NEWSLETTER | The American Philosophical Association

Teaching Philosophy

SPRING 2021

VOLUME 20 | NUMBER 2

ISSN 2155-9708

APA NEWSLETTER ON

Teaching Philosophy

TZIPORAH KASACHKOFF AND EUGENE KELLY, CO-EDITORS VOLUME 20 | NUMBER 2 | SPRING 2021

FROM THE EDITORS

Tziporah Kasachkoff
THE GRADUATE CENTER, CITY UNIVERSITY OF NEW YORK

Eugene Kelly
NEW YORK INSTITUTE OF TECHNOLOGY

We welcome our readers to the spring 2021 edition of the *APA Newsletter on Teaching Philosophy*. We offer this month two articles and some poetry.

Our first paper is "Teaching Some Philosophical Problems through Computer Science." The authors are Daniel Lim, a professor, and Jiaxin Wu, an educational consultant, at Duke Kunshan University in China's Jiangsu Province. They outline a strategy for introducing students to philosophy via computer science. To that end, they show how some problems typical of philosophy emerge from conundrums that computation experts encounter in their efforts to understand the nature and structure of the relatively new discipline of computer science. The underlying pedagogical concept that motivates the authors' attempt to bring two distinct disciplines in contact is that of "interdisciplinarity," which, the authors say, "makes a stark contrast between copying and pasting a conceptual formula in a narrow domain and being able to apply the same concept in a different domain. . . . The ability to think across disciplines may be necessary in confronting many of the pressing problems facing our world."

Students are given computational procedures to implement using Python, a readily available programming language that can be related to some philosophical problems: the functionalist theory of mind, the problem of induction, the problem of the external world and of personal identity, and the validity of (the kalam version of the) cosmological proof of the existence of God. For example, the notion of recursion in computer science, the authors argue, is an entryway into reflections on the notion of infinity—which functions in the kalam argument. The paper concludes with reflections on how the course was evaluated and how it could be improved. One such improvement, they note, might involve using Python to restate and explore the logic of arguments. The paper contains a bibliography.

Our second paper, "Schematics for the Syllogism: An Alternative to Venn," is by Wallace A. Murphree, Professor Emeritus at Mississippi State University. Prof. Murphree describes and argues for the pedagogical value of a computer program he designed for teaching syllogistic logic to college and secondary-school students. The force of his argument is more clearly visible to those who download and enter the program (to which there is a link in the text). The website contains both the program and an extensive series of help screens to lead readers into the program's capabilities so they can quickly familiarize themselves with the techniques of its analysis of the categorical syllogism and assess its pedagogical value. Moreover, Prof. Murphree claims that the schematics of his program permits the easy extension of instruction to the numerically expanded logic for which they were originally designed in his book, *Numerically Exceptive Logic: A Reduction of the Classical Syllogism* (New York: Peter Lang 1991). The author also demonstrates that the representation of syllogisms made possible by his schematics is equivalent to their representation by Venn diagrams. The schematics permit an extension beyond the capacities of the Venn diagrams in that they can make visible as well as check the validity of arguments with numerically flexible quantifiers. Prof. Murphree ends with the enthusiastic invitation to logic instructors "to explore this alternative to the traditional Venn diagram."

Finally, we happily welcome back to our pages Prof. Felicia Nimue Ackerman of Brown University. She offers us two poetic reflections on teaching during the pandemic.

Those of our readers who would like to write of their experiences as teachers for our publication are welcome to do so. We are also glad to consider articles that respond, comment on, or take issue with any of the material that appears within our pages.

We encourage our readers to suggest themselves as reviewers of books and other material (including technological innovations) that they think may be especially good for classroom use. Though we normally list books and other materials that we have received from publishers for possible review in our Books Received section, reviewers are welcome to suggest material for review that they have used in the classroom and found useful. Please remember that our publication is devoted to pedagogy and not to theoretical discussions of philosophical issues. This should be borne in mind not only when writing articles for our publication but also when reviewing material for our publication.

SUBMISSION GUIDELINES

All papers should be sent to the editors electronically. The author's name, the title of the paper and full mailing address should appear on a separate page. Nothing that identifies the author or his or her institution should appear in the body or the footnotes of the paper. The title of the paper should appear on the top of the paper itself.

Authors should adhere to the production guidelines that are available from the APA. For example, in writing your paper to disk, please do not use your word processor's footnote or endnote function; all notes must be added manually at the end of the paper. This rule is extremely important, for it makes formatting the papers for publication much easier.

All articles submitted to the newsletter undergo anonymous review by the members of the editorial committee:

Tziporah Kasachkoff
The Graduate Center, CUNY
tkasachkoff@yahoo.com, co-editor

Eugene Kelly
New York Institute of Technology
ekelly@nyit.edu, co-editor

Robert Talisse
Vanderbilt University
robert.talisse@vanderbilt.edu

Andrew Wengraf
andrew.wengraf@gmail.com

Contributions should be sent to:

Tziporah Kasachkoff, Philosophy Department, CUNY Graduate Center, 365 Fifth Avenue, New York NY 10016, at tkasachkoff@yahoo.com

and/or

Eugene Kelly, Department of Social Science, New York Institute of Technology, Old Westbury, NY 11568, at ekelly@nyit.edu

ARTICLES

Teaching Some Philosophical Problems through Computer Science

Daniel Lim
DUKE KUNSHAN UNIVERSITY

Jiaxin Wu
DUKE KUNSHAN UNIVERSITY

During the second seven-week session of the spring 2020 semester at Duke Kunshan University (DKU is located in Kunshan, China and was established as a partnership between Duke University in the United States and Wuhan University in China), Daniel Lim developed and taught a 2-credit course (17.5 hours of class time) under the title "Philosophy Through Computer Science." One aim of teaching philosophical problems through teaching computer science was to offer an interesting course that attracted students wanting to explore one or both of these disciplines. Consequently, the course was targeted at undergraduates (probably first-year students) with little or no experience with either discipline. The course was an elective and did not count toward the requirements for either discipline. It did, however, satisfy the university's quantitative reasoning requirement, a requirement intended to ensure that all students graduate with critical skills in quantitative analysis and deductive reasoning.

Another aim was to enrich the philosophical learning experience by connecting philosophical issues with computational concepts. Some computational concepts were used as launching points for philosophical discussion while others were used to illuminate philosophical ideas in ways that were intended to be less abstract (and, for some, less fantastic). To explore this, Jiaxin Wu, an educational consultant from the Center for Teaching and Learning, conducted a midterm Small Group Instructional Feedback (SGIF)[1] to collect student feedback and observed the class to help assess its pedagogical effectiveness in improving interdisciplinary learning.[2]

Eighteen students enrolled and successfully completed the class. Eleven students had never taken a philosophy class before, and ten students had never taken a computer science class before. So, more than half of the students were either unfamiliar with philosophy or with computer science. Two students would not have taken a philosophy class if it had not been taught through computer science, and ten students were unsure—they may have taken a philosophy class anyway.

To assess their prior knowledge of philosophy and computer science, the students took a pretest before the class began, which focused on topics that would be covered in the class. Mirroring their reported experience in these fields, a little more than half the students missed or claimed ignorance in response to the questions. For example, when asked to convert a decimal number (111_{10}) into its binary counterpart (1101111_2), a third of the students had no idea what to do, and a third of the students gave incorrect answers. When asked about what external world skeptics claim, fewer than half of the students were able to correctly identify that skeptics deny knowledge of the external world. Finally, when asked about David Hume's views regarding induction, more than two thirds of the students answered incorrectly, thinking induction was based on *a priori* reasoning or that Hume believed that it was rationally grounded.

AUTHOR'S BACKGROUND

After completing degrees in computer science at both the undergraduate and graduate levels, Daniel worked as a programmer for several years. Because of his obsession with philosophical questions, however, his career as a programmer was short-lived. After only a couple years, he

quit his job, went back to school, and completed a PhD in philosophy at Cambridge University in 2011. He has been teaching philosophy at the university level since 2012 and from 2018 (though he is part of the humanities division at DKU) has had (because of student demand) the privilege of teaching introductory courses in computer science.

Having had the content of both fields simultaneously (and continuously) stewing in his mind over the past couple years, he was able to appreciate several areas of conceptual overlap. It's what inspired him to publish two papers in the journal *Teaching Philosophy* titled "Philosophy Through Computer Science" (2019) and "Philosophy Through Machine Learning" (2020). In those papers he explored ways that specific topics in computer science (e.g., recursion) might shed light on classic issues in philosophy (e.g., the existence of God). Because there were a number of such topics that were readily available to be exploited, he made an attempt to turn these ideas into a course.

Marvin Minsky, one of the founders of artificial intelligence research, is credited with saying, "You don't understand anything until you learn it more than one way." We take this to mean that being able to acquire a concept through different points of view is essential to truly mastering it. While we wouldn't go so far as Minsky, the point that learning something in more than one way can be helpful for deepening understanding is a point well taken. Moreover, appreciation of a given field may be heightened and broadened by appeal to one's understanding of the possible relations between that field and another one.[3]

In today's intellectual climate, where the term "interdisciplinarity"[4] is often used, the ability to see connections outside of the context in which a concept is acquired is becoming increasingly important. Not only does this mark a stark contrast between copying and pasting a conceptual formula in a narrow domain and being able to apply the same concept in a different domain, the ability to think across disciplines may be necessary in confronting many of the pressing problems facing our world. The interest in and demand for interdisciplinarity forms an important part of the basic pedagogical motivation behind the construction of this course.

DESCRIPTION OF THE COURSE AS IT WAS TAUGHT

Here are the learning objectives for this course:

1. Summarize and critique philosophical positions on the following topics: external world skepticism, personal identity, the functionalist theory of mind, the existence of god, and the problem of induction.

2. Write rudimentary programs in Python using functions, recursion, and linear regression.

3. Collaborate with others in logically assessing a philosophical issue and putting the issue in natural language that a layperson can understand.

4. Present in oral form an opinion for which one offers justifying reasons.

The textbook used for the computer science elements of the course was John Guttag's *Introduction to Computation and Programming Using Python* (MIT Press, 2016). Select chapters were used from the first half of this book. (In a future iteration of this course a different textbook, Charles Severance's *Python for Everybody*, might be used. Not only is it freely downloadable [https://www.py4e.com/book.php], it is accompanied by exercises and video lectures [https://www.py4e.com/lessons] that can serve as an excellent supplement to the course lectures.) For the philosophical elements of the course, a hodge-podge of summary-style articles and book chapters (for example, the editors' introductions to sections on knowledge and induction from *The Norton Introduction to Philosophy* [2018]) were used to introduce philosophical issues. There follows a breakdown of the topics covered.

WEEK 1: PYTHON SETUP AND ABSTRACTION

Time was spent helping students install Python programming environments on their computers. Python was chosen for its ease of use and its being (arguably) the most popular programming language today.[5] A programming language such as Python is a way for humans to translate actions that they wish the computer to perform into commands that a computer can "understand" and execute. Of course, computers do not literally "understand" the instructions, nor, indeed, do they "understand" anything at all.[6] All that computers do is perform certain actions on receiving certain information provided in accordance with a strict grammar. Nevertheless, it is a powerful tool that, in essence, turns any computer into a *universal* machine—a machine capable of performing *any* task that *any* other machine can do.

We then introduced abstractions in computer science—how computer programs are useful ways of representing a series of 1s and 0s which, in turn, are ways of representing electrical signals. This was then tied into a discussion of external-world skepticism by way of noting that no matter how hard we study a program, it will never yield knowledge of its "true" nature as electricity. This served as a nice analogy for discussing the disparity that sometimes exists between how things may appear to our senses and what these things truly are.

WEEK 2: PYTHON BASICS

The entirety of this week was spent on developing familiarity with the basic elements of Python. Simple data types (integers, floats, Booleans, and strings) and flow-of-control structures (e.g., conditionals and loops) were covered.[7]

WEEK 3: EQUALITY AND IDENTITY

The equality operator, which checks to see if two variables (or expressions or constants) amount to the same value, was used to introduce students to the problem of identity. Because sameness of value does not always imply sameness of object, a discussion of the distinction between numerical and qualitative identity naturally followed. This point was then used to discuss some puzzles regarding identity using, among other things, the Ship of Theseus.[8]

WEEK 4: FUNCTIONS AND FUNCTIONALISM

Time was spent learning how to define functions and

how to use these functions to simplify tasks through modularization. Students were then introduced to the idea of treating images as two-dimensional structures of color pixels and treating colors as a set of numbers that represent the intensities of the colors red, green, and blue. This gave students the ability to perform interesting image-manipulation tasks such as using chroma key effects (sometimes referred to as "green screening," a technique often used in movies to combine two images or video streams to produce the illusion that a given actor is in a different location). Though the image-manipulation tasks were covered primarily as a practical application of the programming concepts acquired so far, it could easily have been used to extend discussions of external-world skepticism.

The critical computational concept introduced this week was that of a function, a concept that was then used to begin a discussion of the functionalist theory of mind.[9] For example, students were asked to write a function, let's call it Unique, that takes a list of numbers as input and generates a new list with duplicate numbers removed as output. If given [1,2,3,3,3] Unique returns [1,2,3]. It becomes evident to students that this function can be implemented in different ways.

One way of implementing unique is to create an empty list ([]), a list without any members, and, by sequentially going through the numbers in the original list, insert into the new list only those numbers that don't currently exist in the new list. Since 1 does not exist in [], the new list becomes [1]. Since 2 does not exist in [1], the new list becomes [1,2]. Since 3 does not exist in [1,2], the new list becomes [1,2,3]. Since the next 3 exists in [1,2,3], the new list remains the same. Finally, the last 3 exists in [1,2,3], so the new list remains the same. The new list can then be sorted so that it will be presented in ascending order.

A different way of implementing Unique would be to create a new empty list ([]) and, by sequentially going through the numbers in the original list, insert into the new list only those numbers that don't appear in the remaining portion of the original list. Since 1 does not exist in [2,3,3,3], the new list becomes [1]. Since 2 does not exist in [3,3,3], the new list becomes [1,2]. Since 3 exists in [3,3], the new list remains the same. Since the next 3 exists in [3], the new list remains the same. Since the final 3 does not exist in [], the new list becomes [1,2,3]. Like the previous implementation, this new list can then be sorted so that it will be presented in ascending order.

We see that the same function, unique, can be implemented in at least two different ways. A bit of reflection on this served as a launching point for students to think about the concept of multiple realizability—the idea that the same mental state can be realized in different ways. This was then used to introduce the functionalist theory of mind as a means of accommodating the multiple realizability of mental states.

WEEK 5: RECURSION AND FILE MANIPULATION

Part of the time spent in this week was focused on recursive programming. A recursive program, as opposed to an iterative program, is one that uses *itself* in order to solve a problem. So, for example, consider the problem of calculating factorials. 5 factorial (written as 5!) is the product of a given positive integer multiplied by all lesser positive integers: (1 x 2 x 3 x 4 x 5) which is 120. Notice that 5! can be calculated if one already knows the value of 4!. We can simply multiply 5 by 4! to calculate 5!. In order to calculate 4! we can, in turn, simply multiply 4 to 3! and so on until we reach a "base" case (i.e., 1!) where the recursion stops. Here is one way to code the factorial function recursively.

```
def factorial_r(n):
        if n == 1:
                return 1
        return n * factorial_r(n-1)
```

First, the function checks to see if the input number n is equal to 1. If it is, the function simply outputs 1. If n is greater than 1, the function takes the next recursive step and outputs the product of n and factorial_r(n-1). In order to calculate this, the product of (n-1) and factorial_r(n-2) is calculated and so on until the base case is reached.

An iterative solution to the problem of calculating factorials would involve the use of a looping flow-of-control structure. A looping structure can be used to specify how many times a block of code is repeatedly executed. Here is one way to code the factorial function Unique iteratively.

```
def factorial_i(n):
 result = 1
 while n > 0:
  result = result * n
 n = n – 1
  return result
```

Here a while loop is used to repeat a block of code so long as the condition n>0 remains true. In this case the block of code will repeatedly be executed n times.

The loop begins by checking to see if n>0 is true. If so, then the product of result and n is assigned to result and n is decremented by 1. On the next iteration, the loop begins by checking again to see if n>0 is true. If so, then the product of result and n is assigned to result and n is again decremented by 1. This cycle repeats until n finally reaches 0. At this point the while loop terminates because n>0 is no longer true and result is returned as the output.

The basic structure for manipulating files was then introduced to students in anticipation of their working (during the following week) with data sets for which understanding of file manipulation is critical.

WEEK 6: GOD AND VISUALIZATION

Using the distinction between recursive and iterative programming, a recent version of the cosmological argument for the existence of God developed by William Lane Craig (1979) was explored. The core argument can be formulated as follows:

1. Whatever begins to exist has a cause of its existence.

2. The universe began to exist.
3. Therefore, the universe has a cause of its existence.

One way premise 2 has been defended is by arguing that it is impossible to form an infinite by successive addition. In other words, it's impossible to count your way to an infinite since whatever number you've counted to (say, 91852105), you can always count to a higher finite number by adding 1. Given that the history of the universe is a temporal series of past events that is a collection formed by successive additions, it follows that the history of the universe cannot contain an infinite number of past events—the universe must have a beginning.

The notion of recursion serves as a nice entry into thinking about infinity because recursion suggests loops that never end. For example, if I write a recursive function without a base case, this function will run forever (if my computer had an infinite amount of memory). Furthermore, the distinction between recursive and iterative programming may provide a distinction that makes one of Craig's key intuitions in support of premise 2 questionable.

Craig argues: "In order for us to have 'arrived' at today, temporal existence has, so to speak, traversed an infinite number of prior events. But before the present event could occur, the event immediately prior to it would have to occur; and before that event could occur, the event immediately prior to it would have to occur; and so on *ad infinitum*. One gets driven back and back into the infinite past, making it impossible for any event to occur."[10]

The idea that it is "impossible for any event to occur" may only pertain to the occurrence of an infinity of events if the events occur recursively. If events occur iteratively, however, then Craig's intuition may not be true. To see why this might be the case, let's return to the factorial function. Even if both recursive and iterative functions solve factorials (as described above), they do so in very different ways. No multiplication operations will be executed in the recursive factorial function *until* the base case is reached. Consider what happens when the following statement is executed:

factorial_r(5)

This results in 5*factorial_r(4). In order to calculate this product, factorial_r(4) must first be executed. But this in turn results in 4*factorial_r(3). In order to calculate this product, factorial_r(3) must first be executed and so on. Until the base case is reached (i.e., factorial_r(1)) *none* of the multiplication operations can be calculated.

Something very different occurs when the iterative version is executed:

factorial_i(5)

In this case, multiplication operations will be executed at every iteration of the loop. In the first iteration 1*5 is executed (since result is 1 and n is 5) and stored as 5. In the second iteration 5*4 is executed (since result is 5 and n is 4) and stored as 20. In the third iteration 20*3 is executed (since result is 20 and n is 3) and stored as 60 and so on. Consequently, multiplication operations are calculated on each iteration of the loop starting with the very first one.

These functions can be modified so that the same statements (factorial_r(5) and factorial_i(5)) enter infinite loops. For the recursive function we can simply remove the base case. For the iterative function we can change the while loop so that the condition (n>0) is replaced with the Boolean value, True—this way the condition is always true and the loop never terminates.

Now we can ask, will any multiplication operations ever get executed given that both functions enter infinite loops? I hope you can see, based on what was said above, that multiplication operations will get executed in the iterative version. In the recursive version, however, multiplication operations will *never* get executed. This is because no base case will ever be reached.

The rest of the course was spent familiarizing students with the matplotlib library, i.e., a set of tools for visualizing data through graphs. A library is reusable code that can be included into one's programs. (The matplotlib library was originally written by John D. Hunter, a neurobiologist, and has since become a popular way of visualizing data through Python.)

WEEK 7: MACHINE LEARNING AND INDUCTION

The final week was spent on the basics of machine learning—using data to shape a computer model that is deployed for making predictions. Though there are a variety of machine-learning techniques, we decided to use the technique of *linear* regression on account of its simplicity. Linear regression attempts to find a straight line that best summarizes the relationship between two or more sets of numerical data.

For example, consider the heights and weights of various people. In general, there is a rough correlation between a person's height and weight—the taller you are the heavier you are. One could, of course, explicitly program a computer with a model to capture this relationship via a linear equation of the form $y = mx + b$ where y stands for the y-coordinate, m stands for the slope, x stands for the x-coordinate, and b stands for the y-intercept.[11] Alternatively, one could provide a computer with lots of examples of people's heights and weights and let the computer, using linear regression, "discover" this linear equation on its own. These two approaches might be analogized with *a priori* and *a posteriori* means of acquiring knowledge. The former might be taken as *a priori* on the grounds that it is "knowledge" that is simply inserted into the computer, perhaps even before the computer is put to use. The latter may be taken to be *a posteriori* on the grounds that it is knowledge that is acquired via experience, namely, the experience of being exposed to lots of examples.

Given that machine-learning algorithms rely on *a posteriori* means of acquiring knowledge it is natural to ask: Why should we trust the subsequent inferences that are made based on the models that these machine-learning algorithms generate? No matter how well a model is made to fit e*xisting* data, what reason do we have to think that

the model will continue to capture future, still unobserved, data? This is, more or less, the essence of the problem of induction as introduced by David Hume in *A Treatise of Human Nature*.[12]

We can go on to other debates in the philosophy of science such as the following: Is there an epistemic difference between a model that "merely" accommodates existing data and one that makes accurate predictions about a phenomenon that has yet to be observed? This question is one that data scientists also ask (often implicitly) when designing their models. It's interesting that philosophers and data scientists have reached similar conclusions with respect to this.[13] They suggest that epistemic preference can be given to prediction because models that do better with prediction have a better chance of avoiding "overfitting" existing data. Overfitting occurs when a model is tuned so closely to existing data that it fails to make good predictions because it fails satisfactorily to generalize.

ASSESSMENTS

The coursework included two written reflection papers (each of 400–500 words), four programming assignments, a final exam, and a group digital poster. The written reflection papers were evenly spaced throughout the course so that students could choose one of two philosophical topics from the first half of the course (external-world skepticism and personal identity) and one of two topics from the second half of the course (functionalism and the existence of God).

These reflection papers were graded according to the degree to which they met the following four criteria: First, the philosophical position discussed (e.g., that there is good philosophical and scientific evidence for the existence of God) had to be clearly stated. Second, a reconstruction of at least one of the arguments used to support the philosophical position had to be concisely stated. Third, critical engagement with the argument (either for or against) had to be demonstrated. And fourth, the language used in the overall paper had to be both grammatically correct and easy to understand.

The programming assignments were made available through an online platform, i.e., a digital service that facilitates interactions between two or more individuals via the internet involving Jupyter Notebooks, a popular platform for providing programming environments online. Students who do not have Python installed on their personal computers can still access a Python programming environment. Moreover, additional software can be added to Jupyter Notebooks to enable automatic grading, making it possible to give students immediate feedback after the deadlines for receipt of their programming assignments.

The final exam consisted of a single question about a topic introduced in each lecture. Since there were fourteen lectures, there were fourteen multiple-choice questions. They included, among others, variations on decimal to binary number conversion, external-world skepticism, and the problem of induction. The exam was not meant to assess depth of understanding; rather, it was meant to ensure that students understood key concepts and were able to use them correctly in the contexts in which they were learned.

The group digital poster was intended to foster discussions *among students* about the way that the study of computer science might shed light on discussion of some philosophical problems. Students were asked to present a philosophical concept using a computational concept. Each group then recorded a video presentation wherein each member of the group talked about an important part of their poster. (A bit more on digital posters is detailed at the end of this paper.)

OUR REFLECTION ON THE COURSE

Based on comments given by students after taking the class,[14] as well as on our own observations, we acknowledge that there was too much content to be covered in a 2-credit class. If offered again, this class should be offered as a 4-credit class while keeping most of the content the same. Not only will this give students more time to get comfortable applying the programming concepts they've acquired, they'll have more time to slow down and read philosophy from the original texts. In the present iteration of the course, philosophical topics (such as external-world skepticism and the problem of induction) were introduced using secondary texts that were easy-to-digest summary essays. Students may get more out of dealing with, say, Descartes's and Hume's actual writings. Not only are the original texts more challenging to understand, but they may also include details that can push the relevant concepts in subtle directions that make them ripe for further discussion.

With an expansion in class credits, at least two additional lectures can be added to the schedule and dedicated to the logic of arguments (with attention to the concepts of premise, conclusion, validity, and soundness) and the art of reconstructing others' (often implicit) arguments. Attention to the logic of arguments is important to introduce early on so that students have a framework through which they can approach the reading and evaluation of subsequent philosophical texts. For example, when they go over Descartes's passages on external-world skepticism, students should be asked to reconstruct Descartes's reasoning by putting it explicitly into premise-conclusion form.

Fortunately, lectures on the logic of arguments fit naturally into the progression of ideas of the current course content. One of the early concepts taught in courses in computer programming is that of the conditional statement. Conditionals are critical for expanding the programmer's ability to manipulate the flow-of-control of a program. And they are useful for thinking through the logic of arguments since they can be seen as direct counterparts of conditionals in philosophical logic. Moreover, conditionals, in most programming languages, can take advantage of Boolean operators ("and," "or," and "not"). Interpreting a philosophical argument as a series of conditionals may be a pedagogically useful exercise.

Take the following deductive argument, for example:

1. Socrates is human.
2. All humans are mortal.
3. Socrates is mortal.

While there are no explicit conditionals here, we could reinterpret this argument using conditionals.

1. Socrates is human.
2. If Socrates is human then Socrates is mortal.
3. Socrates is mortal.

This, then, might look like the following as Python code:

```
Socrates_is_mortal = None
Socrates_is_human = True
if Socrates_is_human == True:
Socrates_is_mortal = True
```

Two Boolean variables are created (socrates_is_mortal and socrates_is_human). socrates_is_mortal is assigned a None value representing a state of ignorance regarding Socrates's mortality. Only if "Socrates_is_human" is true will "Socrates_is_mortal" be assigned a true value. Though not all arguments are smoothly rendered as a series of conditionals, many arguments can be regimented in this way.

Regarding the digital poster projects, it was hoped that students would have ventured into new and creative territory, but for the first iteration of this course, most projects were merely elaborations on connections that were already explicitly made during lectures. More thought needs to be put into this because it's not clear that students had enough preparation or time to generate anything novel. In a future iteration of this course more scaffolding assignments along the way may help students better explore the intersection of philosophy and computer science.

More generally, in future iterations of this course we will focus on several other issues. First, we will revisit the learning objectives in the context of interdisciplinarity and reflect on the long-term goals of offering this course.

Second, we hope to enhance the alignment between the learning objectives and the assessments by developing appropriate assignments to help students explore the intersection of the two disciplines—assignments that go beyond the "traditional" programming or reflection assignments that are common in courses in computer science (e.g., programming assignments) and in courses in philosophy (e.g., writing papers). Third, we will spend more time clarifying the assessment rubrics. Fourth, additional topics that foster philosophical and computational reflection will be developed.

There is definitely a growing interest in computer science, especially as we move rapidly into a world where almost everything we do is being mediated by computers. The growing interest is also evidenced in the enrollment numbers for computer science-related courses. The interest in computer science is even more pronounced when these enrollment numbers are juxtaposed with enrollment numbers for philosophy-related courses at our university. This provides additional incentive for developing classes like "Philosophy through Computer Science" that try to leverage this interest.

This class is by no means the first attempt at pedagogically combining philosophy and computer science. Oxford University and Stanford University, for example, already offer a degree in Computer Science and Philosophy.

However, the class that we describe in this paper may deliver something unique. As far as we can tell there are no classes offered in either the Oxford or Stanford programs that explicitly highlight areas of conceptual overlap between the two fields (students at these two universities take traditional philosophy courses in the philosophy department and traditional computer science courses in the computer science department). Instead of taking classes in their traditional disciplinary silos and asking students to make conceptual connections themselves, this class is built to make such connections explicit.

We end this paper by highlighting an area of conceptual overlap that seemed to be especially effective in stimulating student engagement through their digital projects. Learning about the functionalist theory of mind, after learning how to write their own Python functions, gave students a new way of thinking about functionalism. Instead of simply pondering how there might be different ways of realizing the same function, they learned how this might be done by writing code themselves. Some students, in their digital posters, shared Python code for a "replace" function that they had written in two different ways: one iteratively and one recursively. Not only did this demonstrate their understanding of multiple realizability, but it also gave them room to wonder, on the one hand, whether different ways of carrying out the "same" function really make any difference to how minds generate understanding. On the other hand, they noted a clear example in their own lives in which the way something is carried out does indeed make a difference. There is, after all, a world of difference between a student who writes code merely by copying what someone else has written versus a student who writes code herself.

CONCLUSION

This course provides a new direction in teaching a way of looking at a couple of philosophical issues that may add value to curriculum design at higher education institutions. By connecting some issues in the field of philosophy with computational concepts, (at least some) students in this course will be more interested in the subject matter of both fields, gain some philosophical/critical thinking skills, and come away with rudimentary programming competence.

APPENDIX

Midterm Student Group Instructional Feedback (SGIF) Guide

The semi-structured focus group in the form of a Small Group Instructional Feedback (SGIF) session was conducted by Jiaxin Wu from the Center for Teaching and Learning via Zoom. As a formative midcourse check-in process for gathering information from students on their learning experience, this protocol was designed to foster dialogue between students and instructors and is a transparent way for bringing student concerns to the surface in a thoughtful way.

- What do you like about this course PHIL 109?
- What has contributed to your learning?
- We want to know about the following specifics:
  - Workload of a 2-credit course
  - Remote teaching
  - Helpfulness of assignments and feedback
- Do you think learning philosophy through computer science is helping you to better understand the philosophical content presented? How?
- Do you think learning philosophy through computer science is helping you pay more attention during class? Why?
- Do you think learning philosophy through computer science is making the lectures more difficult to follow? Why?
- What suggestions do you have for making improvements?

NOTES

1. Small Group Instructional Feedback (SGIF) is a structured formative mid-course check-in process for gathering information from students about their learning experience in a course. This technique was first implemented by Dr. Joseph Clark, a professor of biology at the University of Washington in 1974. See Appendix.
2. Besides the instructor's lectures and questions, we looked into the students' responses, that is, their responses to the instructor in the form of questions, assertions, responses to other student's comments, and requests for clarification. Reading a passage or responding to rhetorical questions is not what we mean by a "genuine" learning moment, because they are the same classroom techniques that instructors use to draw student attention to the material at hand, but which requires little reflective thought on their part. Three general questions were adopted to guide the class observation protocol. (1) What are the students' oral contributions to discussion that shows their understanding or their cognitive processes, e.g., responding to a genuine question correctly, assessing the errors in their own assertions, directing critical questions at the instructor, and critically commenting on course materials or on their peer's contributions to the discussion? (2) What does the instructor or the students do, or what is the atmosphere in the classroom that leads the above kind of genuine learning moments? (3) To what extent does the genuine learning moment benefit from the interdisciplinary nature of the course? To begin answering these questions each video recorded class session was analyzed for categories and themes (e.g., "teaching a philosophical concept through a computational concept," "concept covered in pre-test," "thought-provoking philosophical question"). These categories and themes were then converted into codes (i.e., shorthand notation for themes) and compiled into a codebook that listed the codes with their corresponding themes. The codes were then used to systematically label portions of class time in order to assemble data on what was happening in class. We are still in the preliminary stages of analyzing these data.
3. Thanks to an anonymous reviewer for this point.
4. See Frodeman, *The Oxford Handbook of Interdisciplinarity*.
5. Python is considered "easier" than other programming languages (such as C) because it is pitched at a higher level of abstraction. Think of the higher-level abstraction we have as car drivers—to make a car move or stop we simply have to press or release pedals. We don't have to worry about the lower-level details of what these actions do in terms of the mechanics of the car.
6. While this is certainly a popular view it must be noted that there are exceptions—see Preston and Bishop, *Views into the Chinese Room: New Essays on Searle and Artificial Intelligence*.
7. Flow-of-control statements in a programming language allow code to be executed in a non-sequential manner. Without flow-of-control statements, the statements in a program will simply be executed one after another in the order that they appear until all the statements are executed (statement1, statement2, …). With a flow-of-control statement (such as a conditional), some parts of a program can be skipped—so control can be exerted over which statements get executed in certain parts of a program and which statements don't. Consider the following conditional flow-of-control structure: "if boolean1 then statement1 else statement2." This structure, depending on the value of boolean1, will either execute statement1 or statement2 but not both. In this way certain statements can be skipped when executing a program and the flow (of which statements get executed) can be controlled.
8. The Ship of Theseus is a thought experiment that helps one to think through the question of whether an entity (like a ship) can have all of its parts replaced and still remain the same entity.
9. The concept can usefully be thought of in terms of mathematical functions like y = f(x) which suggests an input-output specification—a mapping that relates two domains. This is generally used in math to relate two sets of numbers. But there is much more that is possible with functions in programming languages. A non-mathematical example of a function may be the act of gripping something. This is also a useful way to think about functions because it provides an example of a function that can be implemented in multiple ways. This is also relevant to functions in programming languages since there are multiple ways of implementing a function defined by an input-output specification.
10. Craig and Sinclair, "The Kalam Cosmological Argument," 118.
11. Why was 'm' chosen to represent the slope? This is a question that doesn't have a clear answer—it is standardly used in US math classes but the historical reason for this choice is unclear. See the following website for discussion on this. https://www.matematica.pt/en/faq/letter-represent-slope.php
12. See Thagard, "Philosophy and Machine Learning," where he (convincingly, we believe) makes the claim that the parts of philosophy concerned with induction constitute essentially the same field as machine learning.
13. See Hitchcock and Sober, "Predication Versus Accommodation and the Risk of Overfitting," for a detailed discussion of how models that do a better job of prediction have a higher chance of avoiding overfitting.
14. Here are two student comments: "The course was very cool. But the course in the end was a little rushed because we need to turn [in] four assignments within a week [at the end of course]." "One thing I have to mention is that the programming workload should somehow be reduced, since it is still a 2-credit course."

REFERENCES

Craig, William Lane. *The Kalam Cosmological Argument*. London: Macmillan, 1979.

Craig, William Lane, and James Sinclair. "The Kalam Cosmological Argument." In *The Blackwell Companion to Natural Theology*, edited by William Lane Craig and J.P. Moreland, 101–201. London: Blackwell, 2009.

Frodeman, Robert. *The Oxford Handbook of Interdisciplinarity*. Oxford: Oxford University Press, 2017.

Hitchcock, Christopher, and Elliot Sober. "Predication Versus Accommodation and the Risk of Overfitting." *British Journal for Philosophy of Science* 55, no. 1 (2004): 1–34.

Lim, Daniel. "Philosophy Through Computer Science." *Teaching Philosophy* 42, no. 2 (2019): 141–53.

Lim, Daniel. "Philosophy Through Machine Learning." *Teaching Philosophy* 43, no. 1 (2020): 29–46.

Preston, John, and Mark Bishop. *Views into the Chinese Room: New Essays on Searle and Artificial Intelligence*. Oxford: Oxford University Press, 2002.

Rosen, Gideon, Alex Byrne, Joshua Cohen, Elizabeth Harman, and Seana Shiffrin. *The Norton Introduction to Philosophy*. New York: W. W. Norton & Company, 2018.

Thagard, Paul. "Philosophy and Machine Learning." *Canadian Journal of Philosophy* 20, no. 2 (1990): 261–76.

Schematics for the Syllogism: An Alternative to Venn

Wallace A. Murphree
MISSISSIPPI STATE UNIVERSITY

I have developed a system of schematics that clearly displays syllogistic validity and invalidity by the way the schematics hang together. These can be sketched by hand, but I now have an interactive program which allows the representations of the eight basic propositions to be clicked and dragged into juxtaposition with each other to show what, if any, conclusion follows from any combination of them. I suggest this different approach makes syllogistic study more comprehensible than ever before. I call the program ReasonLines and its web-based version is found at www.reasonlines.com; also, a ReasonLines app is available for android and iOS devices.

An extensive tutorial is provided in the program's Help Page for beginning students of logic, but those already familiar with the syllogism should easily be able to operate it with the information provided below. It would be helpful to have the ReasonLines program open while reading the information—if not on a split screen, at least on an open tab to be able to switch back and forth. (A split screen may not show all the program's features.)

HOW THE PROGRAM APPEARS

The basic conventions used in the schematics are these:

- *Letters in white and black circles represent original (positive) terms and their (negative) complements, respectively;*
- *Green and red arrows connecting the letters represent affirmative and negative statements, respectively; and*
- *Double- and single-ended arrows connecting the letters represent convertible and nonconvertible statements, respectively.*

(Incidentally, I have used A, B, C, etc., as the default letters, but they can be changed with a click/tap.)

The layout of a schematic is a rectangle whose bottom corners have places for two terms and whose top corners have places for their respective complements. Accordingly, the categorical statements are represented by the connection of these left and right terms with the appropriate arrows across the rectangle, as follows:

All A are B = a single-ended, green arrow pointing from A to B,

No A are B and **No B are A** = double-ended, red arrow connecting A and B,

Some A are B and **Some B are A** = a double-ended, green arrow between A and B, and

Some A are not B = a single-ended, red arrow pointing from A to B.

Since each categorical statement is one of four equivalent statements (by obversion, conversion, and/or contraposition) each schematic rectangle is completed by the addition of the appropriate arrows for these equivalents. This involves connections to and/or between the complementary terms; then the rectangular arrow-clusters so completed for the statements given above make up the top row of schematics on the opening screen of the ReasonLines. (See this screen at www.reasonlines.com.)

However, not yet included are the statements below having nonA as their subjects:

All nonA are nonB,

No nonA are nonB (No nonB are nonA),

Some nonA are nonB (Some nonB are nonA), and

Some nonA are not nonB.

So when these together with their equivalents are also schematized, all of the thirty-two possible categorical statements are represented. They are the eight rectangular schematic-arrow-clusters given on the opening screen of the ReasonLines program. Clicking/tapping any one of them lists the four equivalent statements that it represents. (Again, see www.reasonlines.com.)

(It may be well to note here that these representations are exactly what is displayed on the Venn diagram. That is, the overlapping circles for A and B fix four logical spaces, viz., AB, AB′, A′B, and A′B′, and any one of the four spaces can be marked with an asterisk to display a particular proposition, or can be shaded to display a universal one. Moreover, each of these eight possibilities admits of four different statements or "readings," as an asterisk in AB may be read as "Some A are B," "Some B are A," "Some A are not nonB," and "Some B are not nonA"; and the same is the case, *mutatis mutandis*, when each of the other spaces is marked either way. Accordingly, the schematics and the Venn diagrams display the very same thirty-two statements.)

HOW THE PROGRAM WORKS

The opening screen of ReasonLines shows a "premise rectangle" above the eight schematics and the "conclusion rectangle" to the right of them. These contain default terms in place. Now if the syllogism to be tested is Barbara, viz.,

All A are B and
All B are C; so
All A are C,

the user will select the schematic having the single-ended, green arrow that will point from A to B—the top left schematic—and drag it onto the premise rectangle. (Clicking the schematic will show this to be the correct one.) Then another premise rectangle with C-terms will automatically appear alongside the first where the user will then drag the same schematic (top left) to connect B to C.

If at this point the AutoSolve button is activated, the same schematic (top left) will appear again in the conclusion rectangle connecting A to C to represent "All A are C." Of course, the three other equivalents of "All A are C" are represented as well—just as they are on the Venn diagram (although they may be more difficult to discern on the Venn). However, if AutoSolve is off and the schematic is manually inserted for the conclusion, ReasonLines will affirm it; but if any other schematic is selected instead, it will be marked incorrect.

One can tell which, if any, conclusion follows from a set of premises by noting the green arrows of the premises. *Specifically, if the tip of one green arrow connects to the tail of another at a middle term, then those premises do yield a conclusion; and what that conclusion is, is represented by the schematic that properly connects the extreme terms of those arrows when the middle terms are eliminated.* If the green arrows of both premises are single-ended, then the proper connection requires a single-ended green arrow in the same direction, while if one of them is double-ended, the proper conclusion arrow is double-ended. Incidentally, when both premises of a valid syllogism are universal as in Barbara, each term—white and black—of one extreme will be connected to one term of the other extreme by its own "green arrow path." One of these paths will run through the white middle term and the other through the black one. Activating the Hints button (or question mark) located beside the conclusion rectangle will change the red and disconnected green arrows to a dim gray color in order to leave any complete "green arrow path" between the extremes highlighted.

Moreover, if the argument has more than two premises—i.e., if it is a sorites—the schematics are to be read the same way. That is, when there is an unbroken "green arrow path" formed by a tip-to-tail linkage at each middle term, all the middle terms can be eliminated, leaving the extreme terms for the conclusion. But the path may not be broken. When the sorites is long and the busy schematics tend to obscure the situation, the Hints function can be useful.

The schematics display the validity of syllogisms that depend existential presuppositions by employing Fred Sommers' insight that "Some A's exist" can be expressed in categorical logic as "Some A are A"; that is, the presupposition is made explicit by the addition of this premise and the argument is handled as a sorites. Accordingly, since

All A are B, and
All B are C; so
Some A are C

requires the presupposition that some A's exist be valid, the premises are schematized as

Some A are A
All A are B and
All B are C

and the conclusion schematic will show that

Some A are C.

And this method holds for any other universal term required to be existential to make the syllogism valid.

(Note: Letters can be changed by clicking/tapping them and schematics can be entered to the left as well as to the right of the original one.)

THE VENN DIAGRAMS AND THE SCHEMATICS

The Venn diagrams and the schematics display the very same information for the basic syllogisms, and a lattice of connected asterisks placed in the logic areas of a circle of the diagram accommodates conclusions by existential presupposition. But although they seem theoretically equal as far as the basic syllogism goes, I strongly recommend the schematics over the diagrams on two basic counts: 1) I propose that the schematics expose the workings of the basic syllogism more efficiently and more clearly and 2) I propose that the schematics hold a greater utility for categorical reasoning that extends beyond the basic syllogism.

1) For one thing, the equivalent statements of a proposition are easier to distinguish on a schematic than on a diagram, and this makes schematizing "scrambled" arguments much easier than diagramming them. For example, if

All nonB are nonA, and
No B are nonC; so
All A are C

is the argument to be tested, students would schematize it directly and immediately see it to be valid, whereas they likely would need to go through the immediate inferences necessary to "reduce" it to Barbara, viz.,

All A are B, and
All B are C; so
All A are C

to be able assess it by the diagram.

Moreover, students often have trouble diagramming the I and O propositions onto a 3-circle matrix, while these propositions cause no special difficulty on the schematics.

In addition, [quite apart from the AutoSolve feature] schematized premises seem easier to "read" than diagrammed premises. That is, whether any conclusion follows at all is seen at a glance on the schematics by noting whether green arrows meet tip-to-tail at a middle term; and, when they do, the conclusion is clearly indicated by the way those green arrows connect the extreme terms. So at least in these areas the schematics seem more user-friendly than the diagrams.

Furthermore, the schematics reveal how valid arguments take different forms. For example, for any valid argument with a particular conclusion, its Darii (AII-1) form and its Datisi (AII-3) form are always represented by the terms in the green arrow path—according to which converse of the I proposition is the stated one. And for any valid argument with a universal conclusion its two Barbara (AAA-1) forms are always represented by the terms of the two green arrow paths.

Also, whether a term is distributed in a proposition is indicated by whether it lies at the tip of a red arrow or a green one. That is, a red arrow points to the predicate of a negative (=distributed) while a green arrow points to the predicate of an affirmative (=undistributed).

So, I do find the schematics preferrable to the diagrams for basic syllogisms in these respects.

2) Furthermore, I propose that the schematics hold a greater potential for use beyond the basic syllogism. One way is in dealing with sorites. That is, breaking a 4- or 5-premise sorites into its ingredient arguments and diagramming them individually is tedious and time-consuming and tends to splinter the integrity of the argument, whereas the schematization of the sorites is streamlined and allows it to be surveyed as a whole. Moreover, there is no limit to the number of premises that may be accommodated by the schematics (although, of course, the ReasonLines program has computational limits). Furthermore, when multiple premises are combined with the diagramming difficulty of complementary terms mentioned above the problem becomes more severe for the diagrams. For example, displaying the validity of the following argument would likely require extraordinary effort using the diagrams but, although longer than a standard syllogism, it would not be any harder to schematize.

All B are A,
All nonB are C,
Some nonC are not nonD, and
No D are E; so
Some nonE are not nonA.

This is not to suggest that introductory courses should give more coverage to arguments with multiple premises but rather to point out that the schematics make this a more viable option if it is desired.

Another extension of the schematics beyond the diagrams—and the one that has been most important to me—is their capability of handling numerically flexible quantifiers, such as in:

At least 10 A are B, and
All but 6 B are C; so
At least 4 A are C.

In fact, it was for this specific purpose that I devised the schematics since the Venn diagrams seemed inadequate here. The general theory I proposed[1] is that categorical claims are inherently—although often only implicitly—numerical in that

All A are B *literally means* All but 0 A are B,
No A are B *literally means* None but 0 A are B,
Some A are B *literally means* At least 1 A is B, and
Some A are not B *literally means* At least 1 A is not B.

Of course, today this is the generally accepted rendition of the particulars and it may be speculated that it was not recognized for the universals originally because the Greek numerals did not contain a zero. But be that as it may, the traditional statements can appropriately be so considered, and as such they constitute the *terminal contingent instantiations* of

All but x A are B,
None but x A are B,
At least x A are B, and
At least x A are not B.

(These are *terminal contingent instantiations* since numbers smaller than 0—i.e., negative numbers—result in necessarily false universal propositions while numbers smaller than 1—i.e., 0 and negative numbers—result in necessarily true particular propositions.)

Then on this rendition, Barbara becomes

All but 0 A are B, and
All but 0 B are C; so
All but 0 A are C,

which is but one of an infinite number of possible instantiations of

All but x A are B, and
All but y B are C; so
All but x+y A are C.

Also then Darii becomes

At least 1 A is B, and
All but 0 B are C; so
At least 1 A is C,

which, along with the above example concluding "At least 4 A are C," are but two instantiations of

At least x A are B, and
All but y B are C; so
At least x-y A are C.

Likewise, both the multiple-premise argument above and, for example,

All but 7 B are A,
All but 19 nonB are C,
At least 55 nonC are not nonD, and
None but 23 D are E; so
At least 6 nonE are not nonA.

are but two possible instantiations of

All but w B are A,
All but x nonB are C,
At least y nonC are not nonD, and
None but z D are E; so
At least (y-(w+x+z)) nonE are not nonA.

Moreover, infinitely many squares of opposition and other appeals reside in this numerically expanded terrain. But, again, I am not suggesting that introductory courses include such studies. Rather, my claim is that, unlike the Venn diagrams, the schematics provide a gateway into this region for any who may desire to enter. As mentioned earlier, the schematics were specifically designed for this purpose and the capacity can be activated in the ReasonLines program by clicking/tapping the Use Numbers button. (Part Two of the Help Page's tutorial develops this expanded feature systematically.)

Perhaps the greatest drawback to the schematics is that the eight initial designs may be somewhat daunting. However, I think this is not as formidable as it may initially appear. First, all one needs to know in order to use the program is which individual arrow represents which specific statement. Then as the program is used one should become progressively more familiar with the details of each of the eight complete schematics, and the symmetry of the scheme should emerge with progressive clarity. For example, as they are laid out on the screen it can be seen that

(1) the four schematics of the bottom row are the same as those of the top row *flipped down;*

(2) four schematics are basically green (i.e. have two green arrows) and four are basically red (having two red arrows); and

(3) four schematics have a basic Z-shape and four a basic X-shape.

With these distinctions any schematic can be uniquely identified—and *visualized*—by naming its (1) position, (2) color, and (3) shape. For example, the top left schematic is the Up-Green-Z while the next to the bottom right is the Down-Red-X, and so on for the other forms; and users should soon become acquainted with each of these eight schematics individually.

IN CONCLUSION

It is unfortunate that I have no empirical data comparing student achievement using the two methods for the syllogism. The fact is that I did not get the ReasonLines apps produced until after I retired and I have just recently succeeded in getting the web based version (www.reasonlines.com) launched.

In the few times I was able to teach numerically quantified logic before I retired, I had advanced students hand-draw the schematics as they were needed. It was they who suggested schematic cards might be prepared in advance as a way to introduce younger students to the basic syllogism, and since retirement I have had tremendous success in my opportunities to volunteer-teach gifted middle school students using the method.

So, it is on basis of that experience—together with the intrinsic perspicuity of the system and the considerations mentioned above—that I am soundly convinced the adoption of the schematics can greatly facilitate the instruction of categorical logic. I recommend it most enthusiastically.

AFTERWORD

I have never used the schematics in a standard logic class in college.

My initial point of departure was the contention that categorical statements are—or at least can be considered to be—inherently numerical, although this is often implicit. That is, the traditional A, E, I, and O statements can be cast as

All but 0 A are B.
None but 0 A are B,
At least 1 A is B, and
At least 1 A is not B.

Once this scheme is adopted the system extends infinitely as the zeroes and ones are replaced by larger numbers. (Smaller numbers result in tautologies for the particulars and necessarily false universals.) Moreover, with larger numbers infinitely many sets of contradictories, contraries, subcontraries, super- and subalterns and perfect squares of opposition hold, and each general "form-type" (such as Barbara) accommodates infinitely many instantiations. I found I could not display these variations on the Venn diagram, so I devised the system of schematics to display these. But I felt my "discovery" was the numerical logic and the schematics were purely subsidiary. They were not designed to be a rival to the Venn diagram for the basic syllogism.

My *Numerically Exceptive Logic: A Reduction of the Classical Syllogism* (Peter Lang), which relied on the schematics, was published in 1991 and I tried to publicize the numerically extended syllogism as much as I could until I retired in 2001. Instead of introducting the schematics In "Expanding the Traditional Syllogism" [in *Logique & Analyse*, 141-42 (1993)] and in "The Numerical Syllogism and Existential Presupposition" [in *Notre Dame Journal of Formal Logic* 38, No. 1 (Winter 1997)] I appealed to the traditional rules of the syllogism together with a special rule for the numerical quantifiers to establish validity; and in "Numerical Term Logic" [in *Notre Dame Journal of Formal Logic*, 39, No. 3 (Summer 1998)] I calculated conclusions using the plus-minus system of Fred Sommers.

However, during the decade between the book's publication and my retirement I taught the numerical

material four or five times to advanced students and they were required to use the schematics. Of course, they drew their own—line by line—as they were needed, using solid and dotted, rather than green and red arrows. Sometimes they would get confused, so they began drawing these out on cards in advance. Then it was they who claimed that with schematics prepared in advance, the traditional syllogism would become easy enough for precollege students to handle!

This idea seemed right to me, so a student and I took some prepared cards to a gifted section at the local middle school and we were impressed by how easily they seemed to embrace the concepts. Being encouraged, I hired a couple of computer science students to create a web site of the schematics—which was a primitive version of what is now www.reasonlines.com. However, it did not become operative until about the time of my retirement in 2001. Then, upon retirement, my wife and I moved back to our original home community in rural Alabama where I forewent most matters academic until she died in 2013.

After her death I turned my attention back to the schematics and was allowed to present the syllogism to two local classes of gifted middle school students for a semester. Although there was remarkable success, I was confronted with frustrations all along the way. I was especially dismayed to find that a Java update had security-blocked the website the computer science students had created for me. But these students, now in business on their own as Concept House, Inc., provided a "work-around" which allowed me to access the program, although it took time to get it set up each class and it would crash from time to time. Yet when it worked it seemed exactly what the class needed. But in addition to the untrustworthiness of the program, a big drawback was the limited access students had to the one computer.

However, since it seemed each student had a "smart phone," I contracted with Concept House to create an app of the schematics for iOS and android hand-held devices. This took longer than I expected, but I am quite happy with the program. It is called ReasonLines and can be downloaded for $0.99. Subsequently I had Concept House prepare the free web-based version at www.reasonlines.com which has only recently become available.

I demonstrated the smart phone version (projected onto a large screen) at the 2017 conference of the Alabama Association of Gifted Children in hopes some would adopt it (as these teachers have latitude in selecting their materials), but there were no takers. Those I spoke with fully agreed it would be appropriate but demurred that they already had the materials they wanted to offer and also that this would require them to learn a new subject—to master a new skill—that they didn't have time for. So, it seemed if precollege teachers are to use it, they will need to be introduced to it in college.

I propose the schematics as a more efficient, and more insightful way to present the syllogism at whatever level it is considered. Moreover, I have presented the schematics in contrast to the Venn diagram, rather than to the traditional "rules of the syllogism," because I find the traditional rules to be haphazard, spurious, and theoretically misleading. (See pp 175 ff of *Numerically Exceptive Logic*.) On the other hand, I see the Venn diagrams and the schematics to be theoretically proper and equivalent when applied to the traditional syllogism; I only contend that the schematics have important pedagogical advantages.

NOTES

1. In Wallace A. Murphree, *Numerically Exceptive Logic: A Reduction of the Classical Syllogism* (New York: Peter Lang, Inc., 1991).

POEMS ON TEACHING DURING THE PANDEMIC

The Prof Selects Her Social Distancing

Felicia Nimue Ackerman
BROWN UNIVERSITY

Revised from a version that appeared in The New York Times and The Emily Dickinson Society International Bulletin

The prof selects her own society,
Then shuts the door.
She keeps her social distance of
Six feet or more.

Unmoved, she notes the careless crowd
Outside her gate;
Unmoved, she notes the feckless folk
Still tempting fate.

I've known her not to venture outside
Her room
And turn to students that she's teaching
On Zoom.

The Joy of Zoom Teaching

Felicia Nimue Ackerman
BROWN UNIVERSITY

First appeared in The Wall Street Journal

When thoughts are what you exchange,
No need to be at close range.

ADDRESSES OF CONTRIBUTORS

Felicia Nimue Ackerman
Brown University
Email address: felicia_ackerman@brown.edu

Daniel Lim
Duke Kunshan University
Email address: daniel.lim672@dukekunshan.edu.cn

Wallace A. Murphree
Department of Philosophy and Religion
Mississippi State University
274 County Road 804
Wedowee AL 36278
Email address: w_murphree@yahoo.com

Jiaxin Wu
Duke Kunshan University
Email address: jiaxin.wu@dukekunshan.edu.cn