Computer-Based Testing using PrairieLearn in BJC

Bojin Yao
Dan Garcia, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-156
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-156.html

May 21, 2021

Computer-Based Testing using PrairieLearn in BJC

by

Bojin Yao

A technical report submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Teaching Professor Dan Garcia, Chair
Professor Armando Fox

Spring 2021

Computer-Based Testing using PrairieLearn in BJC

Computer-Based Testing using PrairieLearn

by Author Bojin (Max) Yao

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Dan Garcia
Research Advisor

(Date)

* * * * * * *

Professor Armando Fox
Second Reader

(Date)

Abstract

Computer-Based Testing using PrairieLearn in BJC

by

Bojin Yao

Master of Science in Computer Science

University of California, Berkeley

Teaching Professor Dan Garcia, Chair

Starting in the spring of 2019, Professor Dan Garcia turned the focus of his computer science education research and development group to proficiency-based learning for CS10: The Beauty and Joy of Computing (BJC) [1]. The long-term goal was for students to have the opportunity to achieve proficiency through formative assessments that they could practice over and over (and perhaps even summative assessments they could take, and continue to retake until they were above threshold), rather than only being able to showcase their abilities during a few high-stress, high-stakes exams. To achieve this goal, we looked to Question Generators (QG), implemented on the PrairieLearn (PL) platform [2] from the University of Illinois at Urbana Champaign. Question Generators (QG) are computer programs that randomly generate different variants of a question based on predefined parameters. According to [3], randomization is an effective tool to deter collaborative cheating for assessments. This became especially important for the remote-learning environment during the recent pandemic when all assessments had to be conducted online, making exam proctoring a challenging issue.

As one of the technical leads of the group, I was among the first students to dive deep into PL to set up foundational infrastructure for CS10. I assisted in creating and documenting many of the subsequent QGs, best practices, and tools; later, I also led the first contributions to PL's codebase and collaborated with PL's development team. One of the research contributions during that time was advocacy of a better way to formally categorize QGs, into conceptual and numerical variants [4].

As a result of this work, all of the lectures in CS10 had become video-based with randomly generated quizzes that students had access to the entire semester. This helped us to accommodate schedules of all students and free up the usual lecture times for office hours. The randomized quiz questions incentivized students to pay attention to the contents of the video lectures instead of getting answers from their classmates. The fully auto-graded quizzes also came at no additional cost to staff hours.

Additionally, for the first time ever, CS10 was able to introduce a long-exam format to alleviate exam pressure and accommodate the needs of all students. This was possible partly because the randomized questions made collaborative cheating more difficult, and we devised new strategies to detect cheating using comprehensive logs. Furthermore, the entire exams were auto-graded with scores and statistics released *minutes* after the exams' conclusion that saved many hours of handgrading or scanning the exam papers.

The purpose of this master's report is to document and share results from three aspects of my program working with PL: (1) software development, (2) CS10 curricular development using QGs, and (3) student mentorship. I will detail many of the things I've learned, best practices, challenges we faced while working on this project, and how we resolved these challenges. Additionally, I will also mention some important research work related to CS10's PL project that might be informative for others looking to transition to computer-based testing. It is hoped that future PL-course TAs, QG authors, PL developers, and others interested in computer-based testing will benefit from the information presented in this report.

Acknowledgments

This work would not have been possible without the support of many incredible and kind people in my academic life. I'd like to give special thanks to the following individuals:

- My research advisor and mentor, Professor Dan Garcia, for his passion and dedication to his mission to provide world class computer-science education to every student. It is without a doubt that his introductory class, CS10: Beauty and Joy of Computing, inspired me to pursue computer science as a career and completely changed the trajectory of my life. His continued support and influence throughout my academic career largely contribute to my achievements so far, and surely my future achievements as well.

- Professor Armando Fox, the second reader to my program report, for the countless opportunities, immeasurable amount of time, and invaluable feedback he had provided that made parts of this project possible.

- Matthew West, Nicolas Nytko, Mariana Silva and the rest of PrairieLearn (PL) developers from University of Illinois at Urbana-Champaign, Neal Terrell from California State University at Long Beach, and developers from other colleges and universities for their timely answers and assistance with our endless PL related questions and requests. Their advice and support from thousands of miles away made many things possible.

- Erik Rogers and Finsen Chiu for their time and help with deploying and managing ACE Lab's PrairieLearn instances, and always providing (often last-minute) technical assistance.

- My fellow graduate students Qitian Liao and Connor McMahon for their assistance and advice with many parts of the project.

- The entire CS10 staff, especially Yolanda Shen, Lam Pham, and Shannon Hearn, for supporting various tasks relating to my project.

- Members of the ACE Lab, especially Irene Ortega, for assisting with many of the tasks and discussions around PL and Question Generators.

- My family and friends who have been supporting me every step of the way in my pursuit of knowledge, it is thanks to them that all of this is possible.

Some components of this work were partially funded by the California Governor's Office of Planning and Research and the UC Berkeley College of Engineering.

Contents

List of Figures

[1] https://snap.berkeley.edu/snap/help/SnapManual.pdf

Listings

[2] `https://github.com/PrairieLearn/PrairieLearn/blob/master/exampleCourse/questions/element/multipleChoice/question.html`

[3] `https://github.com/PrairieLearn/PrairieLearn/blob/master/exampleCourse/questions/element/multipleChoice/serverFilesQuestion/sky-color.json`

Chapter 1

Introduction

It has always been frustrating when students did poorly in our non-majors introductory course, the Beauty and Joy of Computing (BJC), so we have been exploring ways to resolve it. We believe it is time to move from a "fixed time variable learning" (i.e., A-F grades in one semester) mode to a "fixed learning, variable time" environment (i.e., only A grades in which the students take as long as they want). There are a lot of pieces to put in place to get there, but technically we needed a system that would be able to let a student practice with questions on a particular topic over and over without needing a 1-on-1 tutor to generate those problems for them [5]. We adopted PrairieLearn to write our QGs in hopes of delivering various course content and assessments in the coming semesters.

Since spring 2019, Professor Dan Garcia's CS Education Research and Development Group has been studying and experimenting with Computer Based Testing (CBT) using Question Generators (QG), which are computer programs that randomly generate different variants of a question based on predefined parameters. PrairieLearn (PL) [2] is an open-source, online platform developed and maintained by University of Illinois Urbana-Champaign (UIUC). PL was designed for randomized assessments; these might consist of homework, exams, quizzes, etc., and are often different from class to class. Randomization in assessments is an effective way to deter collaborative cheating [3]; this became especially relevant when courses like CS10 had to transition to remote-learning during the pandemic, which made exam proctoring a challenging issue.

Besides cheating prevention, in CS10, we also believe that QGs are important tools to promote mastery of learning. Thanks to the ability to create different question variants, students have the opportunity to practice concepts until they have achieved understanding. Additionally, when reviewing old concepts, instructors and students will have a clearer scope of students' recall of concepts by assessing students with questions they haven't seen before. Combined, these features of QG can help students master learned materials as they progress through the course.

Over the summer of 2020, student researchers from prospective classes studied and experimented with PL to understand its functionalities to discover potential use cases for the respective classes. I was responsible for the group working for CS10, and during this time, we

identified various problems when transitioning CS10's contents to PL. Some of the problems lie with PL's implementation, others were unique to CS10. I later led the first contributions to PL's source code to help fix some of the issues related to implementation (see Chapter 4). Issues that were unique to CS10 were more difficult to resolve. For example, Snap*!* is a large part of CS10, but it lacked the APIs to be compatible with PL, which made authoring Snap*!* based QGs in PL particularly difficult and time-consuming (see Chapter 3). Fortunately, by the end of the summer, we had explored and decided on various ways PL should be used in CS10 and established many of the best practices that the course still used to this day (see Chapter 5).

When fall semester started, I took the responsibility to oversee the integration and realization of the team's work from summer, and more importantly, fill in any gaps required to deliver the seamless online experience to students. Notably, I assisted in creating many of the lecture "quiz" questions (we call them clicker questions) and ported them to PL as QGs; additionally, I wrote/debugged many of the exam questions as well. In the end, I designed and set up most of the PL infrastructure that the course continued to use in the spring 2021 semester, and likely beyond.

The fall semester also saw some major changes in course policy. We were aware of the incredible stress and anxiety students were facing during the global pandemic, and we wanted to do everything we could to alleviate some of it *without* sacrificing quality of learning. As a result, we *completely* changed the policy and delivery-format of *lecture contents* and *exams* to be as student-friendly as possible thanks to our work with QGs[1]. Section 5.3 covers more details relating to these two aspects of the course. In short, all lectures became video-based; each video had a corresponding quiz question generated by its QG that students were required to answer correctly to receive credit. The quizzes encouraged students to watch and pay attention to these videos and not simply get answers from their classmates. Furthermore, we made these quizzes due at the end of the semester, so students had complete flexibility when to watch the videos and complete the quizzes. We introduced a long-exam format to all students where we gave every student *days* for exams that were designed to take a couple hours. This was possible partly because of randomization with exam questions; if every student had the same exam, it would be too easy to cheat. The exams were also graded with scores and statistics published minutes after. Based on our survey results in Chapter 6, students really liked this new exam format, and we plan on continuing this policy for the future.

In the spring semester, I took up more mentorship and leadership roles relating to the future PL. I took charge of a small group of new members of the research team to learn and create QGs for CS10 and delivered a couple of amazing questions for the final. I also led a small team of students from CS169: Software Engineering, to create a new tool to configure PL assessments as part of their class project.

The purpose of this master's report is to document and share results from three aspects

[1]There were also various small changes relating to assignments and labs that are outside the scope of this report.

of my program working with PL: (1) software development, (2) CS10 curricular development using QGs, and (3) student mentorship. I will detail many of the things I've learned, best practices, challenges we faced while working on this project, and how we resolved these challenges. Additionally, I will also mention some important research work related to CS10's PL project that might be informative for others looking to transition to computer-based testing. It is hoped that future PL-course TAs, QG authors, PL developers, and others interested in computer-based testing will benefit from information presented in this report.

Chapter 2

Related Work

PrairieLearn's [2] core philosophy is based around the revolutionary idea proposed by Benjamin Bloom in 1968 "Learning For Mastery" [6], that most, if not all, students are fully capable of understanding and achieve a level of mastery of materials taught in school, especially when provided with a good tutor. In his original paper, Bloom also touched on many aspects of education relating to formative and summative assessments, and their role in feedback to both instructors and students of the student's level of mastery. Thanks to his ideas, there have been many automated, Intelligent Tutoring Systems (ITS) used in classrooms, that also includes programming classes [7, 8, 9, 10]. In fact, there are many different ITS for programming classes out there, each with different purposes and approaches; studies have been conducted to evaluate these systems' overall effectiveness [11, 12]. The actual number of different ITS being used in classrooms is likely much higher than what's been studied, considering not all organizations have published results of their own systems. For example, student developers within our own EECS Department have authored their own online assessment platform called examtool [13]. This assessment system is very new, and does not have any publications yet; however, it is already being used by huge classes with thousands of students.

The report documents some of CS10's experiences using PL, this builds on similar work from UIUC as well [14, 15]. Additionally, PL was used as the research tool for many studies on cheating in the online environment [16, 17, 18].

Chapter 3

Background

CS10: Beauty and Joy of Computing (BJC) is non-majors introductory Computer Science (CS) course at UC Berkeley and was one of the five national pilots for the College Board AP CS Principles course [19]. The class is designed for students who have little to no prior programming experience. By the end of the class, students learn all the fundamentals of CS Principles, and much more, including recursion, higher-order functions, and concurrency

Historically, CS10 relies heavily on Multiple Choice Questions (MCQ) for its assessment, such as lecture quiz questions and exams. It is well known that well-designed MCQs are good tools for measuring competencies [20, 21, 22], and CS10 uses MCQs in creative ways. The two most common formats of MCQs that CS10 uses are *Multiple Choice Single Select* where one option can be chosen from a list of options and *Multiple Choice Multiple Select* where multiple options can be selected.

The first QGs we wrote were based on MCQs and PL had built-in support for the two main types of MCQs we used; however, PL lacked features for a few common patterns in our MCQs. For example, in many of our MCQs, we presented "All of the above" and "None of the above" at the bottom of the options that would persist for all variants. Due to the shuffling feature of PL, keeping specific options at designated positions was not trivial to override. After communicating with PL's development team, we decided to contribute directly to their codebase to support this feature for all PL users. Later, we added a few other features that PL developers and us believed were useful for others too. Chapter 4 discusses these changes.

In CS10, students learn to program via a block-based programming language called Snap*!*, as shown in Figure 3.1 and Figure 3.2 [23], that allows users to drag and drop blocks of code rather than type with a keyboard.

After having learned the CS fundamentals like abstraction, recursion, and higher-order functions using Snap*!*, students learn Python and apply the same knowledge, while being introduced to a few new concepts, such as object-oriented programming, tree recursion, and data processing [22]. The fact that CS10 has a large component of Snap*!* programming turned out to be challenging when creating QGs for the class.

In a nutshell, PL QGs use results returned by Python's `random` functions as the random elements for a question. If the question needs a random number between `1` and `10`, it can

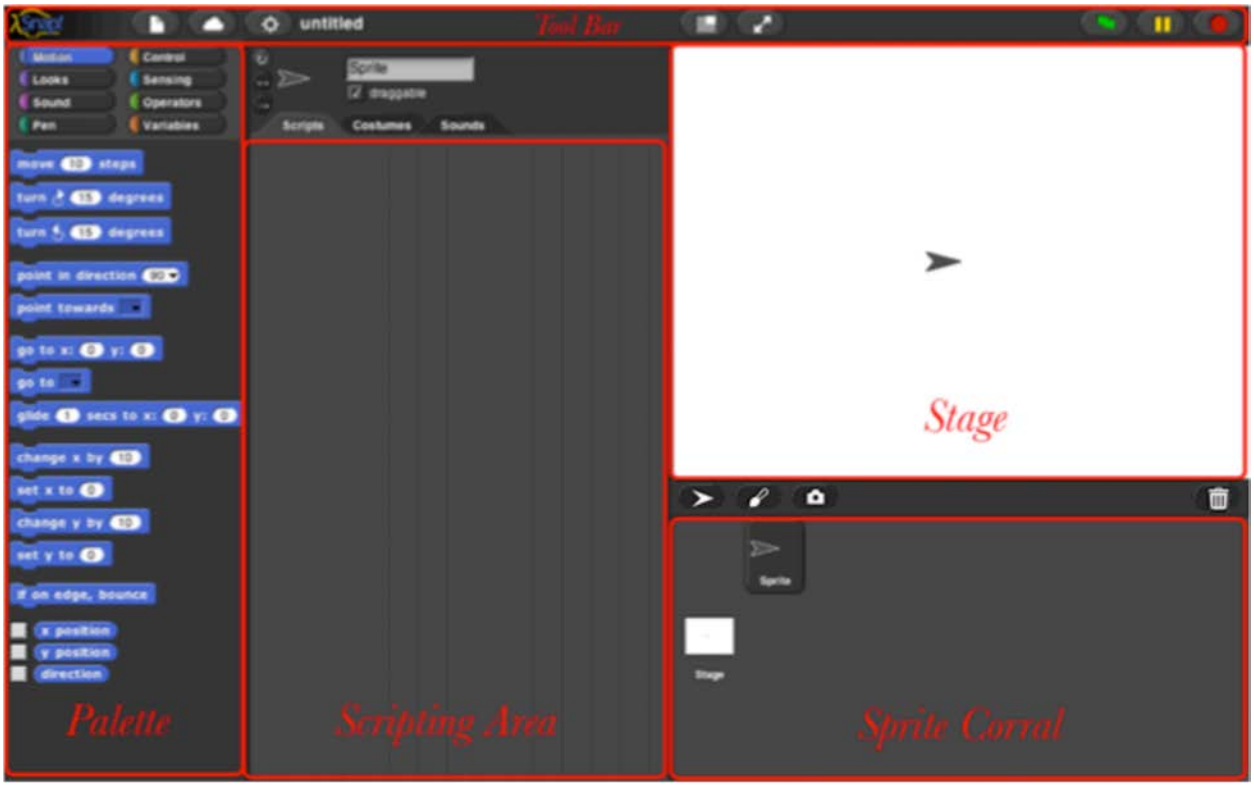

Figure 3.1: Snap*!* layout[1], blocks of various types and shapes inside the palette can be drag-and-dropped to the scripting area to form code-blocks. These code-blocks are run when clicked by a mouse. The blue blocks (shown above) are used to programmatically control sprites in the stage area.

Figure 3.2: Snap*!* blocks[1] snap together to form executable code-blocks. The yellow block shown above functions as the main function in some programming languages, and the green `pen down` block lets the sprite start drawing on stage.

Figure 3.3: The sprite will say "Hello!" for 1 second, 10 times.

be done with a simple Python call: `random.randint(1, 10)`, that's it! Similarly, to generate a random lowercase alphabet, a slightly complicated option might look something like: `chr( random.randint(0, 25) + ord('a'))`, and there are many different ways to do it using various packages. More concretely, let's say we have a Python-based multiple choice question that asks students to identify a list comprehension that would produce a list with numbers ranging from `x` to `y`, in order to vary the values for `x` and `y`, we can do it with two different calls to: `x = random.randint(1, 3); y = random.randint(5, 7)`, then the values of `x` and `y` can be displayed to students as part of the question's page. The variable `x` would be a number between `1` and `3`, whereas `y` would be a variable between `5` and `7`. This way, `x` would always be smaller than `y`, to avoid possible confusions. As pointed out in [24] "special considerations are necessary when computers are used for tests." Since PL can convert any Python string to HTML in its questions page, creating text-based randomization is relatively convenient. Once a question is created after all the randomizing function calls are completed, a new ***question variant*** (or simply, *variant*) is created. A ***variant*** is a question created with *any* combination of random parameters defined by its QG[2]. For the list-comprehension example above, the question would have at least $3 \times 3 = 9$ *unique* variants[3].

However, Snap*!* is not a text-based language, and at the time of this writing, it doesn't have any built-in support for generating random code blocks, nor well-documented APIs for connecting to its runtime environment. As a result, there doesn't exist an easy way to generate random code blocks in Snap*!*. In order to achieve a similar effect of having "different variations" of the same question, one would need to (i) manually create all the variations of the code blocks, (ii) export them as images, (iii) carefully name each one of them, then (iv) write QG code to randomly choose from a list of these images that are named with the same naming convention, that might still require retyping many of the image names. This process takes lots of human labor and is error-prone, worst of all, this way of "generating" question variants is not scalable in any way.

If we want to create a question that asks students how many times "Hello!" will be said

[2]There doesn't seem to be any formal definition of "variants" in literature; thus, this definition is derived from our own practice, but it should be consistent with the general understanding within the context of PL.

[3]The "uniqueness" refers to the appearance of a question from a student's point of view, not necessarily a unique combination of random parameters (see Section 5.2).

as defined by a loop, in Figure 3.3, "Hello!" will be said ten times. We would like to create variants based on the numerical ranges in the for-i block, so that different students may have a different correct response. In Python, this would be a fairly easy task, in fact, we can just use the double `random.randint()` call presented earlier in this section to represent the two variables `x` and `y`, as shown below.

```
def generate(data):
    x = random.randint(1, 3)
    y = random.randint(5, 7)
    data['params']['x'] = x
    data['params']['y'] = y
    data['correct_answers']['question'] = y - x
```

Listing 3.1: Python code to generate numerical ranges. This creates 9 unique variants.

In Snap*!*, we would have to manually create and save images for all these variants (in the earlier example, 9 different images), and that number can quickly increase for more complicated QGs. In reality, we might also vary the duration, so students would need to identify the correct time in addition to the number of times "Hello!" would be said. In CS10, we have had many QGs with dozens of different images, and a few ones with over a hundred. This shortcoming with integrating Snap*!* questions into PL wasted countless hours and hindered development of Snap*!* QGs; past and current members of the research group had dedicated a huge amount of time exploring different ways to resolve this problem.

Chapter 4

Development of PrairieLearn Elements

This chapter describes our contribution to PL's code base in order to transition CS10 to computer-based testing more easily.

In CS10, lots of questions are delivered in Multiple Choice Question (MCQ) formats, that includes Multiple Choice Single Select (MCSS) or Multiple Choice Multiple Select (MCMS). PL has built-in elements[1] that already handle core logic and some functionalities of these two types of multiple choice questions, called `pl-multiple-choice`[2] and `pl-checkbox`[3]. When the CS10 team started using these elements, we realized they were missing certain functionalities that we frequently use. This led to our first contributions to PL's source code. These contributions were all modifications to existing PL elements in the form of HTML attributes[4], and some had been made widely available to all users of PL. All the features mentioned in this section can be found either in PL's official documentation[25] or GitHub repository[26].

Throughout our modifications to PL's source code, we received lots of help and feedback from PL's developers at UIUC, especially Matthew West, the creator of PL, Nicolas Nytko and Mariana Silva, core maintainers of PL's repository. Additionally, Neal Terrell at CSU, Long Beach, provided valuable feedback and discussions on our implementations described below.

Note: throughout this chapter, "option" and "choice" are used interchangeably. In this context, they both refer to the possible statements students can see and choose from in a typical MCQ.

[1] Elements in PL are custom HTML tags for common logic already implemented for QG writers that allows one to write HTML code instead of Python: `https://prairielearn.readthedocs.io/en/latest/elements`

[2] `https://prairielearn.readthedocs.io/en/latest/elements/#pl-multiple-choice-element`

[3] `https://prairielearn.readthedocs.io/en/latest/elements/#pl-checkbox-element`

[4] `https://www.w3schools.com/html/html_attributes.asp`

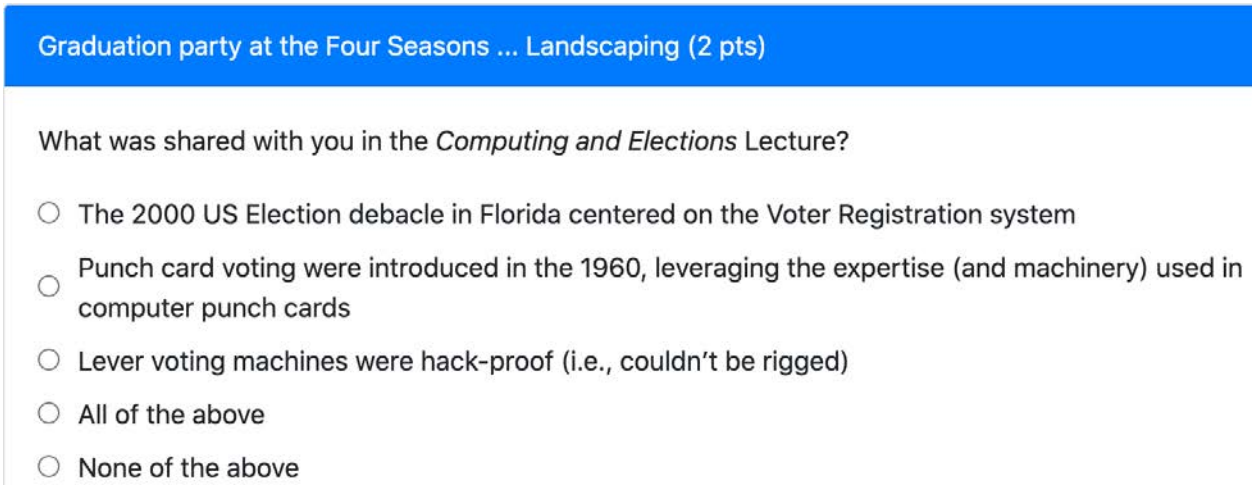

Figure 4.1: A question that has "All of the above" and "None of the above" enabled. "All of the above" is displayed below all regular choices, and "None of the above" is displayed at the very bottom. The question is configured to display 5 choices, and `pl-multiple-choice` randomly picks 3 from a list of choices supplied by a user.

4.1 Changes to `pl-multiple-choice` Element

All of the Above and *None of the Above*

The `pl-multiple-choice` element is used to create randomized MCSS questions that allow students to choose one option from a list of (often randomly selected) options. The original element already handled randomly choosing from a list of possible options, some simple configurations, grading, and rendering. However, the CS10 team quickly realized that we had lots of MCSS questions that contained "All of the above" and/or "None of the above" options, like Figure 4.1, and `pl-multiple-choice` didn't have built-in support for these options. Even though logic for these options can be implemented in individual QGs, the CS10 team believed that a native implementation of this feature would be valuable to other organizations using PL, I took upon the task to investigate and coordinate with PL developers at UIUC to implement these features.

Before starting development, I noted `pl-multiple-choice` had some built-in features already, namely `answers-name`, `weight`, `inline`, `number-answers`, `fixed-order`, and `hide-letter-keys`. If we wanted to have "All of the above" and/or "None of the above" as options, we would need to provide them as one of the possible options. Depending on the number of displayed options, PL would shuffle the order of them to present to students, so these two options might end up anywhere among the displayed options. If we wanted to have these two options at the bottom of all the options for better visibility (which is usually the case), we could configure `fixed-order` to `True` and have all options fixed in place; however, this

approach would lose the benefit of shuffling. Without shuffling of options, as far as MCQs are concerned, other platforms have similar features as PL, such as Gradescope and Canvas, making PL less distinctive for those who rely heavily on MCQs. According to PL developers at the time, to fix these two choices at the bottom of all options, and keep the shuffling feature of `pl-multiple-choice`, we would need to override the mechanism that generated new variants, involving the following steps:

1. Randomly pick from a pool of possible options.
2. Shuffle the order of picked options.
3. Append "All of the above" and/or "None of the above" to the shuffled list.
4. Declare the correct option.

However, this would not be correct because of the special semantic meanings of "All of the above" and "None of the above". If the correct choice were "All of the above", all other choices would have to be correct statements. If the correct choice were "None of the above", all other choices would have to be incorrect statements. As a result, the correctness of either choice would have an influence on the correctness of the rest of the choices. Additionally, when both "All of the above" and "None of the above" are enabled, "All of the above" should be displayed below the regular choices, and "None of the above" should be displayed below "All of the above" as the last choice for logical consistency. In summary, the right way to do this would look like:

1. Check if either "All of the above" or "None of the above" are enabled and do some sanity checks.
    - Toss a coin to determine if either of these two should be correct based on some probability.
    - Check the outcome of the toss, it could be either of them is correct, or neither is correct.
2. Depending on the outcome of the toss, adjust how many correct and incorrect options are chosen.
3. Make the selection, shuffle the order of the options.
4. Append "All of the above" and/or "None of the above" to the shuffled list, in that order.
5. Depending on the outcome of the toss, declare the correct option.

A nuanced detail with this feature is the probability that either "All of the above" or "None of the above" is correct in any given variant. When these two features were implemented, the priority was to make testing easy for QG authors, and since PL lacked the ability for a tester

to quickly go through all unique variants[5], the probability favored going through all correct statements when randomly generating variants. When using `pl-multiple-choice`, the user basically supplies a list of possible options marked as correct or incorrect; the element itself parses this list to generate a list of correct statements and a list of incorrect statements, so that the rest of the code can handle randomization. The current implementation hinges on the following formula:

$$\frac{1}{NumCorrect + AllOfTheAbove + NoneOfTheAbove}$$

It is worth noting that the value for $AllOfTheAbove$ and $NoneOfTheAbove$ would be 1 if enabled, else 0.

The formula basically treats "All of the above" and "None of the above" as *potentially* correct options when they're enabled, making them equally as likely to be chosen as other correct options. If `NumCorrect` is 0, the implementation defaults "None of the above" as correct for logical consistency. If `AllOfTheAbove` is enabled, the implementation also makes sure at least 2 correct options are provided because if only one correct option is provided, technically both that option and "All of the above" would be correct, causing confusion. Following this formula, all correct options have an equal chance to be picked as the right answer when a new variant is generated. The intention is to make sure testers can cycle through all variants where each correct option is chosen as correct to make testing faster. However, as Neal Terrell[6] later pointed out, it might be better in practice to make these two options equally as likely to be true as the number of displayed options, so the probability either option is true would be: $\frac{1}{NumAnswers}$, where $NumberAnswers$ is one of `pl-multiple-choice`'s attributes that configure the number of display options from the pool. In other words, if the question is configured to display 5 different options, then either "All of the above" or "None of the above" should be selected with probability $\frac{1}{5}$. This way, students would have no advantage or disadvantage to select any of the options; whereas in the current implementation, if the total number of correct options exceeds the number of displayed options, it'd be statistically advantageous not to select one of "All of the above" or "None of the above". I'm convinced that Neal is right and will be making adjustments to the current implementation to follow the probability model he proposed[7].

External JSON

The `pl-multiple-choice` element allows one to declare correct and incorrect options by wrapping each option inside a custom HTML element, `pl-answer`, and configure the statement as correct or incorrect. This is fine if only a few options are needed. However, in CS10, we have MCQ based QGs that have dozens of options, especially the lecture based questions, that would result in massive HTML files.

[5] See Chapter 8.

[6] `https://www.csulb.edu/college-of-engineering/neal-terrell`

[7] For safety and stability, PL maintainers refrain from making releases during regular semesters.

```
<pl-question-panel>
    <p>
    Here, the <code>pl-multiple-choice</code> element displays "All of the
    above" and "None of the above" as options below the correct and
    incorrect options provided.

    Both of these options could be correct with the same probability as
    all of the individual correct options. The number of options displayed
    is also consistent across
    all variants, and is determined automatically if <code>number-answers<
    /code> is not set.
    </p>

    <p> What is the color of the sky? </p>
</pl-question-panel>

<pl-multiple-choice answers-name="sky-color5" none-of-the-above="true" all
    -of-the-above="true">
    <pl-answer correct="true">Blue</pl-answer>
    <pl-answer correct="true">Dark Blue</pl-answer>
    <pl-answer correct="true">Light Blue</pl-answer>
    <pl-answer correct="true">Clear Blue</pl-answer>
    <pl-answer>Pink</pl-answer>
    <pl-answer>Purple</pl-answer>
    <pl-answer>Orange</pl-answer>
    <pl-answer>Yellow</pl-answer>
    <pl-answer>Brown</pl-answer>
    <pl-answer>Red</pl-answer>
</pl-multiple-choice>
```

Listing 4.1: example multiple choice question from PL's official repository[8]. Correct options are marked as `True`.

Looking at the code above, it is immediately apparent the `pl-answer` tag is repeated for every single option. This makes the question statements less portable and doesn't adhere to the Computer Science maxim: "Don't repeat yourself" (here the formatting is duplicated and "baked" into every possible choice). For situations like this, we wanted to have a dedicated, easily-parsable file to store these options, and simply inform the element to use that file. This led to a new attribute to this element called `external-json` where the user can specify a path to a JSON file that stores all correct and incorrect options in their own separate lists.

The implementation of this feature was more straightforward, once I figured out the file path configuration of PL, I wrote code to read options from a JSON file besides parsing through an HTML file; thus, this attribute is compatible for scenarios where options are split between the JSON file and the HTML file. Thanks to this implementation, questions with many options are quite condensed, as shown in below:

```
{
    "correct": [
        "Blue", "Dark Blue", "Light Blue", "Clear Blue"
```

```
    ],
    "incorrect_choices": [
        "Pink", "Purple", "Orange", "Yellow", "Brown", "Red"
    ]
}
```

Listing 4.2: example multiple choice question from PL's official repository[9]. This is the same as Listing 4.1, but all the options are stored in a JSON file.

4.2 Changes to `pl-checkbox` Element

The `pl-checkbox` element is for MCMS questions, where students can choose more than one option from a list of selected options. In CS10, this element is usually used for lecture video questions[10], where students are expected to choose all the correct answers to receive full credit, and students have unlimited tries with no penalty.

`hide-score-badge`

Early on, we discovered that `pl-checkbox` would display correctness badge next to the student-selected options after each submission, as shown in Figure 4.2.

This behavior would allow students to discover the correct options by submitting once with all options checked; thus, defeating our best intentions. I modified this behavior by adding a new attribute to `pl-checkbox` called `hide-score-badge`, that will suppress the display of the badges. Students would still be able to tell when they finished the question because the percentage badge would read "100%", noting that they had received full credit for the question.

`allow-none-correct`

The previous implementation of `pl-checkbox` asked students to check at least 1 box, otherwise it would raise an error[11]. In CS10, this made it impossible to create certain QGs. For example, a boolean-based question that asked students to check variables that would be true, this constraint prevented us from having variants where all variables would be false. PL developers suggested adding the option "None of above to the list of correct options and overriding the variant generation logic similar to `pl-multiple-choice`. However, this approach suffered the same issues mentioned in Section 4.1, and we believed this was a common need for other developers of PL. As of this writing, my implementation hadn't been merged

[10]Clicker questions are like lecture quizzes. During in-person lectures, students would participate by answering questions with their iClickers, provided free of charge by CS10.

[11]It was actually more advanced. The element could be configured to have a minimum and/or maximum number of correct options (≥ 1) and raise an error when a student selected too few/many choices.

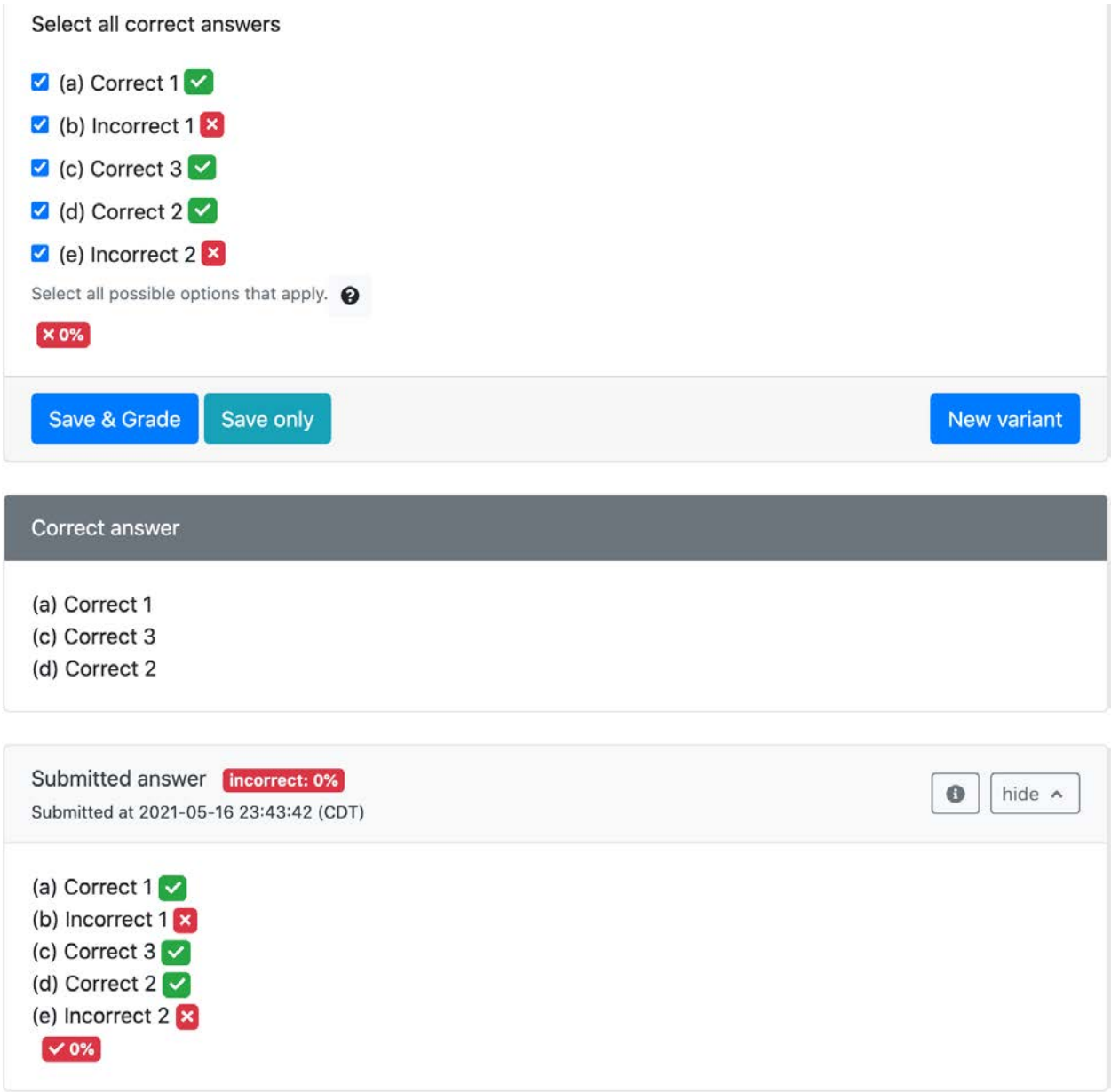

Figure 4.2: This is an example question. Correctness badge displayed next to student-selected options, giving away the answer.

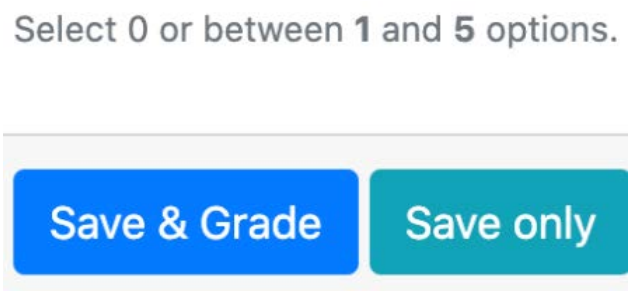

Figure 4.3: help text for `pl-checkbox`.

to master branch yet, but was similar to the solution presented in Section 4.1, where a probability model based on $\frac{1}{NumCorrect+1}$ determined whether all the options would be incorrect. With this attribute enabled, if a student believed all the options were incorrect for their particular variant, they would simply click "submit" directly, and no error would be raised.

Looking at the current implementation, I believe it suffers the same issue as the current implementation in Section 4.1; additionally, it might be a bad design to allow students to submit without selecting any option. I think a better design would be to add it as an option like "None of the above" that students can check; and once it is checked, raise an error when any other options are checked. As to the probability of it being correct, I think it can default to $\frac{1}{NumberAnswers}$ but allow the users to configure it via an additional attribute.

Bug in `help-text`

`pl-checkbox` has the option to display help text at the bottom of a question to inform students of the range of options to choose.

In Figure 4.3, the help text displayed two numbers that should be consistent across all variants of the same question. If this range was different for different variants, it could provide unfair advantage to some students. I discovered that when `min-correct` and `detailed-help-text` attributes were set to `True`, the second number would *always* display the actual number of correct options. Consequently, for students that received variants where all the options were correct, they'd immediately know and receive full credit.

4.3 Changes to `pl-figure`

In PL, QGs can display figures such as images. As explained in Chapter 3, images are very commonly used in the context of CS10 to accommodate practically all Snap*!* questions. The original `pl-figure` displayed all figures as block-level elements in HTML[12], but CS10 had a lot of small Snap*!*-code images that should be displayed as inline elements. To provide a

[12] `https://www.w3schools.com/html/html_blocks.asp`

solution to this, I implemented a new attribute to `pl-figure` called `inline` that when set to `True`, would override the HTML to display the figure as an inline-level element.

Chapter 5

Computer-Based Testing in CS10

This chapter will describe our experiences developing and delivering CS10 curriculum using PL. This work had many components and was a collaborative effort that involved many passionate and hardworking individuals. Notably, Erik Rogers deployed and maintained our PL instance where all the assessments were hosted; Shannon Hearn and Irene Ortega had significant contributions to CS10's course repository; Qitian Liao, Connor McMahon, and Professor Dan Garcia co-authored our poster to SIGCSE 2021. The following sections document how we set up the course, categorized QGs, and delivered course content and assessments on PL; as well as some of the things we've learned in retrospect.

5.1 Course Repository

By default, PL handles different courses by parsing through directories at the root that contain a file called `infoCourse.json`, through which PL would load contents from these directories[1]. In production deployment, PL can sync up directly with a remote repository, so that course maintainers only need to make changes to the remote repository, like GitHub, to update their respective course(s). Additionally, PL allows maintainers to define instances of the same course (for example, CS10 has fall and spring offerings), and have an instance for each offering in a course repository. These instances consist mostly of JSON files that define assessments because QGs are all shared across course instances, making each course instance relatively small in terms of disk space. For most users of PL, these resources should already be set up by course repository maintainers.

For CS10, Erik Rogers[2] managed the deployment and maintenance of its production PL instances, I was in charge of maintaining the course repository on GitHub. One of the very important lessons we learned early on was the importance of coordination between PL-instance and course-repository maintainers. At the time of this writing, PL course ad-

[1]PL instances can also be customized with a configuration file: `https://prairielearn.readthedocs.io/en/latest/configJson`

[2]https://eecs.berkeley.edu/people/staff/it-staff

Figure 5.1: CS10 QG metadata tags describe (i) the type of assessment (formative or summative) the QG is intended for, (ii) exam level it is based on, (iii) difficulty level, iv) question format, v) whether the QG is ready for release, (vi) GitHub username of the creator of the QG, (vii) the school the question comes from, (viii) the semester the question is based on, ix) conceptual vs. numerical variants, and x) other useful metadata.

ministrators had limited access to its database and certain configurations, and there were a few times we needed some last-minute changes that only Erik could accomplish. Erik also managed scaling of our PL instances, especially around exam times, so it was generally a good idea to keep him informed about important upcoming events that might impact our PL instances.

Metadata in CS10[3]

At the course-level, PL lets users define colorful custom tags[4] as metadata for QGs. When we started transitioning to PL, we realized the importance of metadata as we knew we were going to have hundreds of QGs; these metadata would be helpful for organizing and searching for QGs. The CS10 team spent a lot of time indexing all of its course concepts and discussing tags we needed to help us organize all the QGs. All of this was documented in our internal documentation for future users. So far, our tagging scheme has made organizing and searching convenient since we can quickly glance at useful information for each QG, see Figure 5.1 for an example.

When correctly used, our tagging scheme is a very useful form of documentation; in fact, it would be a good idea to have a strict rule requiring correct tags for *reusable* QGs. In Figure 5.1, the third to last row was a QG specifically written for a final exam, so it wasn't a big deal that the tags weren't complete. In our experience, it was very difficult to go back

[3]This work was mostly done by Irene Ortega and Shannon Hearn.

[4]`https://prairielearn.readthedocs.io/en/latest/course/#colors`

and correct tags for past QGs due to lapse in time, so it would be a better practice to update tags as the QG is developed.

CS10 Concept Map[5]

Concept maps are graphical tools for organizing and representing knowledge. They include concepts, usually enclosed in circles or boxes of some type, and relationships between concepts indicated by a connecting line linking two concepts [27]. In order to provide students with a better view of their learning trajectory as well as promoting mastery of learning, Shannon's team painstakingly compiled all the concepts in CS10 and constructed CS10's first ever concept map, as shown in Figure 5.2. Using this map, we were able to identify specific concepts that we needed to create QG(s) for and make sure we had good exam coverage for midterms and finals. At the moment, the concept map is one of the internal documents that CS10 staff references. This is an active area of development for the CS10 CBT team.

5.2 *Numerical* versus *Conceptual* Variants

An important contribution we have made to the computer-based testing space is a formal categorization of QG variants that we believe can be informative to others utilizing QGs.

For context, after authoring many QGs, we realized two common problems that arose over and over again.

1. How many variants do we have?
2. How do we test all the variants?

The first problem concerns the chance a student will have a different answer than their neighbor, and generally the more variants we can create for a QG the better. The second problem concerns the process we take to ensure good test coverage of all variants.

In our poster to SIGCSE 2021 [4], we advocated for two categories of variants for QGs: numerical and conceptual variants. We think of *numerical variants* as variants that vary (easy) question parameters; these tend to be the low-hanging fruit when creating randomization to a question. On the other hand, *conceptual variants* vary significant parts of a question, such as the question setup. Using a mathematical question as a small example, if the question asks students to determine the value of y given the equation $y = x + 1$and the value of x. Numerical variants might simply vary the value of x, so that students would have different answers for the value of y. Conceptual variants might change the equation itself to something like: $y = x \times 2$, $y = x^3$, $y = \frac{x}{4}$ etc. so that given the same value of x, the value for y will still be different. However, one important consideration when creating any variants is maintaining the same difficulty level across *all* possible variants. In the previous examples,

[5]This work is mostly done by Shannon Hearn and her team.

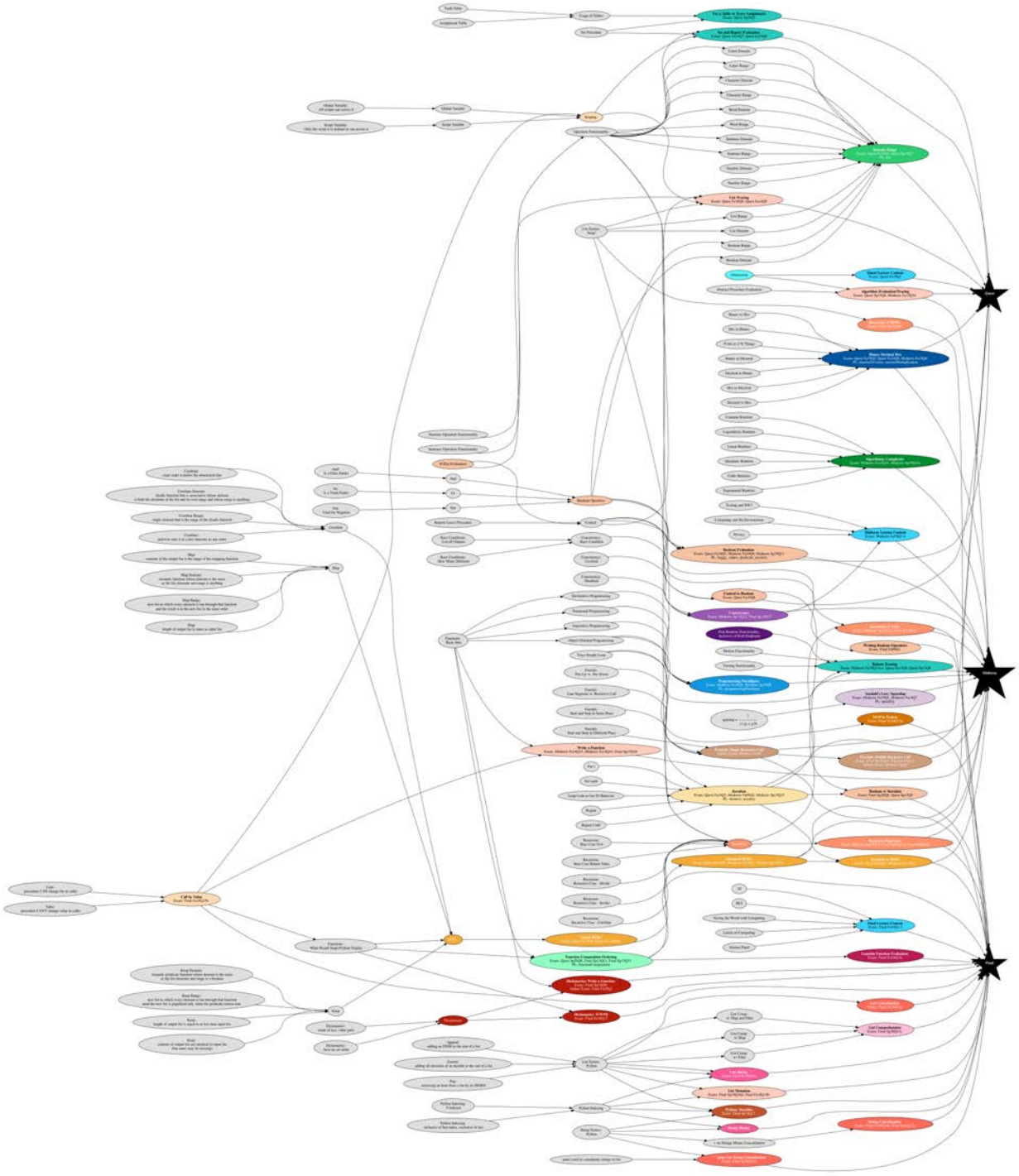

Figure 5.2: CS10's concept map. The goal was to make the map visually pleasing and interactive for students in the future, so that students would have a clear idea of the direction of the course, and more importantly, a visual representation of their progression through the course.

Conceptual Variants	Numerical Variant #1	Numerical Variant #2	...	Numerical Variant #N
Conceptual Variant A	Variant A,1	Variant A,2	...	Variant A,N
Conceptual Variant B	Variant B,1	Variant B,2	...	Variant B,N
Conceptual Variant C	Variant C,1	Variant C,2	...	Variant C,N
...	...	...	...	...

Figure 5.3: If a QG has N different numerical variants for each conceptual variant, instead of checking every single combination of conceptual and numerical variants, the QA team can test the first few numerical variants (in green) and skip the rest (in red) for each conceptual variant; thus, saving valuable time.

raising x to the power of 3, or dividing by 4 are quite different conceptually from just adding 1. Instructors are encouraged to beta-test conceptual variants with their teaching staff to assure they are of equal difficulty.

It is worth noting that the term "variant" in our description of these categories has a "uniqueness" property attached to it, in terms of the resulting question. Sometimes, different combinations of parameters can produce the same question as far as students can tell; as QG authors, we care only that the questions look different to students.

After a QG is completed, before it is used on an exam, the QG is always tested to ensure that the unique variants are working as expected, such as the display of the prompt, images, and choices, as well as the grading. Depending on the number of possible unique variants, this could be a daunting task; taking forever to cycle through all possible combinations of a QG's parameters. Fortunately, our proposed categories of variants can provide a methodology for testing efficiently.

We believe that when testing a QG, after having checked the correctness of a few numerical variants, we can be very confident about the correctness of the rest. After all, numerical variants should be generated with the same solution model and should be relatively easy to reason about. On the other hand, conceptual variants should be checked individually because of the more complex variability. Figure 5.3 illustrates this process.

Figures 5.4, 5.5, 5.6 are some real examples of conceptual variants in CS10. Applications of conceptual and numerical variants can be different for other classes. For example, a QG that tests students on Depth-First Search (DFS) can have different tree-node values as numerical variants, since these values have no effect on the traversing order; the QG can use different DFS orders as conceptual variants, such as pre-order, in-order, and post-order. In testing, the QA team can check a few numerical variants' correctness for different orders of traversal to conclude the QG's overall correctness.

In CS10, we added metadata tags "CvN" to designate the number of conceptual and

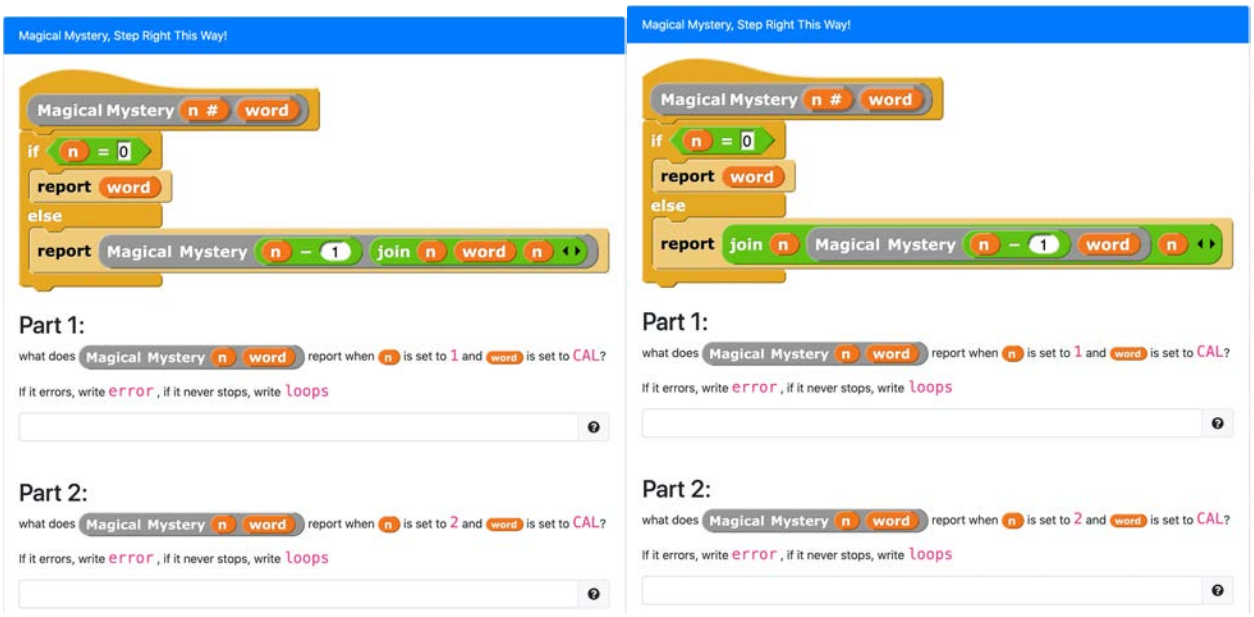

Figure 5.4: A question we authored that appeared in Fall 2020 Midterm 2. The question has no numerical variant, and these are the only two conceptual variants. The only difference is in the recursive call, everything else is the same. 238 students took the exam, the average for the variant on the left is 78.58%, on the right is 81.93%, the question itself is worth 8 points and has 5 subparts (only first 2 subparts are shown), the 3.35% difference translated to less than 0.3 points for an exam worth 100 points.

numerical variants to our QGs, see Figure 5.1. This made it convenient to select which QGs to be included in the assessments, without having to check out the source code. The categorization also simplified counting the number of unique variants when documenting existing QGs. We recommend following a similar approach when documenting the QGs; additionally, other categorization methods with varying granularity are possible. We welcome and look forward to new categories being added to our list.

5.3 Content Delivery and Assessments

In CS10, we were able to integrate lectures and assessments into PL with modifications to policies of each. This section details our approach and results.

Lectures

Due to COVID-driven online learning, CS10 changed its lecture format completely. Instead of having live lectures, Professor Dan Garcia recorded and uploaded high-quality, short clips

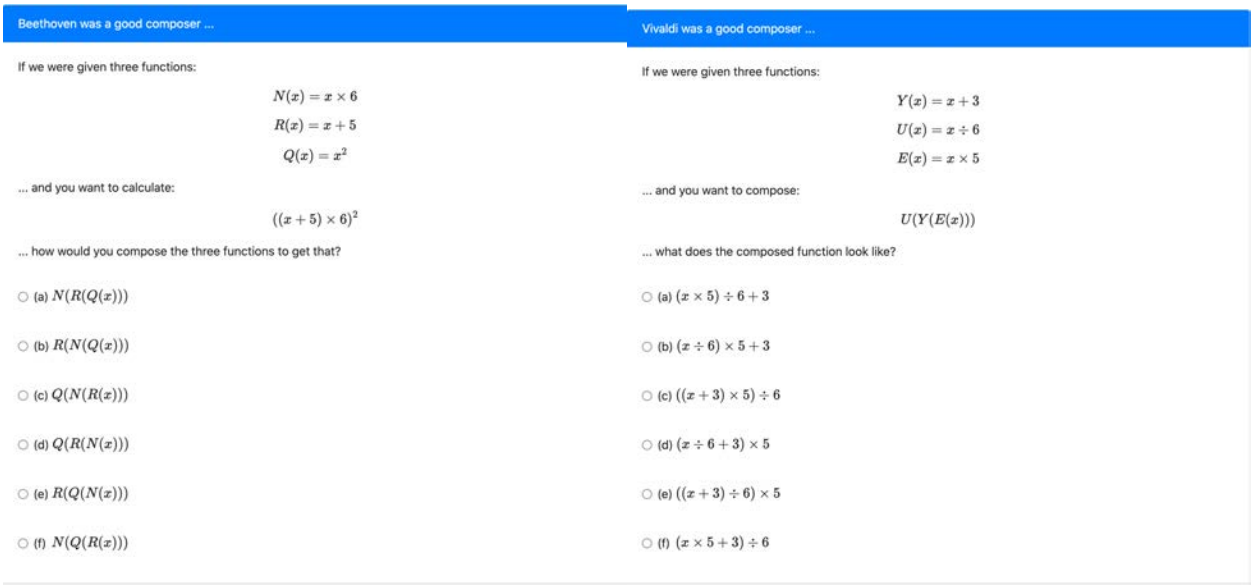

Figure 5.5: A question we authored that appeared in CS10's Fall 2020 first Midterm,. There are 2 conceptual variants (whether the student was moving from compound expression to nested subexpressions, or vice-versa), each with 5,120 numerical variants (the individual three functions listed at the top). Combined, this QG had over 10 thousand unique variants. The two conceptual variants are, in a sense, inverses of each other, the one on the left had an average of 90.26%, the one on the right 87.38%. This question was worth 2 points, so the difference translates to less than 0.06 points for an exam worth 20 points.

of videos that students could watch any time on their own. Each video usually covers a single topic and is between 5 to 10 minutes long; a typical one-hour lecture would consist of 4 to 5 videos. Students were *recommended* to finish watching the videos corresponding to each lecture *before* the usual lecture times; and the usual lecture times were turned into Dan's office hours that were always recorded, with recordings made available afterwards.

To ensure students paid attention to lecture videos, each video came with its own quiz known as a "clicker question" below the embedded video itself. A lecture with 5 short videos would have 5 different clicker questions. These questions were all constructed using the `pl-checkbox` element where 5 correct and incorrect options were supplied, and only 5 options were displayed[6] where any number of them could be correct. The options range from matter-of-fact to simple-concept, all based on the corresponding video[7]. Figure 5.7 is an example of a lecture video.

Students were also told that they had an unlimited number of tries, without penalty for

[6]This yielded on average 1000 unique variants, given that the pool of correct and incorrect statements averaged around 10

[7]The instructor and I typically collaborated in writing these questions.

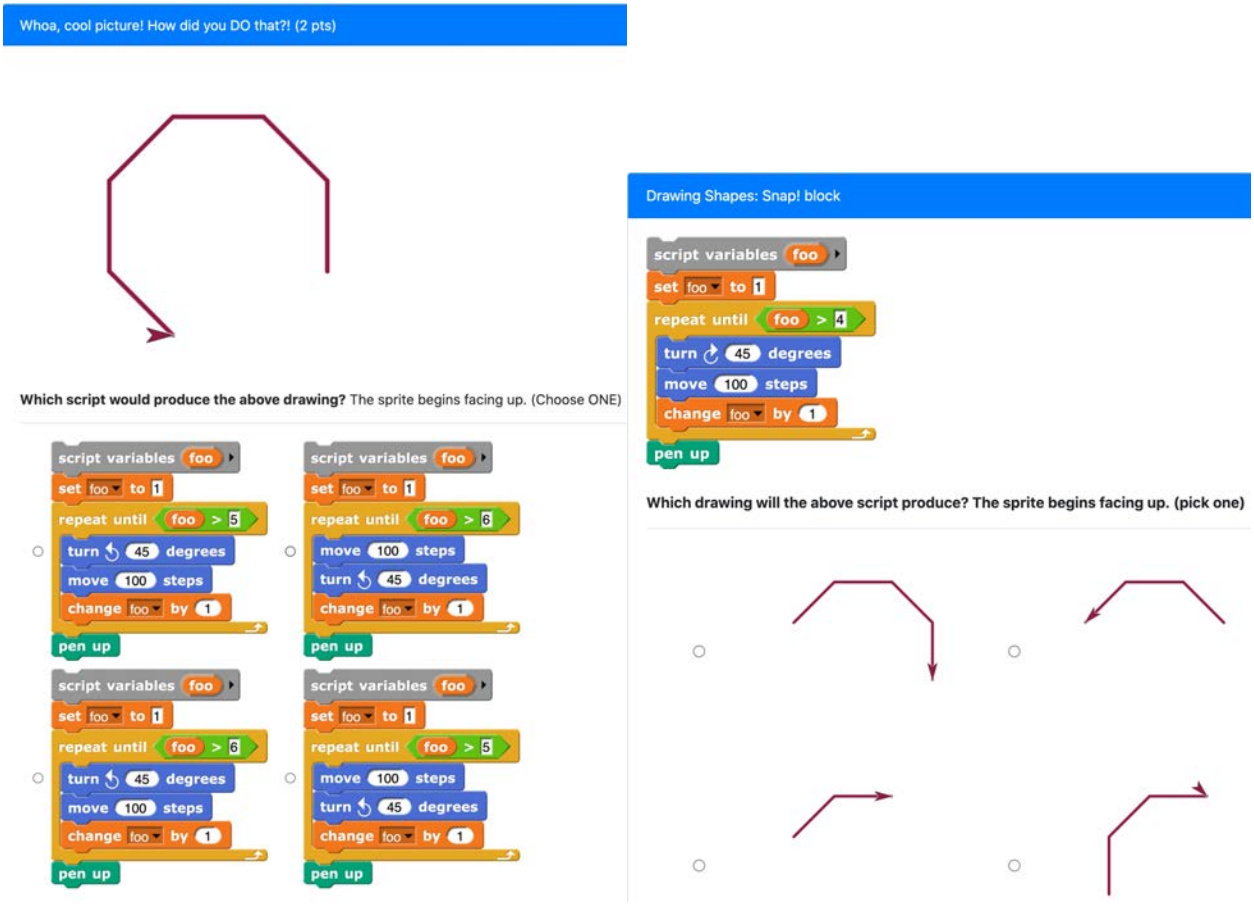

Figure 5.6: A question authored by a small group of new members of the R&D Group in spring 2021, under our supervision. There are 2 conceptual variants, each has 62 numerical variants, with a combined 124 unique variants. The variant on the left appeared on the spring 2021 final exam. The reason the variant on the right was not chosen was that it was felt students could too easily re-code the script and immediately see the answer on the Snap*!* stage.

incorrect attempts, for each clicker question; and the questions were all-or-nothing completion points that they could access any time until the end of the semester. We also set up a Piazza thread for each lecture where we answered student questions to the best of our abilities. The intention was to incentivize students to pay attention while watching these videos and deter cheating, without causing unnecessary stress or frustration. At the end of the Fall semester, average scores on all lectures were over 96% with the exception of a couple lectures that had averages in the lower 90s. The clicker questions had a combined worth of 11 points out of 500 total points.

Figure 5.7: Video 2 of Lecture 8 from Fall 2020 semester

Assessments

One of the unique challenges with remote exams is accommodating time zones and special needs requests. Another challenge is the prevention of cheating. The remote setting made it difficult for classes to come up with ways to promote fair exams, and some classes opted for some form of online proctoring.

CS10 had the benefit of not being one of the required classes for major declaration and has traditionally had a low rate of cheating in the past; students normally over-collaborate on individual projects or plagiarize when writing their essay. Sometimes students give in to pressure and share code during our "in-lab, with-computer" exams, but that is quite rare – once every other year or so (four semesters). For the Fall 2020 semester, we decided that we were going to trust the students and introduced non-proctored, open-book, take-home exams. These exams were all offered using PL, and I assisted in writing many of the QGs. For Midterm 1, known as the Quest, students were given 24 hours to complete an exam designed to take less than 1 hour. Students could (i) work on the exam at any time in those 24 hours, (ii) leave in the middle of it, then come back later, and (iii) use all resources offered on CS10's website (including lecture videos), just not open Snap*!* or search online for answers. For Midterm 2, the testing format was mostly the same, and the exam was designed to take less than 4 hours, but students were given 48 hours to complete it from the moment they started. We kept the exam open for 72 hours because the exam spanned over a weekend, so some students could choose to allocate half of their time for Monday. For the Final, we gave students 72 hours to complete the exam that was designed to take less than 5 hours.

For Midterm 2 and the Final, there were programming components to the assessments in which students were allowed to use live instances of Snap*!* and Python. Since students were not allowed to have these during some parts of the exams, we divided both exams into two parts: the first part consisted of regular questions (mirroring a traditional, on-paper exam), and the second part consisted of questions that required the live instances. Before students started the first part, we had them sign a pledge stating that they would not open any instances of Snap*!* or Python while working on the first part.

In Spring 2021, the same testing formats were followed, and we believe this reduced a lot of the unnecessary stress from exams. Students seemed to like these formats as shown in Chapter 6. Additionally, the exam statistics remained consistent with past semesters, and there was no indication of widespread collaborative cheating in any of the exams so far.

Cheating Cases

At the end of the fall semester, the CS10 course staff caught a few (6) cases of collaborative cheating. At the time of this writing, PL didn't have a built-in cheating detection tool (see Chapter 8); however, it did have logs of student activities based on button click events, such as the opening, closing, submitting a question, etc. The course staff looked through logs

of students who submitted within minutes of each other and was able to identify groups of students who worked on the exam together based on their activity timestamps.

As mentioned in the previous section, students were allowed to use live instances of Snap*!* and Python for part two of Midterm 2 and Final. Unlike part one of the exams that were generated *and automatically graded* on PL, part two were only *generated* on PL, then hand graded by course staff. This was because there wasn't an integrated tool to automatically grade Snap*!* code; however, this did allow course staff to give generous amounts of partial credits. Some students that collaborated on this part of the exam did not realize the questions were randomized, and submitted solutions not meant for the variants they received. Additionally, when hand-grading coding questions, course staff sometimes noticed similar patterns in solutions among students, which also helped catching collaborative cheaters.

Chapter 6

Survey Results

For the Spring 2021 semester, with the help of CS10's course staff, especially Lam Pham and Yolanda Shen, we sent out a course-wide post-Midterm 2 survey that focused on students' test-taking experiences on PL. Students were incentivized with 1 extra-credit for completing the survey. Of the 138 enrolled students, 60 completed the survey.

Based on the responses received from students, the examination experience on PL was overwhelmingly positive with no reported negative experience. The long-exam format was very much preferred by students even though they didn't need all the extra time to finish the exam. Most students also expressed a belief that this form of exam would prevent cheating; coupled with the fact very few cheating cases were found, the exam format we have introduced might be worth experimenting with in similar classes.

Figure 6.1 and Figure 6.2 indicate that students had an overall positive test-taking experience using PL. It is generally important to ensure that students are comfortable with the technologies the class uses; in this case, it seems safe to conclude that students preferred using PL and that we should continue to use it for assessments, *even when we return to face-to-face instruction.*

Figure 6.3 shows that students liked the longer exam format. This was expected, and it was nice to confirm our hypothesis. Many students in the survey noted that the longer exam format was less stressful, more flexible and accommodating. Some students noted that the longer exam format caused them to spend more time because they felt pressured to double check their answers to avoid simple mistakes; whereas with the normal exam format, they would only double check their answers if they had time left. Understandably, this was one of the drawbacks of the longer exam format; however, we believe the significantly lessened anxiety is worth the tradeoff, and this question seemed to confirm that.

Figure 6.4 and Figure 6.5 indicate that for the vast majority of students, the long exam format was more than enough time in students' opinion to complete both parts of the exam. It also reveals that there is a long tail, and that some students used far more hours than they would normally have been allowed to take. If they needed those hours, great, and this model of assessment is a real win for them. They also might have just burned hours making sure every answer was perfect (possibly at the detriment of their other classes). Either way, the

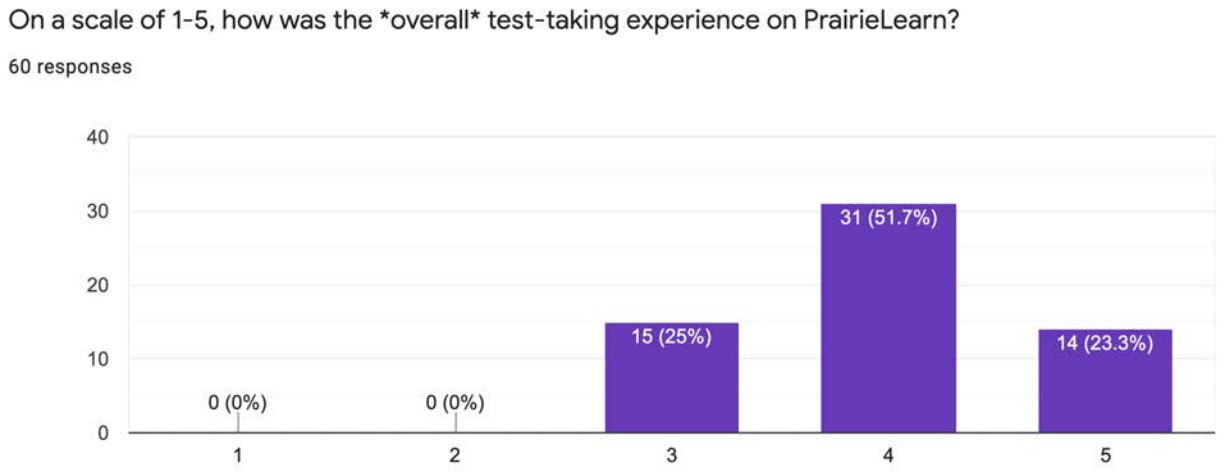

Figure 6.1: Students rate test-taking experience on PrairieLearn. 1 for "Worst" experience, 5 for "Best" experience

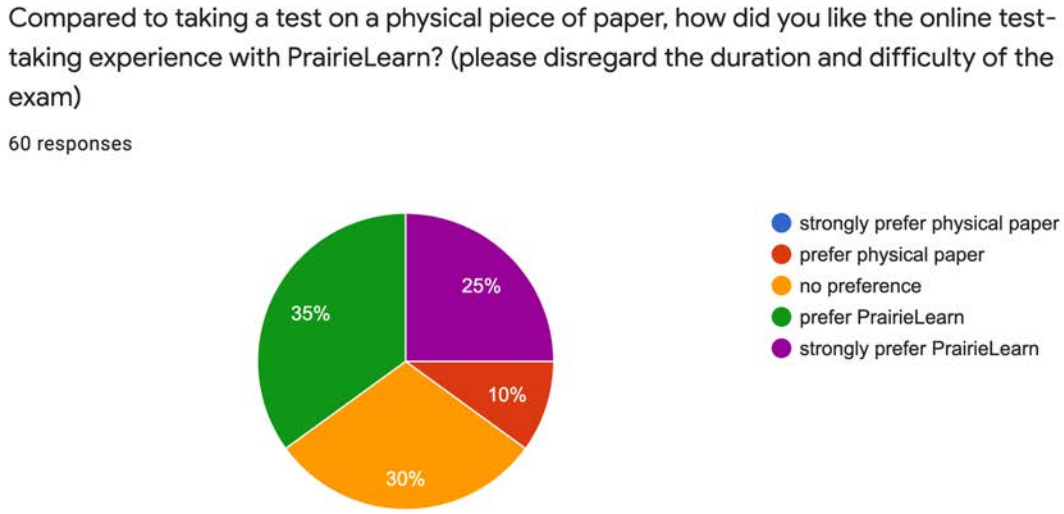

Figure 6.2: Students' preference for PL compared to paper testing

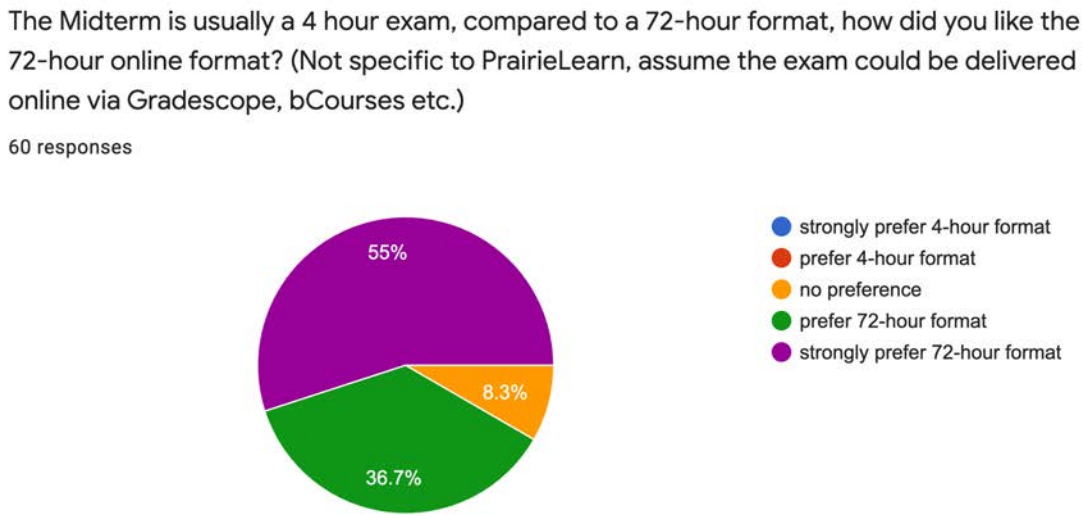

Figure 6.3: Students' preference for long vs. regular length exam

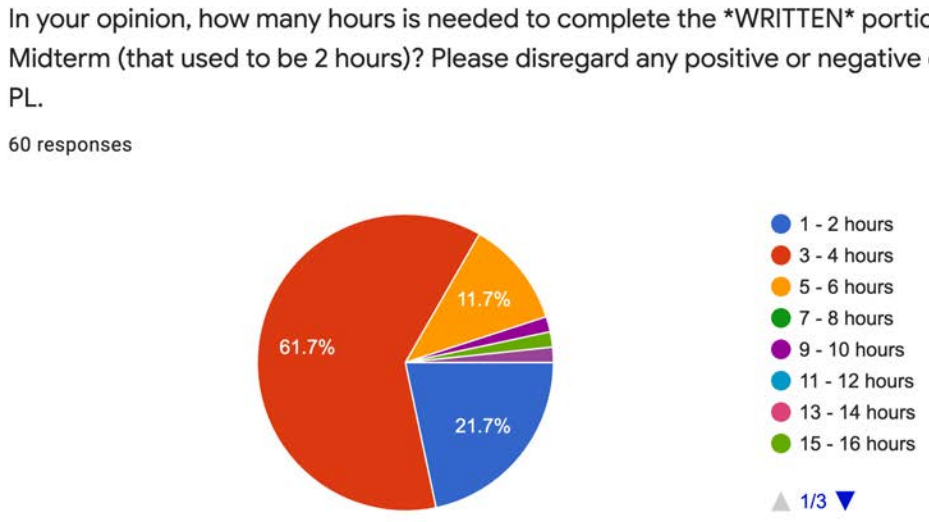

Figure 6.4: Students' perception of part one of the exam

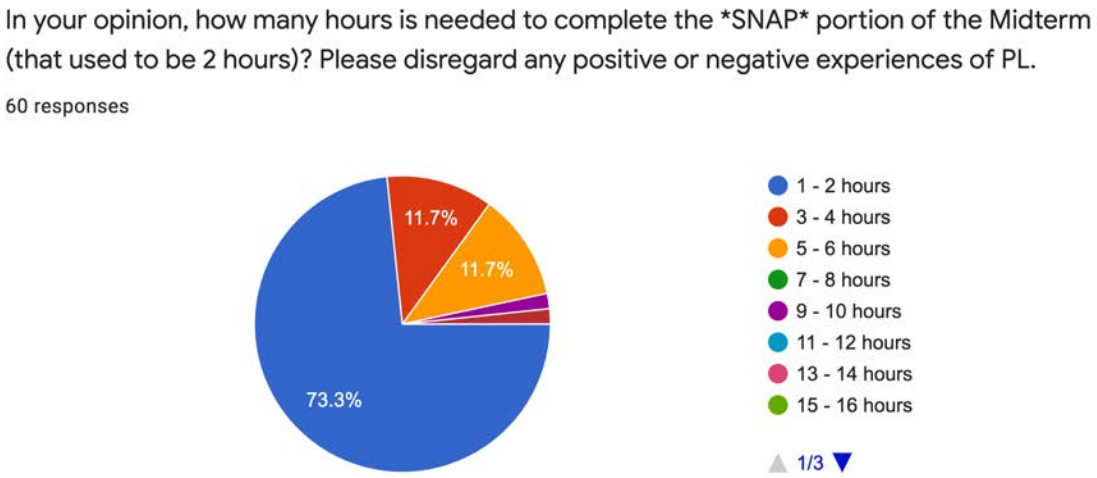

Figure 6.5: Students' perception of part two of the exam

overwhelming evidence presented in Figure 6.3 says we should continue untimed, take-home exams.

When writing the exam, we spent many hours coming up with ways to create more variants and for some questions we couldn't create any variant at all, so we were interested to learn if students could tell when a question had no variation; or more specifically, does having more/less variants make a question easier/harder for students to identify it as having more than one unique variant.

In Figure 6.6, we reminded students that PL can randomize questions for different students, then we asked them to try to identify questions that were different for other students. Of all the questions on the exam, some of them had no randomization at all, some had thousands of unique variants. We hypothesized that there would be some positive correlation between a question's number of unique variants, and number of students to identify that question for having randomization.

The vertical axis in Figure 6.6 is the normalized number of unique variants the horizontal axis shows the normalized percent of students who thought that question was different for others[1]. The second-to-last choice said "Other: everyone has the same questions", the last choice says "Other: I'm not sure". Of the 60 students who answered, 7 of them (11.6%) thought that the entire exam had no randomization; 24 of them chose to not identify any of the questions.

Of the 14 questions, 4 questions had no randomization at all, 6 of them had less than or equal to 24 unique variants, 2 of them had hundreds of unique variants, lastly, 2 of them had thousands of variants. Based on the R-Squared analysis of a linear regression in Figure 6.7,

[1]Full names of the questions were displayed on the actual survey, here, the names are condensed.

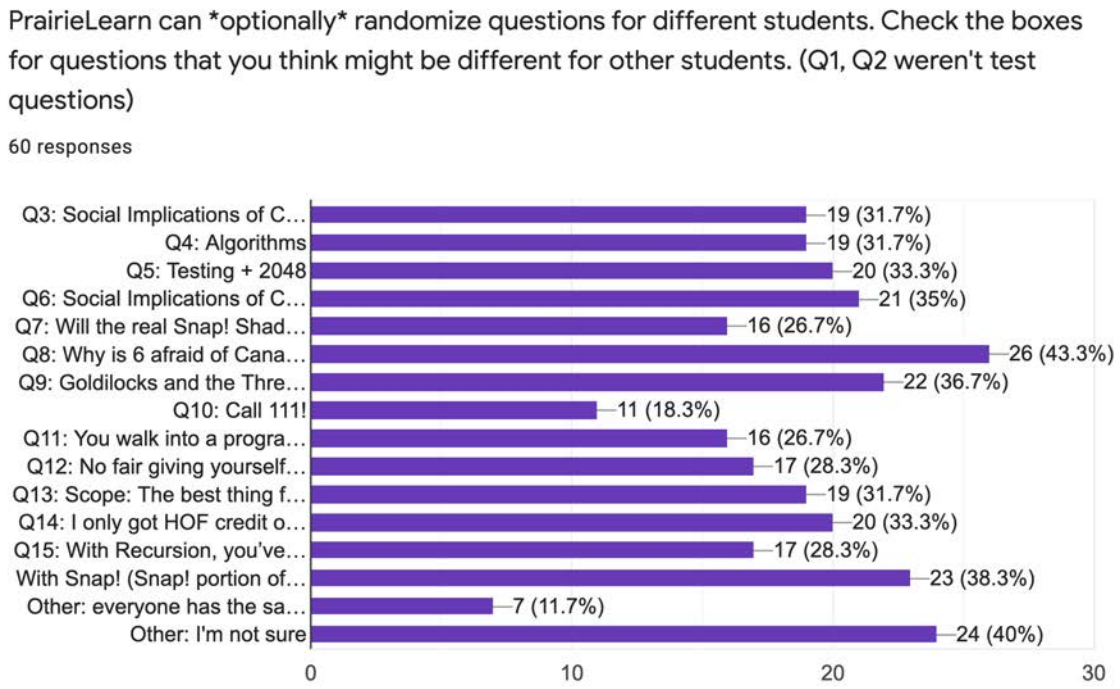

Figure 6.6: Students' guesses on whether each question was randomized

there was *very little correlation* between a question's number of unique variants, and the percentage of students who were able to identify that question for having randomization. This was surprising to us since from QG authors' perspective, it seemed intuitive the more "entropy" (number of unique variants) a QG has, the easier it is to identify its variability, but this was not the case. For certain questions, a QG author could easily imagine different ways to create randomization, so it is possible that we had developed a sense of "smell" for randomizable questions. This finding could be helpful because it showed that QG authors had a "home field" advantage, and this could be an opportunity to mix-in more complex questions with no or few variants *as long as the students don't find out.*

We were also interested in whether students were confident the exam format would be able to prevent cheating.

In Figure 6.8 and Figure 6.9, it seems that students were generally trusting that PL would prevent cheating but had mixed opinions about its ability to catch cheaters. Figure 6.9 is especially interesting in that the histograms resemble a tri-modal distribution. Some students noted that it would be very difficult to detect students who used live instances of Snap*!* or Python when they were not supposed to; others believed that PL had advanced tracking techniques to detect cases like these (it doesn't). Even though individual cheating detection

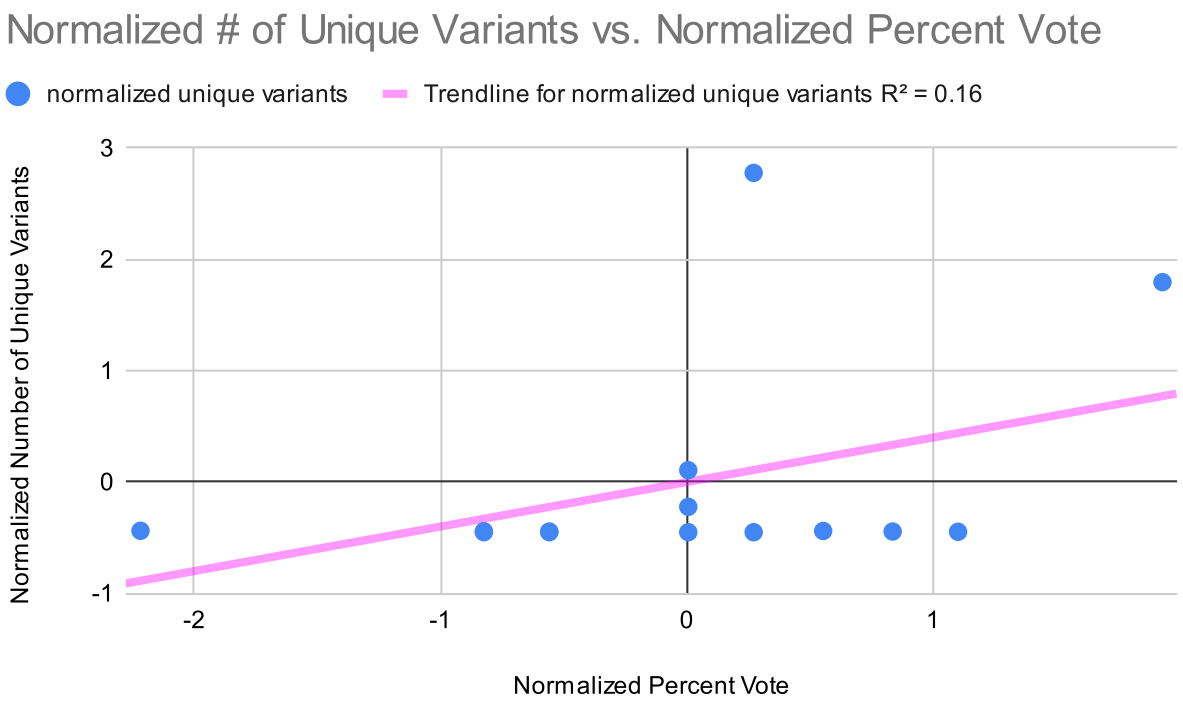

Figure 6.7: Normalized Number of Unique Variants vs. Normalized Percent Vote

is not the main goal of PL, it would be worth investigating whether some existing techniques could be integrated into it (see Chapter 8).

We also simply asked the students if they knew anyone who cheated (Figure 6.10). It was good to know that of the students who participated in the survey, no one claimed to know anyone that cheated.

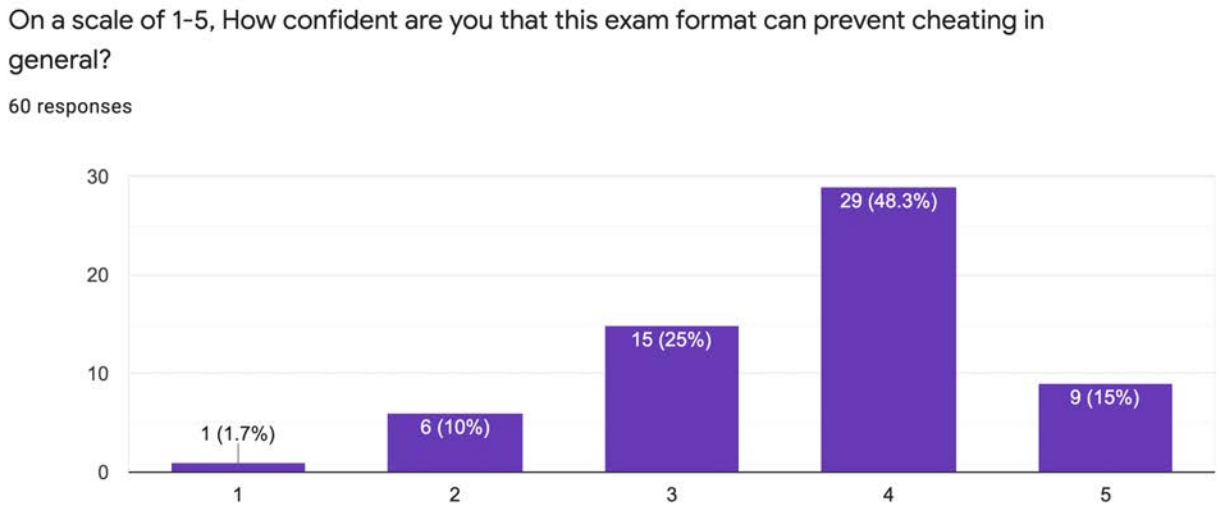

Figure 6.8: Students' confidence level in cheating prevention. 1 for "No confidence", 5 for "Very confident"

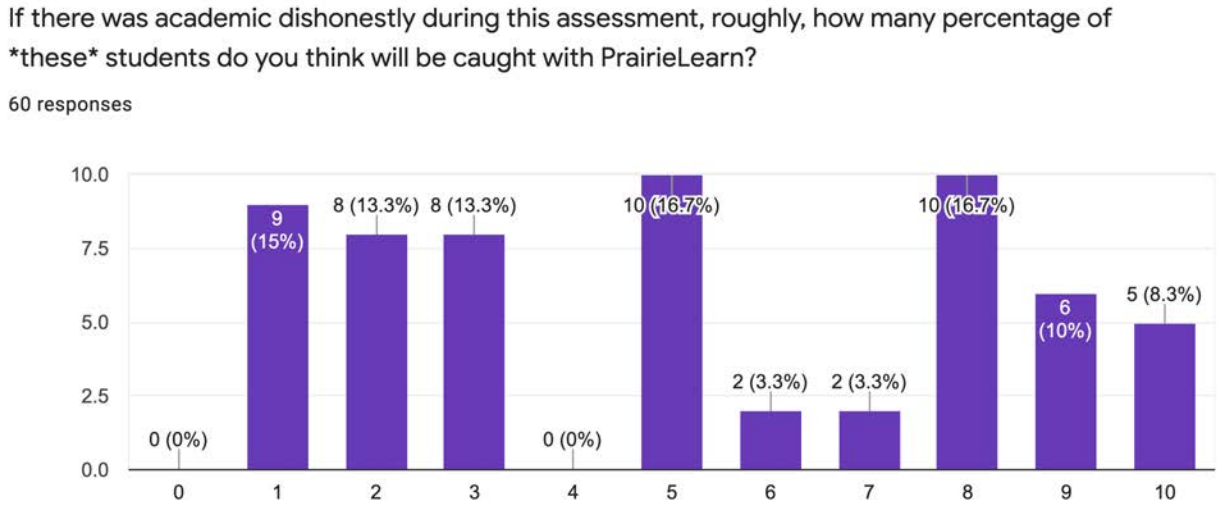

Figure 6.9: Students' confidence level in cheating detection. 0 for 0%, 10 for 100%

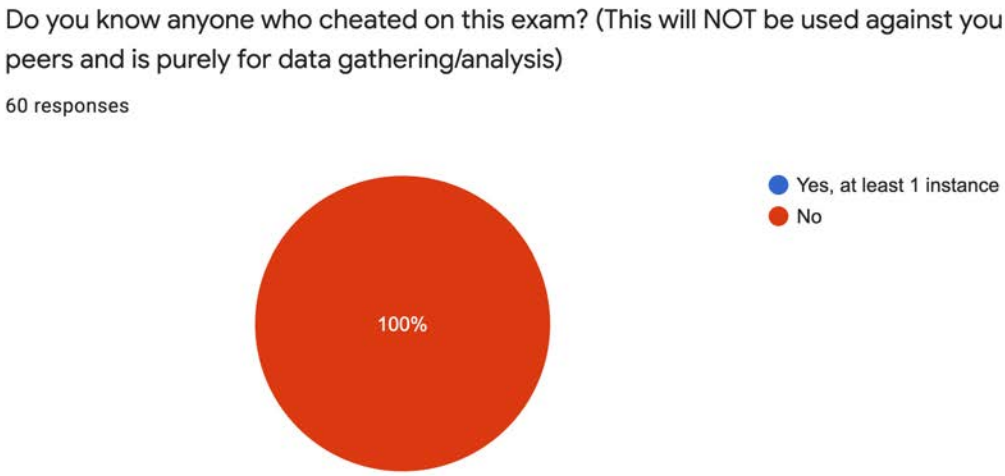

Figure 6.10: Do you know anyone who cheated?

Chapter 7

Leadership Development

In the Spring 2021 semester, I voluntarily took charge of two separate groups of students that worked on PL-related projects. In the end, both groups were able to achieve their respective goals. This chapter summarizes their work and details my experience mentoring these students.

7.1 QG Team

With the R&D Group, there were many new members who joined our efforts in CS Education research. New and old members were separated into smaller groups to work on topics that interested them. As one of the most experienced authors of QGs, I took charge of the QG Team that consisted of members who were interested in learning about authoring QGs in PL. By the end of the semester, the subgroups were able to deliver two, brand new, final-level QGs that were actually included in the Final. This also marked the first time new members of the R&D Group contributed to one of CS10's exams.

I found that having new members to start by writing some simple QGs modeled after existing ones was very helpful for them to learn PL and discover potential confusions. In fact, the learning process was similar to that of programming languages; by reading and understanding examples, they were able to gauge the capabilities of the system as well as learning some new tricks. It is also worth noting that members with limited prior programming experience struggled a lot the entire semester; they were more likely to encounter roadblocks so it was important to pay close attention to their progress.

Additionally, writing QGs for the first time could be frustrating since numerous revisions were typically required before a QG was released to students; sometimes the QG might never get accepted due to various reasons. As a mentor, it was important to actively re-engage a group's attention after it became obvious their implementation wasn't going to be accepted because the wasted efforts might very well cause members to lose interest in the project.

Based on my experience onboarding new members for the QG team, I have compiled a short list of recommended steps to onboard new members in Appendix A.

7.2 CS169 Student Group

CS169 (Software Engineering) has semester-long projects where groups of students help clients to implement features or products. Students from one of the groups were interested in working on a PL-related project, and I volunteered to design and advise them on implementing a working prototype of a PL assessment configurator. For context, assessments on PL are created and configured with a JSON file as shown in Appendix C.

PL has many parameters to customize aspects of its assessments, its official documentation does a reasonably good job of outlining and explaining various parameters relating to assessment configurations[1]. However, in our experience, configuration issues were *very* common due to many details of typical assessments.

To make the assessment configuration process easier and less error-prone, the student group from CS169 built a web application with an interactive graphical user interface that syncs up with the course repository. The application allows users to browse, drag-and-drop QGs, and make basic assessment configurations, then output the configurations as a JSON file. I planned and designed all functionalities of the application; the team implemented the application following the design decisions. We met every week to check on the progress of the project, and I'd give advice on existing implementations, then, I'd plan out the next steps and assign tasks for the team before our next meeting. By the end of the semester, the group of students were able to deliver and present a working prototype to the class (see Figure 7.1), and we had a new tool to help configure assessments.

[1]For more assessment configurations: `https://prairielearn.readthedocs.io/en/latest/assessment`

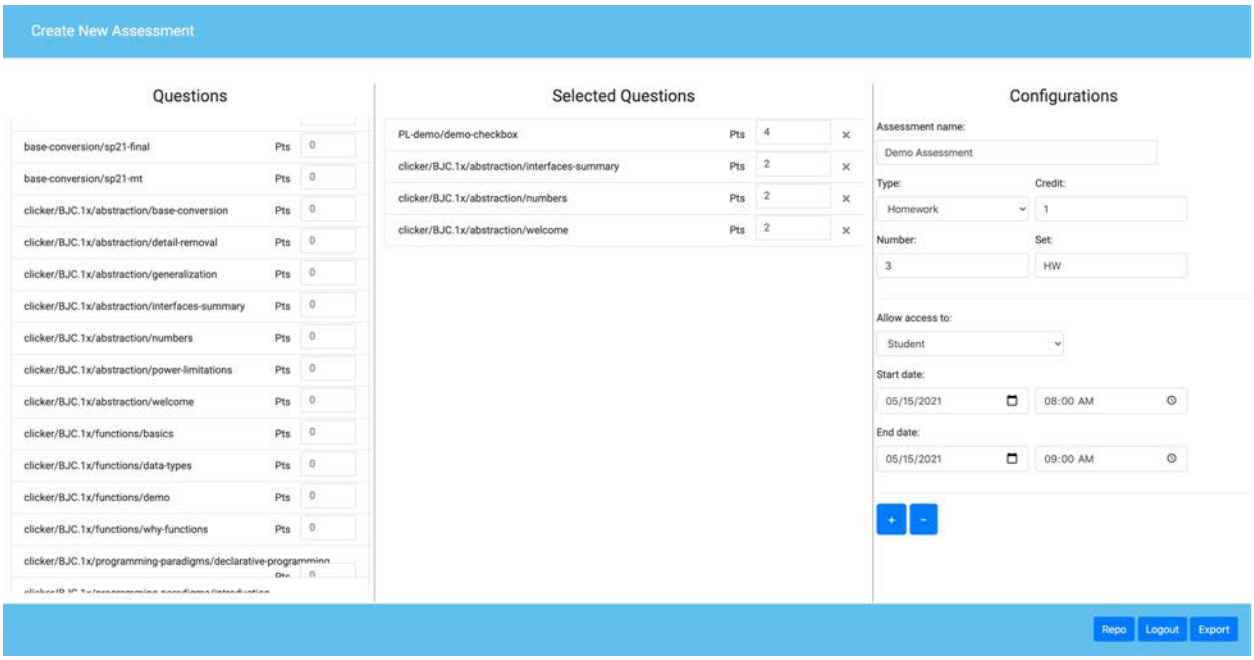

Figure 7.1: Configuration page for students' project. This is one of the three pages the students implemented.

Chapter 8

Future Work

Besides finishing merging pull requests related to elements mentioned in Chapter 4, there are many much-needed improvements to PL that are important to its role in CS10. This section briefly outlines some of the notable ones.

- A new feature to PL that will allow QG authors to quickly generate unique variants in a controlled and predictable manner; instead of generating a random variant each time.
- Score statistics across the unique variants generated for particular assessments, to help identify possible unfair variants post-assessment.
- A better integration of conceptual variants in PL. Right now, conceptual variants are sometimes mixed with numerical variants as a single QG or are saved as separate QGs.
- A (possibly) new PL element based on checkboxes that can be configured and displayed as a grid. This will be useful for pattern-based questions.
- A new PL element for matching. The existing `pl-drawing` elements are too flexible and general purpose to be used for matching questions.
- A new feature to PL that will generate a different variant after a student's failed attempt.
- A cheating, or collaboration detection tool to flag students during, or after an assessment.

These items are just a few of a myriad of possible projects to improve PL further for the purpose of CS10, and possibly other courses. Additionally, many existing elements, features, and configurations of PL can be improved to become more powerful or user-friendly. For those interested, it will not be difficult to find new projects to work on.

Chapter 9

Conclusion

This report documents my work relating to PrairieLearn (PL) in the context of CS10. In the span of a year, I helped transition the course to this new platform, contributed back to the Question Generator (QG) community, and grew as a mentor.

To future PL developers, I strongly recommend following the agile methodology[1] when contributing to PL and gathering feedback from a wide range of users. Additionally, don't be hesitant to propose and defend new feature ideas to PL's core team. Some of the features mentioned in Chapter 4 were not accepted by PL's core team at first; however, we were able to convince them after explaining our needs.

The lecture and exam formats for a non-majors course described in this report were novel, and supported institutional resilience (and reduced student stress) in the COVID-driven online learning format. The course staff enjoyed the convenience and streamlined experience of autograded assessments, and we were proud to be able to offer substantial flexibility to our students. Furthermore, based on results from Chapter 6, it appeared that these course formats yielded better class experience for the students too. As a result, we believe these changes are worth preserving for future iterations of the class.

We were pleasantly surprised by the lack of widespread academic dishonesty under our relaxed policies; it might be due to the fact that CS10 was not a required course for majoring in computer science. It'd be worth investigating if the randomized exams worked really well to deter cheating or because of other factors.

[1] `https://www.infoworld.com/article/3237508/what-is-agile-methodology-modern-software-development-explained.html`

Bibliography

[1] BJC. Cs10: Beauty and joy of computing.

[2] Matthew West, Geoffrey Herman, and Craig Zilles. Prairielearn: Mastery-based online problem solving with adaptive scoring and recommendations driven by machine learning. *2015 ASEE Annual Conference and Exposition Proceedings*, Jun 2015.

[3] Binglin Chen, Matthew West, and Craig Zilles. How much randomization is needed to deter collaborative cheating on asynchronous exams? In *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*, L@S '18, New York, NY, USA, 2018. Association for Computing Machinery.

[4] Bojin Yao, Qitian Liao, Connor McMahon, and Daniel D. Garcia. Formal categorization of variants for question generators in computer-based assessments. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 1244, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Ursula Wolz, Gail Carmichael, Dan Garcia, Bonnie MacKellar, and Nanette Veilleux. To grade or not to grade. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 479–480, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Benjamin S. Bloom. Learning for mastery. instruction and curriculum. regional education laboratory for the carolinas and virginia, topical papers and reprints, number 1., Apr 1968.

[7] J. R Anderson and E. Skwarecki. The automated tutoring of introductory computer programming. *Commun. ACM*, 29(9):842–849, September 1986.

[8] P.l. Brusilovsky. Intelligent tutor, environment and manual for introductory programming. *Educational and Training Technology International*, 29(1):26–34, 1992.

[9] Stephen Cooper, Yoon Jae Nam, and Luo Si. Initial results of using an intelligent tutoring system with alice. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, page 138–143, New York, NY, USA, 2012. Association for Computing Machinery.

[10] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas Van Binsbergen. Ask-elle: an adaptable programming tutor for haskell giving automated feedback. *International Journal of Artificial Intelligence in Education*, 27(1):65–100, 2016.

[11] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. Intelligent tutoring systems for programming education: A systematic review. In *Proceedings of the 20th Australasian Computing Education Conference*, ACE '18, page 53–62, New York, NY, USA, 2018. Association for Computing Machinery.

[12] John Nesbit, Li Liu, Qing Liu, and Olusola Adesope. Work in progress: Intelligent tutoring systems in computer science and software engineering education. *2015 ASEE Annual Conference and Exposition Proceedings*, Jun 2015.

[13] Cal-CS-61A-Staff. Cal-cs-61a-staff/cs61a-apps.

[14] Craig Zilles, Robert Deloatch, Jacob Bailey, Bhuwan Khattar, Wade Fagen, Cinda Heeren, David Mussulman, and Matthew West. Computerized testing: A vision and initial experiences. *2015 ASEE Annual Conference and Exposition Proceedings*, Jun 2015.

[15] Craig Zilles, Matthew West, and David Mussulman. Student behavior in selecting an exam time in a computer-based testing facility. *2016 ASEE Annual Conference & Exposition Proceedings*, Jun 2016.

[16] George|Sottile Watson. Cheating in the digital age: Do students cheat more in online courses?., Nov 2009.

[17] Donald L Mccabe. Cheating among college and university students: A north american perspective. *International Journal for Educational Integrity*, 1(1), 2005.

[18] Curtis G. Northcutt, Andrew D. Ho, and Isaac L. Chuang. Detecting and preventing "multiple-account" cheating in massive open online courses. *Comput. Educ.*, 100(C):71–80, September 2016.

[19] Ap computer science principles, Jan 2021.

[20] Steven M. Downing and Thomas M. Haladyna. *Handbook of test development*. Erlbaum, 2006.

[21] Pedro Henriques Abreu, Daniel Castro Silva, and Anabela Gomes. Multiple-choice questions in programming courses: Can we use them and are students motivated by them? *ACM Trans. Comput. Educ.*, 19(1), November 2018.

[22] Clark David. Testing programming skills with multiple choice questions. *Informatics in Education*, 3(2):161–178, 2004.

[23] Bernat Romagosa. Welcome to snap!

[24] Alan C. Bugbee. The equivalence of paper-and-pencil and computer-based testing. *Journal of Research on Computing in Education*, 28(3):282–299, 1996.

[25] PrairieLearn. Prairielearn.

[26] PrairieLearn. Prairielearn/prairielearn.

[27] Joseph D. Novak and Alberto J. Cañas. The theory underlying concept maps and how to construct and use them. *Technical Report IHMC CmapTools 2006-01 Rev 01-2008*, 2008.

Appendix A

Onboarding new QG authors

To onboard prospective QG authors, the process is relatively straightforward. First, the maintainer of the respective course repository should give the proper permissions to the authors. If the repository is part of ACE Lab, the maintainer can simply add the user to the team corresponding to the course on GitHub. Existing QG authors should help the new authors setting up their PL docker image and getting it running, simply follow the steps outlined under Installing PL for local development[1] in Installing and running PL locally section[2].

New QG authors should *carefully* read and understand contents in Question Configuration[3], Elements for writing questions[4] (can skim), clientFiles and serverFiles[5] sections in PL's documentation. More specifically, new QG authors should understand the following key features of PL to get started with writing QGs:

1. Contents of `info.json` along with metadata the course uses.
2. Basics of HTML and preferably the Bootstrap library[6] for `question.html` file.
3. Enough of Python to get started with `server.py`, and preferably Mustache 5[7] to help customize the `question.html` file.
4. How to use common PL elements, such as `pl-multiple-choice`, `pl-checkbox`, `pl-figure` etc. and checkout examples of them in PL's `exampleCourse`[8].

[1] `https://prairielearn.readthedocs.io/en/latest/installing/#installing-pl-for-local-development`
[2] `https://prairielearn.readthedocs.io/en/latest/installing`
[3] `https://prairielearn.readthedocs.io/en/latest/question`
[4] `https://prairielearn.readthedocs.io/en/latest/elements`
[5] `https://prairielearn.readthedocs.io/en/latest/clientServerFiles`
[6] `https://getbootstrap.com`
[7] `http://mustache.github.io/mustache.5.html`
[8] `https://github.com/PrairieLearn/PrairieLearn/tree/master/exampleCourse/questions/element`

5. Conditional Elements of PL, such as `pl-question-panel`, `pl-submission-panel`, **pl-answer-panel** etc., especially their effects.

It seemed helpful to have the new authors try to write a simple QG first before moving on to more ambitious QGs. Additionally, it is recommended for new authors to read and understand some past QGs that might be similar to QGs they're planning on writing.

Appendix B

PL Element Guide

Depending on the purpose of the PL element, the developer should obtain a local copy of PL source code. The easiest way I found was following directions in installing with local source section[1] of PL's developer documentation. Once the local instance works, the question element writing section[2] covers the inner workings of PL's elements. It is best to read and try to understand source code for existing elements before attempting to start writing a new element. Additionally, it'd be a good idea to reach out to PL's development team to express your interest in writing or modifying elements; the developers can add you to their Slack channel so you can ask questions and get help when you need it. PL developers are also a good source of information for existing projects that could be related, and it is generally courteous to keep the people maintaining the source code informed. This is a good idea even if the element is private, PL developers might want to include it as one of its official elements.

[1] `https://prairielearn.readthedocs.io/en/latest/installingLocal`
[2] `https://prairielearn.readthedocs.io/en/latest/devElements`

Appendix C

Example `infoAssessment.json`

```
{
    "uuid": "c0576193-3a56-43b0-abaa-25fb4ff31523",
    "type": "Exam",
    "title": "Demo",
    "set": "Practice Exam",
    "number": "1",
    "allowRealTimeGrading": false,
    "autoClose": false,
    "allowAccess": [
        {
            "role" : "TA",
            "credit": 100,
            "startDate" : "2020-09-11T00:00:01",
            "endDate" : "2020-12-14T23:59:00"
        },
        {
            "role" : "Student",
            "credit": 100,
            "password": "snap",
            "startDate" : "2020-09-11T00:00:01",
            "endDate" : "2020-09-13T23:59:00"
        }
    ],
    "zones": [
        {
            "title": "Directions, READ FIRST!",
            "questions": [
                {"id": "PL-demo/DIRECTIONS", "points": 0}
            ]
        },
        {
            "title": "The pledge you will need to sign...",
            "questions": [
                {"id": "PLEDGE", "points": 0}
```

```
            ]
        },
        {
            "title": "Example multiple-choice",
            "questions": [
                {"id": "PL-demo/demo-multiple-choice", "points": 1}
            ]
        },
        {
            "title": "Example checkbox",
            "questions": [
                {"id": "PL-demo/demo-checkbox", "points": 1}
            ]
        },
        {
            "title": "Example fill-in-blank",
            "questions": [
                {"id": "PL-demo/demo-fill-in-blank", "points": 1}
            ]
        }
    ]
}
```

Listing C.1: an example `infoAssessment.json` file similar to what we might use for an exam.

Appendix D

Personal Reflections

Throughout my journey, the most important thing I learned was the value of companionship. All aspects of my work would not be possible without the support, help, and collaboration of those around me. I was truly fortunate to have everyone to be part of my life. The rest of this section summarizes my personal learnings that I believe are worth sharing.

When I took responsibility for quizzes and exams while working for CS10, my QGs could have a significant impact on students' grades. I was inclined to see concepts from students' perspectives and identify mis-conceptions to create questions that actually test understanding. I believe working on exam questions made me a better teacher and helped me gain a deeper appreciation for the process of writing exams. It was very helpful to document things we did, whether it was how QGs worked, the number of variants, the intended difficulty, etc., documentation was what allowed us to avoid careless mistakes and made other people's jobs easier. Additionally, whenever a bug was discovered with a QG, it was paramount to fix it right away; instead of putting it off. There was a ton of work to be done to keep the class running smoothly, and there would be a ton of work to make the class better in the future. Whenever a mistake is found, fix it right away.

After mentoring two groups of students on separate projects, I realized the importance of communicating deadlines and expectations. This might seem cliche, but it's true. Mentorship on projects posed challenges *as well as pressure and anxiety* to students. From their perspective, the difficulty of upcoming tasks was never obvious, so their time-management would typically be less ideal. However, as the mentor, I had a more realistic understanding of potential challenges. I found it helpful to outline the exact steps students should take to approach their problems and pointing out to them any potential problems that might arise, *and* how to solve them. Additionally, it never hurts to repeat the same information over and over, some of the "obvious" details might be quite nuanced to students, and the repetition of information gives students time to process and realize these details to ask questions. Lastly, it was important to demonstrate humility and understanding. Students could have various problems outside of school, and they typically aren't completely comfortable with sharing their potential issues. It is important to demonstrate understanding, flexibility, and room for error to make students' lives not that much harder.

At the end of my program and having had a few years of experience in CS education, I consider myself an educator. From my perspective, besides the teaching of knowledge and the completion of responsibilities, one of the most helpful and humane things an educator can do for students is seeing things from their perspective. Teaching and my companionship with my peers helped me realize the incredible privileges I've enjoyed in life. As a student myself, I wouldn't have gotten this far without tremendous luck and the loving people around me. As a result, it is so important to try to understand students from their perspective because *each student is a unique human being.* I was very glad CS10 was able to introduce the extended exams format, and based on our survey results, it was obvious students loved it. I wish from the bottom of my heart that those reading this report would try their best to accommodate students and create an enjoyable learning environment for everyone.