Mathematical Proofs

In this part, you will write a *full mathematical proof* in response to an induction question. Please make sure to show all of your work and clearly indicate which part(s) are the proof.

1. Induction
Define f_n and g_n as follows for $n \in \mathbb{N}$:

$$\begin{aligned} f_0 &= 1 & g_0 &= 1 \\ f_1 &= 5 & g_1 &= 5 \\ f_2 &= 10 & g_2 &= 10 \\ & & g_3 &= 0 \\ & & g_4 &= -40 \\ f_n &= 2f_{n-1} - 4f_{n-2} \text{ for } n \geq 3 & g_n &= 2g_{n-1} - 3g_{n-2} - 2g_{n-3} + 4g_{n-4} \text{ for } n \geq 5 \end{aligned}$$

Prove that $f_n = g_n$ for all $n \in \mathbb{N}$. You may assume that $f_n = g_n$ for $n \in \{0, 1, 2, 3\}$ without proof *for this proof only.*

Post-Test

Participant ID:

INSTRUCTIONS:

- There are two sections on this Post-Test.
- The first section will ask you to find errors in Mathematical Proofs.
- The second section will ask you to write a proof using Mathematical Induction.
- Please read all of the instructions carefully and write down all of your work.

Part I: Reviewing Mathematical Proofs

In this part, you will review a proof. Assess the proof by first reviewing the base case(s) and reviewing any options related to them. Once you've done the base cases, move on the induction hypothesis, and then finally move on to the induction step. As you select things wrong with the proof, you should assume that all errors you've selected judgements for are *corrected* for the remainder of the proof.

1. Induction

Consider the following problem:

Prove $\exists(c > 0).\ \exists(n_0 \geq 0).\ \forall(n \geq n_0).\ A(n) \leq c2^n$, where $A(n)$ is defined as follows:

$$\begin{aligned} &A(0) = 0 \\ &A(1) = 0 \\ &A(n) = 1 + A(n-1) + A(n-2) \text{ for } n \geq 2 \end{aligned}$$

Consider the following proof:

We want to prove $\exists(c > 0).\exists(n_0 \geq 0).\ \forall(n \geq n_0).\ A(n) \leq c2^n$.
Let c, n be arbitrary and fixed as $c = 1$, $n = 0$. We go by induction on n.

Base Case $(n = 0)$**.** $A(0) = 0 < 2^0 = 1$.

Induction Hypothesis. The claim is true for all k which are $\leq n$ but greater than n_0.

Induction Step. $A(k+1) = 1 + A(k) + A(k-1)$ by definition of A
$\iff A(k) \leq 2^k$ and $A(k-1) \leq 2^{k-1}$ by the IH.

It follows that $A(k+1) \leq 1 + 2^k + 2^{k-1} \leq 2^k + 2^k = 2^{k+1}$ by algebra, and because $1 \leq 2^{k-1}$.

Thus, since our base case and induction step hold, our claim is true for all $k \geq n_0$.

The actual question is on the next page. Feel free to flip between this page and the next one while answering this question.

Directions:

- For each of the statements below, write T if it applies to the proof and F otherwise.
- For each of the statements below, indicate supporting locations in the proof by using the statement number.

---

1 ( ) Uses at least one variable without declaration or quantification

2 ( ) Quantifies at least one variable multiple times

3 ( ) Several consequtive steps of the proof are unrelated or unconnected.

4 ( ) Starts from the conclusion and arrives at something which is true without explicitly stating the work is reversible

5 ( ) Uses $P(n)$ or refers to "the claim" without defining it.

6 ( ) Uses a function as a statement or the reverse (e.g. proves "$P(n) = n + 1$" by induction on n)

7 ( ) States "assume $P(k)$" without defining k (either before or after)

8 ( ) Too few (but at least one) base case(s) are given to establish the induction hypothesis.

9 ( ) The proof includes an unnecessary base case.

10 ( ) The induction hypothesis assumes the conclusion of the proof.

11 ( ) The induction hypothesis does not say "assume," "suppose," or something similar.

12 ( ) The induction hypothesis assumes $P(k) \implies P(k+1)$ instead of just $P(k)$.

13 ( ) The induction hypothesis assumes the statement holds for "some $k \leq L$, $k \geq 0$" instead of "all $k \leq L, k \geq 0$" where L is the variable inducted on in the strong induction hypothesis.

14 ( ) States in the IH "assume for some L that $P(k)$ is true for all $0 \leq k \leq L$", but then goes on in the Induction Step to show that $P(k+1)$ is true.

15 ( ) The induction step assumes the conclusion (independently of the induction hypothesis).

16 ( ) The proof does not explicitly invoke the induction hypothesis.

17 ( ) The induction step invokes the induction hypothesis in a part of the argument unrelated to the induction hypothesis.

Part II: Writing Mathematical Proofs

In this part, you will write a *full mathematical proof* in response to an induction question. Please make sure to show all of your work and clearly indicate which part(s) are the proof.

1. Induction
Prove that $n^n > 3 \cdot n!$ for all $n > 3$ by induction.

Bibliography

[ABD+10] SA Ambrose, MW Bridges, M DiPietro, Marsha C Lovett, and Marie K Norman. *How learning works: Seven research-based principles for smart teaching*. John Wiley & Sons, 2010. 2.1.1

[ACM09] J Aronson, Geoffrey Cohen, and W McColskey. Reducing Stereotype Threat in Classrooms: A Review of Social-Psychological Intervention Studies on Improving the Achievement of Black Students. Issues & Answers. *Regional Educational Laboratory . . .*, (076), 2009. 2.1.1

[AKMS98] A Arcavi, C Kessel, L Meira, and JP Smith. Teaching mathematical problem solving: An analysis of an emergent classroom community. *Research in collegiate mathematics education, . . .*, 1998. 2.1.5

[Bal12] Andreia Balan. *Assessment for learning: a case study in mathematics education*. PhD thesis, Malmö University, 2012. 7

[BN86] Ruth Butler and Mordecai Nisan. Effects of no feedback, task-related comments, and grades on intrinsic motivation and performance. *Journal of educational psychology*, 78(3):210–216, 1986. 2.1.3

[Bos97] Nigel Boston. A Use of Computers to Teach Group Theory and Introduce Students to Research. *Journal of Symbolic Computation*, 23(5-6):453–458, May 1997. 2.2.3

[BSS10] Randal E. Bryant, Klaus Sutner, and Mark J. Stehlik. Introductory computer science education at carnegie mellon university: A deans' perspective. Technical Report CMU-CS-10-140, Carnegie Mellon University, Computer Science Department, August 2010. 2.5.1

[CCD+10] Quintin Cutts, Emily Cutts, Stephen Draper, Patrick O'Donnell, and Peter Saffrey. Manipulating mindset to positively influence introductory programming performance. *Proceedings of the 41st ACM technical symposium on Computer science education - SIGCSE '10*, page 431, 2010. 2.1.1

[Con11] Mike Conley. *The Wisdom of Peers: A Motive for Exploring Peer Code Review in the Classroom*. PhD thesis, University of Toronto, 2011. 7.1.3

[Cou05] P. Cousot. Abstract interpretation. MIT course 16.399, `http://web.mit.edu/16.399/www/`, Feb.–May 2005. 5.4.2

[CP97] John Cannon and Catherine Playoust. Using the Magma Computer Algebra

System in Abstract Algebra Courses. *Journal of Symbolic Computation*, 23(5-6):459–484, May 1997. 2.2.3

[Dev01] Keith Devlin. Viewpoint: the real reason why software engineers need math. *Communications of the ACM*, 44(10):0–1, 2001. 1

[DKR99] E L Deci, R Koestner, and R M Ryan. A meta-analytic review of experiments examining the effects of extrinsic rewards on intrinsic motivation. *Psychological bulletin*, 125(6):627–68; discussion 692–700, November 1999. 2.1.3

[DKR01] E. L. Deci, R. Koestner, and R. M. Ryan. Extrinsic Rewards and Intrinsic Motivation in Education: Reconsidered Once Again. *Review of Educational Research*, 71(1):1–27, January 2001. 2.1.3

[Dwe08] Carol S Dweck. Mindsets and Math / Science Achievement. 2008. 2.1.1

[ECP99] Robert Eisenberger, Judy Cameron, and David W. Pierce. Effects of Reward on Intrinsic Motivation–Negative, Neutral, and Positive: Comment on Deci, Koestner, and Ryan, 1999. 2.1.3

[Fra06] DJ Frailey. What math is relevant for a CS or SE student?: an employer's perspective. *ACM SIGSOFT Software Engineering Notes*, 31(3):6–7, 2006. 1

[GR07] Cornelia S Große and Alexander Renkl. Finding and fixing errors in worked examples: Can this foster learning outcomes? *Learning and Instruction*, 17(6):612–634, 2007. 7.3

[Hen01] Peter B. Henderson. Mathematical Reasoning In Software Engineering Education. *ACM SIGCSE Bulletin*, 46(9):45–50, 2001. 1

[HM00] Dianne Hagan and Selby Markham. Does it help to have some programming experience before beginning a computing degree program? *ACM SIGCSE Bulletin*, pages 25–28, 2000. 2.5.3

[HT07] John Hattie and Helen Timperley. The power of feedback. *Review of educational research*, 77(1):81–112, 2007. 6.1, 6.2.2

[Kal97] Erich Kaltofen. Teaching computational abstract algebra. *Journal of Symbolic Computation*, 05363:503–515, 1997. 2.2.3

[Loc] Paul Lockhart. *A mathematician's lament.* 2.1.1

[LR14] Ed Lazowska and Eric Roberts. Tsunami or sea change? responding to the explosion of student interest in computer science. NCWIT 10th Anniversary Summit, May 2014. 1

[Mar10] Bill Marion. Computer algebra systems in discrete mathematics. *Journal of Computing Sciences in Colleges*, 26(2):139–148, 2010. 4.5.4

[Maz09] E. Mazur. Farewell, lecture? *Science*, 323:50–51, 2009. 3.2.1

[ML00] Barbara M Moskal and Jon A Leydens. Scoring rubric development: Validity and reliability. *Practical Assessment, Research & Evaluation*, 7(10):71–

81, 2000. 6.4.2

[MT08] Laurie Murphy and Lynda Thomas. Dangers of a fixed mindset: implications of self-theories research for computer science education. *ACM SIGCSE Bulletin*, pages 271–275, 2008. 2.1.1

[oCC13] ACM/IEEE-CS Joint Task Force on Computing Curricula. Computer science curricula 2013. Technical report, ACM Press and IEEE Computer Society Press, December 2013. 1

[Pie13] Benjamin Pierce. Software foundations, 2013. 2.1.4

[PJ08] Dwayne E Paré and Steve Joordens. Peering into large lectures: examining peer and expert mark agreement using peerscholar, an online peer assessment tool. *Journal of Computer Assisted Learning*, 24(6):526–540, 2008. 7.1.2

[Ral05] Anthony Ralston. Do we need ANY mathematics in computer science curricula? *ACM SIGCSE Bulletin*, 37(2):6, June 2005. 1

[RK89] KA Ross and DE Knuth. A programming and problem-solving seminar. 1989. 2.1.5

[She97] Gary J. Sherman. Trying to Do Group Theory with Undergraduates and Computers. *Journal of Symbolic Computation*, 23(5-6):577–587, May 1997. 2.2.3

[SHM08] Beth Simon, Brian Hanks, and Laurie Murphy. Saying isn't necessarily believing: influencing self-theories in computing. *Proceedings of the Fourth International Workshop on Computing Education Research*, pages 173–184, 2008. 2.1.1

[SS94] Richard Scheines and Wilfried Sieg. Computer environments for proof construction. *Interactive Learning Environments*, 4(2):159–169, 1994. 7.1.1

[Sut05] Klaus Sutner. Cdm: Teaching discrete mathematics to computer science majors. *J. Educ. Resour. Comput.*, 5(2), June 2005. 4.5.4

[Tan11] Mason Tang. Caesar: a social code review tool for programming education. Master's thesis, Massachusetts Institute of Technology, 2011. 7.1.3

[TMM$^+$09] Dimitra Tsovaltzi, Erica Melis, Bruce M Mclaren, Michael Dietrich, Georgi Goguadze, and Ann-Kristin Meyer. Erroneous examples: A preliminary investigation into learning benefits. In *Learning in the Synergy of Multiple Disciplines*, pages 688–693. Springer, 2009. 7.3, 7.4.2

[Wil09] By Daniel T Willingham. Is It True That Some People Just CanâĂŹt Do Math? *American Educator*, 33(4):14–20, 2009. 2.1.1

[WS01] BC Wilson and Sharon Shrock. Contributing to success in an introductory computer science course: a study of twelve factors. *ACM SIGCSE Bulletin*, pages 184–188, 2001. 2.5.3

[ZZ11] Jessica M Zerr and Ryan J Zerr. Learning from their mistakes: Using students' incorrect proofs as a pedagogical tool. *PRIMUS*, 21(6):530–544, 2011.

7.3