

(1) THE CASE FOR USING FOREIGN LANGUAGE PEDAGOGIES IN
INTRODUCTORY COMPUTER PROGRAMMING INSTRUCTION
(2) A CONTEXTUALIZED PRE-AP COMPUTER PROGRAMMING
CURRICULUM: MODELS AND SIMULATIONS FOR EXPLORING
REAL-WORLD CROSS-CURRICULAR TOPICS

A Thesis and Project Report

Presented to

The Faculty of the Departments of Computer Science

and

Division of Curriculum and Instruction

California State University, Los Angeles

In Partial Fulfillment of the Requirements for the Degree

Master of Science

in

Interdisciplinary Studies: Computer Science Curriculum and Pedagogy

By

Scott R. Portnoff

June 2016

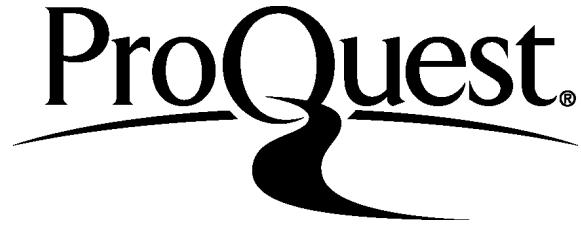
ProQuest Number: 10132126

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10132126

Published by ProQuest LLC (2016). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

© 2016

Scott R. Portnoff

ALL RIGHTS RESERVED

The thesis of Scott R. Portnoff is approved.

Russell J. Abbott, Committee Chair

Raj S. Pamula

Chogollah Maroufi

Robert E. Land

Karin E. Brown, Dean of Graduate Studies

California State University, Los Angeles

June 2016

ABSTRACT

(1) The Case for Using Foreign Language Pedagogies in

Introductory Computer Programming Instruction

(2) A Contextualized pre-AP Computer Programming Curriculum:

Models and Simulations for Exploring Real-World Cross-Curricular Topics

By

Scott R. Portnoff

Large numbers of novice programmers have been failing postsecondary introductory computer science programming (CS1) courses for nearly four decades. Student learning is much worse in secondary programming courses of similar or even lesser rigor. This has critical implications for efforts to reclassify Computer Science (CS) as a core secondary subject. State departments of education have little incentive to do so until it can be demonstrated that most grade-level students will not only pass such classes, but will be well-prepared to succeed in subsequent vertically aligned coursework.

One rarely considered cause for such massive failure is insufficient pedagogic attention to teaching a programming language (PL) as a language, per se. Students who struggle with acquiring proficiency in using a PL can be likened to students who flounder in a French class due to a poor grasp of the language's syntactic or semantic features. Though natural languages (NL) and PLs differ in many key respects, a recently reported (2014) fMRI study has demonstrated that comprehension of computer programs primarily utilizes regions of the brain involved in language processing, not math. The implications

for CS pedagogy are that, if PLs are learned in ways fundamentally similar to how second languages (L2) are acquired, foreign language pedagogies (FLP) and second language acquisition (SLA) theories can be key sources for informing the crafting of effective PL teaching strategies.

In this regard, key features of contemporary L2 pedagogies relevant to effective PL instruction – reflecting the late 20th-century shift in emphasis from cognitive learning that stressed grammatical knowledge, to one that facilitates communication and practical uses of the language – are: (1) repetitive and comprehensible input in a variety of contexts, and (2) motivated, meaningful communication and interaction.

Informed by these principles, four language-based strategies adapted for PL instruction are described, the first to help students acquire syntax and three others for learning semantics: (a) memorization; (b) setting components in relief; (c) transformations; and (d) ongoing exposure.

Anecdotal observations in my classroom have long indicated that memorization of small programs and program fragments can immediately and drastically reduce the occurrence of syntax errors among novice pre-AP Java programming students. A modest first experiment attempting to confirm the effect was statistically unconvincing: for students most likely to struggle, the Pearson coefficient of -0.474 ($p < 0.064$) suggested a low-modest inverse correlation. A follow-up study will be better designed. Still, a possible explanation for the anecdotal phenomenon is that the repetition required for proficient memorization activates the same subconscious language acquisition processes that construct NL grammars when learners are exposed to language data.

Dismal retention rates subsequent to the introductory programming course have historically also been a persistent problem. One key factor impacting attrition is a student's intrinsic motivation, which is shaped both by interest in, and self-efficacy with regards to, the subject matter. Interest involves not just CS concepts, but also context, the domains used to illustrate how one can apply those concepts. One way to tap into a wide range of student interests is to demonstrate the capacity of CS to explore, model, simulate and solve non-trivial problems in domains across the academic spectrum, fields that students already value and whose basic concepts they already understand.

An original University of California "G" elective (UCOP "a-g" approved) pre-AP programming course along these principles is described. In this graphics-based *Processing* course, students are guided through the process of writing and studying small dynamic art programs, progressing to mid-size graphics programs that model or simulate real-world problems and phenomena in geography, biology, political science and astronomy. The contextualized course content combined with the language-specific strategies outlined above address both interest and self-efficacy. Although anecdotally these appear to have a positive effect on student understanding and retention, studies need to be done on a larger scale to validate these outcomes.

Finally, a critique is offered of the movement to replace rigorous secondary *programming* instruction with *survey* courses – particularly *Exploring Computer Science* and *APCS Principles* – under the guise of "democratizing" secondary CS education or to address the severe and persistent demographic disparities. This group of educators has promulgated a nonsensical fiction that programming is simply one of many sub-disciplines of the field, rather than the core skill needed to understand all other CS topics

in any deep and meaningful way. These courses present a facade of mitigating demographic disparities, but leave participants no better prepared for subsequent CS study.

DEDICATION

This thesis is dedicated to my husband of 29 years, Albert Joseph Winn, who passed away in the midst of my studies. Born July 9, 1947, Al was an accomplished artist, photographer, teacher and storyteller/writer. He was a devoted spouse, who encouraged me throughout this endeavor even as he dealt with illness in his final year. He was an enthusiastic uncle, son, brother, brother-in-law and dear friend, missed now by many, many more people than just myself. Not to mention how doggedly he cared for each of our many pets. He died on May 20, 2014 of complications from neuroendocrine carcinoma. Portions of his work may be viewed at *AlbertJWinn.com*.

TABLE OF CONTENTS

Abstract.....	iv
Dedication.....	viii
List of Tables	xii
List of Figures.....	xiii
List of Abbreviations	xv
Part 1. The Case for Using Foreign Language Pedagogies in Introductory Computer Programming Instruction.....	1
Chapter 1. The Efficacy of Memorization on Learning	
Programming Language Syntax.....	2
Section 1. Introduction.....	2
Section 2. Methods and Implementation	5
Section 3. Results and Evaluation.....	11
Chapter 2. Teaching CS Using a Second Language Pedagogic Paradigm	19
Section 1. The <i>Anti-Programming Interest Group (APIG)</i> :	
Abandoning Novice Programmers.....	19
Section 2. Technological Innovation is Not Pedagogic Innovation.....	37
Section 3. SLA Theories and Programming Languages	42
Subsection 1. Theoretical Considerations for Using a SLA Model in CS Instruction	42
Subsection 2. A Review of SLA Theories Pertinent to Programming Languages.....	49
Subsection 3. Deaf ESL Learners	62
Subsection 4. The Role of Context	65

Subsection 5. Neurocognitive and fMRI Studies.....	68
Section 4. SLA Instructional Strategies for Computer Programming	72
Subsection 1. Syntax: Memorization	73
Subsection 2. Semantics: Setting Components in Relief	79
Subsection 3. Semantics: Revealing Underlying Transformational Structure	83
Subsection 4. Semantics: for-loop Transformations.....	86
Subsection 5. Semantics: Ongoing Exposure	97
Subsection 6. Summary	100
Part 2. A Contextualized pre-AP Computer Programming Curriculum: Models and Simulations for Exploring Real-World Cross-Curricular Topics ...	102
Chapter 1. Contextualization in CS Education	103
Section 1. Introduction.....	103
Section 2. Contextualization Efforts in CS Education.....	108
Section 3. Potential Impact on Institutional Change.....	114
Section 4. Core Computing Concepts and Computational Competencies....	116
Section 5. Broadening Participation	119
Section 6. Principles for Implementation.....	126
Chapter 2. The Course Outline for CPRWE: Computer Programming as if the Rest of the World Existed	136
Section 1. Introduction.....	136
Section 2. Piet Mondrian Painting	139
Section 3. Ricocheting Comets	142

Section 4. Rotating McClure Painting	148
Section 5. Word Clouds	153
Section 6. CodingBat: Boolean logic, Strings and Arrays.....	158
Section 7. Nested For-Loops, Regular Patterns, and T-Tables.....	164
Section 8. The Right to Vote.....	167
Section 9. Around the World in 24 Days	172
Section 10. Galileo's Revolution and Astronomy	178
Section 11. Molecular Modeling and DNA	185
References.....	192
Appendices	
A. A Critique of the ECS Curriculum.....	211
B. The Influence of the APIG on CSTA Policy.....	225
C. The APIG's Impact on CSTA's Curricular Framework.....	228
D. The APIG Fallacies of Computer-Free CS Instruction.....	236
E. Differences between Natural and Programming Languages	240

LIST OF TABLES

1. Experimental Results	11
2. Assignment to Memorization/Control Group as Independent Variable, Pearson Coefficients and (2-tailed) p-values	12
3. CSP Composite Score as Independent Variable, Pearson Coefficients and (2-tailed) p-values	16

LIST OF FIGURES

1A. Case Study Program Code (CSP) and Translation	6
1B. Case Study Program (CSP) Output	7
2. Novel Assessment Program (NAP) Instructions and Output.....	8
3. Novel Assessment Program (NAP) Optimal Solution.....	8
4. Top: What instructors imagine students see. Bottom: What students might as well be seeing	15
5. Non-Gifted Subgroup: CSPCS scores vs. NAP#SE	17
6. rotateLeft3() instructional code and diagram.....	54
7. rotateRight3() direction error	55
8. Output and Detail of Starter Code for Piet Mondrian Painting	74
9. Surface Syntax. Constructor Call Arguments → Assignment of Values to Constructor Parameters	84
10. Transformational Model: Positing Intermediate Structures that clarify both (a) assignment of argument values to parameters and (b) how parameters behave like local variables	84
11. Concentric Squares. Left: Desired Output. Right: Red Lines to help with calculations	87
12. CS Gender Performance Gap, 2004-2015	120
13. %Female vs. Number of STEM Test Takers, 2015	121
14. %Female vs. Total Number of APCS Test Takers, 2004-2015	122
15A. East bound traveler is just to the west of the sector at Time 1.....	176
15B. East has moved just to the east of the next sector at Time 2. No sunrise is detected!	176

16A. West bound traveler is at the west edge of the sector at Time 1. A sunrise event is detected.....	177
16B. West has moved to the east edge of the next sector at Time 2. A 2nd sunrise event is detected!.....	177
17A. East bound traveler is to the west of the widened sector at Time 1. No change in behavior.....	177
17B. East is within the widened sector at its east edge at Time 2. A sunrise event is now detected!	177
18A. West bound traveler is at the west edge of the narrowed sector at Time 1. A sunrise event is detected.....	177
18B. West is now just outside the east edge of the next sector at Time 2, because the sector has been narrowed at this end. No 2nd sunrise event is detected!	177
19. Derivation of Additive Trigonometric Identities	190
20. Derivation of Formulas for Calculating New Coordinates after Rotation about the Origin	190

LIST OF ABBREVIATIONS

ACM	Association for Computing Machinery
API	Application Program Interface
APIG	Anti-Programming Interest Group
APCS	Advanced Placement Computer Science
ASL	American Sign Language
CL	Computer Literacy
CPRWE	Computer Programming as if the Rest of the World Existed
CS	Computer Science
CS0	College-level pre-Introductory Computer Science
CS1	College-level Introductory Computer Science
CSEA	Computer Science Equity Alliance
CSP	Case Study Program
CSPCS	Case Study Program Composite Score
CSP#SE	Case Study Program Number of Syntax Errors
CSS	Cascading Style Sheets
CSTA	Computer Science Teachers Association
CT	Computational Thinking
ECS	Exploring Computer Science
ESL	English as a Second Language
FLP	Foreign Language Pedagogy
fMRI	Functional Magnetic Resonance Imaging
G	Gifted

HTML	HyperText Markup Language
IDE	Integrated Development Environment
IT	Information Technology
LAD	Language Acquisition Device
LAUSD	Los Angeles Unified School District
L1	First (native) Language
L2	Second (non-native) Language
NAP	Novel Assessment Program
NAP CS	Novel Assessment Program Composite Score
NAP #SE	Novel Assessment Program Number of Syntax Errors
NG	Non-Gifted
NSF	National Science Foundation
NL	Natural Language
OO	Object-Oriented
PL	Programming Language
PLTW	Project Lead the Way
RGB	Red Green Blue (color values)
SIGCSE	Special Interest Group: Computer Science Education
SLA	Second Language Acquisition
SMC	Subject Matter Competent
STEM	Science, Technology, Engineering & Math
UCOP	University of California Office of the President
URM	(Traditionally) Under-Represented Minority

PART 1

**THE CASE FOR USING
FOREIGN LANGUAGE PEDAGOGIES IN
INTRODUCTORY COMPUTER PROGRAMMING INSTRUCTION**

CHAPTER 1.
THE EFFICACY OF MEMORIZATION
ON LEARNING PROGRAMMING LANGUAGE SYNTAX

Section 1. Introduction

It would be disingenuous to say that the difficulty experienced by large numbers of novice programmers is a crisis; rather it is a long-time, defective *feature* of first programming courses in a CS curriculum. The literature has been documenting the failure of such students to acquire proficiency in the skills and concepts taught in secondary and postsecondary introductory CS programming courses for nearly four decades (Bennedsen & Caspersen, 2007) (Watson & Li, 2014) (Lahtinen, AlaMutka, & Järvinen, 2005) (McGettrick, et al., 2005). It is also a constant that enrollments in secondary AP CS courses are skewed demographically more so than for any other AP subject, and that absolute numbers for such enrollments are currently 4-6 times lower than for AP Calculus, AP Statistics or AP Biology (College Board, 2015). Attempts during the preceding decade at remedying any part of this status quo – such as contextualization (e.g. robotics) and drag-and-drop programming interfaces within self-contained 2-D and 3-D worlds (Scratch, Alice) – have failed to move the needle (Kelleher & Pausch, 2005).

Researchers have occasionally looked for predictors of success, considering, for example, such student characteristics as the ability to abstract (Bennedsen & Caspersen, 2008). Correlations have not been found. Extra tutoring and long hours in the lab have yielded inconsistent and weak claims of success for only a portion of failing students (Azemi & D’Imperio, 2011). Such interventions also have given no insights into

understanding why students fail. In the absence of any remedies, many educators have asserted that students who are otherwise intelligent or successful in other subjects, but who fail in CS, lack innate talent, perseverance or some other enigmatic quality.

Ascribing fixed mindsets to such students – that they simply have low ability in this particular subject area – conveniently allows educators to let themselves off the hook in terms of reflecting on their own teaching practice, in particular that their teaching may be contributing to these outcomes.

Retention of students in the discipline is also problematic, as attrition rates are high for all but the so-called "high-fliers". A very few elite postsecondary institutions that educate talented and privileged students have employed social support strategies to increase this different, but related, issue (Alvarado & Dodds, 2010). It is unlikely, though, that these interventions can be replicated at the secondary level outside of an extra-curricular computer club setting.

The first obstacle that beginning programmers encounter, and that instructors universally observe when teaching a text-based programming language (PL), is a stubborn, and in many cases unsuccessful, struggle with syntax errors. This phenomenon has long been reported:

... students using an unfamiliar or new programming language waste considerable time correcting syntax errors. Studies have shown that excessive time spent on correcting syntax problems can be detrimental to long-term success as students become disheartened with programming. (Kummerfeld & Kay, 2003)

For several years, though, I have observed that requiring grade-level high school freshmen to memorize program fragments and small programs resulted in a drastic reduction of students repeatedly making the same syntax errors. Although this did not magically turn all students into high-fliers, memorization – together with other language-

based instructional strategies – did greatly raise the performance of those who formerly would have failed to learn much of anything at all. At the end of a 20-week high school semester, what would have formerly been the bottom performing group of students was able to successfully do basic introductory programming tasks, similar in both difficulty and variety to the first 50% of problems found in the *Logic-1* and *Array-1* Java modules on Stanford professor Nick Parlante's *Codingbat.com* website.

The most convincing way to demonstrate efficacy for language-based pedagogies would have been to set up experimental and control classes, teach each for an extended period using different instructional methods, give pre-and post-treatment assessments, match students for characteristics that might affect the outcome, and run the statistical analyses. The availability of resources required for such a lengthy experiment is rare at a single secondary site, so a more limited study on the effect of memorization on syntax acquisition was designed and executed. Statistical confirmation of a memorization effect on syntax acquisition was weak and did not reach the 95% confidence level. A better-designed study will be performed in the near future.

Section 2. Methods and Implementation

The study was conducted in August 2015 at the end of the first week of classes, by which time enrollment had stabilized. The student programming tool was *Processing*, a Java-based Integrated Developer Environment (IDE) for visual artists. The **Case Study Program (CSP)** contained a main `setup()` method and 3 user-defined methods. Graphic output was a simple design consisting of rectangles and lines. Guided instruction was given to a class of grade-level¹ high school freshmen as they incrementally built and tested the program. Primitive drawing methods and RGB concepts were demonstrated and explained. Counterexamples underscored common novice misunderstandings regarding program operation and logic. With guidance from the instructor, students observed changes in output as they experimented with the program – modifying method parameters, changing the order of primitive methods, commenting out method calls – to gain an appreciation of the flow of control and how primitive drawing methods worked in concert with one another.

Students had been randomly assigned to two groups, each with equal numbers of gifted and non-gifted² students. Students were given a study sheet of the CSP (*Figures 1A, 1B*), consisting of (1) the program code, accompanied by line-by-line comments; and (2) an image of the final graphic output. The control group was instructed to *study* the CSP, while the experimental group was directed to *memorize* the CSP code. To make clear what was expected from the memorization group, it was suggested that they repeatedly (a) study the code; (b) set it aside; and (c) write out the program, using either pencil/paper or a text editor, until they could reproduce the code *perfectly* from memory,

¹ Grade-level students are those proficient in Algebra 1 and co-enrolled in either Algebra 2 or Geometry.

² Data indicating previous assignment of gifted status was a part of each student's demographic record.

paying particular attention to details such as punctuation and letter-case. Students had the weekend to study. Both groups were told that they would be tested on writing similar code for a different novel problem. The memorization group was told that they would also be tested on how well they had memorized the CSP.

Java Code

```
void setup() {
  size(300, 200);
  background(0);
  drawYellowRect();
  drawThinVertBlueLine();
  drawThickHorzGreenLine();
}
```

```
void drawYellowRect() {
  rectMode(CENTER);
  fill(255, 255, 0);
  rect(300/2, 200/2, 280, 180);
}
```

```
void drawThinVertBlueLine() {
  strokeCap(SQUARE);
  stroke(0, 0, 255);
  strokeWeight(5);
  line(300/2, 10, 300/2, 190);
}
```

```
void drawThickHorzGreenLine() {
  strokeCap(SQUARE);
  stroke(0, 128, 0);
  strokeWeight(20);
  line(10, 200/2, 290, 200/2);
}
```

Translation

Write the body for the system method setup (),
return type (RT) = void.
 Set window size to width = 300 px, height = 200 px.
 Set the window background color to black.
Call a method named drawYellowRect().
Call a method named drawThinVertBlueLine ().
Call a method named drawThickHorzGreenLine ().

Write the method header for **drawYellowRect()**,
 RT=void.
 Set the rectangle drawing mode to CENTER.
 Set the interior brush color to Yellow.
 Draw a rectangle, center at (150,100), w=280, h=180

Write the method header for **drawThinVertBlueLine()**,
 RT=void.
 Set the line end style to SQUARE.
 Set the drawing pen color to blue (opaque).
 Set the drawing pen width to 5.
 Draw a line from (x1=150, y1=10) to (x2=150, y2=190).

Write the method header for
drawThickHorzGreenLine(), RT=void.
 Set the line end style to SQUARE.
 Set the drawing pen color to green (opaque).
 Set the drawing pen width to 20.
 Draw a line from (x1=10, y1=100) to (x2=290, y2=100).

Figure 1A. Case Study Program Code (CSP) and Translation.

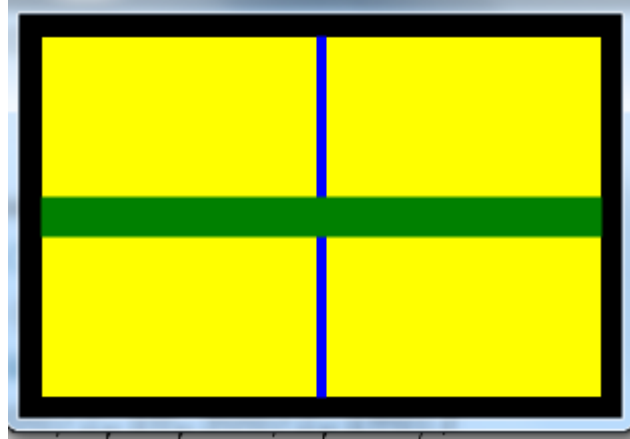


Figure 1B. Case Study Program (CSP) Output.

It had been anticipated that there would be substantial non-compliance with the study instructions from both groups. In this respect, the division of students into two groups was a ruse. The plan was to assess all students on how well they had memorized the CSP and use the memorization scores as the independent variable when looking at the ability to solve a similar, but novel, problem.

At the next class meeting, students were given a chance to ask questions and clear up any misunderstandings about the CSP code. All students were then given the *Memorization Assessment*. The exam consisted of a handout with an image of the CSP output (**Figure 1B**), but with all dimensions labeled. Students were asked to write the CSP code from memory with paper and pencil to the best of their ability. All exams were collected when the students had finished.

Students were next given a diagram of rectangles and lines with different positions and colors, and asked to write a *Processing* program – referred to here as the **Novel Assessment Program (NAP)** – that would output the new design. They were supplied with specifications for all dimensions and colors, as well as the names and descriptions of 3 user-defined methods they were to use in their programs (**Figure 2**).

Students were additionally given the CSP study handout (*Figure 1A, 1B*) for reference to help them write the new program. Without analogous program code to refer to, students who had not memorized the program would have been at a profound loss to perform the task in any meaningful or substantial way. *Figure 3* shows optimal code for the NAP.

Using the Case Study Program as a model, write a program that produces the output at right using the specs below in the order given.

1. The window's width=400, height=600. The window color is **white**.
2. The method **drawRedRect()** draws an opaque **red** rectangle with a 1-px thick **black** border, width=300, height=500. Its center is in the middle of the window (this leaves a white 50 px margin on all sides).
3. The method **drawThinHorzYellowLine()** draws a horizontal opaque **yellow** 5-px thick line with square ends. It bisects the red rectangle.
4. The method **drawThickVertBlueLine()** draws a vertical opaque **blue** 20-px thick line with square ends. It also bisects the red rectangle.

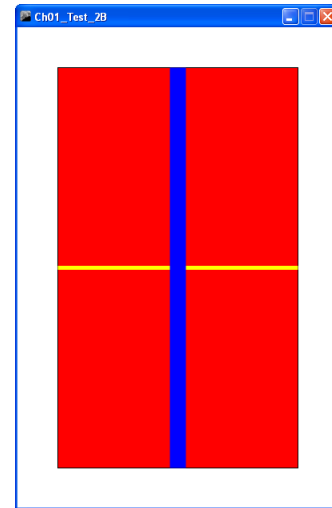


Figure 2. Novel Assessment Program (NAP) Instructions and Output

```

void setup() {
    size(400, 600);
    background(255);
    drawRedRect();
    drawThinHorzYellowLine();
    drawThickVertBlueLine();
}

void drawRedRect() {
    rectMode(CENTER);
    fill(255, 0, 0);
    rect(400/2, 600/2, 300, 500);
}

void drawThinHorzYellowLine() {
    strokeCap(SQUARE);
    stroke(255, 255, 0);
    strokeWeight(5);
    line(50, 600/2, 350, 600/2);
}

void drawThickVertBlueLine() {
    strokeCap(SQUARE);
    stroke(0, 0, 255);
    strokeWeight(20);
    line(400/2, 50, 400/2, 550);
}

```

Figure 3. Novel Assessment Program (NAP). Optimal Solution

Having access to the CSP program does not automatically ensure that students can write new programs perfectly. The reason: teachers routinely are asked for help during instruction when students inadvertently insert errors into program code that they are

simply copying. It was therefore hoped that statistical analysis might be able to discern small differences in the occurrence of such errors and confirm a memorization effect.

Composite Scores for both programs were calculated by combining three measures: (1) Completeness, (2) Number of Syntax Errors and (3) Number of Logical Errors.

Completeness

A complete 4-method program counted as 100 points. A missing method, or one so incomplete as to be useless, resulted in a deduction of 25 points. The composite score was heavily weighted using the completeness measurement because it was the parameter best reflecting overall how well the student memorized the program.

Number of Syntax Errors

Four categories of syntax errors occurred, and each error counted as a one-point deduction. These were:

- 1) Semi-colons that were either absent, or used incorrectly;
- 2) Curly braces that were missing or unpaired; or the wrong type of brace was used (e.g. using an opening brace in place of a closing brace);
- 3) Errors in primitive drawing method calls:
 - a) A named-constant intended for one method was used incorrectly as a parameter in a different method;
 - b) A method contained the wrong number of parameters;
 - c) Parameters appeared in the wrong order.
- 4) User-defined methods:
 - a) Headings lacked parentheses;

- b) Method bodies were incomplete or missing.
- 5) A missing method counted as a 1-point syntax error. Without this (small) adjustment, a blank paper would count as having no syntax errors.

Number of Logical Errors

Logical errors, each counting as a one-point deduction, included (1) setting pen attributes after – rather than before – calls to draw lines or rectangles; (2) coordinate errors affecting margins, positions, size, orientation and the like; (3) RGB errors.

Composite Score

The formula for the overall composite score appears below:

$$\text{Composite Score} = \underline{\text{Completeness}} - \underline{\# \text{Syntax Errors}} - \underline{\# \text{Logical Errors}}$$

Section 3. Results and Evaluation

		CSP Scores				NAP Scores			
Group	Gifted	% Incomplete Program	# Syntax Error	# Logic Error	Composite Score	% Incomplete Program	# Syntax Error	# Logic Error	Composite Score
Control	N	100	15	0	-15	0	8	7	85
Control	N	100	11	3	-14	0	1	2	97
Control	N	75	11	0	14	0	0	3	97
Exper	N	75	9	1	15	0	5	3	92
Exper	N	75	9	1	15	75	4	3	18
Control	N	75	10	0	15	0	2	2	96
Control	N	20	22	3	55	0	6	3	91
Control	N	30	5	4	61	0	2	3	95
Control	N	25	7	4	64	0	0	3	97
Exper	N	0	2	3	95	0	1	3	96
Exper	N	0	5	0	95	0	0	3	97
Exper	N	0	2	0	98	0	0	3	97
Exper	N	0	0	1	99	75	4	2	19
Control	N	0	0	0	100	0	0	3	97
Exper	N	0	0	0	100	0	0	4	96
Exper	N	0	0	0	100	0	2	2	96
Control	Y	100	11	2	-13	0	0	4	96
Control	Y	100	11	1	-12	0	0	5	95
Exper	Y	100	8	1	-9	0	0	4	96
Exper	Y	75	15	1	9	0	5	3	92
Exper	Y	30	12	8	50	0	3	3	94
Control	Y	25	3	0	72	50	3	1	46
Control	Y	20	3	4	73	0	1	4	95
Control	Y	0	8	3	89	0	0	4	96
Control	Y	0	9	0	91	0	3	1	96
Control	Y	0	0	4	96	0	0	3	97
Exper	Y	0	0	2	98	0	0	2	98
Exper	Y	0	0	1	99	0	4	2	94
Exper	Y	0	0	0	100	0	0	0	100
Exper	Y	0	0	0	100	0	0	0	100
Control	Y	0	0	0	100	0	0	0	100

Table 1. Experimental Results

Scores for each student appear in **Table 1**. Data is sorted by **Gifted/Non-Gifted** status, and secondarily by **CSP Composite Score**. Pearson bivariate correlation analyses were performed on the 31 students as a whole, and on groups disaggregated by Gifted (G) and Non-Gifted (NG) status.

Table 2 shows the correlations of scores with the group – Memorization (Experimental) vs. Control – to which students were assigned.

Data Sub-Group	CSP Composite Score (CSPCS)	CSP # Syntax Errors (CSP#SE)	NAP Composite Score (NAPCS)	NAP # Syntax Errors (NAP#SE)
All N=31	.258 (.162)	-.326 (.073)	-.159 (.393)	.056 (.766)
Gifted N=15	.021 (.941)	-.060 (.833)	.241 (.387)	.242 (.385)
Non-Gifted N=16	.492 (.053)	-.555 (.026)	-.353 (.179)	-.077 (.776)
<i>Table 2. Assignment to Memorization/Control Group as Independent Variable, Pearson Coefficients and (2-tailed) p-values</i>				

The data show no correlation of Group Assignment with NAP scores. More importantly, though, these data show poor correlation of Group Assignment with successful recollection of the CSP code. Only the NG group showed moderate, but statistically significant, correlations with the CSP Composite Score (CSPCS) ($p < .053$) and the CSP Number of Syntax Errors (CSP#SE) ($p < .026$).

As mentioned earlier, the lack of correlation with successful recollection of CSP program code was not unexpected, as substantial student non-compliance with study instructions had been anticipated for both groups. In a one-time exercise, simply assigning students to a group does not ensure that the desired experimental "treatment" actually takes place. A much better measure of compliance with the memorization study instructions in this case is the CSPCS.

The use of the CSPCS score as the independent variable, however, comes with its own set of concerns. First, there is the possibility for some contribution to selection bias from a student's Gifted status. Among the 15 students whose scores clustered at or above the natural cutoff point of 89, the NG:G ratio is 7:8. The scores of the remaining 16 students ranged from -15 to 73 with a NG:G ratio of 9:7 (*Table 1*). Although these shifts from parity are small, there is a workaround: one can eliminate any potential selection bias by disaggregating the data into distinct Gifted and Non-Gifted groups.

A second concern is speculation that students with better scores were simply more motivated, i.e. motivation might be a source of selection bias for the CSPCS. Note, however, that motivation did not appear to be a problem when students worked on the NAP coding task: NAP Composite Scores (NAPCS) Scores for 28 of the 31 participants were 85 or above (*Table 1*). One would be hard-pressed to explain how low motivation was the causative factor for both the poor CSPCS scores of half of the study's participants and for that same sub-group's high NAPCS scores assessed a few minutes later. It should also be recalled that 11/16 students in the Control group and 10/15 students in the Experimental group (*Table 1*) appeared to comply with the instructions given to their respective groups; therefore no inference can be made about the motivational level of 2/3 of the study's participants. One might also argue that situational motivation for memorizing the CSP program *outside* of class was lower than when students worked on the NAP program *in* class, but this would only apply to 5 students in the Experimental group with poor CSPCS scores. Finally, one should remember that all students had a history of high academic motivation and were – at minimum – grade-level proficient in mathematics and English Language Arts.

It seems far more likely that students who had poor CSPCS scores, but earned high NAPCS scores a few minutes later, were able to do so by referring to the CSP handout as they wrote the analogous NAP program.

The research goal was to try to statistically tease out / discern small differences in NAP coding measurements with the CSPCS score as the independent variable. The reason for suspecting that such differences might exist comes from the common observation that novice programming students routinely insert errors into programs that they are simply copying, even when the text is sitting right in front of them. Most instructors would view this from a behavioral perspective and blame the errors on a lack of attention, care and/or precision on the part of the copier. Seen from within a language framework, though, the errors can be ascribed to an incomplete but ongoing and developing acquisition process of the meaningful syntactic markers of the language, a phenomenon no different from the many syntax errors that foreign language instructors observe their beginning students make on a daily basis. Although hyperbolic, *Figure 4* may help instructors appreciate how foreign a program written in a PL may appear to students when first encountered.

```

void draw() {
  backgroundTransparent();
  fill(clr);
  ellipse(x,y,30,30); // use x,y for drawing
  x = x + directionX; // update x
  if (x >= width || x <= 0) {
    directionX = -directionX; // update directionX
    clr = color( random(255), random(255), random(255) );
  }
  y = y + directionY; // updating the variable
  if (y >= height || y <= 0) {
    directionY = -directionY; // update directionY
    clr = color( random(255), random(255), random(255) );
  }
}

```

```

void δραω() {
  βαγκρουνδΤρανσπαρεντ();
  φιλλ(χλρ);
  ελλιπσε(ξ,ψ,30,30); // υσε ξ,ψ φορ δραωινγ
  ξ = ξ + διρεχτιονΞ; // υπδατε ξ
  ιφ (ξ >= ωιδτη || ξ <= 0) {
    διρεχτιονΞ = -διρεχτιονΞ; // υπδατε διρεχτιονΞ
    χλρ = χολορ( ρανδομ(255),ρανδομ(255),ρανδομ(255) );
  }
  ψ = ψ + διρεχτιονΨ; // υπδατινγ τηε παριαβλε
  ιφ (ψ >= ηειγητ || ψ <= 0) {
    διρεχτιονΨ = -διρεχτιονΨ; // υπδατε διρεχτιονΨ
    χλρ = χολορ( ρανδομ(255),ρανδομ(255),ρανδομ(255) );
  }
}

```

Figure 4. Top: What instructors imagine students see.
Bottom: What students might as well be seeing.

Data Sub-Group	CSPCS vs. NAPCS	CSPCS vs. NAP#SE
Gifted N=15	.039 (.891)	-.036 (.898)
Non-Gifted N=16	-.050 (.855)	-.474 (.064)
<i>Table 3. CSP Composite Score as Independent Variable, Pearson Coefficients and (2-tailed) p-values</i>		

Table 3 shows correlation values using the CSPCS as the independent variable. Unsurprisingly, there are no correlations of CSPCS scores with NAPCS scores. A reasonable interpretation is that most students were able to construct the complete *structure* for each method of the NAP from the CSP handout via analogy.

However, the correlation values of the CSPCS scores with the number of syntax errors (NAP#SE) made when students coded the NAP showed clear differences between the two disaggregated groups.

The Gifted subgroup showed no correlation, i.e. so long as CSP code was available, prior memorization of the CSP made no difference for Gifted students in terms of syntax errors made when they wrote the NAP.

However, for the Non-Gifted subgroup, the result was a low-moderate inverse correlation of $-.474$ ($p < .064$) between CSPCS score and NAP#SE. The result is statistically unconvincing, but nonetheless suggestive that Non-Gifted students made fewer syntax errors if they had successfully memorized the analogous CSP. Stated conversely, Non-Gifted students who had not memorized the CSP were less successful at deciphering meaningful syntax information from the CSP handout when constructing the NAP. **Figure 5** shows a plot of the data points with the linear trend line.

There were logistical and design problems with the experiment: the disaggregated sample (N=16) was small; the magnitude of the penalty for calculating the NAP#SE score for the 2 Non-Gifted students who had incomplete programs was minimal, but also

arbitrary; and that all students had access to the CSP handout when coding the analogous NAP ensured that any detected differences in performance would be small.

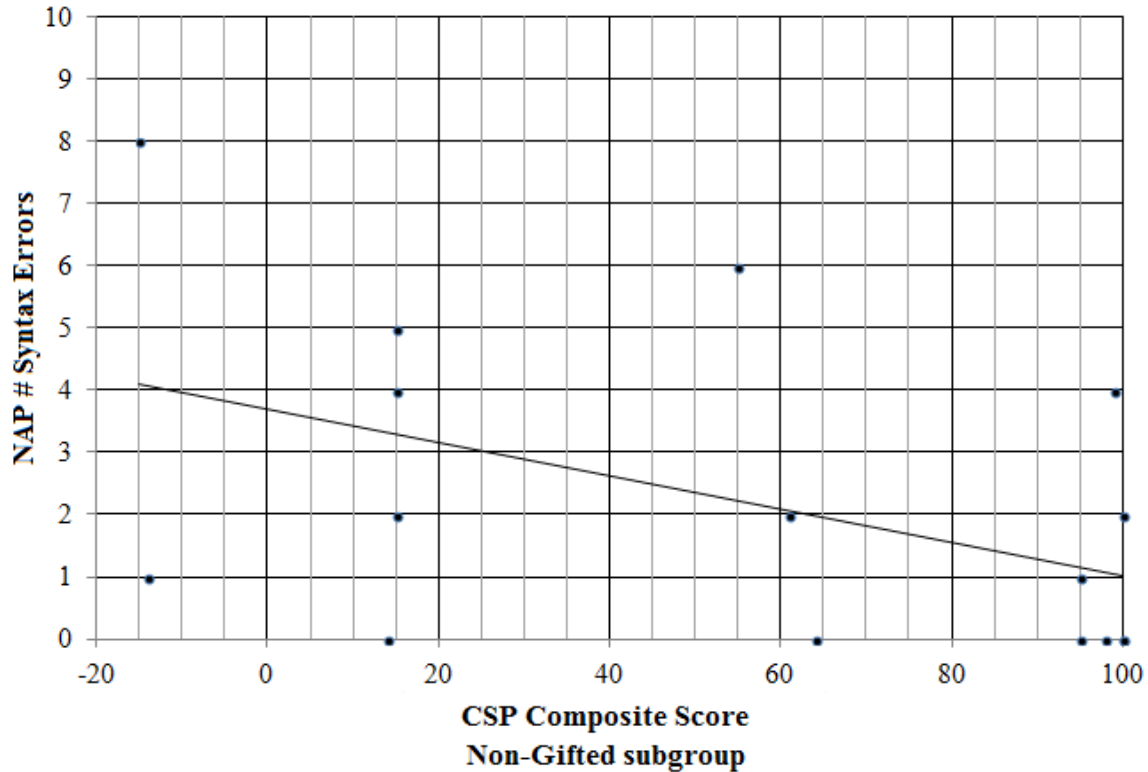


Figure 5. Non-Gifted Subgroup: CSPCS score vs. NAP#SE

A better-designed follow-up study will be done. The current results suggest that reference to the analogous CSP sufficiently compensates students who have not memorized it when tasked with constructing the NAP. What the results do not discern, though, is a difference in *learning*, i.e. the internalization of PL syntactic schemas, per Piaget. In the new study, learning will be assessed by simply asking students to *identify* the syntax errors in the code of a novel program.

In summary, the results of this study are unconvincing, but still suggestive of the hypothesis – and the anecdotal observations – that *among novice programmers most likely to be struggling* – memorization is an effective instructional strategy for

minimizing syntax errors, i.e. for internalizing the syntactic rules of a PL. Together with fMRI data reported in the literature that link brain language processes to computer program comprehension (Chapter 2, Section 3, Subsection 5), they form an intriguing argument that language-based pedagogies which address both syntactic and semantic issues merit further research to see whether they can better help students learn to program. What I have been observing in the classroom is that language-based instructional strategies (Chapter 2, Section 4) improve learning and understanding for all first-year programming students, and not just strugglers. Better-designed follow-up studies will be attempted to confirm this.

CHAPTER 2.

TEACHING CS USING A SECOND LANGUAGE PEDAGOGIC PARADIGM

Section 1. The *Anti-Programming Interest Group (APIG)*:

Abandoning Novice Programmers

Prior to addressing second language pedagogies, it would be instructive to provide some background about how the intellectual climate of secondary CS education has been shaped by recent attempts to address the low enrollments and inability of most grade-level high school students to succeed in both pre-AP programming (CS0) and AP programming courses (CS1).

One instructor has described the university-level CS1 course as comparable in its complexity to writing "a fifteen-page paper ... on Napoleon's invasion of Russia... in Swedish, using a quill pen" (Bloch, 2014). Likewise, novice programmers are asked to master several conceptual skills quite quickly:

- a) Learn to use an Integrated Developer Environment (IDE), a software program with special features to ease the process of writing computer programs, and to run, test and debug them.
- b) Develop moderate proficiency in a programming language; and
- c) Write programs to correctly solve problems in wildly different subject domains.

A daunting task, yet many students are able to do all of this, taking to the subject like fish to water. This has also been the initial experience of virtually all CS instructors. As such, appreciating the learning perspectives of their struggling students is next to impossible, limiting their ability to pin down, or even imagine, exactly where the learning process is breaking down.

The literature documents numerous attempts to help struggling novice programmers, some of which have involved the creation of IDEs that circumvent the preponderance of syntax errors that novices make when writing programs in a text-based language (Daly, 2009) (Kelleher & Pausch, 2005). There is, however, little to no evidence that these interventions can help students transition to programming in a text-based language.

A handful of programs at elite universities, whose learners comprise the top tier of academically talented and motivated students, have implemented contextualized curricula and social support mechanisms to increase the participation and retention of poorly represented demographic groups (females, in these particular studies), with some measure of success (Frieze, Quesenberry, Kemp, & Velazquez, 2012) (Alvarado & Dodds, 2010). However, there are few expectations that these methods can be successfully replicated in less selective colleges or broadly at the secondary level.

These two issues, self-efficacy and attrition, factor into the dismal and well-known demographics disparities in participation rates of females and under-represented minorities (URMs). The problem has proved so intractable and exasperating that the UCLA-based *Exploring Computer Science* (ECS) group, whose mission is to broaden participation in computing through equity and increased access, has from the outset, abandoned the idea of teaching programming to high school students, aside from a superficial and below-grade level exposure. This group has instead compiled a poorly conceived, simplistic and, in places, dumbed-down and bizarre *survey* course, penned by authors whose subject-matter competence appears to extend no further than the first year

of a college-level programming curriculum³. The lessons are from a diverse range of sources, but often have a questionable connection to CS and little to no relationship with one another. The result is a confused curriculum with disjointed units and poor cohesion. A detailed unit-by-unit critique is given in *Appendix A*.

Even were this survey course somehow of adequate quality, it would be a nonsensical response. The universal postsecondary consensus is that without a firm programming foundation, it is impossible to understand topics in subsequent coursework, including algorithms, on anything beyond a trivial level.

Many introductory programming courses have programmability as a core activity and learning goal, and for good reasons since programmability is the defining characteristic of the (digital) computer. This is also echoed in the ACM/IEEE curriculum recommendations *the programming-first model is likely to remain dominant for the foreseeable future*. (Bennedsen & Caspersen, 2012)

Indeed, instructors of postsecondary courses geared to non-majors, such as Wheaton College's *Computing for Poets*, echo these same sentiments:

This course teaches computer programming as a vehicle to explore the formal symbol systems currently used to define our digital libraries of text. Programming facilitates top-down thinking and practice with real-world problem-solving skills such as problem decomposition and writing algorithms." (Computer Science 131. *Computing for Poets*.)

Removing the goal of programming proficiency from an introductory CS course and replacing it with a survey of the "major concepts of the field of computer science" (Goode, Chapman, & Margolis, 2012) is nonsensical in several ways.

First, the idea is as poorly thought through as neglecting French language skills in a first-year course, opting instead to focus on literature-in-translation. Schemes of this

³ The impetus for the ECS survey course was a failed attempt by the *Computer Science Equity Alliance* to expand offerings of the AP Computer Science course throughout LAUSD. The response to create a survey course, as opposed to a foundational programming course, was both an illogical choice and a strategic mistake. This episode is discussed in more detail in *Appendix A*.

sort would ignore both the interrelationship of language and culture, and the dead-end role of perpetual outsider for those with limited language proficiency. A similar analogy would be an Algebra 1 course taught without variables, where students plot specific lines and calculate slopes, but aren't taught line equation formulas. Whether going by the name "general", "technical" or "vocational" math, course sequences comprised of content that regurgitated grammar school math and *culminated in pre-Algebra* at one time served a generation of American high school students not bound for college. It's hard to imagine secondary math educators advocating a return to a curriculum as regressive and trivialized.

Second, there are pedagogic parallels between modifying programs and viewing the outputs, and the use of graphing calculators in Algebra 2 classes. Graphing calculators allow students to quickly observe the effect on curves when parameter values are changed. In the time it would take students to manually graph one curve, they can instead examine, compare and reflect on 25 variations in a family of curves. Similarly, in the course of examining a CS concept, the constant interplay of making small modifications to a program and observing the output allows students to learn more deeply and quickly. Students with little programming facility will learn far less, if anything.

Third, in terms of vertical alignment, failing to rigorously teach the programming concepts found in a typical CS1 programming curriculum will leave students without the conceptual foundation they will need to understand ideas in subsequent coursework. In the vocational math example given above, it would be absurd to propose a hypothetical survey course of the "major concepts" of algebra, geometry, trigonometry and calculus for a target audience of pre-algebra students because they lack the requisite background

for any real comprehension. Educators advocating such courses would, for good reason, be driven out of the profession. In like manner, the ability to program provides the tools to test, manipulate and explore in depth concepts studied in the many subfields of CS. To teach "major concepts" without a programming foundation would require that any content be simplified to such an extent that the "concepts" part is for all practical purposes gutted. As an example, the simplest concept – and one of the first taught – in a course in Machine Learning is the Perceptron, the mathematical model of a neuron from which neural networks are built. To understand the Perceptron concept in any rigorous way requires writing an implementation, observing the program's running behavior, and studying the Perceptron Convergence Theorem – the proof that a Perceptron will always be able to classify sets of linearly separable data. Simply watching a graphics demonstration of a Perceptron would be a surface exercise, leaving students with no appreciation, deep or shallow, for its internal workings or its potential uses in neural networks.

Practically speaking, though, the ECS course doesn't even live up to the promise of including the "major topics" of CS. Someone holding an undergraduate degree in CS would recognize nothing of the sort in this course. Rather the overall sense of many of the course's topics can be more accurately described as a truncated and much simplified version of a first-year programming curriculum, but one which has been gutted of PL instruction. What remains are the domains of practice problem sets that, in a programming course, had been specifically chosen because their solutions required the use of particular PL control structures or concepts. The domains themselves – what formerly served as the material for practicing the *application* of PL concepts – have been confused for and presented as essential CS topics (for examples, see *Appendix A*).

In the world of secondary CS education, the notion of the central place of programming in a CS curriculum has been under attack for the past decade, with a goal to ideologically relegate programming to the periphery as just another subfield of CS. A suitable moniker for the group of educators responsible, which includes the authors of ECS, might be the *Anti-Programming Interest Group* (APIG), borrowing H.M. Kliebard's term "*interest group*" to describe factions that attempted to influence and shape American education in the first half of the 20th century (Kliebard, 2004). Along with the dissemination of ECS, the *APIG* has managed to inject its flawed ideological premises into the policies of the Computer Science Teachers Association (CSTA) (*Appendix B*), a K-12 CS advocacy and resource subgroup under the aegis of the ACM (Association for Computing Machinery), the largest umbrella organization for scientific and educational computing groups.

In the eight years since ECS first appeared, it has persisted with the spin that its, at best, middle-school level course is a rigorous introductory pre-AP curriculum. Furthermore, with the imprimatur of their National Science Foundation (NSF) and CSTA backers, ECS has managed to persuade several large underperforming urban school districts nationwide to offer it. Most administrators at secondary institutions, failing or otherwise, have little to no competence or familiarity with the subject matter and would be hard-pressed to distinguish a useful, rigorous CS curriculum from one of poor quality, or even one having no CS content whatsoever. After eight years, there is unsurprisingly zero evidence that ECS has propelled any students into further CS study, or at an even more basic level, that students actually learn its simplified content. Moreover, ECS has provided no evidence that students learn any skills or concepts that will substantially

prepare them for – or even give them an increased chance of success were they to take – the Advanced Placement Computer Science (APCS-A) course or its CS1 college equivalent, or any subsequent CS coursework for that matter.

Worse, ECS, despite its rhetoric of "equity", "access", and "broadening participation" in CS, has replicated in LAUSD – the Los Angeles Unified School District – the very institutional problem it set out to resolve: a de facto two-tier system. The lower tier consists of poor minority students at failing urban campuses who now learn a below-grade-level ECS curriculum⁴ taught by instructors, the vast majority of whom are not subject matter competent (SMC), i.e. they possess neither a CS college degree nor proficient programming skills. The higher tier comprises privileged students at half a dozen of the approximately 200 district high schools, either located in wealthier areas or academically selective in some other way. These schools offer an APCS-A course taught by SMC instructors, with annual AP exam pass numbers in the double digits. It also needs mentioning, that, as anyone who has taught in a large failing urban high school knows, there are students in *every* high school who are at or above grade-level and have the academic ability to succeed in rigorous coursework. ECS' success in propagating their inferior curriculum in poor urban school districts across the country comes at the cost of depriving such students of the opportunity to access a high-level CS course. No sense of irony is lost that Margolis, who first raised awareness of the social injustice aspects of secondary CS education in LAUSD by describing students (females, URM)

⁴ ECS has taken the place of Computer Literacy courses (the MS-Office suite). One might argue that the former had more practical value.

without access to quality CS education as "stuck in the shallow end"⁵ (Margolis, Estrella, Goode, Holme, & Nao, 2008) – and is the first to admit that she has no expertise in CS – has allied herself with this distraction from efforts to identify and remedy the contributing causes of ineffective programming instruction.

ECS has also promoted a "Teacher Support Model":

To carry out the curriculum, teachers are key. For this reason, we couple the curriculum with a teacher professional development program, offering ECS teachers intensive professional development during summers and throughout the school year that:

- 1) address CS content, pedagogy, and belief systems (including stereotypes about which students can excel in CS);
- 2) provide in-class coaches who help reinforce skills learned at workshops; and
- 3) offer participation in a teacher community for reflection, discussion, collaboration and support. (Margolis, et al., 2012)

Offering vague ideological-sounding phrases like "radical transformation of teaching is required to broaden participation in computing" (Goode, Chapman, & Margolis, 2012) as its rationale, ECS' professional development model "radically" dispenses with all notions of accountability regarding its effectiveness in demonstrating *measurable*, positive educational outcomes for students, or – to set the bar even lower – that students are even learning the below-grade level curriculum. The goals of the teacher model are explained:

The aim of this program is to help support teachers learning of the ECS content while simultaneously *strengthening teachers' pedagogical content knowledge* that supports the teaching of ECS. For many teachers, building a pedagogy that supports inquiry-based teaching and equity practices in the classroom is a multi-year process.

The instructional strategies featured in these workshops – inquiry-based and project-based learning and multicultural education – are, however, long-taught staples of teacher-

⁵ "Stuck in the Shallow End" can be a metaphor for multiple areas/activities from which minorities have historically been excluded: from public swimming pools during the segregation era, for the disproportionate number of people of color who don't learn to swim, for the higher rates of drowning deaths in minority communities.

credential programs. Although the workshops make frequent reference to the demographic challenges of the subject area in all of its manifestations, there are no pedagogies taught whose value in addressing issues of minority or female participation or success in secondary CS classrooms has been validated by research studies.

Also absent from this teacher model is the collection and evaluation of evidence of student learning, and reflection on that *evidence*. The focus rather is on a pre-conceived notion of teacher instruction as opposed to critical observation of students as they learn. The errors that students make provide insights into the *process* by which they learn, and are not just a collection of misunderstandings that require correction. The former has far deeper implications and utility for informing instruction (Corder, 1967) (See: Chapter 2, Section 3, Subsection 2).

Regarding the claim that their model strengthens "teachers' pedagogical content knowledge", first-hand knowledge can reveal quite the opposite. When I attended the week-long training workshop at UCLA in August 2009, one early exercise included teams of teachers brainstorming how to visually demonstrate several sorting algorithms. My team of 3 and a second team of 5 were assigned *Quicksort*⁶ and given *CS Unplugged* handouts explaining the algorithm. The 5-member team presented first. The team's members lined up single-file in front of a table. The teacher at the rear of the line handed each object up the human chain, bucket-brigade style. When an object reached the teacher at the head of the line, he simply placed it in its correct position by eyeballing it, widening spaces between already-placed objects whenever he needed to! Despite that many of these instructors were highly proficient in their own subject areas (physics and

⁶ Note that even the APCS-A course declines to teach *Quicksort*, opting instead for its more comprehensible divide-and-conquer cousin *Mergesort*.

math), and some had even taught APCS-A, there was a complete absence of even the most basic understanding of the sorting algorithm mechanics of comparing and swapping. The fact is they were simply not SMC and hadn't taken the time to remedy that deficiency by enrolling in a university-level introductory CS class. As such, even the simple explanation in the *CS Unplugged* handouts was beyond their ability to comprehend without the guidance of a SMC instructor. The end of their presentation was met with unanimous applause, with the ECS team and authors joining in and smiling! No one on the ECS team offered any corrections and the presentations continued. This was my first realization that the course was subpar and its authors themselves lacked subject-matter competence. Unfortunately, this anecdote turned out not to be unique, and my continued involvement with the ECS project over the next academic year only confirmed the inferior academic nature of the course.

This episode of mediocrity is in fact typical of most public school CS programs. The educational system from top to bottom simply refuses to put the resources into training and staffing high schools with SMC CS instructors. The subject area is such a low priority that current thinking among administrators is that as long as the teacher can stay one step ahead of the students, where's the harm? Note that this is the best scenario, which assumes that teachers actually understand a lesson's concepts. Standards this low for core subject instructors would be inconceivable.

In the past several years, also with funding from NSF, there has been a parallel development of the College Board *APCS Principles* course. The subject matter and programming instruction in this course is generally at an appropriately higher level due to its having been piloted by CS departments at universities nationwide. Course instructors

have developed a host of creative and imaginative lessons that are extremely clever and engaging, and hence quite useful in both a pre-APCS programming course and the APCS-A course itself. Nevertheless, the *APCS Principles* course, like ECS, has been conceived as a counterweight to the so-called "programming-centric" (Goode, Chapman, & Margolis, 2012) focus of the APCS-A/CS1 course. Although the programming examples studied in some versions of the course are quite advanced, in the version of the course in which I was trained, students are expected to only modify program code, not write entire programs from scratch, and little attention is paid to teaching programming competence. Compounding this, during a two-week *APCS Principles* training workshop developed by *Project Lead the Way* (PLTW, summer 2015), I witnessed several smart but not SMC instructors struggle with the initial step of simply trying to *choose* an appropriate program to modify because of their unfamiliarity with Python syntax, data structures and control structures. This elephant in the room – finding competent CS instructors – is a huge problem:

Applying the current AP training model for new CS teachers is similar to asking a teacher with no mathematics background to initiate a new AP Calculus course with just one week of training. This situation would seem absurd to most administrators in the mathematics context, but it is the common expectation for promoting new AP CS courses. (Gray, 2013)

Given all this, it would be magical thinking to expect *APCS Principles* to provide much help to novice programmers in overcoming the problems they will inevitably encounter in the course. Likewise, students taking *APCS Principles* will benefit little from it in a subsequent *APCS-A* course, should they choose to continue study.

One may wonder how a course as grossly substandard as ECS can be disseminated in plain sight. The answer lies in an educational milieu where : (a) there is

massive ignorance about CS by K-12 stakeholders, and (b) school systems are headed by administrators who essentially don't care, responding to news publicity about the impoverished state of CS education with proposals that superficially play well to the public, but that require no additional allocation of financial resources. In such an environment, ECS was readily embraced by LAUSD and earned easy approval as a University of California college preparatory "G" elective. With no evidence of effectiveness, NSF touted ECS as an "exemplar" for developing curricular materials for *APCS Principles* (Cuny, 2011). The logical contradiction, however, is that had ECS been an effective and rigorous pre-AP curriculum, there would have been no need for *APCS Principles*.

Even more astounding, although the low quality of the course is an open secret on the order of "*The Emperor's New Clothes*" – quietly acknowledged by even NSF staff in the same office which funded its continuing development (\$1.5 million in 2010 and \$2.4 million in 2012) – to this day, NSF's CISE (Computer & Information Science & Engineering) Directorate continues to support the expansion of ECS by funding grants up to \$1 million for proposals that "*focus on efforts that enable teachers to successfully offer either or both of two new courses: Exploring Computer Science (ECS) or the new Advanced Placement (AP) CS Principles*" (STEM + Computing Partnerships Program Solicitation NSF 16-527, 2016).

The phenomenon of substandard curricula being embraced by public schools is not without historical precedent in the haphazard evolution of the modern American public educational system. From its origins around the turn of the 20th century, American education has been an amalgam of accumulating and competing ideas pressed by four

principal interest groups. Their chief philosophical emphases can be succinctly summarized as: (1) humanist/academic; (2) social efficiency; (3) psychological / developmental; and (4) social reconstruction/social justice (Kliebard, 2004). Each had its heyday and each left its mark, for better or worse. The more significant reforms that remain are:

- a) subject realignment, e.g. botany and physiology were replaced with biology, a.k.a. the life sciences (*social efficiency*);
- b) a subject-centered college-preparatory curriculum (*humanist*);
- c) egalitarianism, in the sense that all should have the opportunity to learn (*humanist*);
- d) the factory-model, "platoon" system of managing resources, e.g. moving students from room to room (*social efficiency*);
- e) vocational education (*social efficiency*);
- f) child-centered psychological learning theory and constructivism (*developmentalist*);
- g) multiculturalism and critical theory, especially as represented in the social sciences (*social reconstruction/justice*);
- h) accountability, charter schools / privatization of education, and the standards movement (*social efficiency*).

One failed – and in retrospect much-ridiculed – reform, with uncanny parallels to the *APIG*, was the avowedly and unapologetically anti-academic *Life Adjustment* curriculum of the cold-war era. *Life Adjustment* education was the second major initiative pressed by Charles W. Prosser, an early proponent of "social efficiency" who was involved in the

passage of the 1917 Smith-Hughes Act that conferred federal support for vocational education in the public schools. *Life Adjustment* advocated a high school curriculum that was relevant to the social needs of teenagers, including areas such as dating, learning to be an effective consumer, social relationships, family living, and the like. Courses included *School and Life Planning*, *Preparation for Marriage*, *Boy-Girl Relationships*, *Learning to Work*, and *How to Make Friends and Keep Them*. The movement secured a place in educational policy by professing to be a solution for a perceived dropout crisis based on regional dropout rates as high as 30-45%. The curriculum was backed by the National Association of Secondary-School Principals, and even the U.S. Department of Education supported implementation of Prosser's ideas at one time. *Life Adjustment* education eventually came under heavy attack in the early 1950s by academics who had long viewed the school's job as the development of the intellect. By the time of Sputnik and the subsequent 1958 *National Defense Education Act*, the *Life Adjustment* movement had lost most of its credibility among the general public (Kliebard, 2004).

Kliebard's historical examples support the thesis that American educational interest groups have succeeded in influencing the public school system when they proposed solutions to a perceived or manufactured crisis. In the case of the *APIG*, the "crisis" is an educational system that produces only a fraction of the CS college graduates needed to staff an ever-expanding and lucrative computing job market. Pointing to universally acknowledged dismal levels of enrollments in the AP Computer Science course, and to a stubbornly persistent and severe minority and gender participation gap, its proposed solution is to expand participation in CS by these traditionally underrepresented groups. ECS has indeed shown that it can easily fill its academically

low-level classes with students in these demographics. This exercise-in-cynicism, however, can only succeed in the environment already described, where ignorance in the K-12 educational establishment about CS is pervasive, and apathy by boards of education and district administrators that prefer the appearance of doing something on the cheap to substantive efforts that would require the budgeting and expenditure of significant financial resources.

Although ECS' original mission was to respond to this "crisis" – low numbers of CS college graduates – its mission has "evolved" after the fact, one might even say conveniently:

Our mission goes beyond the “pipeline” issue of who ends up majoring in CS in college. Rather, our mission is to democratize CS learning and assure that all students have access to CS knowledge. (Margolis, et al., 2012)

"Beyond" is a euphemism; *discarding* "the pipeline issue" would be more accurate. ECS' idea of democratization is simply to expose large numbers of minority urban and female students to whatever content is in their course. Accountability and even the much lower bar of simply inspiring students to engage in further study of CS are disregarded⁷. In an article with a section entitled *Appropriate Measures of Reform Success Must be Used*, the authors of ECS state:

Disrupting the traditional secondary curriculum with the introduction of computer science education can entrap reformers into proving that ... students are more likely to choose to enroll in computer science classes in the future. These are *incredibly high, if not impossible standards for any one high school class to obtain*, and the high possibility of *confounding variables* makes it difficult to ultimately make any conclusions. Additionally, these types of measures dilute the importance of computer science as essential knowledge for 21st century students in its own right... As a community of computer science educators, we must allow

⁷ In contrast to the core subjects, goals of secondary CS0 courses are often nothing more than to "inspire" students, rather than preparing them for future coursework with rigorous instruction and skills. A math course whose primary objective was to inspire students to take more math classes would never be taken seriously.

access to learning computer science education in high school serve as a fundamental right to learn, rather than a stepping-stone for a future purpose. (Goode, Chapman, & Margolis, 2012)

The claim about "incredibly high, if not impossible standards for any one high school class to obtain" and "confounding variables" has, however, apparently not hindered the College Board from conducting several longitudinal studies on the positive quantifiable effects of the APCS-A course on choice of college major. One study correlated participation (not just passing scores) in the APCS-A exam with a 6- to 8-fold higher probability of choosing a CS college major (Morgan & Klaric, 2007). Another found that 20% of students who score 2 or higher on the exam choose a CS major, and that fully 27% of students earning a 5 go on to major in CS (Mattern, Shaw, & Ewing, 2011).

In this regard, what is missing and what no one seems to care about, is accountability in the form of evidence that any of the many thousands of students who have taken ECS have taken and *succeeded* in subsequent CS coursework. In sharp contrast, in this same 8-year time period, the half dozen successful APCS programs taught in LAUSD have produced over 500 students who have passed the AP exam (earning scores of 3, 4 or 5), including the 47 students in my small program in the 6 years from 2010 through 2015, plus 16 more who earned a score of 2. Applying the College Board's finding that statistically 20% of such students choose a CS major (and confirmed by reports from former students returning to visit, even some with scores of 1), the effectiveness of this author at a small school with not a penny in federal funding – or of any effective APCS teacher, really – dwarfs the non-existent results of the ECS program over a comparable time period in mitigating the *APIG's* so-called crisis.

Such statistics accentuate the stark contrast in student outcomes between ECS and APCS-A. LAUSD's current two-tier system for these two courses is eerily reminiscent of the (earlier described) track system of previous decades in which college-bound students took a math sequence culminating in calculus and pre-calculus, while students not bound for college took a general mathematics sequence terminating with pre-algebra (Brahier, 2005).

Goode et al. also asserted that "access to learning computer science education in high school" should be a "fundamental right". One can hardly disagree – as long as there is a caveat that CS instruction be rigorous. However, if the "CS knowledge" taught has been simplified to such an extent that it does not align vertically, it becomes a dead end, teaching disjointed concepts and skills whose level is so minimal that their real-world applications and capacity for vertical academic preparation are nil.

Lastly, the past several years have seen the emergence of campaigns like the "*Hour of Code*", meant to increase the numbers of K-12 students studying CS, and in particular programming. The goal is right on the mark, but, like the *APIG*, its means is largely hype: generating enthusiasm to bait students and their naïve teachers, and relying on the same ineffective and failing pedagogies that have left most students confounded since the beginnings of CS education. In March 2013, *code.org* posted a video on *YouTube* entitled "*Anybody Can Learn – code.org*" featuring software company CEOs and other high-level officers (all from privileged educational backgrounds) touting the message that coding is an easily acquired skill, despite the overwhelming evidence to the contrary. A second more minor, but still indicative, example of their campaign strategy, is the *Hour of Code* website's encouragement of pair programming, stating incorrectly

that "research shows students learn best with pair programming, sharing a computer and working together" (Hour of Code, 2015). In fact, pair programming requires continual teacher monitoring and is much more than "sharing a computer and working together". Moreover, rigorous studies that found mixed results and modest benefits of pair programming in postsecondary full-semester CS1 courses (Braught, Wahls, & Eby, 2011) have not been replicated in secondary classrooms, although recent middle-school studies have shown benefits under certain conditions, particularly when the members of a pair were friends (Werner, et al., 2013) (Denner, Werner, Sampe, & Ortiz, 2014). In ways such as these, the campaign trivializes and misrepresents the subject and raises false expectations and hopes.

These types of pitches have become fairly common, most recently playing out with adult learners. *The Washington Post* reported on the rapid expansion of coding *bootcamps* that promise high-paying jobs upon completion, but with courses costing upwards of \$10,000. The article critiqued such educational models, now being copied by some universities, with critics arguing metaphorically, "You emerge from a bootcamp fit to do an oil change, but not design a car".

Longtime tech recruiter Dave Fecak is worried about the push towards fast-paced, truncated coding programs. "We as a nation, as we talk about the STEM shortage, we're fostering a gold rush mentality that leads to these bootcamps with the promise of employment, promise of strong employment with strong demand and stability and a lot of money," he said. "And a lot of the people that may get coerced into signing up for these bootcamps may end up with a lot of debt and not a lot of job offers." He likens the trend to throwing bodies at the problem rather than addressing the industry's real need for **highly skilled developers**. (Turner, 2016)

Section 2. Technological Innovation is Not Pedagogic Innovation.

The largely unquestioned assumption spurring the manufacture of the *ECS* and *APCS Principles* frameworks has been that teaching programming to any but the most "naturally talented" of students is an insurmountable obstacle at the secondary level. An alternative narrative, however, is that the problem may lie with ineffective instructional strategies, that is, with pedagogy. To this end, instructor-developers have created new technologies to simplify the process of learning to program. Two extremely innovative and engaging programs that have exhibited real staying power over the last decade are MIT's *Scratch* and Carnegie-Mellon's *Alice*, drag-and-drop IDEs that allow students to skirt many of the syntax markers errors they are prone to make when programming in a text-based PL.

It has been suggested, however, that these and scores of similar changes to the *instructional technology* over three decades – be they in language, paradigm, platform, or context – have contributed little, if anything, to solving the persistent problems that confound novice programmers, and that their categorization as *pedagogic* interventions is only in the most superficial of senses (Kumar, 2013). Superficial because instructional resources that accompany these technologies follow a textbook formula/structure modeled after a traditional math curriculum: direct instruction and problem sets. The technologies may be novel, but pedagogically, there is no innovation in how they are used for instruction. Crucially, little or no thought has been given to how students actually learn. What's curious is that there are self-paced mathematics programs (e.g. ALEKS) that, while largely using Socratic methods, constantly assess student learning and adjust lessons and instruction accordingly. The equivalent in the CS educational

world is simply online problem sets that allow a learner to see whether solutions work for a range of input/parameter values.

When one reflects on why all of the attention in K-12 CS education has gone to technologic innovation, it seems only natural that CS instructors might innovate in the single area where they feel most competent, the technology component. Secondary CS courses are generally modeled after university courses, and college CS professors have had little to no training in education, particularly psychological theories of learning and child development. Because the way they were taught succeeded (for them), they might not naturally think about entirely different approaches to pedagogy, let alone different overall learning models for the core skill of the discipline (computer programming).

It took decades of K-12 math instruction to arrive at (a) strand categories that persist throughout these 13 years, (b) vertically aligned standards for each grade level, and (c) a rich variety of pedagogies for each level, strand and sub-discipline. CSTA has drawn up standards, but its results have been feeble in comparison, and generally useless in practice. CS and in particular computer programming have two things working against them in K-12 education. The field is young and rapidly changing, and it is not universally or consistently taught at these levels. There are few SMC secondary CS instructors, a situation that limits the types of day-in, day-out observations of student learning that can lead to reflection on how instruction might be made more universally effective.

Moreover, SMC secondary CS instructors are themselves not immune from the same biases as their postsecondary counterparts, thinking that students who are successful in their classes have "innate" talent; and the corollary, that otherwise intelligent students who lack these characteristics will fail, though studies to identify such traits have found

no correlations. If such ideas lie at the heart of instructors' belief systems, there's little incentive to investigate or change one's pedagogies.

It has been claimed that drag-and-drop programming interfaces have a pedagogic value, in that they lower a learner's cognitive load by circumventing the unforgiving need for proper use of syntactic markers when programming in a text-based PL. The scope of this effect, however, is extremely narrow, as it targets just a miniscule portion of a programming language's syntax – markers for command termination, signifiers for blocks of commands, a partial improvement on parameter passing – all areas that can be quickly and effectively addressed by memorization. Moreover, evidence is lacking that instruction that utilizes these IDEs have any benefits for novice programmers post-course, e.g. that whatever skills learned transfer to text-based PLs; or that they aid students in learning how to write and organize programs to solve novel problems, even within the narrow domains of the IDEs themselves. The advantage – the bang for the buck – that these impressive IDEs provide is thus theoretically very limited and unproven, at best. It is not an overstatement to say that if one views these IDEs as pedagogic interventions, that they represent the only significant innovation of the last decade.

Scratch and *Alice* also contextualize programming domains within their remarkable 2-D and 3-D virtual worlds, respectively, and there is no question that students find their initial encounters with these worlds compelling. However, the trade-off has been that programming problems and examples are constrained to the limited list of objects these 2-D and 3-D worlds provide. My experience has been that secondary student interest in these virtual worlds largely exhausts itself after a couple of months,

and the slate of categories of problems that can be done within them similarly runs its course.

There is also good reason to doubt that transfer of programming concepts and skills to other PLs can occur in the absence of programming fluency, if our experience with second language acquisition is any guide. There is some evidence that acquiring a third language is easier after fluency or comfort in communicating in a second language has been attained (Abu-Rabia & Sanitsky, 2010). In this vein, the practice of teaching novices one PL only to switch PLs each semester or year is as pedagogically disruptive and self-defeating as teaching a semester of Spanish followed by a semester of German, then a semester of Russian. It's true that the goal is to impart CS programming concepts, but one can't ignore the fact that those concepts are mediated by and tightly intertwined with a language. Students need to *acquire* sufficient language proficiency in how their first PL operates – how it handles the basic set of introductory programming concepts – before both language-related and CS concepts can transfer over and into a subsequent PL.

In broad terms, students choose a course of study with two considerations in mind: self-efficacy and interest. While interest is non-negotiable – students need to find some sort of personal connection to the subject matter – they must also feel confident in their understanding of the subject's concepts and their ability to apply them in non-trivial situations. Educators who contextualize programming instruction may effectively be addressing student interest. However, no amount of interest will give students the confidence required to continue studying the subject if they experience little progress or success in composing working programs on their own.

Pedagogical innovation gleaned from an ongoing cycle⁸ of *Teach; Observe; Collect and analyze data; Reflect* has for decades been the primary, ongoing methodology for improving student learning, and there is no reason to believe that CS educators are any exception. The results of this process for the core K-12 subjects – particularly mathematics – have been formalized pedagogies that have statistically demonstrated positive outcomes vis-à-vis student learning. The history of CS education, to the contrary, has consisted almost entirely of glossy technological "fixes" that fix little, at the near total neglect of considering – let alone investigating – alternative pedagogic approaches. It is a sad fact that CS instructors have not moved the needle one iota in terms of student learning since CS education began, and none can pinpoint why some students succeed while others struggle and fail.

⁸ The emphasis is on the word *cycle*, in that any of the four processes can be an entry point. In constructivism and the ideas of Dewey, there is much mention of the phrase "student-centered", crafting instruction to meet student needs. The corresponding process in the cycle, and often the most useful starting point, is *Observe*, paying attention to students – in particular noting the mistakes that students make, seeing their errors as evidence that teachers can use to speculate about theories of how they learn particular topics. Theories, whether correct or not, can inform instruction. Instruction can in turn be tested to ascertain its effectiveness.

Section 3. SLA Theories and Programming Languages

Subsection 1. Theoretical Considerations for Using a SLA Model in CS Instruction

CS1 instructors often dismiss the importance of learning any particular PL, because a primary objective of a programming course is to teach programming concepts, i.e. the use of a PL. Instructors have often rationalized this dismissal with arguments that languages come and go, and that CS majors will inevitably have to program in multiple languages. The illogic of how one is expected to learn concepts *mediated by a language* one doesn't fully understand how to use, though, has never seemed to be a serious consideration.

Only a handful of papers exist in the literature suggesting that language issues may impact CS education (Robertson & Lee, 1995). One researcher stated:

Language acquisition studies provided the theoretical basis and much of the relevant research for the [current] study. Drawing parallels between language classrooms and introductory computer science classrooms does not require much imagination. (Applin, 2001)

Applin's insight about the similarities to language classrooms notwithstanding, in reality there are vast differences between natural languages (NL) and artificial PLs, and one must consider whether a NL acquisition model is even applicable at all. Some of the extensive differences between NLs and PLs are summarized in *Appendix E*, though four characteristics of PLs in particular have critical implications for pedagogy. These are: (a) PLs are not spoken; (b) PLs are visual languages; (c) PL syntactic structures are less specific than those of NLs; and (d) in linguistic theory, PLs belong to a different grammar type than NLs. Nonetheless, the generative nature characteristic of all languages is commonality enough to argue that a NL acquisition model be employed in PL instruction.

A PL has no spoken counterpart. Learning to read and write a language without extensive prior speaking knowledge of the language is not a trivial undertaking, an assertion backed by decades of research into congenitally Deaf ASL (American Sign Language) signers learning English (as a second language because they have no access acoustically for acquiring the language). Nonetheless, a strong correlation between proficiency in ASL and English literacy has repeatedly been demonstrated (Kuntze, 2004). Studies of Deaf populations learning written English (or other spoken NLs) have consistently emphasized the importance of "mastery of a primary language" (i.e. ASL) for achieving reading competence in a second language (Perfetti & Sandak, 2000)⁹. Students learning a PL are in a similarly difficult position.

A corollary to the fact that PLs exist only in written form is that, like ASL, they are visual languages. The *sequence* of signs in the visual signal of a signed NL is temporally linear – like the auditory signal in a spoken NL. However, signed NLs are also *spatially* non-linear, not just because hands and arms move throughout a 3-D space, but because qualities of space, like direction, carry meaning. In like manner, the spatial positions of components in a PL also encode meaning. A prime example is the requirement or convention (depending upon the language) to utilize indentation to delineate scope and code blocks. Another is the directionality of assignment statements, and the requirement that L-values (in most languages) be on the left. A third is the spatial location of (a) local variables, (b) method/constructor parameters, (c) class/instance

⁹ The situation may be more nuanced, as other researchers have described a "bilingual" approach as best facilitating the acquisition of English by Deaf learners (*Subsection 3: Deaf ESL Learners*).

variables, and (d) global variables, whose positions define their scope and behavior¹⁰.

Although all three of these are seemingly syntactic qualities, anecdotally, memorization does not appear to improve their acquisition to any great degree. This may be because their patterns of occurrence are less clear-cut or because their meaning – the function they perform – depends more on context than do other syntactic markers. Counterexamples using varied contexts may help clarify the meaning of these positional features, but again are no guarantee.

Because PLs are visual, it also seems plausible that they might be acquired visually, at least in part. As mentioned above, congenitally Deaf ESL learners of written English would seem to have comparable, if not similar, L2 acquisition difficulties. There are, however, crucial differences between (hearing) PL learners and Deaf ESL learners. First, the brains of native Deaf ASL signers are organized for visual and spatial processing of language, while those of hearing PL learners are organized for auditory processing. Second, the brains of the two groups may be organized differently in terms of how they handle written language. Hearing learners process written languages either (a) orthographically/syllabically, that is, the letters / syllabograms / words correspond to sounds; or (b) logographically, where the symbols bear no relation to sound (as in traditional Chinese script or Japanese kanji). Note that PL learners have already internalized the orthographic writing system that PLs use. Moreover, visual word processing is lateralized to the same brain hemisphere as other classic language functions, indicating some degree of intertwining. Hence, learners may not necessarily visually acquire and/or process PLs at all. Interestingly, Deaf signers may differ among

¹⁰ Although method parameters are in effect local variables, their positioning in the method signature allows for them to be initialized outside of the scope of the method body.

themselves in how they process written English – they may do so orthographically or logographically, and/or they may associate words with ASL signs.

Although the syntactic footprints of PLs are at least an order of magnitude smaller than those of NLs, this does not mean that PLs are less complex or easier to learn. Each syntactic component in a PL is semantically broader than any of the more numerous NL syntactic elements. One way that PLs compensate for this increased generality in meaning is by combining components into precise sequences or blocks. These blocks can themselves be encapsulated into methods with user-defined names and reused, a generative capability that undercuts the first impression of a small footprint when viewing the body of a main() method. This generative capacity is magnified by the ability to create new "vocabulary" items ("identifiers") that allow one to perpetually extend the language in practice. The upshot is that there are virtually no syntactic components in NLs that have PL counterparts. The implication for learners is that one early, crucial and often-used L2 learning strategy – reference to analogous syntax structures in L1 – is simply not available to students learning a PL. Each and every syntax component in a PL is foreign in all senses of the word.

In linguistic theory, NLs and PLs are also classified as different grammar types. In 1959, Chomsky described a hierarchy of "formal grammars" that underlie "formal languages". In very general terms, a formal language is a set of symbols and the phonemic, syntactic and semantic rules and transformations for combining those symbols into statements that native speakers would recognize as correctly formed. Chomsky called the syntactic grammars of NLs "Type 3" *regular* grammars (finite automata). They are a subset of "Type 2" *context-free* grammars (systems of phrase structure), which

are the category to which PLs belong. The crucial difference between the two is that context-free grammars are "self-embedding" (a form of recursion, as it is understood in a CS context), giving them "excess generative power" (Chomsky N. , 1959). Were there no contradicting neurocognitive data, the theoretical differences between the two grammar types might also argue that the brain processes PLs differently from NLS.

That errors involving mismatched or orphaned braces are common may be a symptom of this difference in grammar types. Note that errors involving mismatched or orphaned parentheses are rare – although omissions of an entire pair of opening and closing parentheses to signal a parameter-less method call are not. Possible factors may be: (a) Proximity: the distance between opening and closing braces is variable and may be quite large, as in **class** definitions; (b) Familiarity: students have experience with the use of parentheses in math and English; (c) Explicitness: there is no clear-cut mechanism for indicating the affinity of a closing brace for its opening counterpart. One primarily linguistic factor, however, may be the *self-embedding* nature of braces (or indentation in Python) because PLs permit the nesting of blocks. This "excess generative" recursive power is a feature of Type-2 PLs that is absent in Type-3 NLS, and the language apparatus of the brain may simply not have the capacity for processing this kind of structure. One way to help compensate for this deficiency is to require that students add a comment after all closing braces to unambiguously indicate their matching partner:

```
void methodA( int x ) {
    for (int i = 0; i < 10; i++) {
        if (x > 100) {
            statement;
        } // if
    } // for
} // methodA
```

Although the numerous and severe differences between NLs and PLs seem to argue against the use of a foreign language model in a CS1 classroom, there is one overwhelming commonality: the grammars of both are *generative*, i.e. all languages are capable of generating an infinite number of statements or utterances. The theoretical implication of the generative nature of both NLs and PLs for instruction is that features of open-ended and infinite systems like languages cannot be taught like concepts in a mathematics curriculum, where computation is relatively straightforward and the critical thinking piece is recognizing and choosing an appropriate formula. Moreover, the meaning of words in NLs is learned from repeatedly hearing them used in multiple contexts, a process that allows one to differentiate them in varied environments. As such, the predominant CS pedagogy of demonstrating introductory programming paradigms and expecting students – under the guise of problem solving – to intuitively guess how ambiguous PL features can be combined, or interpreted in other, often barely recognizable, contexts, is an unrealistic expectation.

The alternative is a second language model, where the chief principles are repetition in meaningful and varied contexts, and production that is interactive and communicative. Foreign language curricula model language features as they are used in conversation in idealized, but typical social settings, over and over and over, using multiple contexts that allow learners to perceive the particular features *in relief* against varied backgrounds. Likewise, the meaning of PL features is facilitated by (1) exposing students to their use in multiple, but slightly altered contexts, and (2) giving students opportunities to use these features in a good number of slightly altered contexts.

Once competence in the use of basic PL features has been acquired, students are in a position to investigate problems within specific domains, again using the principle of repetition in meaningful ways, allowing students to assemble an increasing number of paradigms in their problem solving toolkits. "Problem solving" is actually not the best phrase for describing a key goal of programming instruction. Rather programming "literacy" would be a more accurate term, going beyond the minimum requirements of PL acquisition and even fluency. One might define PL literacy as the ability to use the language with ever increasing organization, proficiency and sophistication, and to grow increasingly adept at weighing clarity, conciseness and efficiency of algorithmic designs among possible coding alternatives. Kuntze discussed the difference between language acquisition and literacy as they relate to Deaf students learning English:

The definition of literacy as a specific way of using language posited by Olson and his colleagues offers a useful way of departing from the conventional ideas related to literacy and thinking about how literacy development can take place through face-to-face discourse. The current practice is to focus on the acquisition of English as an educational priority for children who do not have a native knowledge of English. The reasoning is that they need to have English competence, for without it, they will not be able to learn to read and thus acquire literacy. What Olson and Torrance are saying is that the prospect of literacy development is enhanced through exposure to the use of language in a particular way and exposure to a specialized mode of thought. Becoming literate is in effect more than merely the acquisition of special skills, such as word recognition. This perspective on literacy has far-reaching implications for deaf children, as well as for other children of language minority backgrounds. (Kuntze, 2004)

The challenge for CS educators, then, is to discover second-language-like pedagogies that, from day one, help their students first develop competence, then fluency, and then build on that foundation to develop literacy.

Subsection 2. A Review of SLA Theories Pertinent to Programming Languages

The literature has long expressed doubts about the utility of SLA research to inform the day-to-day pedagogic practices that foreign language teachers use in the classroom (Nassaji, 2012). One reason is a difference in focus: researchers have generally thought of the process of second language acquisition as being mediated through daily interactions with native speakers or other exposure to native speech, rather than through formal classroom instruction. Nevertheless, the broad outlines and implications of a handful of ideas that have arisen in SLA theory have spurred major shifts in foreign language instruction. Chief among these is the idea that second languages are acquired via subconscious processes, in much the same way that one acquires a mother tongue. The pedagogic model consistent with this idea is that learning in a second language classroom – actual language acquisition – occurs *following* repetitive and meaningful exposure to the language and interactive communication. There is a role for cognitive learning, but it is not the primary mechanism for acquisition.

Theories about second language acquisition prior to 1960 were second-hand versions of first language acquisition theories, which themselves drew upon ideas from descriptive linguistics (Bloomfield, 1933) and behaviorism (Watson J. , 1925) (Skinner, 1938). Chomsky's theory of transformational-generative grammar (Chomsky N. A., 1957) moved the focus of linguistics from description to explanation. Although the theory in no way claimed to explain the neurological processes for language, it did seek to model, explain and predict the brain's mental representation of linguistic knowledge. A transformational-generative model of language consists of a set of rules, and word/phrase/sentence structures on which these rules operate, that together can generate

(i.e. account for) all conceivable grammatically correct utterances/sentences. Although it is possible to build a generative model using only phrase structure rules – e.g. "Sentence → Noun-Phrase + Verb-Phrase", a transformational model provides key advantages. Using transformations not only greatly reduces the number of phrase structure rules needed to describe a language, but allows for a mechanism to see the common "logical form" of grammatically related sentences (e.g. declarative and interrogative forms, or active and passive forms). Transformations also allow for a deeper understanding of sentence structure. Given the sentence "Joe is going outside" and its interrogative form "Is Joe going outside?", one can formulate a rule that moves the auxiliary verb to the front of the sentence. Given a more complex sentence like "Those who have any information can contact the police", transformations help to explain why the auxiliary "can" (associated with the phrase "those ... can contact") moves to the front and not "have" (inside the relative-adjectival clause).

Transformations in a natural language cannot be observed directly; hence they are an inferred, underlying mechanism that illustrates the relationships between sentences. Although the same hold true for PLs, some transformational-generative structures in PLs can be visible surface syntactic forms, like any other. The most obvious examples are *iterative* control structures, which allow for a huge economy in program size.

Chomsky was a vociferous opponent of behaviorism, and in particular, as it impacted linguistic theory. He held that the preponderance of evidence, particularly the ease with which children learn language, argued overwhelmingly for an innate "language acquisition device" (LAD). This "nativist" stance also stood in contrast to Piaget's cognitive view, which held that general learning processes could explain language

acquisition. One's perception of language acquisition as nativist or cognitive directly impacts the types of pedagogic strategies one would presuppose to be most effective for foreign language instruction.

As Chomsky's ideas came to dominate linguistics over the next decade, researchers began to look at "what learners actually do and produce, as well as the context in which they learn, rather than merely focus on the description of source and target languages" (Myles, 2010). What Chomsky meant by "language acquisition" was a subconscious process via which all children naturally learn a first language using an innate LAD. The phrase as it is used in SLA, however, has little precision, as end-state fluencies can vary widely. Young children, and even many of those into their teen years, can eventually learn a second language with native fluency. Although it is not unusual for adults to acquire syntactic and semantic fluency or near-fluency, lack of phonemic or phonological fluency often leaves a detectable accent. Some adults may attain semantic fluency, but persist in making the same syntax errors. On a different tack, the focus of some researchers might be literacy. Here, there is again a wide variance. Second language learners may have native fluency but often poor literacy skills. Others might have thick accents, but otherwise speak and write flawlessly. In either case, it is possible that literacy is more a representation of first language (L1) literacy than second language (L2) proficiency.¹¹ When weighing competing SLA claims, then, it is prudent to keep in mind what types of end-states researchers might be addressing. Given the wide range of outcomes, a safe baseline definition of acquisition within an SLA context might be

¹¹ One might speculate, too, that PL proficiency/literacy might reflect the level of a learner's native language literacy, similar to the correlation of proficiency in ASL with English literacy (Kuntze, 2004)

semantic fluency, the ability to communicate as easily and effortlessly as competent native speakers.

In the area of foreign language pedagogies (FLP), the mid-1960s was the heyday of the audio-lingual method, a popular curriculum grounded in behaviorism that inundated students with drills, prodding them to repeat phrases and sentences uttered by teachers and audiotapes, with prompts signaling modifications that students were supposed to incorporate. An example of such a sequence appears below:

Teacher: I am studying history.
Students: I am studying history. (*repeating*)
Teacher: You.
Students: You are studying history. (*replacing*)
Teacher: She.
Students: She is studying history. (*replacing*)

Soon after the audio-lingual method appeared, however, arguments against it began to surface, suggesting that instructors follow more natural (i.e. communicative) approaches (Newmark, 1966).

In 1967, Corder proposed that second languages were learned in fundamentally the same way as first/native languages (L1); that is, speakers acquire a second language (L2) when they have both (1) sufficient motivation and (2) ongoing exposure to the language data. He also proposed that L2 learner errors – *systematic* errors (i.e. evidence of an underlying language system) as opposed to *performance* errors, after Chomsky's competence/performance dichotomy – be viewed not as "mistakes", but rather as evidence of the ongoing process speakers undergo in constructing an internal grammar of L2. Theories of second language acquisition, Corder claimed, would need to explain and take into account such phenomena. He offered the following interaction as an example:

Mother: Did Billy have his egg cut up for him at breakfast?
 Child: Yes, I **showeds** him. (1)
 Mother: You what?
 Child: I **showed** him. (2)
 Mother: You showed him?
 Child: I **seed** him. (3)
 Mother: Ah, you saw him.
 Child: Yes, I saw him. (Corder, 1967)

The three errors made by the Child are not random, but rather evidence of an L1 system in which the rules for certain grammatical distinctions are still in development. The Child first uses past and present tense inflection markers simultaneously on a single verb. She/he then uses the verb *show* in place of the verb *see*. The relationship between the verbs and the use of one for the other may be difficult to understand within an English grammar context. However, in languages like Hebrew, the two verbs have the same 3-consonant root, and the semantic distinction between them manifests when *show* is inflected using the *causation* or *causal* marker. The third error is an exception to the general past tense conjugation rule (*see* + *past-tense-marker* → *saw*) that the Child has not fully assimilated.

Systematic syntax errors can also be observed in the programs written by beginning programming students. In the experimental study (Chapter 1), 8 of the 31 students – 6 from the group with poor memorization scores and 2 from the group with high memorization scores – incorrectly appended semi-colons to Java method headings in two ways:

```

(1) void drawThickVertBlueLine() ; {
(2) void drawThickVertBlueLine() ;
  
```

The 8 students made this error despite (a) having the demo program literally at their fingertips for reference, and/or (b) having memorized and written method headings

correctly in the demo program. The simplest explanation is that having internalized the syntax rule for semi-colon termination of *statements*, these students have not yet perceived a clear distinction between *method headings* and *statements*, and therefore over-generalized the rule to what seemed to be a similar-looking syntactic structure. The same type of error can be observed in loops and conditional statements commonly and universally observed and reported elsewhere (Kummerfeld & Kay, 2003) (Spohrer & Soloway, 1986) (Garner, Haden, & Robins, 2005):

- (1) `for (int i = 0; i < 10; i++) ; {`
`}`
- (2) `if (x == 7) ; {`
`}`
- (3) `while (!done) ; {`
`}`

In the course of the year, a novel spatial error related to the direction of assignment statements was observed. The topic, a variation of swapping in place, asked students to rotate the elements of a 3-member integer array. Students were first shown how to rotate the elements left one position using a temporary 4th variable named **save**, positioned to the left of the array in an instructional diagram (Figure 6).

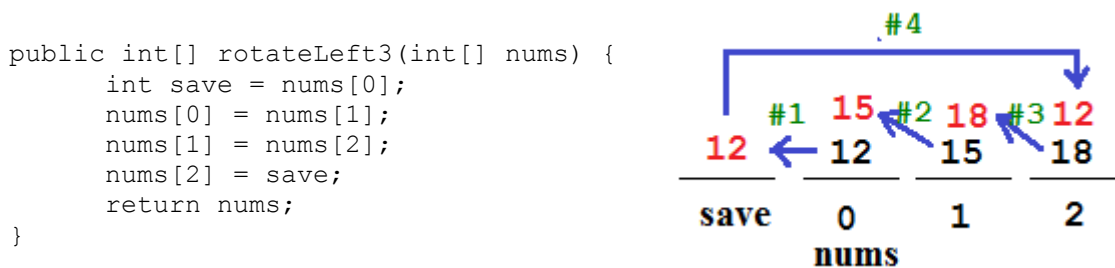


Figure 6. rotateLeft3() instructional code and diagram

When then asked to write the method for `rotateRight3()` , several students submitted code and a diagram like those in Figure 7.

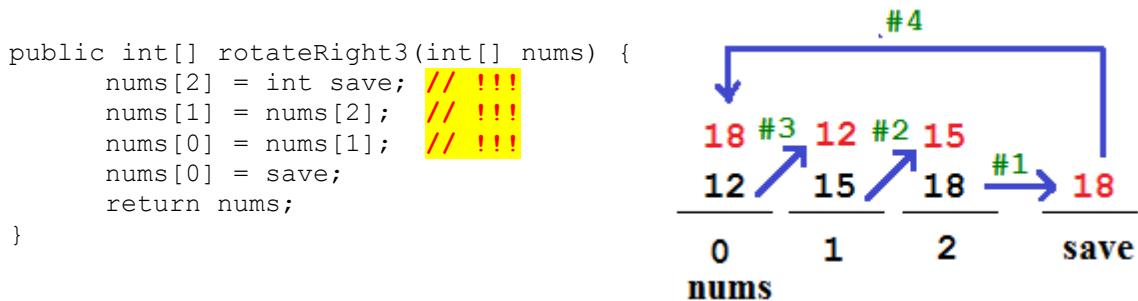


Figure 7. `rotateRight3()` direction error

The assignment statements that students wrote mimicked the variable locations used in the diagram they'd constructed (with the swapping variable arbitrarily placed to the right of the array). Had the original instructional diagram illustrating the movement of values for `rotateLeft3()` represented the array and the `save` variable *vertically*, the error might not have occurred – although the names of the methods themselves implicitly convey the assumption of a directionally-specific left-to-right array model. Crucially, though, the appearance of this error psycho-linguistically argues for the spatial, visual character of programming languages. Instructionally, this kind of problem might be used as a counterexample for the types of information one can and cannot glean from diagrams. The example also serves as a cautionary note against an automatic instructional assumption that the direction of assignment is self-evident.

Corder used the term "*transitional competence*" to describe the developing language system revealed by systematic errors. This intermediary language system became known later on as an *interlanguage* (Selinker, 1972). One way to think about the systematic errors of an interlanguage is that they are evidence of a learner's application of hypotheses about the input language data to which he/she has been exposed. The

examples of semi-colon termination and assignment direction errors discussed above are consistent with an interlanguage stage in PL acquisition.

The first major study of systematic errors concluded that the vast majority of errors were developmental, meaning simply that the acquisition of certain language features are age-dependent (Dulay & Burt, 1973). The study of 145 five to eight year-old native Spanish speakers found that the L2 English errors these children made were the results of over-generalization and the omission of syntactic *functors*¹² – the same types of errors observed in the developing language of native English speakers. They concluded that the source of L2 errors was not caused by L1 "interference" – children in this age group borrowing features from L1 "habits" – as a behaviorist model would have predicted. In the study's discussion section, they advocate for a natural communication pedagogic model (which they called "creative construction"), where the "attention of the speaker and hearer is on the "message," or content, of the verbal exchange rather than on its form [structure, syntax]," arguing that syntax will be acquired naturally via innate processes.

Several studies demonstrating predictable stages in the acquisition of specific syntactic features of children's first language had been done by this time. One well-known case involved asking children aged 5 to 10 whether a blindfolded doll was "easy to see" or "hard to see". The children who responded that the doll was hard to see, and were then asked to make her easy to see, removed the blindfold. Older children who responded that the blindfolded doll was easy to see, and were then asked to make her hard to see, employed a variety of means to place the doll out of view (Chomsky C. ,

¹² Functors are grammatical morphemes that play a secondary role in semantics, such as "noun and verb inflections, articles, auxiliaries, copulas and prepositions", such as the underlined elements in "the cat is meowing at the moon".

1969). Whether similar stages occurred in L2 acquisition was unknown until Dulay and Burt, in an additional study of three geographically diverse groups of children, reported that L2 speakers acquire functors in a predictable order; the order, however, was different from the children's L1 acquisition of functors.

In a study of 73 adult subjects like the one just described, another team of researchers found a predictable sequence for acquisition of functors in adult L2 English learners, despite the subjects having (a) different amounts of ESL instruction, (b) different first languages, and (c) different levels of exposure to English. Again, one would expect individual differences in the sequence of functor acquisition were the subjects relying heavily on features borrowed from their different first languages (Greek, Farsi, Italian, Turkish, Japanese, Chinese, Thai, Afghan, Hebrew, Arabic, and Vietnamese). Moreover, this sequence was similar to that found by Dulay and Burt in their study with child L2 learners. The researchers concluded that even though adult L2 learners overall do not attain the native-level skill in L2 as child L2 learners, nevertheless "they process linguistic data in ways similar to younger learners" (Bailey, Madden, & Krashen, 1974).

Although certain aspects about language acquisition can be deduced from systematic errors, later researchers made the point that relying primarily on erroneous data ignored the potential for identifying strategies, particularly *communication strategies*, that native and L2 language learners actually used. A team of researchers in the 1980s identified a hierarchically ordered list of twelve learning strategies that the group of children they studied employed when acquiring a second language. The three

initial, and most frequent, strategies were *repetition, memorization* and *formulaic expressions* (phrases that function as units) (Chesterfield & Chesterfield, 1985).

Several researcher-theorists in the late 1970s, including Bialystok, Swain, Long and Krashen, worked on different aspects of a model for second language acquisition that presupposed a dichotomy: conscious, active, intentional learning strategies (*explicit*) versus subconscious, passive language acquisition processes (*implicit*). There was general agreement that the evidence supported the idea that L2 learners used primarily implicit processes to acquire grammar. Explicit strategies were used for "monitoring", that is, noting exceptions to grammar rules, or for corrective tasks such as adjusting patterns/rules that were learned incompletely or with minor errors. The two primary requirements for L2 acquisition in this model, per Krashen, are *comprehensible input* and *meaningful communication / interaction*.

Language acquisition does not require extensive use of conscious grammatical rules, and does not require tedious drill. It does not occur overnight, however. *Real language acquisition develops slowly*, and speaking skills emerge significantly later than listening skills, even when conditions are perfect. The best methods are therefore those that supply "comprehensible input" in low anxiety situations, containing messages that students really want to hear. These methods do not force early production in the second language, but allow students to produce when they are "ready", recognizing that improvement comes from supplying communicative and comprehensible input, and not from forcing and correcting production. (Krashen S. , 1982)

Describing the function of the classroom, Krashen continues:

...we have to provide students with enough comprehensible input to bring their second language competence to the point where they can begin to understand language heard "on the outside", read, and participate in conversations. Since they will be less than fully competent, we also need to provide them with tools for encouraging and regulating input...

...all second language classes are transitional, and no second language class can be expected to do the entire job ... second language classes are best thought of as

places to gain comprehensible input in early stages, when the acquirer does not yet have the competence to understand the input provided on the outside.

Interestingly, the implicit/explicit dichotomy echoes a concrete/abstract paradigm, modeled after Bruner (Brahier, 2005), and developed for teaching Algebra 1 to middle school students. "*Implicit*" is bottom-up, data driven, subconscious, concrete. "*Explicit*" is top-down, rule-based, conscious and abstract.

Many algebra topics can be addressed in the morning class in a concrete, manipulative, or real-world context. Then the same topic can be revisited in the afternoon at the more abstract level typically seen in a secondary algebra class. The students make a much stronger connection to the subject matter this way.

... higher mathematical content is not out of the reach of younger minds if we present it concretely. (Fulton, 2005)

Krashen later proposed the *Input Hypothesis* (Krashen S. , 1985), which claimed that comprehensible input – and not production (output) – was what spurred language acquisition. At about the same time, however, Swain argued "that learners not only need comprehensible language input, but that they also need to produce output in order to develop their communicative abilities in the L2 to a *high standard*" (Myles, 2010). Krashen has continued to dispute this (Krashen S. , 1998), but has only referred to the ineffective strategy of "forced" production, saying nothing about the role of production once it occurs naturally. More to the point, one might wonder how Krashen could ascertain or assess a learner's language acquisition if not by his/her speech or writing, i.e. one's productive capabilities. Swain, Long and other researchers seem to form a solid front opposing this view: whatever processes occur in speakers once they comfortably speak and produce unquestionably accelerate competence. Certainly young children are motivated to communicate to satisfy their wants and needs more effectively than they could by showing signs of distress (though that's always available as an emphatic last

resort). Children access the subconscious hypotheses they have about language as they take part in a back-and-forth productive negotiation with proficient speakers until their communication has been understood. Their hypotheses are honed in this process, with their language system adapting itself more precisely to the language surrounding them. Meaningful communication is just as vital to this kind of language acquisition process for second language learners.

A similar core principle is involved in mathematics education: one can't learn mathematics effectively by just reading or listening to lecture; rather you learn mathematics by doing, applying mathematics to solve problems. In so doing, you activate knowledge and reinforce it by making connections. Certainly, the same holds true for computer programming. Although "communicative and comprehensible input" may be enough for language acquisition, the higher standard of literacy requires production.

The next decade saw a back-and-forth between the *nativist* camp and those returning to the prevailing psychology model (which placed much less emphasis on subconscious processes than is generally accepted today) and more explicit *cognitive* models of L2 acquisition, particularly as general learning theories like constructivism influenced L2 acquisition models. In the mid-1990s, the beginnings of a sociocultural theory relying heavily on the work of Vygotsky emerged asserting that L2 acquisition was more complicated than the individual-centered input/output model. Rather it was heavily influenced by social interaction mediated by symbolic mental functions, in particular language, using either L1 or L2 or both. Researchers in this camp, for example, studied Japanese ESL learners and surmised a role for *private speech* in L2 learning (Ohta, 2001). A first criticism of this study (and others like it) is that, although

Vygotsky-like functions/strategies may have been *observed* in qualitative studies, evidence that they actually *impact* acquisition is assumed. A second criticism is that Vygotsky's notions of audible private speech and silent inner speech applied to children (Woolfolk, 2004), not to the college-aged students in this study.

One interesting study reinforced Krashen's assertion that the correction of production grammar errors by "recasting" (responding to an error by vocalizing the correct form and having the student repeat it) had little to no influence on acquisition. Other correction methods, however, were demonstrated to be effective, undermining the broadness of Krashen's claim (Lyster & Ranta, 1997). A connectionist theory was also being articulated around this time asserting that a machine learning model based on the probabilities of patterns, and not innate rules, drives L2 acquisition. The theory, however, adds little understanding to the discussion, since the complex task of assigning probabilities cannot possibly be a conscious process; the model simply presupposes a different mode of neural functioning inside a black box.

Subsection 3. Deaf ESL Learners

As mentioned earlier, research about Deaf ESL learners may offer fresh insight into how PLs might be learned/acquired. A recent qualitative study looking at how Deaf children become proficient in English literacy – although lacking in conjecture or theory – provides a model for how CS educators might fundamentally rethink PL instruction. The children in the study, who were raised by Deaf parents in a bicultural / bilingual household, (a) learned ASL as their first language, and (b) grew up in an English immersion environment, that is, one where written English and signed English were used extensively. The researchers identified several "themes" that facilitated literacy within this overarching framework (Mounty, Pucci, & Harmon, 2014).

The first theme is that "each language supports the development of the other," that there is a "bidirectional relationship" between ASL and English literacy, wherein the latter also improves skill in the former. One girl is described as at first translating English words using their literal signed counterpart, but "increasingly choose[ing] signs that matched the meanings conveyed by words and phrases in a given passage," gaining increasing automaticity in translation ability as English literacy progressed.

The second theme is "providing a print-rich culture for *ongoing exposure* to English." Parents guided their children in the use of computers for communication or literacy tasks, read them bedtime stories, and encouraged them to write in "daily journals" and read "newspapers, magazines, books and comics". Like their hearing counterparts, the Deaf parents would read books, like Dr. Seuss, to their children over and over. They would not only translate the stories into ASL, but would use prosodic features (facial

expressions, gestures) to clarify meaning. Crucially, though, they would fingerspell key English words, turning each written word into a living visual language form.

This last parental interaction is in fact the third theme, that fingerspelling is a bridge between ASL and English and that children should be exposed to fingerspelling even as toddlers. The unspoken implication is that the ongoing exposure to these fingerspelled English words, intermediate forms occupying a space between the ASL signs for the English words and the written English words themselves, provides these Deaf children with a visual English counterpart to their hearing peers' spoken English. Critically, this would partially make up for the advantage the latter have in already being native speakers of the language they are learning to read. Although this would not seem to substantially benefit Deaf children in the task of acquiring English syntax, there is no question that it would enrich their English language lexicons and lessen the cognitive load when trying to learn other English language features.

The fourth theme is related to socio-cultural theory: children "must have opportunities to interact with and observe ASL/English bilingual adults" both in school and at home. The types of activities described are (a) situations that clarify metalinguistic awareness, i.e. reveal the relationship of language to cultural behaviors, (b) metacognitive strategies to infer the meaning of unfamiliar words, (c) strategies for recognizing errors, (d) differences between formal and informal language use, and (e) providing multiple contexts for words – *and signs* – that have multiple meanings so that correct meanings can be deduced from the context. This last activity seems to be of particular importance. Hearing children have a distinct advantage in their ready exposure

to the use of English language features in multiple contexts, whereas opportunities for Deaf children in this regard are much more constrained.

Subsection 4. The Role of Context

A research team at MIT studying the acquisition of words by one child videotaped over the first three years of his life has shown "that words used in distinctive spatial, temporal, and linguistic contexts are produced earlier, suggesting they are easier to learn" (Roy, Frank, DeCamp, Miller, & Roy, 2015). One impetus for the study was the idea by first language acquisition theorists – including J. Bruner, E.V. Clark and E.A. Cartmill – that the quality of extra-linguistic contexts can affect language learning in children. The measure of "word learning" in this study was the Age of First Production (AOFPP): "the point at which the child first made use of a phonological form with an identifiable meaning." Although repetition was a useful predictor for acquisition of concrete nouns, it was not predictive for predicates or closed-class words (e.g. *here, more, if, but*). A better predictor for these latter two classes of words were shorter MLUs (mean length of utterance), defined as the complexity of the sentences in which they were used. However, the best predictor of when the child would connect meaning (semantics) to the sound (word) – even higher than frequency/repetition – turned out to be the *distinctive context* of a word. Three distinctive contexts were defined: "the location in physical space where it [the word] is spoken, the time of day at which it is spoken, and the other words that appear nearby it in the conversation", i.e. space, time and language. The authors concluded:

The more tied a word is to particular activities, the more distinctive it should be along all three measures, and the easier it should be to learn. Consistent with this hypothesis, contextual distinctiveness (whether in space, time, or language) was a strong independent predictor of the child's production...

...The three distinctiveness variables showed strong correlations with one another and striking consistency as predictors of the age at which words were first produced. This consistency supports the hypothesis that each is a proxy for a

single underlying pattern: Some words are used within coherent activities like meals or play time (e.g., *breakfast, kick*), whereas others are used more broadly across many contexts. *These **differences** may be a powerful driver of word learning.*

In a TED talk, principal investigator Deb Roy spoke informally about this study:

Every time my son would learn a word, we would trace back and look at all of the language he heard that contained that word. And we would plot the relative length of the utterances. And what we found was this curious phenomenon, that caregiver speech would systematically dip to a minimum, making language as simple as possible, and then slowly ascend back up in complexity. And the amazing thing was that bounce, that dip, lined up almost precisely with when each word was born -- word after word, systematically. So it appears that all three primary caregivers -- myself, my wife and our nanny -- were systematically and, I would think, **subconsciously restructuring our language to meet him at the birth of a word and bring him gently into more complex language.** And the implications of this -- there are many ... feedback loops. Of course, my son is learning from his linguistic environment but the environment is learning from him. That environment, people, are in these tight feedback loops and creating a kind of scaffolding that has not been noticed until now. (Roy D. , 2011)

It's a near certainty that this precise interaction is outside of Krashen's notions of *comprehensible input* and *meaningful interaction*, but it would fit comfortably into a socio-cultural language acquisition model. The problem is that this type of scaffolding, per se, doesn't explain how neglected children acquire language. It might, however, help delineate the types of interactions that children need to achieve higher levels of literacy.

Catherine Snow, an education professor at Harvard, reiterated the importance of encountering words in multiple contexts, both for native and non-native English language speakers:

Well, what we decided to do was go beyond the kind of vocabulary instruction that we know doesn't work very well. A list of twenty words, study the definitions, use the words in a sentence, take a quiz on it at the end of the week, then next week go on to another twenty words. That doesn't work for lots of reasons. It's easy enough to do it, and to forget those words. In order to have a high probability of learning a word, you need to encounter it fifteen, twenty times.

Typically these are words that have slightly different meanings in different contexts. So it's important to encounter them in different content areas. If you think about a word like variable, in math a variable is, of course, a technical term. But in history you might talk about variable responses to the change in government. Or in science you might talk about a variable in an experiment. **So recurrence in a variety of contexts. Opportunities for students not just to read the words and write with the words, but also to use them.** (Arditti & Skirble, 2010)

Subsection 5. Neurocognitive and fMRI Studies

Finally, there remains the critical issue of whether the human brain processes NL and PL tasks in fundamentally similar ways. Although evidence demonstrating that NLs and PLs are *acquired* in similar ways would make for a much more convincing argument for the use of language pedagogies in PL instruction, the use of similar or identical brain areas when *processing* both language types would provide a strong argument as well.

Linguistic theories about syntax have been greatly informed by early aphasia studies, in which injuries to the brain affected features of language processing. Broadly speaking, Broca's aphasia was found to impact expression/production; Wernicke's aphasia caused receptive/understanding difficulties; and conduction aphasia caused an impaired ability to repeat prompted input, as if the communication channel between seemingly well-preserved expressive and receptive functions was damaged. Imaging studies later correlated several of the 52 Brodmann areas of the cerebral cortex with these aphasic conditions. For example, Broca's aphasia mapped to Brodmann areas 44 and 45; Wernicke's aphasia mapped to Brodmann area 22; and conduction aphasia mapped to Brodmann areas 39 and 40.

An internet search for possible links between aphasia and computer programming resulted in just one anecdotal reference that supported a causal link with language:

Working as an engineer for an aeronautical firm, Christopher was 33 when he had a stroke, which left him with some physical weakness and moderately severe aphasia. His job was '*relatively complicated*', involving computer programming, mathematics and other technical skills... Six months after his stroke, Christopher went to see his boss and persuaded him to let him try some programming: '*I told him my predicament and I said: "Could I borrow a spec ... a specification?" – Because this was the specification I used to programme a computer software. And I couldn't do it. – I thought I could do it and I ... thought I could do it and ... then again it wasn't surprising. – That event told me I could never do it again.*' (Parr, Byng, & Gilpin, 1999)

Within the last few years, researchers have used fMRI – a non-invasive technique – to visualize and associate regions of the brain that become activated with specific neural processes/tasks. One such neurocognitive study demonstrated that the "syntax-like" operations involved in understanding and solving algebra tasks are processed in areas of the brain associated with "representation of quantity, Arabic numerals, and calculation", distinct from areas associated with language tasks. The authors had cited previous studies of neurological patients suggesting that the two abilities were physiologically dissociated:

Particularly relevant is the case of patients with agrammatic aphasia, who exhibit intact understanding of the rules, structure, and operations of abstract algebra but perform at chance levels in standard assessments of language (Monti, Parsons, & Osherson, 2012).

Key evidence that PLs are processed like NLs came in a recent fMRI study of 17 first-year programming students that looked specifically at comprehension of computer programs. Short programs (fewer than 20 lines) which performed string-reversal, array searching/sorting, and integer-related tasks were shown to participants. To measure comprehension, research subjects were asked to determine the output for each program. For control tasks, participants were also given a duplicate set of these same programs, but containing a variety of syntax errors, such as (a) missing identifiers, and (b) opening parentheses, brackets, and quotations that were matched with incorrect closing characters. A baseline of brain activity for the task was gotten by asking participants to detect these errors. The reasoning was that because participants were occupied with scanning the surface form of each error-containing program – rather than trying to understand what the program did – these measurements could serve as controls to filter out the contribution

from non-relevant baseline visual and audio brain processing. When measuring program comprehension, the researchers found activation over baseline in Brodmann Areas (a) 6 and 40, and (b) 21, 44 and 47. The first group is associated with comprehension, the second group with language processing (Siegmund, et al., 2014). More specifically, BA 40 is associated with semantic and phonological reading processes. BA 21 is associated with semantic retrieval, linguistic inference, word form processing, and recognizing words with similar meanings. BA 44 is associated with semantic tasks and, together with BA 45 in the left hemisphere, forms Broca's area, which when damaged results in an impairment of the ability to speak fluently and grammatically, or to write. Specific tasks include syntactic and grammatical processing and sentence comprehension. BA 47 has been associated with a range of language tasks: semantic processing, coding and retrieval; phonological processing, reading single words, lexical inflection and intonation aspects of prosody. CS Professor Chris Parnin at North Carolina State University summarized the study's most significant results:

The team found a clear, distinct activation pattern of five brain regions, which are related to language processing, working memory, and attention. The programmers in the study recruited parts of the brain typically associated with language processing and verbal oriented processing (ventral lateral prefrontal cortex). At least for the simple code snippets presented, programmers could use existing language regions of the brain to understand code without requiring more complex mental models to be constructed and manipulated.

Interestingly, even though there was code that involved mathematical operations, conditionals, and loop iteration, for these particular tasks, **programming had less in common with mathematics and more in common with language.** Mathematical calculations typically take place in the intraparietal sulcus, mathematical reasoning in the right frontal pole, and logical reasoning in the left frontal pole. These areas were not strongly activated in comprehending source code. (Parnin, 2014)

Christian Kästner, a professor at Carnegie Mellon University and the study's second author, was upbeat about the possibility of psycholinguistic similarities between PLs and NLs:

There is no clear evidence that learning a programming language is like learning a foreign language, but our results show that there are clearly similarities in brain activations that show that the hypothesis is plausible. (Amirtha, 2014)

It is presently not possible to suggest, that because PLs are *processed* in the same areas of the brain as L1s and L2s, that they are *acquired* in parallel ways as well. Other types of studies would need to be done to convincingly demonstrate this kind of association.

Nonetheless, this study provides sufficient and intriguing evidence to argue for investigating the efficacy of language pedagogies in computer programming instruction.

Section 4. SLA Instructional Strategies for Computer Programming

...we cannot really teach language, we can only create conditions in which it will develop spontaneously... (Corder, 1967).

*It [language acquisition] does not occur overnight, however.
Real language acquisition develops slowly. (Krashen S. , 1982)*

Most contemporary language teachers employ interactive instructional strategies, be it immersion or some variation of the communicative approach. Both of these emphasize activities where students communicate in, and hear and read authentic examples of, the target language. This stands in stark contrast to the dominant pedagogic model used by college and secondary CS0 or CS1 instructors. The abstract top-down approach they overwhelmingly employ – lecturing about, then referring to, the syntax rules of a programming language's grammar to construct statements and programs – might as well have been lifted from an early 20th-century Latin classroom, as if instructors were teaching a dead language.

SLA theory is hardly settled. If anything, though, there is sufficient evidence and agreement about the following principles:

- a. Second language acquisition is implicit, not consciously learned.
- b. Second language acquisition is facilitated by repeated exposure and interaction in varied meaningful contexts.
- c. Second language acquisition takes considerable time.

Specific instructional strategies informed by these principles were designed and used in my introductory CS classes and are described in this section. Anecdotally, they appear to help all students deepen their understanding of PL concepts and quicken the acquisition of PL proficiency.

Subsection 1. Syntax: Memorization

Although the experiment described in Chapter 1 was not convincing – a moderate inverse correlation that fell short of the 95% confidence level – nevertheless, for several years now, I have observed that a memorization strategy helps students acquire the ability to minimize or avoid syntax errors. What might be the mechanism underlying this? Successful memorization of programming paradigms – written by a "native speaker", i.e. a SMC programming instructor – requires that students study a text meticulously, over and over, until they can reproduce it perfectly. One might think of this process as functionally mimicking Krashen's comprehensible input requirement: bombarding the brain with data. Like young children learning a NL, subconscious, passive and innate processes internally construct syntactic rules by induction from the patterns in the data. Even a program fragment composed of a few methods has enough statements and headings to sufficiently define the patterns for inferring the placement of a statement's syntactically meaningful punctuation markers (semi-colons, commas, parentheses, curly braces) with respect to its identifiers. An apt metaphor can be found in the film *The Karate Kid*, when Mr. Miyagi directs Daniel to perform four days of repetitive chores – sand-the-floor, wax-on/wax-off, paint-the-fence, side-to-side. The motions – surrogates for the comparable movements of defensive blocks – worked their way into his muscles' memories.

In preparation for their first project, students typically receive instruction in (a) the mathematics of the inverted y-axis graphing plane that ***Processing*** employs; (b) the mechanics of RGB color; and (c) primitive methods for rendering lines and rectangles. They are then given the task of writing a program to reproduce a *Piet*

Mondrian painting (Part 2, Chapter 2, Section 2). They are asked to memorize the starter code below, which produces the drawing in **Figure 8**:

```

void setup() {
  size(479,550);
  background(0);
  yellowRect();
  redRects();
}

void yellowRect() {
  noStroke();
  fill(255,215,0);
  rect(5,3,32,93);
}

void redRects() {
  fill(255,0,0);
  noStroke();
  rect(43,3,162,93);
  // draw horz black line
  stroke(0);
  strokeCap(SQUARE);
  strokeWeight(8);
  line(42,11,42+163,11);
}

```



Figure 8. Output and Detail of Starter Code for Piet Mondrian Painting

Students are given no instruction regarding *syntactic rules* for method headings, bodies or primitive statements. Rather they appear to acquire these rules after memorizing the paradigm starter code and are easily able to complete the program with the approximately 20 parameter-less methods needed for rendering the remaining rectangles. They are also instructed to fabricate accurate and meaningful names for the methods. Language-wise, this "production" part of the assignment prods students to discover how the language operates via the ongoing feedback they receive as they run the program-in-construction rendering the image-in-progress. The pedagogic *program-run-reconsider-modify* loop serves the function of "meaningful interaction", allowing the user to internally construct

and adjust semantic information pertaining to primitive method parameters, the ordering of statements, method calls, and so forth.

After completion of this unit, students are then asked to memorize progressively more complete versions of the *Ricocheting Comets* program (Part 2, Chapter 2, Section 3). Memorization at this stage has two objectives: (a) students are quickly, but progressively, exposed to the *syntactic form* of conditional statements, variables, iteration and arrays in the PL equivalent of simple, but authentic, sample dialogues in the target language; and (b) having been thus primed for familiarity with the building blocks of the program, the cognitive load is lessened when students subsequently perform exercises meant to convey the meaning of those blocks (described in *Subsection 2*). The syntactic expectation for the initial memorization part is that by exposing students to Java's most common syntactic structures and markers in multiple contexts¹³ in a short and uncomplicated, but evolving program, they will implicitly learn and acquire their correct syntactic usages. The foreign language classroom counterpart is memorization or repeated study of a unit dialog.

At first, students are guided incrementally through the construction of a *procedural* version of the program, which renders a single comet. Guided instruction then helps them convert this to an *object-oriented* (OO) program, so that multiple comets can be rendered. The semantic expectation is not that students *master* any of these PL constructs, but that they begin to gain familiarity by seeing their use in a complete and working program for this particular domain type (e.g. using conditional statements to

¹³ Note that the strategy of *multiple contexts* has long been used in mathematics teaching. Counter-examples are one way to emphasize pertinent features and applicable uses. Algebraic concepts – simple examples being perfect squares or the quantity $(a+b)^2$ – are presented with their geometric counterparts and representations. Algebra tiles are wildly successful in visually modeling algebraic factors and products.

reverse the vector of direction when the comet objects come into contact with window boundaries). Both the procedural and OO versions of the program for a single comet appear below, followed by the final OO version for multiple comets, which employs iteration to traverse an array (a for-loop that uses a counter variable for initialization and a for-each loop for movement). By the end of the unit, most students will have acquired the ability to program with proper syntax, to find and correct unintentional syntax errors, and to carry this knowledge forward in subsequent programming assignments. There will be a few students who will have not fully learned these objectives, but they will make many fewer of the kinds of novice systematic errors discussed in Section 3, allowing them a quicker and less overwhelming "monitoring" process than would normally be the case with struggling students.

Procedural Version of Ricocheting Comets

```
// variable declarations
int x;
int y;
int directionX;
int directionY;
color clr;

void backgroundTransparent() {
  fill(0,0,0,12);
  rect(0,0,width,height);
}

void setup() {
  size(800,600);
  // variable assignments
  x = 15;
  y = 15;
  directionX = 10;
  directionY = 4;
  clr = color(255,0,0);

  ellipseMode(CENTER);
  background(255,0,0);
}

void draw() {
  backgroundTransparent();
```

```

fill(clr);
ellipse(x,y,30,30); // use x,y for drawing
x = x + directionX; // update x
if (x >= width || x <= 0) {
    directionX = -directionX; // update directionX
    clr = color( random(255),random(255),random(255) );
}
y = y + directionY; // update y
if (y >= height || y <= 0) {
    directionY = -directionY; // update directionY
    clr = color( random(255),random(255),random(255) );
}
}

```

Object-Oriented Version of Ricocheting Comets

```

// main.pde for a single comet
Comet c; // declare object variable

void backgroundTransparent() {
    fill(0,0,0,12);
    rect(0,0,width,height);
}

void setup() {
    size(800,600);
    ellipseMode(CENTER);
    c = new Comet( 15,15,10,4,color(255,0,0) ); // calls constructor
}

void draw() {
    backgroundTransparent();
    c.display();
    c.move();
    c.bounce();
}

// Comet.pde
class Comet {
    // instance variables
    float x;
    float y;
    float directionX;
    float directionY;
    color clr;

    // constructor
    Comet(float xIn, float yIn, float dirX, float dirY, color clrIn) {
        this.x = xIn;
        this.y = yIn;
        this.directionX = dirX;
        this.directionY = dirY;
        this.clr = clrIn;
    }

    void display() {
        fill(this.clr);
    }
}

```

```

    ellipse(this.x, this.y, 30, 30);
}

void move() {
    this.x = this.x + this.directionX;
    this.y = this.y + this.directionY;
}

void bounce() {
    if (this.x >= width || this.x <= 0) {
        this.directionX = -this.directionX; // update directionX
        this.clr = color( random(255), random(255), random(255) );
    }
    if (this.y >= height || this.y <= 0) {
        this.directionY = -this.directionY; // update directionY
        this.clr = color( random(255), random(255), random(255) );
    }
}
}
//end of class

```

```

// main.pde for multiple comets

Comet[] comets; // declare object variable
final int NCOMETS = 500;

void backgroundTransparent() {
    fill(0,0,0,12);
    rect(0,0,width,height);
}

void setup() {
    size(800,600);
    ellipseMode(CENTER);
    comets = new Comet[NCOMETS];
    // for loop using a counter variable
    for (int i = 0; i < NCOMETS; i++) {
        Comet c = new Comet( 15, 15, random(1,10), random(1,10),
color(255,0,0) );
        comets[i] = c;
    }
}

void draw() {
    backgroundTransparent();
    // for-each loop
    for (Comet c : comets) {
        c.display();
        c.move();
        c.bounce();
    }
}

```

Subsection 2. Semantics: Setting Components in Relief

Once students have finished memorizing the *Ricocheting Comets* program in its three incarnations, they have intimate familiarity with the structures of both the procedural and OO versions of the program. However, their understanding of how the individual components work together – the semantics, the logic, and the program flow – is poor. Instructors often make the assumption that presenting a concept once with practice – or even re-teaching – is sufficient for students to learn it. To the contrary, language arts teachers have internalized the statistic, earlier cited, that it takes 15-20 times of hearing a word, phrase or pattern in multiple contexts before a student acquires mastery. Foreign language instructors often put that number 3-4 times higher.

With access to both versions of the program, students are now charged with the task of converting the procedural version of the *Ricocheting Comets* program to the OO version in 9 discrete progressive steps. Students are cautioned to incorporate only those components needed to effectuate the specific changes for each step, taking care to avoid adding variables or commands responsible for other behaviors. The learning objective is for students to discover exactly which code fragments working together elicit which specific runtime behaviors, and through this process gain a fuller understanding of the program logic and its flow. The instructions (below) guide students to initially create a class module with a completely empty body; write a default constructor and call it in `setup()` to construct a new object; create instance variables and initialize them; transfer procedural drawing methods to the class; and call these object methods in the main module using dot notation. The stepwise nature of the exercise pinpoints areas of student misunderstandings, and facilitates their clarification through guided discovery.

A key feature of this strategy is that the instruction not be about re-teaching, but rather about re-constructing the program in a different order from the way it was taught, coming at it from a different angle, as it were. This slightly altered approach provides a fresh perspective for each functional component of the program, setting it *in relief* against the seemingly amorphous program background as a whole, incrementally giving each its due, providing enough of a different context for each constituent part that its meaning and function can better become apparent. The foreign language classroom counterpart is to what follows memorization or repeated study of a unit dialog: exercises that pinpoint particular features in that dialog for students to focus on. In spirit, this strategy has much in common with Deb Roy's *Predicting the birth of a spoken word* study, described earlier (Roy, Frank, DeCamp, Miller, & Roy, 2015).

This strategy is often implemented as a group assignment, with two or three students crowding around one computer as the academically weakest group member "drives", does the actual keyboarding. A group needs the instructor to "sign off" on each step before they can proceed, to verify/assess each member's understanding of the step's primary learning objective, and his/her appreciation for how the program design has advanced from the immediately previous version.

Converting **bounceDiag** to the **Comet class** program

Name of program: **Comet01**

Methods: **setup()**, **draw()**, **size()**, **ellipseMode()**, **backgroundTransparent()**

Run: Window is 800 W x 600 H, with a black transparent (transparent/alpha value=12) background.

Name of program: **Comet02**

Add a tab for the Comet class. Write a simple Comet class, its body is empty.

Run: Same as Comet01

Name of program: **Comet03**

In the Comet class, add the **constructor**.

In the main file, create a Comet **variable c** and initialize it in **setup()** with the **new** command, which calls the **constructor**, which creates a **Comet** object.

Run: Same as Comet01

Name of program: **Comet04**

Add the 5 instance variables to the **class**: **x**, **y**, **directionX**, **directionY**, **clr**.

Initialize these instance variables in **setup()** using the Comet variable **c** with dot notation. Their values will be, respectively: 15, 15, 10, 4, color(255,0,0)

Run: Same as Comet01

Name of program: **Comet05**

Add an **int parameter xIn** to the constructor, and – **IN THE CONSTRUCTOR** - initialize the instance variable **x** using the new parameter **xIn**, and using the keyword **this** with dot notation.

In **setup()**, pass the value 15 as an **argument** for this new **parameter**.

Comment **OUT** (but do not delete) the line in **setup()** that initialized the instance variable **x**.

Run: Same as Comet01

Name of program: **Comet06**

Same directions as **Comet05**. Create the 4 remaining instance variables **y**, **directionX**, **directionY**, **clr**. The constructor parameters corresponding to those instance variables will be called **yIn**, **dirX**, **dirY**, **clrIn**.

Run: Same as Comet01

Name of program: **Comet07**

In **setup()**, delete the 5 lines that were commented out.

Add a class method called **display()** that draws the red circle at the initial point (**x**, **y**). Use the keyword **this** with dot notation inside this method when referring to any of the class's instance variables.

Use/call this method in **draw()**, again using the Comet variable **c** with dot notation.

Run: A red circle is drawn in the top left corner of the window.

Name of program: **Comet08**

Add a class method called **move()** that causes the red circle to move.

Use the keyword **this** with dot notation inside this method when referring to any of the class's instance variables.

Use/call this method in **draw()**, using the Comet variable **c** with dot notation.

Run: A red circle moves right and down until it disappears off screen.

Name of program: **Comet09**

Add a class method called **bounce()** that causes the red circle to reverse its horizontal or vertical direction depending upon which window edge it encounters.

Use the keyword **this** with dot notation inside this method when referring to any of the class's instance variables.

Use/call this method in **draw()**, using the Comet variable **c** with dot notation.

Run: A red circle move across the screen, bouncing off of all 4 edges.

Subsection 3. Semantics: Revealing Underlying Transformational Structure

Even after students have incrementally converted the procedural version of the *Ricocheting Comets* program to an OO version, there remains considerable confusion on two aspects of constructor semantics: (1) the *sequence* of arguments in the constructor call must correspond to the order of parameters in the constructor definition; and (2) the mechanism by which argument values in a *new* constructor call are *implicitly* transferred into the constructor and assigned to its parameters.

Instructors already know how to clarify the confusion regarding the *sequence* of arguments by using the concept of overloaded methods with different *method signatures*. The example and counter-example below are typical of such instruction.

Overloaded Constructors that can distinguish between argument calls because the variable types of sequence positions differ

```
c = new Comet( 15, 10, color(255,0,0) );  
c = new Comet( color(255,0,0), 15, 10 );
```

```
Comet(float x, float y, color clr) { ... }  
Comet(color clr, float x, float y) { ... }
```

Overloaded Constructors that cannot distinguish between argument calls because the variable types of sequence positions are identical

```
c = new Comet( 15, 10 );  
c = new Comet( 10, 15 );
```

```
Comet(float x, float y) { ... }  
Comet(float y, float x) { ... }
```

However, the confusion over the *implicit* assignment of argument values to method/constructor parameters is predictable given students' familiarity with the *explicit* syntax for assignment of values to variables using the assignment operator. Although the mechanism by which arguments are assigned to parameters may appear to be self-evident

to instructors, the reality is that the surface syntax of the language does not explicitly make clear what is happening¹⁴ (**Figure 9**).

```
// main.pde
c = new Comet( 15,15,10,4,color(255,0,0) );

// implicit assignment of argument values from constructor call to parameter variables
// Comet.pde
Comet(float xIn, float yIn, float dirX, float dirY, color clrIn) { ... }
```

Figure 9. Surface Syntax.
Constructor Call Arguments → Assignment of Values to Constructor Parameters

The mechanism can be made explicit via a transformational model that posits intermediate underlying syntactic structures that clarify the implicit nature of the PL grammar definition (**Figure 10**).

```
// main.pde
c = new Comet( 15,      15,      10,      4,      color(255,0,0) );

// Comet.pde
Comet( float xIn, float yIn, float dirX, float dirY, color clrIn ) { ... }

// #1: Pair argument values with corresponding parameter variables
Comet( float xIn=15, float yIn=15, float dirX=10, float dirY=4,
      color clrIn=color(255,0,0) ) {
}

// #2: "promote" parameter-value pairs directly into method body and
// transform them into syntactically correct assignment statements
Comet() {
    float xIn=15;
    float yIn=15;
    float dirX=10;
    float dirY=4;
    color clrIn=color(255,0,0);
}
```

Figure 10. Transformational Model: Positing Intermediate Structures that clarify both
(a) assignment of argument values to parameters and
(b) how parameters behave like local variables

¹⁴ Python does have a mechanism for assigning values to parameters in function calls, e.g. `printinfo(age=50, name="miki")`, though how these values are explicitly assigned in the *function body* is still left obscure. Like C++, Python also has a mechanism for default argument values, which provides partial transparency for how such assignments occur in general. Unfortunately, Python also has a steep learning curve, particularly for high school students. (Konidari & Louridas, 2010)

The first intermediate structure shown in **Figure 10** illustrates how argument values can be paired with their corresponding parameter variables using the assignment operator (=). The second intermediate structure depicts how parameter-value pairs are then "promoted" to the method body and *transformed* into independent assignment statements, underscoring that parameter variables behave just like any other locally defined variable. Instruction that uses these transformations to model a number of successive constructor/method calls with different argument values allows students to integrate this alternative assignment scheme into the explicit assignment schema they already possess.

Like Chomskyan transformations, the transformations in this example are explanatory, but in a way slightly different from the former. Chomskyan transformations operate on a single hidden underlying structure to generate different surface structures (sentences). The common underlying structure clarifies how the different outputs are related. In contrast, the hidden intermediate structures in this PL example are generated by transformations that combine elements from both a constructor or method *call* and the constructor or method *definition*. These underlying intermediate structures demonstrate the implicit mechanism that connects the public face of a syntactic element (method / constructor) to its inner workings using the kind of PL statements that students are already familiar with. Unlike NL generative grammars, no claim is (yet) made that these underlying structures reflect a mental representation of the PL, though semantically they are certainly plausible. Instead, the PL structures are *pedagogic* in nature, intended to *scaffold* conceptual understanding. Like generative structures, however, they provide deeper insight into how the surface structures of the language operate.

Subsection 4. Semantics: for-loop Transformations

Understanding the workings and usages of for-loops is another formidable semantic construct with which novices struggle. When the counter variable functions only to enumerate the number of repetitions, grade-level (i.e. proficient in Algebra 1) freshmen encounter few logistical problems. When given direct instruction along with custom graphics programs with which they can experiment – by modifying the counter variable's initial value, terminating condition, and increment/update value – and inspect the runtime output, they readily learn how to calculate the number of repetitions, including whether the settings will result in an infinite loop or execute for zero iterations. Note that this does not mean students are completely comfortable with the language semantics, or understand how loops operate beyond these surface calculations. They can, though, program with loops in this simple manner without much difficulty. A graphics program for such instruction appears in the *Ricocheting Comets* unit (Part 2, Chapter 2, Section 3).

Difficulties arise, however, when the counter variable is used to calculate values *within the body* of the loop. Consider a problem asking students to use iteration to draw four equally spaced concentric squares inside a window's client space that measures 200 x 200 pixels, per **Figure 11**. This exercise is the first in a set of 4 or 5 problems with similar solutions.

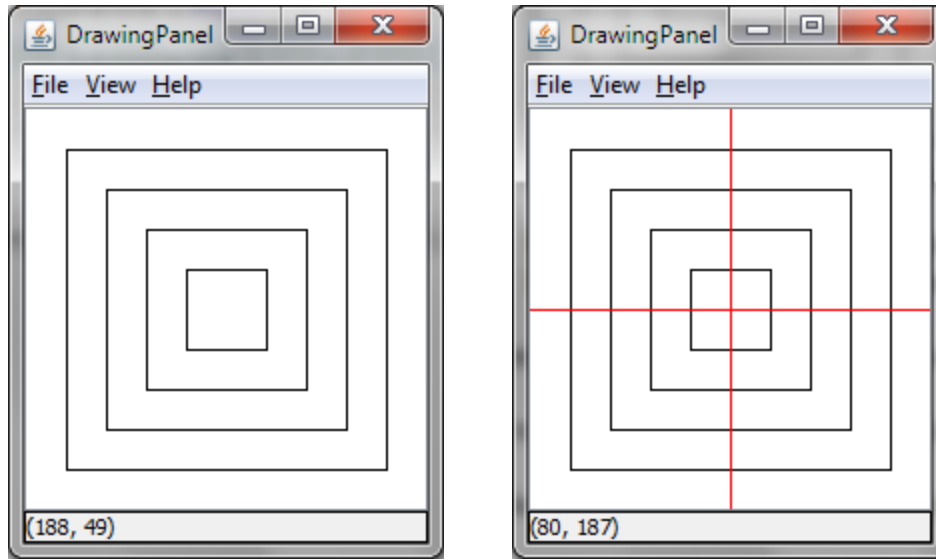
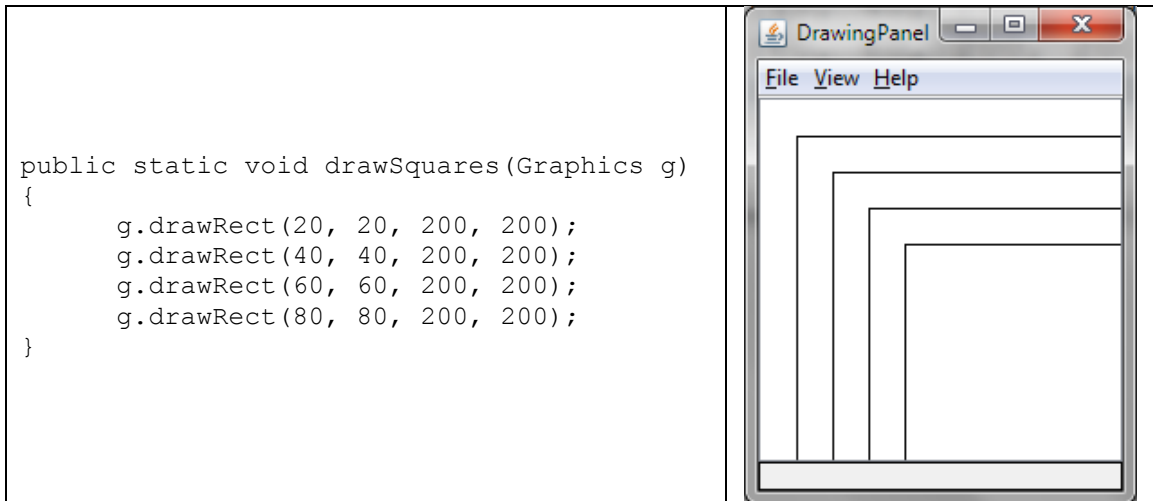


Figure 11. Concentric Squares. Left: Desired Output.
Right: Red Lines to help with calculations

This type of problem is amenable to both the concrete-abstract approach (Part 1, Chapter 2, Section 3, Subsection 2) and the syntactic transformations approach. Solving this problem in a concrete manner is not difficult, but even this part of the instruction is not left to chance. Helper lines (in red above) are drawn to help students discern the 10 equal subdivisions of the 200-pixel width and height for calculating intervals and offsets.

Novices can often be overwhelmed at the start of new unit, not even knowing where to start, because too many things are required of them at once. The mathematical calculations may be trivial, but their initial occurrence in an unfamiliar context can be disorienting. For this reason, the calculations for the parameters needed for drawing the square were separated into two parts. Students were first asked to calculate just the top-left coordinates of the 4 squares, using a constant 200 pixel value for the size, and to postpone dealing with squares extending beyond the right/bottom window borders.



Once students correctly calculated the top-left coordinates, they were asked to calculate the width and height of each square. The *Concrete* program with values hard-coded for all 4 squares appears below.

```
// Concrete
public static void drawSquares(Graphics g) {
    g.drawRect(20, 20, 160, 160);
    g.drawRect(40, 40, 120, 120);
    g.drawRect(60, 60, 80, 80);
    g.drawRect(80, 80, 40, 40);
}
```

The gap between this solution and its iterative equivalent (*Abstract 2*, below) is generally too wide for most students to bridge. This is the case despite the fact that students had just completed a unit where they used the same Algebra 1 *T-Table* technique to calculate slope-intercept expressions as that used in the *Abstract 2* iterative solution. The context, however, was different: nested for-loops that generate arithmetic patterns of line-text output (Part 2, Chapter 2, Section 7). Until students have enough experience applying a concept in multiple contexts, they are often either reticent, or have no firm idea where, to begin in a new environment. Note also that had 0-based counting been used, the

algebraic expressions would have reflected the *Concrete* method's values for the largest square drawn (for-loop values and assignment statements appear in **comments** below).¹⁵

```
// Abstract 2: 1-based counting
// T-Table: Implementing slope-intercept algebraic expression
public static void drawSquares(Graphics g) {
    for (int i = 1; i <= 4; i++) { // i = 0; i < 4; i++
        int xy = 20 * i;           // xy = 20 * i + 20
        int wh = -40 * i + 200;    // wh = -40 * i + 160
        g.drawRect(xy, xy, wh, wh);
    }
}
```

Semantically, the two methods are equivalent. The *Abstract 2* one, though, is a complex, abstract transformation of the *Concrete* method in which the algebraic expressions for each square's top-left corner and width/height are dependent upon the counter variable. There is, however, an intermediate solution (*Abstract 1* below) that sets initial values and uses the slopes to increment/update the square variables *without* the need to involve the counter variable.

```
// Abstract 1
// Assign initial values, Increment/decrement steps each iteration
public static void drawSquares(Graphics g) {
    int xy = 20;
    int wh = 160;
    for (int i = 1; i <= 4; i++) {
        g.drawRect(xy, xy, wh, wh);
        xy = xy + 20;
        wh = wh - 40;
    }
}
```

Students are asked to solve the set of 4-5 problems in these 3 different ways – with instruction preceding each round to address the particulars of each problem –

¹⁵ Whether one uses 0-based or 1-based counting is immaterial, as no array elements or String characters are being indexed. 1-based counting was used in the text line output exercises students solved in the prior unit, as it was consistent with line counting (what would a line 0 be?) The decision to not introduce 0-based counting was taken to avoid diffusing the focus from the lesson's primary objectives.

implementing, in order, the (a) concrete, (b) intermediate abstract, and (c) complex abstract solutions. When done, students will have had enough experience to feel comfortable using for-loops to solve problems of this type

The purpose of this approach – of all the approaches really – is two-fold: (1) to scaffold learning, breaking the lesson into smaller more comprehensible steps; and (2) to focus on language acquisition through repetition in multiple contexts. Studying the particulars of the *Concrete* method – (a) the four `drawRect()` calls corresponding to the number of repetitions; (b) the parameters for drawing the largest square reflected in the initial values of variables in *Abstract 1* or the y-intercept values of *Abstract 2*; and (c) the intervals between the parameter values echoed in the step or slope values – focuses students' attention on the critical semantic details of the for-loops. These are the elements common to all three methods which students carry forward as they *transform* one solution into another.

As alluded to earlier, the idea of revealing connections between different semantic or syntactic structures has parallels in Algebra instruction. For example, geometric proofs of the algebraic factor formulas for difference of squares ($a^2 - b^2$) and addition of squares ($a^2 + b^2$) not only reveal the connections these concepts have to both disciplines – like two sides of the same coin – but provide students the additional tools to derive the formulas, rather than having to rely only upon memorization. The passage below cites the importance of connections in mathematics, but the same holds true for deeper understanding of the way programming languages work:

When students can connect mathematical ideas, their understanding is deeper and more lasting. They can see mathematical connections to the rich interplay among mathematical concepts... By emphasizing mathematical connections, teachers can help student build a disposition to use connections in solving mathematical

problems, rather than see mathematics as a set of disconnected, isolated concepts and skills. (National Council of Teachers of Mathematics, 2000)

Psychologically, connections facilitate retrieval of information from long-term memory.

When trying to recall a specific memory, related concepts are also activated and remembered (Woolfolk, 2004). The more connections attached to a concept, all things being equal, the easier it will be to remember it when needed.

* * * * *

A progression of alternate coding choices from concrete to abstract can also help to clarify the use of counter variables used in algebraic expressions in other domains, such as when they are used (a) to access array/list elements, or (b) to iterate between the two parameters of the **String** *substring(int beginIndex, int endIndex)* method. The four methods below incrementally demonstrate behavioral characteristics of *beginIndex*.

Incremental methods to clarify the meaning of *beginIndex*.

// concrete for guided discovery of how the 1st parameter works

```
public static void method1A() {
    String s1 = "abcdefghijklm";
    System.out.println( s1.substring(0) );
    System.out.println( s1.substring(2) );
    System.out.println( s1.substring(4) );
    System.out.println( s1.substring(6) );
    System.out.println( s1.substring(8) );
}
```

// abstract 1

```
public static void method1B() {
    String s1 = "abcdefghijklm";
    int start = 0;
    for (int i = 1; i <= 5; i++) {
        System.out.println( s1.substring(start) );
        start = start + 2;
    }
}
```

// abstract 2

```
public static void method1C() {
    String s1 = "abcdefghijklm";
    for (int i = 1; i <= 5; i++) {
        int start = 2 * i - 2;
    }
}
```

```

        System.out.println( s1.substring(start) );
    }
}

// general abstract method using all possible values for beginIndex
// Note: output is the empty string when i == s1.length()
public static void method2() {
    String s1 = "abcdefghijklm";
    for (int i = 0; i <= s1.length(); i++) {
        System.out.println( s1.substring(i) );
    }
}

```

The three methods below likewise demonstrate behavioral characteristics of *endIndex*.

Incremental methods to clarify the meaning of *endIndex*.

// concrete for guided discovery how

// the 2nd parameter operates in combination with the 1st

```

public static void method2A() {
    String s1 = "abcdefghijklm";
    System.out.println( s1.substring(2,3) );
    System.out.println( s1.substring(2,4) );
    System.out.println( s1.substring(2,5) );
    System.out.println( s1.substring(2,6) );
    System.out.println( s1.substring(2,7) );
    System.out.println( s1.substring(2,8) );
}

```

// concrete for extracting individual characters

```

public static void method2B() {
    String s1 = "abcdefghijklm";
    System.out.println( s1.substring(0,1) );
    System.out.println( s1.substring(1,2) );
    System.out.println( s1.substring(2,3) );
    System.out.println( s1.substring(3,4) );
    System.out.println( s1.substring(4,5) );
    System.out.println( s1.substring(5,6) );
}

```

// abstract for extracting individual characters

```

public static void method2C() {
    String s1 = "abcdefghijklm";
    for (int i = 0; i < s1.length(); i++) {
        String ch = s1.substring(i,i+1);
        System.out.println( ch );
    }
}

```

Combining the *concrete-abstract* and *parameter-transformation* strategies is especially useful in a unit where one is introducing common methods of a class's API. The **String-1** unit at *Codingbat.com* contains a nicely designed progression of problems, starting with those that require the use of concatenation, `substring()` and `length()` for their solutions. Below is an example of how one would scaffold instruction for the second problem, `makeAbba()`.

```
public static String makeAbba(String a, String b) {
    return "-";
}

public static String makeAbba() {
    String a = "Hi";
    String b = "Bye";
    return "HiByeByeHi";
}

public static void main(String[] args) {
    System.out.println( makeAbba("Hi", "Bye") );
    System.out.println( makeAbba("Yo", "Alice") );
    System.out.println( makeAbba("What", "Up") );
    System.out.println( makeAbba() );
    System.out.println( makeAbba2() );
    System.out.println( makeAbba3() );
}
```

The unsolved problem appears at top, followed by its parameter-transformed version, where the parameter variables have been promoted to the local space and initialized with the arguments from the first call of the method in `main()`. The return statement contains the correct output for those values (Codingbat's description for each problem provides these). Two intermediate steps, one concrete and the other abstract, are constructed (below): the return statement is first subdivided into constituents corresponding to the parameter values, and the variables corresponding to those parts are then substituted in.

```

public static String makeAbba2() {
    String a = "Hi";
    String b = "Bye";
    return "Hi" + "Bye" + "Bye" + "Hi";
}

public static String makeAbba3() {
    String a = "Hi";
    String b = "Bye";
    return a + b + b + a;
}

```

Running the *parameter-transformed* program locally in an IDE (e.g. in *Eclipse*) and seeing the correct output demonstrates the equivalencies of the concrete and abstract versions. The abstract method body – minus the promoted initialized local parameter variables - is then pasted into the unsolved problem at top, and finally into the web page.

```

public static String makeAbba(String a, String b) {
    return a + b + b + a;
}

```

Below is similarly scaffolded code for the problem **makeOutWord()**.

```

public static String makeOutWord(String out, String word) {
    return "-";
}

public static String makeOutWord() {
    String out = "<<>>";
    String word = "Yay";
    return "<<Yay>>";
}

public static void main(String[] args) {
    System.out.println( makeOutWord("<<>>", "Yay") );
    System.out.println( makeOutWord("<<>>", "WooHoo") );
    System.out.println( makeOutWord("[[]]", "word") );
    System.out.println( makeOutWord() );
    System.out.println( makeOutWord2() );
    System.out.println( makeOutWord3() );
    System.out.println( makeOutWord4() );
    System.out.println( makeOutWord5() );
    System.out.println( makeOutWord6() );
    System.out.println( makeOutWord7() );
}

```

```

public static String makeOutWord2() {
    String out = "<<>>";
    String word = "Yay";
    return "<<" + "Yay" + ">>";
}

public static String makeOutWord3() {
    String out = "<<>>";
    String word = "Yay";
    return "<<" + word + ">>";
}

public static String makeOutWord4() {
    String out = "<<>>";
    String open = "<<";
    String close = ">>";
    String word = "Yay";
    return "<<" + word + ">>";
}

public static String makeOutWord5() {
    String out = "<<>>";
    String open = "<<";
    String close = ">>";
    String word = "Yay";
    return open + word + close;
}

public static String makeOutWord6() {
    String out = "<<>>";
    String open = "<<>>".substring(0,2);
    String close = "<<>>".substring(2);
    String word = "Yay";
    return open + word + close;
}

public static String makeOutWord7() {
    String out = "<<>>";
    String open = out.substring(0,2);
    String close = out.substring(2);
    String word = "Yay";
    return open + word + close;
}

```

Note the discipline of postponing the substitution of an abstract variable for a String literal until all steps in a progression of intermediate concrete statements have been exhausted. For example, the local variables **open** and **close** are initialized with string

literals, postponing the two **substring()** extractions of the variable **out**. This scaffolding can be reinforced with direct instruction articulating general methodologies or principles for solving problems of this kind, e.g. the requirement that one must be able to trace back everything in a return statement's abstract expression(s) to the input parameters.

Recall Deb Roy's chief finding that all three of his son's caregivers "subconsciously restructured [their] language to meet him at the birth of a word and bring him gently into more complex language". This is the type of language instruction that is required of teachers who want to progressively move as many programming-capable students as possible into literacy. Feedback from students of mine who tend to struggle with new programming constructs is that instruction like the kind described above helps to lessen, if not completely dispel, the utter confusion or mental blocks they initially experience as they begin to see ways to utilize previously learned, familiar concepts as they work towards understanding the new material.

Subsection 5. Semantics: Ongoing Exposure

A general strategy for ongoing exposure to the PL – originally articulated by Corder for L2 acquisition – is modeled after Mountry et al.'s second theme of providing a continuous print-rich English-cultural home presence, exemplified by the ritual of Deaf parents reading the same bedtime stories over and over to their children, and taking the time to point out, explain and reinforce new vocabulary with fingerspelling. This strategy is also informed by the finding that the use of words in multiple contexts (*wordscapes*) predicted word acquisition in a toddler.

The predominant university-level instructional model for CS1 – and the secondary pedagogic model that echoes it – is for students to be given general instruction and assigned problem sets. Answers are posted after the homework is turned in. They may optionally choose to attend study sessions led by T.A.'s for going over these exercises, where the type of re-teaching that is common in secondary classrooms might occur.

In the two secondary courses that I teach, equivalent to CS0 and CS1, students receive instruction (direct or otherwise) on a particular PL feature. Then, depending upon the exercise, they are given guidelines or a framework, or told that certain past problems can help them solve problem sets utilizing the new feature. As students work in class, the teacher circulates, giving hints to students who may be at a roadblock. When most students appear to have solved a problem, the instructor shows one or more optimal solutions. Following a back-and-forth discussion, students implement (not copy) these solutions. Students work through each problem in the set in this same way.

The *ongoing exposure* strategy occurs when all of the problems have been completed. Students are again asked to study the optimal solutions, and when ready,

implement *from scratch* all exercises in the problem set in a workspace where they cannot refer to their previous work. Students may refer to written notes, but not programming code. As such, this is not an exam, but rather a production strategy for *repetition* and *meaningful interaction*, giving students more practice, and fostering an environment in which optimal language semantics can be internalized. The teacher is always available to give hints so that students who are stuck can get beyond roadblocks.

This last interaction, where a teacher gives individual help to a student who is motivated to progress, is reminiscent of Deb Roy's scaffolding feedback loop. As an example, one student who had still not internalized the difference between array indices/positions and values/elements, could not fathom why he could not get his program to output the median value in a list of 15 sorted integers. He had actually coded the program to output the index (`length/2`) rather than the value (`list[length/2]`), but because the number 7 was one of the elements, he kept mistaking the output for the wrong element value. Feedback by the teacher came at the moment when the student was primed to finally understand the distinction.¹⁶

The *Ongoing Exposure* strategy is admittedly time-intensive, but as students progress through the introductory course and into the second year, time becomes less of an issue because their programming skills become more automatic. This strategy is often tacked onto the tail end of instructional strategies described earlier, allowing students to reacquaint themselves with each problem, perhaps glean new insights, and more firmly secure its solution as a paradigm in their toolkits. For example, after students have completed the for-loops transformations unit, where they have solved each problem in the

¹⁶ Alternately, the problem might have been avoided altogether had the list been instead of a type other than integer, such as a list of Strings; confusion of the value with the index would then simply not have occurred.

set 3 different ways, they are then asked to solve each problem again from scratch in all 3 ways in a workspace where they have no access to their programming code.

Subsection 6. Summary

The instructional strategies discussed in this section all employ ideas from general teaching theory. No concept or problem-solving piece is taken for granted. As much as possible of a lesson's instruction is broken down into "baby" steps, consistent with Vygotsky's zone of proximal development (ZPD) and scaffolding notions. Revisiting concepts in contexts where they are used with increasing levels of sophistication utilizes principles from Bruner's *Spiral Curriculum*. The difference between the pedagogies discussed here and those typically used in CS programming instruction is that scaffolding and revisiting is not limited to program logic and mathematics, but extended to features of the PL used to mediate the logic and mathematics.

The near universal pedagogic model for CS instruction is taken from the playbook used in the mathematics classroom: instruction followed by practice. In contrast, language learning pedagogies provide students with an environment where (a) comprehensible input is repeated in multiple contexts, and (b) meaningful interaction and production are crucial pieces of instruction. In a language model, following instruction there are two or more rounds of practice in which students are asked to use concepts and features in slightly different ways and from different angles. The altered contexts for language features allow students to differentiate the features from the whole, helping them clarify their meaning, and their basic and nuanced uses.

One chief impact of this kind of instruction is that the pace in which one traverses the traditional introductory CS curriculum will be significantly slower and extend over a longer time frame. This is not a novel idea:

Ideally, people would learn to program the same way “normal” people learn to play instruments: Slowly over several years, with lots of practice. However, this is not practical at the university level.

The foreign language model is closer to being practical. At Grand Valley, students study a foreign language for four semesters before beginning a serious study of literature and composition in that language. In theory, I think a similar model for programming would be much more effective. (Kurmas, 2011)

Secondary CS courses are generally one-semester electives, a learning model that guarantees that students will never become proficient at this core CS skill. Rather, a minimum two-year sequence of *programming* courses is necessary: a foundational pre-AP programming course and the APCS-A course. Both courses need to employ pedagogies addressing how students learn both programming concepts and programming language features. The extension of time is the necessary condition for ensuring deeper and more lasting comprehension, and an increase in the number of successful students.

The "programming-centric" criticism of programming courses that has been voiced over the last decade, though nonsensical at its heart, does have one valid point: such courses generally have a very narrow focus, and domains of application are just as narrow and uninteresting. A student whose experience is limited to just the content of the APCS-A course may have gained substantial skill and proficiency in programming, but would be at a near total loss to articulate how those fundamental, but still very basic, skills could be put to use in any real-world or practical terms. If such students end up unable to connect what they have learned with the far-reaching role that computer programming plays in nearly every aspect of 21st-century life, how can one expect students with no exposure to this subject to enroll in courses that teach what appears to be an esoteric skill? Part 2 of this thesis is a project report describing a foundational pre-AP programming course that challenges this mold.

PART 2

A CONTEXTUALIZED

PRE-AP COMPUTER PROGRAMMING CURRICULUM:

MODELS AND SIMULATIONS FOR EXPLORING

REAL-WORLD CROSS-CURRICULAR TOPICS

CHAPTER 1.

CONTEXTUALIZATION IN CS EDUCATION

Section 1. Introduction

Despite the pervasiveness of the applications of Computer Science throughout daily life, most high school students – and most of the population at large – have no idea as to the content of CS courses. And why should they? Rather than being integrated into the curriculum starting in the early grades, the breadth of the applications of CS are either completely ignored by all K-12 core subject areas or limited to what is already known in the popular culture at large. Unfortunately, this also characterizes what the overwhelming majority of secondary CS instructors know. When the curriculum described in this project report was first developed, it was thought, naively, that a CS curricular sequence whose organizing principle is multidisciplinary – one that applies CS concepts to the Sciences, Humanities and Arts, subject areas with which students are already familiar – might in and of itself augment enrollments and retention. This did not turn out to be the case. Interest without self-efficacy, as discussed earlier (Part 1, Chapter 2) is insufficient. This does not mean, however, that interest is negotiable or a lesser factor. Students must be able to forge a personal connection to the subject matter to sustain long-term interest and engagement. Personal interest in the subject matter fuels intrinsic motivation, which psychological research has long linked causally to engagement, creativity and high-quality conceptual learning:

Intrinsic motivation enhances a learner’s conceptual understanding of what they are trying to learn. When high, intrinsic motivation promotes flexibility in one’s way of thinking, active information processing, concentration and effective use of learning strategies, and learning in a way that is conceptual rather than rote. When intrinsically motivated, learners concentrate process information deeply, and think about and integrate information in a flexible, conceptual, and less rigid way,

rather than engage in rote learning such as memorizing and simply trying to reproduce an other-prescribed right answer. (Reeve, 2014)

Furthermore, the decision to frame course content within the context of CS as an *applied science* – to expand the constricted popular view of CS beyond gaming, social media and the like in order to broaden non-trivial interest in the field – is certainly a better way to academically represent and teach the subject as it actually operates in the 21st-century. The alternative is the predominant model, one that fails to recognize that it is in effect a truncated, incomplete curriculum that propagates a distorted view of the field's wide range of applications.

The introductory- and intermediate-level interdisciplinary units described in this project showcase the reach, importance and utility of CS applications to other academic fields, including Astronomy, Geography, Molecular Modeling and Political Science (units in Music and Bioinformatics are taught in advanced courses, and units in Environmental Science and Holocaust Studies are ideas for future development). The curriculum goes by the name *Computer Science as if the Rest of the World Existed* (CPRWE).

At the start of a typical multi-week unit, students use an existing complex and engaging application – such as Planetarium software or a Molecule Viewer program – to explore a specific problem. They then begin to build a scaled down, simplified version of the program which they use to investigate that same problem when completed. The intent is that students be able to envision, with little imagination, their ability to construct – in theory – the larger real-world applications after which their smaller programs were modeled. In the process of writing the smaller scale program, students focus on one or more core algorithms key to its intended function. This approach (a) gives students

experience in constructing complete programs that can be used to examine actual phenomena; (b) shows them how the programming concepts they've learned can be put into practice to solve a lengthy, open-ended problem; and (c) provides a purposeful context for utilizing academic knowledge learned in other coursework.

At the end of each unit, students study a theater piece, film and/or screenplay – often biographical – where some aspect of the unit problem plays out in the lives of the characters. This last piece anchors the problem to an historical and social context and offers students a humanistic vantage point to consider the ramifications of the unit problem and their program's historical – or potential – societal impact: to wit, the knowledge gains a human, and perhaps personal, purpose.

In addition to creating new areas of interest for students, an argument can be made that this kind of approach might also counter and rehabilitate the prevailing constrictive and generally uninformed image of CS, which may be feeding systemic, bureaucratic and attitudinal obstacles to inclusion and implementation of CS education at the secondary level. Layperson stakeholders may not ever learn to appreciate the subject matter on anything but a surface level, but they can certainly comprehend how extensively 21st-century society relies upon the applications of this field.

It needs to be acknowledged that, within the time constraints of a course lasting a single college semester or high school year, the tension between the two goals of enabling self-efficacy and generating interest involves trade-offs. Ten units is a tight fit for a year-long course if one wants to teach each unit in any degree of depth, particularly if one wants to delve into the kind of details software engineers encounter regarding

unintended side-effects caused by one's particular choice of competing programming models or implementations.

Were the course taught in a foreign language-like context, there would be a minimum time allotment of two years for achieving substantial language goals. In my experience, this kind of time period is required for students to develop substantial programming competence, the traditional high-fliers included. Although the most talented students may appear like wizards compared to the majority of their grade-level classmates, by no means do they have a solid conceptual foundation, and like their fellows, require a second year of the kind of subject matter content and practice found in a rigorous APCS-A course, such as one that uses the textbook *Building Java Programs* by Reges & Stepp (BJP). With these considerations in mind, the current incarnation of my freshman CS0 course teaches Units 1-6 (Dynamic Art and *Codingbat*) in the first semester. The second semester comprises the first 4 chapters of BJP – the textbook for the APCS-A course – and concludes with Unit 7, *The Right to Vote* (scoring optical ballots using multiple recursion), and Unit 10, *Molecular Modeling and DNA*. Units 8 and 9 can be taught in subsequent years.

Note that because the IDE used in this course is *Processing* – a graphics library built on top of Java, with the *Processing* language itself being a simplified form of Java that fully conforms to Java syntax, even allowing the intermingling of Java and *Processing* statements – there is a seamless transition to Java when students begin to work in *Codingbat* and the first part of the APCS-A curriculum. As such, the disruption that would occur when switching languages – with the unrealistic expectation that programming skills will automatically transfer – is avoided.

The initial stages of this effort were acknowledged by a *SIGCSE Special Projects* grant in the August 2011 cycle. An interim report was presented in a 75-minute "special session" at the *SIGCSE 2012* national conference in Raleigh, NC, and described in a short publication (Portnoff, 2012). The full course outline was approved by *UCOP* (University of California Office of the President) as a college-preparatory "A-G" "g"-elective in December 2013.

Section 2. Contextualization Efforts in CS Education

CS educators have long suggested that science education not be fenced off from the social sciences:

... *insuring science and technology are considered in their social context*... may be the most important change that can be made in science teaching for all people, both male and female. (Rosser, 1990)

In her article *In Search of Gender Free Paradigms for Computer Science Education*, [CS professor Dianne Martin discusses] "a premise for the gender bias in computer science: the *existing educational paradigm that separates studies of science, math, and computer science from studies of the humanities*, starting in the secondary schools." She speculates that an integrated approach to computer science would attract more women students, and that we should *pay "greater attention to values, human issues, and social impact* as well as to the mathematical and theoretical foundations of computer science." (Margolis, Fisher, & Miller, 1999/2000)

The acceptance and integration of social issues into computing curricula is still a work in progress *twenty years* after it was first incorporated into the ACM Computing Curricula. ...most institutions include the societal impact of ICT in their programs. However, topics often concentrate on computer history, codes of ethics and intellectual property, while *neglecting broader issues of societal impact*. (Goldweber, et al., 2011)

What's more, the idea of a multidisciplinary approach to CS education is not a novel one:

...most computer science programs in their early years are narrowly focused on programming and the more technical aspects of the field, *with applications and multidisciplinary projects deferred to the very end*. This gives beginning students the false message that computer science is "only programming, programming, programming," *abstracted away from real world contexts* (Margolis, Fisher, & Miller, 1999/2000).

Interdisciplinary learning "fosters two related goals: exposure to multiple bodies of knowledge that are not connected through a traditional course of study; and the ability to integrate those bodies of knowledge in pursuit of a shared understanding or answers to a larger question" (Holley, 2009). Benefits of interdisciplinary learning are well-documented. Students engaged in "interdisciplinary programs are more likely to acquire

integrated perspectives and solution-focused strategies, rather than content-specific knowledge derived from a single discipline" (Ivanitskaya, Clark, Montgomery, & Primeau, 2002). Research showing that "the brain is a parallel processor that makes meaning by patterning" supports the view that "sophisticated levels of learning" are not as well learned when "studying subjects separately" (Klein, 2005). Compared to disciplinary courses, "interdisciplinary curricula tend to make more explicit connections between the subject matter and the student's prior experiences through active, constructivist, student-centered pedagogies." Researchers report that interdisciplinary curricula enhance critical-thinking and problem-solving skills (Holley, 2009). Calling something "interdisciplinary", however, is no magic bullet. Researchers caution that "outcomes stem as much from the way in which the courses are taught as they do from their interdisciplinary nature" (Newell, 1994).

Over the last decade, postsecondary CS educators have begun to offer CS courses across diverse disciplines in increasing numbers. Leading the pack is bioinformatics with approximately 150 major/minor programs in North America. "Modern experimental techniques, including automated DNA sequencing, gene expression microarrays, and X-ray crystallography are producing molecular data at a rate that has made traditional data analysis methods impractical. ...Computational modeling and prediction methods, such as comparative modeling of protein structure, are now reaching a level of sophistication that *allows some experimentation to take place entirely within a computational framework*" (Doom, Raymer, Krane, & Garcia, 2003).

A UMass Lowell course sequence called *Performamatics* employs Scratch to write *music* related programs and tackle issues like synchronization and harmony

(Ruthmann, Heines, Greher, Laidler, & Saulters, 2010). The Computing and Music Committee of SIGCSE was established in May 2010 to explore ways "to enhance computer science education through music applications" (Beck, Burg, Heines, & Manaris, 2011). The North American *Computational Linguistics* Olympiad (NACLO) is a high school competition that highlights study and career opportunities "that involve linguistics, computation, and human language technologies" (Radev & Levin, 2009). An *Interdisciplinary CS Research Projects* program at KCC-CUNY motivates students by emphasizing the "interplay between fundamental principles arising from *mathematical* theory, and computational complications involved in applying these principles to real world problems" (Balsim & Feder, 2008). Georgia Tech offers a course on *computational journalism* and Columbia University has a dual Master of Science program in CS and Journalism. The discipline applies "algorithms and principles from computer science and the social sciences to gather, evaluate, organize and present news and information" (Pulimood, Shaw, & Lounsberry, 2011). In 2007, the Electrical and Computer Engineering department at UNCC offered a course entitled *Design of Intelligent Spacecraft* that integrated concepts from the history of astronomy and space exploration, mathematics, celestial mechanics and navigation, engineering and computer science (Willis & Conrad, 2008). This course went further than most in attempting to anchor the topic within an historical context. The *Computational Science* minor program at Wittenberg University, in place since 2004, involves students majoring in *biology, biochemistry-molecular biology, chemistry, economics, physics, geology, mathematics and computer science*, as well as "in several non-traditional disciplines including *art, psychology and international studies*" (Caristi, Sloan, Barr, & Stahlberg,

2011). Missouri University of Science and Technology has an introductory CS course for engineering students in which they solve problems in *aerospace engineering* and *civil engineering*, such as calculating rocket trajectories of lunar craft from Cape Canaveral to the moon (Hurson & Sedigh, 2010). The Penn State Worthington / Scranton IST and physics departments designed a project utilizing the Satellite Tool Kit to model *astrodynamics* simulations for various orbital scenarios, and plan high school projects in *applied physics*, the *physical sciences* and *engineering* (Smarkusky, et al., 2011). Florida Tech offers a survey course targeting non-major freshmen and sophomores that covers *digital media, bioinformatics, business, molecular modeling, engineering, geography, physics, sociology, AI* and *cryptography*. A parallel course for majors is an upper-class elective that explores problems in *sociology*, geography, *marketing, finance, law* and *biology*. The Harvard University Dept. of *Anthropology* and the MIT CS and AI Lab looked for archaeological mounds using *remote sensing* data acquired from time series of multispectral satellite images. They adapted an old technology – using satellite radar to collect surface elevation data in order to generate topographical maps of rocky planetary surfaces – and applied it to an *archaeological* problem (Menze & Ur, 2012). A working group (in which I was a co-author) at *ITiCSE 2012* detailed the outlines for 14 lessons in a wide range of applications addressing social concerns (e.g. *humanitarian relief, environmental* issues), including Red Cross disaster relief, species population simulations, the use of nuclear power, and water pollution (Goldweber, et al., 2013).

As these many examples show, the applications of CS now encompass a hugely diverse range of academic and scientific fields. That secondary CS educators have failed to incorporate such connections in their curricula in any serious or comprehensive way is

more than a wasted opportunity. Consider teaching the fingerings of a musical instrument using only practice exercises, with no attempt to play real music compositions, even simplified ones. The metaphor is exact. Including the reach of CS across the academic spectrum is not a gimmick to recruit students: this material is a vital and essential – but missing – part of the secondary CS curriculum. By paying lip service to the idea, but ignoring it in practice, secondary CS educators are acquiescing in a *truncated, incomplete curriculum*.

Importantly, this approach also has the potential to alter the image secondary CS projects. The popular conception of CS pigeonholes the field as one whose principal applications are limited to Gaming, Social Media, the Internet and Mobile Devices. Student IDEs like Alice and Scratch do initially engage students, and the Scratch *Performamatics* program at the University of Massachusetts appears to have sustained that interest. On the other hand, curricula that focus on tasks such as *Alice* skaters performing pirouettes around holes in the ice trivialize CS and narrow the lens through which students might find and connect with something of lasting import and value in the field. One might concede that story-telling might have a valid place in K-8 curricula. However, by the time students reach high school, my experience is that most students – girls included – quickly tire of make-believe scenarios, and using these IDEs with such content does nothing to slow attrition from the CS program. Juxtapose such curricula against those that create simulations of the solar system to calculate and view the trajectory of a space mission to Mars, or ones that construct biologically important molecules to elucidate disease or drug mechanisms. The latter acquaint students with powerful examples of Applied CS and provide more desirable engagement experiences

because: (a) they have purposeful and multiple real-world connections; (b) the problems presented are complex, but not insurmountable; and (c) they utilize and extend academic knowledge that students already possess.

Section 3. Potential Impact on Institutional Change

CSTA's 2010 report *Running on Empty* describes nationwide systemic issues that promote a climate that devalues K-12 CS education (Wilson, Sudol, Stephenson, & Stehlik, 2010). Such factors include: (a) the miscategorization of CS as a CTE course and nothing more, and/or confusing it with Computer Literacy; (b) a lack of comprehensive and dedicated CS state standards; (c) a widespread failure to view CS as part of the "core" curriculum or to count CS coursework towards math or science graduation credit requirements, and (d) a lack of support or interest on the part of school staff (Carter D. , 2012).

The report's Executive Summary begins: "Computer science and the technologies it enables now lie at the heart of our economy, our daily lives, and scientific enterprise." Yet the single reference and sole example of the "*broad interdisciplinary* utility of computers and algorithmic problem solving" is "programming a *telephone answering machine*." That one would be hard-pressed in 2010 to imagine a less inspiring example¹⁷ is indicative of the meager thought given this aspect of CS by those leading the discussions on secondary educational policy. Although NSF has funded *postsecondary* grants that incorporate multidisciplinary approaches into CS curricula, conversations by policy makers and national teacher organizations have been devoid of meaningful or considered efforts to encourage or implement curricula that include the interdisciplinary reach of CS across the *secondary* academic curriculum.

Carefully choosing the most engaging examples of interdisciplinary applications to showcase in an integrated CS curriculum has the potential to attract interest from

¹⁷ The example is even dated, superseded by voice mail. The phrasing appears to have been lifted verbatim from CSTA's October 2003 *A Model Curriculum for K-12 Computer Science* (p 12).

secondary *educational administrators*. For example, if a more accurate awareness of the ever-increasing dependency of biological, medical and genetic research on CS algorithmic approaches can be conveyed to administrators responsible for science education, the systemic dynamics that have kept CS as an academic underdog might be positively altered.

Moreover, an outcome showing increased retention rates would have broad implications for an alternate model curriculum, one in which cross-disciplinary units are the vehicles through which CS concepts are learned and applied. Although much effort and hope are going into the new *APCS Principles* course, what has not been addressed is a pipeline that leads to it. Most AP courses assume a supporting foundation of core subject areas whose coursework students have been studying for years. Presently, the only course that might conceivably fulfill this supporting role is Computer Literacy (CL), which typically covers the MS Office suite or the equivalent. Recently, however, some postsecondary instructors have developed CL modules that "cover topics of interest to a particular major or group of majors" (Carter L. , 2007). Middle or high school CL classes where students learn to navigate applications in the STEM fields and the Arts can similarly be re-imagined. Because CL is often a requirement, courses with multidisciplinary content could naturally channel students into CS courses.

Section 4. Core Computing Concepts and Computational Competencies

Although CSTA has proposed standards, there are four problems with its objectives and outlines. First, it is a top-down approach, which – lacking any research to support it – consists simply of educated guesses. Second, the standards are topics without contexts; it is inadequate to stipulate **Iteration**, without specifying the intended task (e.g. search a list; perform a repetitive action with changing parameters; circular queue; implementing recursion iteratively). Third, algorithms are dealt with on only the most rudimentary level. Fourth, even the most recent version of CSTA's 63-page K-12 Computer Science Standards (2011) is a document that is long on generalities and short on specifics: there is not a single code fragment to serve as illustration. Compare to NCTM's 402-page *Principles and Standards for School Mathematics* (2003). The latter is rich in illustrations and specifics for how to teach concepts from each strand; includes insightful counterexamples; examines concepts on many levels; and makes explicit connections between strands.

A curriculum whose units include complex central problems, given a large enough number and range of modules, might be a contributing source for deciding what might be the specific computing concepts and skills we want to teach at the high school level. Consider, for example, the modulus operator, which surprisingly is mentioned in none of the three levels of CSTA's Model K-12 Curriculum. In contrast, three of the CPRWE CS units described here utilize a circular buffer and explicitly implement the iterative wrap around using the modulus operator.

One might divide programming standards into 4 conceptual levels: (a) **Coding**: Programming Language Concepts; (b) **Structure**: Organization, Modularization, Method

Hierarchy; (c) **Abstraction**: Object-Oriented Model; and (d) **Algorithms**: Object-Oriented Problem Solving Strategies.

Open-ended problems explored in depth lend themselves to unforeseen, but fascinating, sub-problems that crop up en route as one crafts the larger solution. The sub-problems encountered in the CPRWE units are more often on the levels of *Abstraction* and *Algorithms*, and turn out to be the more powerful and engaging ideas in each unit. Studying these sub-problems can give rise to algorithmic standards CS educators may never have previously considered.

For example, in the *Around the World* geography unit, when an east- or west-bound traveler detects a sunrise, using a point-sector intersection model, it increments a count variable. At the end of any number of days, when the two have completed one full circumnavigation, the east-bound traveler should experience one more day than a stationary observer, and the west-bound traveler should experience one fewer. However, when *both traveler and sunset are moving*, a bug arises where the east-bound traveler misses a sunrise, and the west-bound traveler counts one of the sunrises twice. The solution involves slightly narrowing the sector for the west-bound traveler and expanding it for the one traveling east, but the mathematics for doing so present the most profound concept in the unit. The CS principle in this sub-problem reveals the limitations and problems that arise when *discrete* models are used to approximate idealized *continuous* systems or processes. Other similarly complex CS and software engineering concepts are detailed in the unit descriptions.

There is a consistent focus in these units on deciding among the best of competing solutions to a complex problem – whether at the levels of coding, abstraction, algorithms

or design – giving students the kind of experience in problem-solving typical of project based learning, but at a deep and complex level. Programming and PL concepts are re-encountered and reinforced as students implement these sub-problem solutions using specific combinations of control structures and logic. Critically, students learn that software logic in one part of a program can cause side-effects or have implications for design decisions down the line, both positive and negative.

Section 5. Broadening Participation

An argument can be made that finding ways to increase enrollments of all students in the APCS-A course may positively impact gender diversity as well.

College Board data for 2015 (College Board, 2015) shows a small difference in APCS scores by gender, a pattern that had been consistent since 2004 – when *Java* supplanted *C++* as the PL of instruction – but that has narrowed over the last decade¹⁸. Mean 2015 scores were 3.11 for males and 2.94 for females, a difference of about 6%. Research into the cause of what might be a related performance gender gap in math – and its disappearance – found that "the likeliest explanation for the dramatic improvement in math performance by U.S. females lies in ...girls in general taking more mathematics and science courses during high school due, *in part*, to changes in requirements for graduation and admission to colleges..." (Kane & Mertz, 2012). The performance problem was the visible symptom, but the underlying problem appears to have been one of a disparity in *participation numbers*. Note that participation numbers rose "in part" because of a policy change enacted nationwide. No curricular or pedagogical cause was given credit, although their influence cannot be ruled out.

Bearing this in mind, a reasonable hypothesis regarding the small CS performance gap measured by the APCS exam is that it, like the math gap, has been driven by the low female participation numbers. This hypothesis is supported by a shrinking gender performance gap, falling from a high of 16% in 2004 to 5.6% in 2015 (**Figure 12**). The R-square value for the correlation between the two variables using either quadratic or cubic polynomial regression is 0.688.

¹⁸ The gap was 11-16% from 2004-2007, 8-10% from 2008-2012, 5.6-7% from 2013-2015.

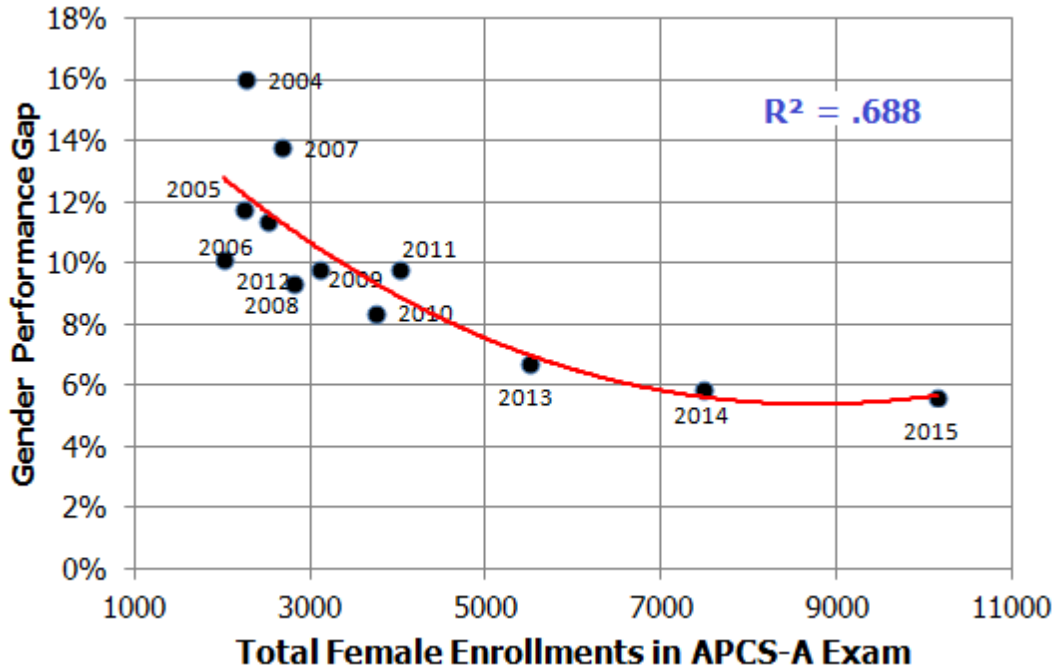


Figure 12. CS Gender Performance Gap, 2004-2015

College Board data also show a positive correlation between the total number of students who take a particular *STEM* subject exam and the participation rate of female students in that AP course (**Figure 13**). The graph suggests that low total enrollment numbers may contribute to the gender disparities seen in U.S. secondary CS participation rates. Note that this is not a novel idea. In a study looking at the effect of culture and environment on women's participation in CS, Blum et al. said as much, stating their "belief that the reasons for under-representation [of women] in CS are very much the same as the reasons for the huge decline in interest in the field generally." (Blum L. , Frieze, Hazzan, & Dias, 2006)

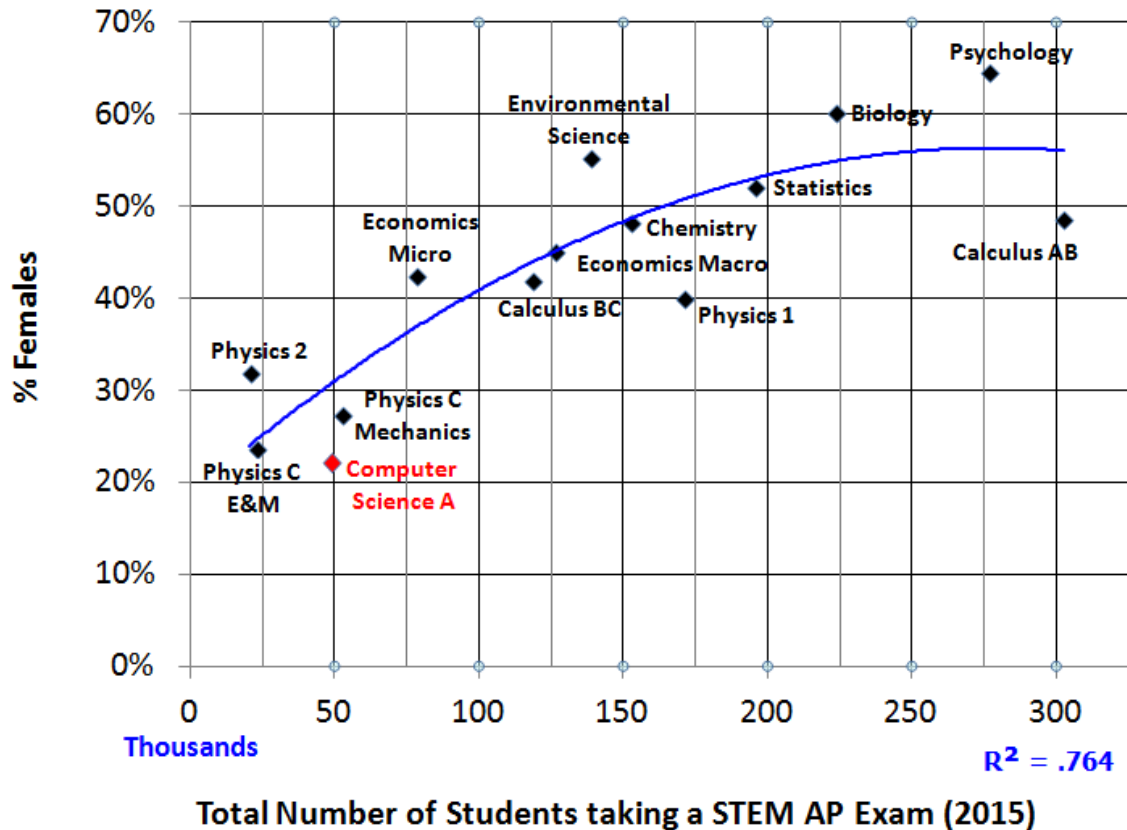


Figure 13. %Female vs. Number of STEM Test Takers, 2015

The corollary would predict that an increase in overall enrollment numbers in CS classes would narrow the gender gap. There was, in fact, an improvement from 2004 to 2015, as female participation grew from 16.2% to 21.9% as overall enrollments rose from 13,872 in 2004 to 46,344 in 2015 (Figure 14, the quadratic regression differing little in position and significance from the linear regression). The improvement is extremely modest considering that enrollments more than tripled, but the trend is still significant. There are – at least – two interpretations: (a) Low total enrollments are a minor cause of CS gender disparities, or (b) much higher total participation rates will be required in order to see substantial increases in female enrollments.

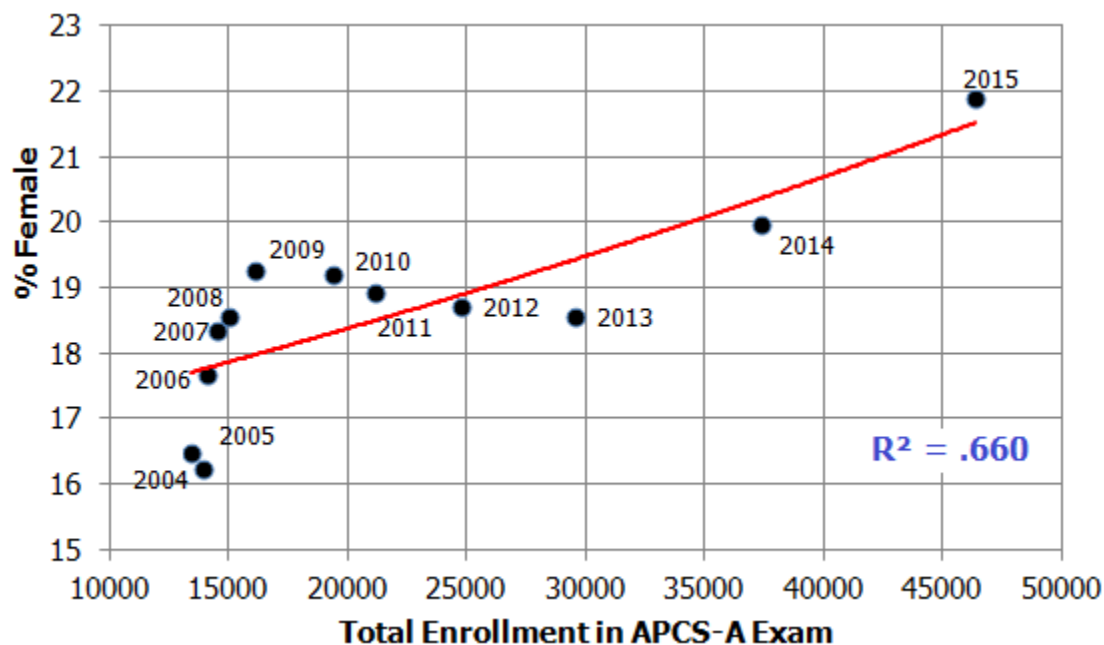


Figure 14. %Female vs. Total Number of APCS Test Takers, 2004-2015

Historically, the AP Physics courses have also experienced nearly as severe gender disparities, but among them, those courses with *substantially* higher total enrollments have had better female participation rates, an argument that favors the second interpretation above. Through 2014, the Physics courses consisted of two calculus-based "C" courses and one algebra-trigonometry-based "B" course. In 2015, the latter was discontinued and split into two courses called *Physics 1* and *Physics 2*.

Physics C: Electricity & Magnetism and *Physics C: Newtonian Mechanics* had total enrollments of 20,110 and 48,207 with female participation rates of 23.2% and 27.1%, respectively. Despite total enrollments that were lower (17,758), the new algebra-trigonometry-based *Physics 2 (Fluid Mechanics & Thermal Physics, Electricity & Magnetism, Optics and Nuclear Physics)* had a higher female participation rate of 31.2%, somewhat lower than its *Physics B* predecessor. However, its sibling *Physics 1 (Rotational Mechanics, Intro to Electricity, Waves & Sound)*, with a whopping total

enrollment of 162,796, had a female participation rate of 40.1%, 9 points higher. This is also 5½ points higher than its *Physics B* predecessor's rate of 34.5% in the years 2011-2014 (constant during those 4 years) with half the enrollments of *Physics I*: 71,395 in 2011 and rising to 87,495 in 2014. These statistics do seem to support the hypothesis that very high numbers are needed for female *participation* rates to begin approaching parity in this subject area. At this point in time, one can only speculate what differences might make Physics and CS so much more resistant to achieving gender parity than Environmental Science, Statistics and Chemistry.

In terms of gender *performance* gaps, the two *Physics C* courses and *Physics 2* had small gaps (mean scores) in the range 6-11% favoring males. *Physics 1*, however, had a much greater gender performance gap: 22%, a rise of 7 points from *Physics B*'s previous 4-year gender performance gap average of 15%. This behavior is the opposite of that observed in CS and would argue that the correlation in **Figure 12** is not causative.

If higher enrollments do have a causative role in CS gender disparities in participation or performance, how to bring about increases in total CS enrollments, and female enrollments in particular, is very much an unknown. Some CS educators have suggested that curricula centered on story-telling using IDEs like Alice or Scratch might attract girls to computing (Kelleher, Pausch, & Kiesler, 2007). However, Blum et al. warned:

The implications are that women do not need handholding or a "female friendly" curriculum in order for them to enter and be successful in CS or related fields, nor is there need to change the field to suit women. To the contrary, ***curricular changes***, for example, ***based on presumed gender differences can be misguided***, particularly if they do not provide the skills and depth needed to succeed and lead in the field. ***Such changes will only serve to reinforce, even perpetuate, stereotypes and promote further marginalization.*** (Blum L. , Frieze, Hazzan, & Dias, 2006)

Despite reports of 29% of high school CS instructors using Alice as a first language in introductory courses (Carter D. , 2012), this has not translated into greater gender diversity or an increase in female participation rates for the APCS-A exam.

Although it is widely acknowledged that such IDEs initially generate a high level of engagement, it has also not been demonstrated that this interest can be sustained. One likely explanation is that the fault lies with the curricula used in conjunction with Scratch or Alice. If the contexts for class instruction are game-related or make-believe, the message teachers send is that CS has no real world applications of any consequence or value beyond entertainment or play. One study has shown that gaming curricula actually discouraged non-majors' interest in CS (Rankin, Gooch, & Gooch, 2008). It has also been reported that games "have been tried and have failed to have an effect on steeply declining female enrollment" (Stross, 2008).

It is also true that these curricula employ the same default educational paradigm criticized by Margolis et al, i.e. "*programming, programming, programming, abstracted away from real world contexts.*" There's actually a very good reason that this is the case. Introductory courses focus on imparting the basic programming infrastructure that will allow students to write methods, both procedurally and using an OO approach. Applications, particularly those that would hope to model "real world contexts", require a software engineering approach that involves organizing methods hierarchically and within classes and modules, something that has pedagogically been postponed until later. This sequence builds programming skills from the bottom up. Changing this to a top-down paradigm, in which both sets of skills could be taught simultaneously, is a tough act to pull off. You can't exactly read *Madame Bovary* if you're still learning elementary

French vocabulary and grammar. Even intermediate-level news web sites like *News in Slow French* (newsinslowfrench.com) require a full year and a half of high school French.

Nevertheless, one might term the basics-first approach *self-referential*, a curriculum that has a strict focus on programming topics and that ignores CS's role as an applied or engineering science with applications in numerous domains outside of CS. A self-referential approach has limited appeal beyond the traditional demographic. Thus, despite being contextualized in a virtual, self-contained 2-D or 3-D world, the core content and pedagogic approach of such curricula differ little from that of curricula that they've replaced. This is not the first time this criticism has been articulated:

Typical computing programs of study today have changed very little over the years and decades while the computing field has evolved and transformed itself many times over... ***Are we doing our students an injustice? Have we cheated them by not keeping pace with the field?*** ... We claim that it is time to think beyond the bounds of our own learning and to consider new ways of learning computing. (Impagliazzo & McGettrick, 2007)

Section 6. Principles for Implementation

An alternative to the *self-referential* curricular paradigm is one where units consist of cross-curricular central problems which students solve over a period of several weeks using whatever CS tools are needed. The structure of this type of curriculum resembles *IMP* (Interactive Mathematics Program), whose development was funded by NSF in the 1990s. *IMP* is a 4-year program of problem-based mathematics that replaces the traditional *Algebra I-Geometry-Algebra II-Trigonometry/Pre-calculus* sequence.

IMP units are generally structured around a complex central problem. Although each unit has a specific mathematical focus, other topics are brought in as needed to solve the central problem, rather than narrowly restricting the mathematical content. Ideas that are developed in one unit are usually revisited and deepened in one or more later units. (Fendel, Resek, Alper, & Fraser, 2008)

For example (although the real-world application is a stretch), the complex problem posited by *IMP*'s *High Dive* unit (Year 3) has students calculate at what point a circus performer on a turning Ferris wheel must dive so as to land in a tub of water on a moving cart. To solve the problem, students extend right-triangle trigonometric functions to circular functions; learn about the graphs of the sine and cosine functions; and study polar coordinates, inverse trigonometric functions, the Pythagorean identity, and the physics of falling objects.

Extending this approach to an introductory CS course is not feasible in the short-term. Even small cross-curricular problems in CS require a working knowledge of all basic programming constructs. In this sense, CS is like learning a foreign language: one needs to acquire a basic foundation before one is able to read even simple literature. The multi-week cross-curricular units in the curriculum described in this project report appear in the second half of the course, following students' acquisition of reading/writing

proficiencies in using conditionals, iteration, variables, variable types, basic data structures, classes/objects and functions. Getting students up to this minimum level quickly needs to include a short, but sustained, period of practice with units comprised of sequences of progressively more complex problems, à la websites like *Codingbat.com*, and can be done in approximately one semester and change.

Moreover, the goals of this cross-curricular course are different from *IMP*. The *IMP* thematic units are carefully sequenced over a three-year period to build algebra, geometry, trigonometry and pre-calculus proficiencies. The sequencing may differ from the traditional sequence of math strands, but is critical nonetheless.

In contrast, this CS curriculum has two goals. The first is to ***expand student repertoires*** for how they can use programming constructs to solve a variety of tasks, to facilitate *self-efficacy*. The second is to ***provide purpose***, to counter the traditional paradigm of disembodied knowledge, to spur *interest*. This is done by giving students opportunities to solve engaging, motivational problems set in intriguing real-world contexts designed to answer the question: "*How can I use this knowledge?*" The sequencing of the CS modules is important, but not particularly critical, because students will have acquired basic programming competencies by the time they encounter the material.

Moreover, to attempt to design a CS curriculum that parallels *IMP* would be premature because of the near total ignorance of how students learn in this subject area. The first part of this thesis claims that language-acquisition-based pedagogies play a critical role in the first year of learning fundamental programming concepts. However, the fact that even simple programming problems are open-ended, with many possible

solutions, points to pedagogies that should be developed and employed to facilitate the higher goal of programming literacy. Programming literacy – writing hierarchically organized, concise, clear, maintainable and *elegant* programs – is a craft with similarities to creative writing. We know that the only way to become a better reader is to read, and that reading helps one learn to become a better writer (Krashen S. , 2004). Reading programs would seem to be a non-starter due to the difficulty and patience needed to decipher them. Rather, effective pedagogies for programming literacy will probably follow Corder's ideas that language learning happens spontaneously when conditions in the learner's environment encourage it.

The guidelines for acceptance of a Computer Science course that meets University of California A-G requirements is that it both "teaches students to express algorithms in a standard language" and "requires students to complete substantial programming projects", both of which contribute to the rigor of this curriculum. The Applied CS units for this study are built using the following guidelines:

1. The units employ a *software engineering approach*. CS's natural relationship to other disciplines is Software Engineering, a field which uses programming and CS concepts, along with expert knowledge of specific target topics, to model and solve societal problems and needs. Topic areas will largely lie outside the field of CS proper.
2. Central problems are *multi-part* and *multi-week*. Each unit lasts several weeks and revolves around solving a central problem in the target topic area. A software solution evolves in an incremental way, utilizing whatever CS concepts and control

structures are required. Various programming and algorithmic strategies are attempted at each step in a problem's solution, and students weigh the advantages and disadvantages.

3. Programs are *modeled after real applications*. One way instructors can foster connections is by having students create small scale versions of engaging and complex real world applications. Students should be able to envision logical extensions of their projects to the already existing and more complex programs from which they were drawn.
4. Interdisciplinary knowledge provides *context*. To solve a unit's central problem, students need to become familiar with relevant concepts in geometry, trigonometry, biology, physics and so forth. This reflects typical conditions that software engineers encounter in their day-to-day work, i.e. programmers must not only be proficient in their own field, but must have knowledge of the specific non-CS systems they are modeling in order to write accurate, robust and logically organized programs. Pedagogically, this gives students multiple contexts for recalling and integrating what they learn.
5. Unit problems are considered within a *social and/or historical context*. This provides students an explanation why solving the central problem is important in the first place. These so-called back-stories may utilize Literature, Theatre, History, Social Studies, Economics, Film and/or Art in making those connections.

In the Astronomy unit, for example, a short program written in *Processing* simulates a heliocentric (Copernican) model of the solar system and allows students to view the complete cycle of Venus's phases as seen from Earth. A companion program that simulates the geocentric (Ptolemaic) epicycle model will demonstrate the impossibility of observing both a completely dark Venus and a fully lit Venus. When students subsequently read Bertolt Brecht's play *Life of Galileo*, they learn that it was this single celestial observation – made possible by the newly invented telescope – that was a pivotal point in the ongoing erosion of papal power, already weakened by the Reformation. Although Galileo himself was put under house arrest for the remainder of his life, his discoveries loosened the Church's capacity to impede the pace of science during the Renaissance.

On the pedagogic level, this unit taken in its entirety extends and clarifies students' understanding of events 400 years old. It does so through the use of student-written software that can clarify the true nature of a celestial phenomenon, one whose logical implications had huge historic, social and political ramifications. Because of its many facets, the unit contains multiple points at which students can make engaging connections.

If placing problems in real-world contexts answers the question "*How can I use this knowledge?*", providing historical and social contexts allows students to ask "*What is the human/societal impact?*" At this point, students step back, gain perspective and reflect on the big picture to observe how their work can be used *and* misused. For example, IBM, in pursuit of its bottom line, conducted business as usual by supplying pre-war Nazi Germany punch cards for the Hollerith machines it

used to automate its data collection methods in pursuit of its racial laws. The Galileo and Holocaust units bring up powerful ethical questions for scientists and engineers that dwarf such standard topics as intellectual property. On the constructive side, this part of each unit can also be a time for students to ask the question: "*What do I want to create with this knowledge?*" and begin to envision themselves as technology creators.

6. Units should make credible *connections* to academic fields and topics students have already studied. This allows them to extend prior knowledge, and facilitates quicker engagement with the material. Detailed descriptions of the course's cross-curricular units appear in Chapter 2. Below are outlines for potential units, not yet fully developed.

a) **Evolution and Social Reaction (Evolution, Genetics, Bioinformatics).**

Use of genomic databases and software tools to align DNA and protein sequences from related species to build phylogenetic (evolutionary) trees. Study and modeling of bioinformatics algorithms: LCS algorithm, global alignment, local alignment, scoring matrices, clustering – all used by the free software MEGA (Molecular Evolutionary Genetics Analysis).

- Earthquake Epicenter Algorithms using Triangulation. (Science Courseware.org, CSU). Study of the 3/11/2011 Japanese Earthquake and Tsunami and its aftermath.

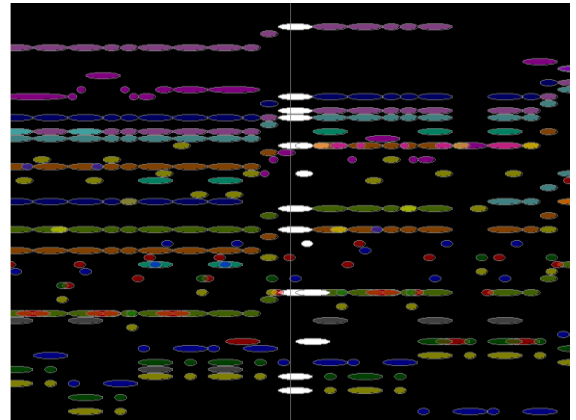
c) **On the Road (Geometry / Geography)**

GPS and Routing Programs

- A GPS program based upon triangulation of satellite data, equations for calculating longitude and latitude on a sphere, and a geographic database.
- A routing program using Dijkstra's shortest path algorithm (like Google Maps).

d) **Music Visualization (Music Animation Machine)**

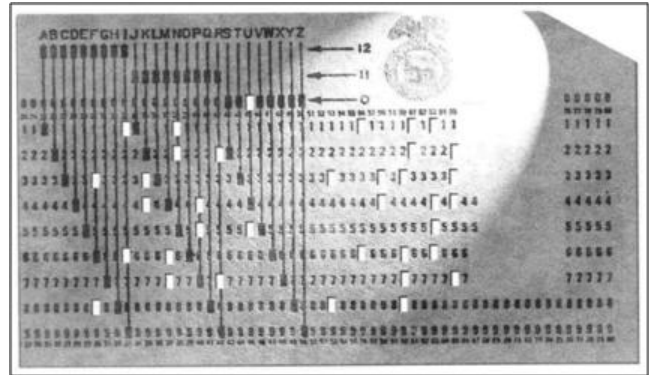
Synchronization of the orchestral instruments of musical works – such as Beethoven's 7th Symphony or Vivaldi's Four Seasons (Winter) – with colored geometric shapes moving across the screen that represent notes (pitch, duration, volume). Study of the life and work of *Walter/Wendy Carlos*, a 1960s pioneer in electronic music, and an examination of the nature of prejudice against sexual minorities, particularly transgender people.



CS concepts: Proportional reasoning and sequence/series concepts are used to determine the frequencies of musical notes and for calculating vertical positions; parsing of string input; synchronization of midi synthesizer output with visuals; discontinuities between system timer and midi timer.

e) **IBM's Strategic Contribution to Facilitate the Holocaust (Holocaust Studies)**

Before the invention of computers, **punch card technology** was used to solve database-related problems. The **sorting and tabulating algorithms** used to process these cards were ancestors of methods used in present-day databases.



IBM and its German subsidiary *Dehomag* were active participants in the processing of population data used to identify, transport and exterminate the Disabled, Jews, Gypsies, gay men and Communists throughout Europe from 1933 through 1945.

Excerpts from the book: *IBM and the Holocaust*. (Black, 2001)

Film/Book: *Sarah's Key*. Deportation of a young Parisian girl and her family during the *Vel d'Hiv* roundup of July, 1942.

Film: *A Film Unfinished* (footage from the Warsaw Ghetto)

Film/Play: *Copenhagen* (ethical conflict between physicists Niels Bohr and Werner Heisenberg)

Websites: *United States Holocaust Memorial Museum*; *USC Shoah Foundation*

Institute: IWitness; Yad Vashem, the Holocaust History Museum

Students build a simulation of a Hollerith machine and implement sorting and tabulating algorithms to order punch cards coding for census data.

Candidate Unit Topics that were considered, but not fleshed out, included:

1. ***DNA Fragment Assembly*** – used in the Human Genome Project. (DNA Learning Center)
2. ***SNP*** (Single Nucleotide Polymorphism) ***Mapping of Genes*** (Sadée) (Altshuler, et al., 2000)
3. Classical ***Population Genetics*** using Blood Types.

CHAPTER 2.

THE COURSE OUTLINE FOR CPRWE: COMPUTER PROGRAMMING AS IF THE REST OF THE WORLD EXISTED

Section 1. Introduction

CPRWE (Computer Programming as if the Rest of the World Existed), is a year-long introductory programming course intended to give students (1) a rigorous overview of and basic literacy in the uses of a structured programming language, using the Java-based language *Processing*; and (2) familiarity with algorithmic problem-solving. Within the context of programs of mid-level complexity and size, and cross-curricular fields of application (science, art, humanities), students learn the uses of variables, Boolean expressions, and iterative and conditional control structures. They learn to encapsulate code within methods, pass input (arguments) via parameters, and calculate return values. Students learn to think of programs as interactions of objects having attributes and methods that they describe in classes. They learn software engineering principles for top-down design, resulting in hierarchically organized programs for optimal maintenance, modification and extendibility. They examine criteria for deciding which of competing code styles and algorithms to implement. Equally important, they learn the possibilities for non-trivial applications of programming to study and solve diverse problems across the STEM, Humanities and Arts curriculum. To write accurate programs, students learn and use cross-curricular concepts from such core areas as math (e.g. algebra, trigonometry), chemistry (electro-negativity, covalent and hydrogen bonds) and biology (DNA structure and genetics). In order to give purpose and context to the programming task, students study a film or play that situates the target

problem within an historical / societal context. For example, in the Astronomy unit (*Galileo's Revolution*), students build (a) a Copernican simulation of the solar system to understand such observational phenomena as the phases of Venus, Mars in retrograde, and the infrequency of solar eclipses; and (b) a Ptolemaic simulation (using *epicycles*) to prove that such a model cannot account for all the phases of Venus. They then study Bertolt Brecht's play *Life of Galileo*, and consider the repercussions of the discovery of the phases of Venus (a) in accelerating the pace of both the scientific Renaissance and the religious Reformation, and (b) in weakening the Church and eventually the monarchies of Europe.

The course is divided into three sections:

- a) Basic Programming Skills and Introductory Projects: Graphics basics, Primitive Methods / Arguments, Coordinate Plane Manipulations, Processing Mouse and Keystroke Events, Animation. `setup()` Initialization Code; `draw()` Animation Code, User-Defined void Methods / Blocks / Indentation, Variables, System Variables, Classes / Objects, Arrays, Iteration.
- b) Building Programming Skills: Methods that return values, Primitive Types (int, boolean), Method Parameters, Hierarchy / Nested Conditional Statements
- c) Intermediate Projects: Software Engineering Principles, Multiple Structural Recursion, Inheritance, Polymorphism.

The first section is intended as a "*whole language*" approach where students learn to recognize and use basic programming components to build four **dynamic art** programs sequenced in increasing levels of sophistication. The second section uses **CodingBat** to help students gain proficiency in programming skills and recognize programming issues

that involve Boolean logic, strings, arrays and iteration. The third section is comprised of four multi-week projects where students build scaled down, but functional, **applications of real-life software programs**, and use them to examine or solve specific problems in **government, geography, astronomy and molecular biology**.

Section 2. Piet Mondrian Painting (PART I: INTRODUCTORY PROJECTS)

Essential Question

How does one **design** a computer program?

Supporting Questions

Where does a computer program begin **execution**?

What is a **method**?

How does a **primitive** method differ from a **user-defined** method?

Does the order of **arguments** in a method **call** matter?

What is **hierarchical** organization?

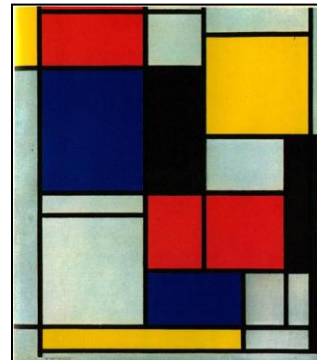
What are the advantages to **organizing** one's code?

How do **syntax** errors differ from **logic** errors?

How do methods that set **modes** operate?

How does one fix **bugs** in a program?

How does the **RGB** color system – and transparency – work?



Description

The unit introduces students to the **Processing** programming environment and familiarizes them with its **inverted Cartesian coordinate plane** (origin at the upper left corner). They learn basic **drawing** and **mode** methods for rendering regular and irregular shapes. They learn the programming concept of **hierarchical organization** by defining **methods** with **meaningful names** and grouping primitive functions into the **method bodies**. They learn to call these methods sequentially in the program's **entrance point** method **setup()**. They learn **indentation** conventions to organize lines of code for legibility. They learn to add **comments** to their program to clarify intent. They learn that the settings of **mode** methods persist beyond the methods in which they are used – until they are next changed. They learn simple **debugging** techniques for locating the source of **logical** errors. They learn about programming language **syntax**, such as matching parentheses and curly braces, and the order of method arguments; they learn to debug **syntax** errors. Students learn the **RGB** color system and transparency.

Key Assignments

Following an introduction to the Processing programming environment and basic drawing methods, students are given an image of the Piet Mondrian painting and shown how to determine coordinates of rectangle vertices and line endpoints using the system tool **Paint**.

Students then write a hierarchically organized program that renders a full-scale and close approximation of the image. In the course of completing the task, students:

1. Create parameter-less user-defined methods made up of primitive methods.
2. Call primitive methods using the correct coordinates and widths/heights.
3. Call user-defined methods in `setup()`;
4. Use primitive **mode** methods at the beginning of each user-defined method to avoid persistence side-effects.
5. Employ simple debugging strategies to locate and correct syntax and logic errors.

Teaching Strategies

Instructor uses direct whole-class instruction to demonstrate the **Processing** programming environment and its inverted Cartesian coordinate plane (origin in the upper left corner) as students practice at their workstations and instructor and advanced students circulate to help students having problems. Instructor similarly guides students through the use of basic drawing methods, and accessing/reading online documentation. Using guided discovery, students learn by examining program output: (a) how colors are defined using RGB values; and (b) what the various attributes do that **mode** methods set.

Instructor clarifies how primitive methods work by using counterexamples, e.g. a different ordering of primitive methods or method arguments (signatures), to show incorrect or unintended program output.

Instructor helps students individually and via direct whole class instruction to debug syntax and logic errors in their program. Syntax errors in this assignment are limited to orphaned opening or closing curly brackets or parentheses; and using a different ordering or number of method arguments than those specified in the documentation. Instructor teaches (a) students to recognize these specific errors, (b) procedures for avoiding these errors, and (c) simple strategies for locating these errors when they occur (commenting out lines, use of auto-indent feature). The most common logical error in this assignment is calling a **mode** method in one part of the program and not resetting the attribute in a subsequently called method. Teacher shows students that routinely calling mode methods at the beginning of user-defined methods, though seemingly repetitive, avoids this kind of error. Instructor shows students the use of print statements and comments for debugging logic errors.

Section 3. Ricocheting Comets

Essential Question

How does one program the **simulation** of **movement**?

Supporting Questions

How can the **draw()** method be used to simulate movement?

What is difference between using the **background()** statement in **setup()** vs. **draw()**?

How are **variables** in programming similar to and different from variables in Algebra?

How and where do you **declare**, **initialize**, **use** and **update** variable values?

What is an assignment statement, and what are the various forms it can take?

What is a **system variable**? [width, height]?

When in the execution of a program do system variables acquire meaningful values?

How can a **conditional** statement detect when an object reaches a specific location?

What's the relationship between a dividend being evenly divisible by a divisor and the **MOD** function?

Why use **parentheses** in conditional expressions if there is no difference in expression evaluation, i.e. if **order of operator precedence** yields the same result?

How do you use a variable to set the **speed** or **direction** of an object?

What does the **random()** method do; in what kind of situations would you want to use it?

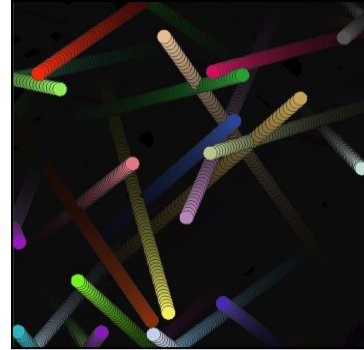
How does a **class** allow you to create multiple objects of a given type?

What are **instance variables**?

What is a class **constructor**?

What's the difference between a **class** and an **object**?

How does a class allow you to alter **attributes** of objects so that they look and behave differently?



Description

This unit introduces students to programming strategies that **simulate movement**.

Students learn to combine the use of drawing methods, **variables** and **conditional** statements to move a circular object across the screen and make it ricochet off the edges.

Students learn the **modulus** function and some of its uses in conditional statements.

They learn the advantages to using variables instead of hard-coded values. Students learn to combine simple conditions into complex conditional expressions using the logical

AND && and OR || operators. They learn to use **parentheses** inside of complex

conditional expressions in order to make the **intent** of their code **clear** and to avoid ambiguity. They learn how to use the **random()** method to dynamically change the color of objects. Students learn to define **classes** and use them to instantiate multiple **objects** of that class.

Key Assignments

The instructor guides students through the construction of a small program in which a circular object moves horizontally, reversing direction when it reaches the left and right edges of the window. Students then:

1. Write a program in which a circular object moves vertically, bouncing off of the top and bottom edges.
2. Write a program in which a circular object moves diagonally and ricochets off each of the four edges.
3. Modify #2 so that the circular object simulates *realistic* ricocheting behavior, i.e. bounces when its outer edge touches the boundary, rather than its center.
4. Calculate algebraic expressions for the slopes of the diagonal segments; and calculate the slope-intercept form of the equations for the lines lying on those diagonal segments.

In the course of completing these tasks, students learn to create different variables, each of which accomplishes a single task. For example, they must create separate variables for horizontal and vertical movement (*position* and *increment*); they must create a radius variable for modeling an object's outer edge.

Instructor introduces students to the **random()** method, and shows them how it can be used to change the color of an object. Students then:

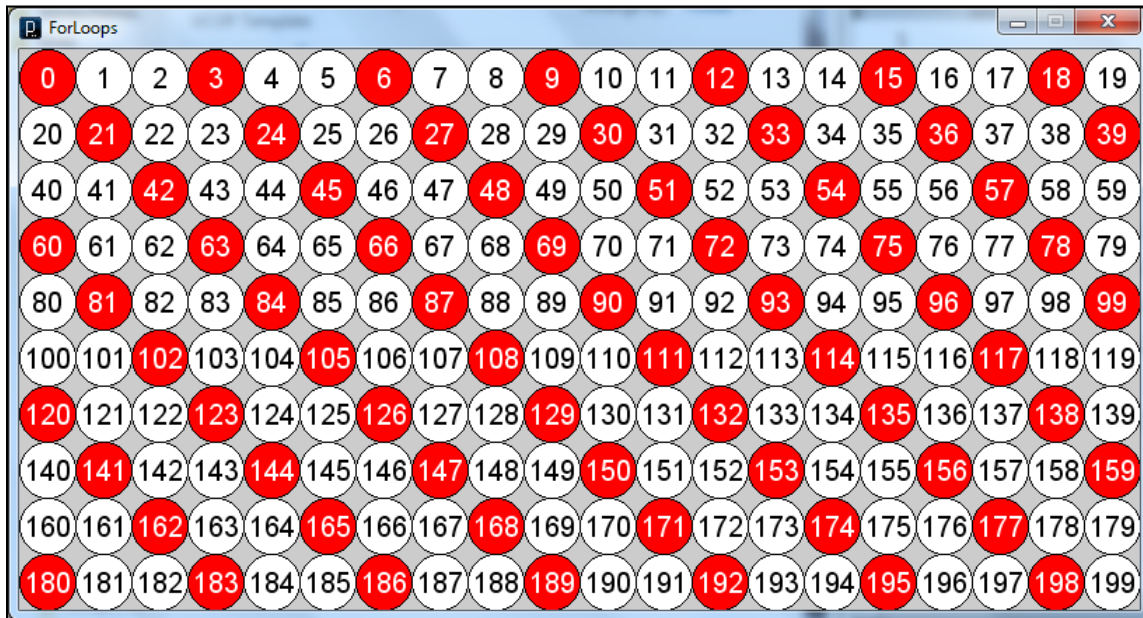
5. Modify program #3 to change the color of the circular object whenever it bounces off of an edge.

Instructor shows students the syntax for defining a **class**. Instructor shows how to write a **constructor** for initializing an object's attributes, and how to transfer the code developed previously into class methods with meaningful names. Instructor shows students how to **instantiate** an **object** of the new class, and how to call the object's methods in **setup()** and **draw()**. Students then:

6. Add 15 objects to the program, each having different starting directions and speeds.

Finally, instructor shows students how to use a combination of transparency values and drawing methods to give the illusion of a fading trail to a moving object. Students then:

7. Modify their programs to give the illusion of fading trails to their moving objects.



8. Students solve a series of increasingly complex problems using the **200-circle matrix program** (above), modifying only the conditional statement to produce the correct output of correctly numbered red circles. For example, the conditional statement that produces the output shown above is:

```
if (i % 3 == 0) {  
    drawRedCircle(i);  
}
```

Teaching Strategies

Instructor uses direct whole-class instruction, and circulates to help students. Students learn by **guided discovery**, observing the runtime output of **counterexamples**. Two cases detailed below show the use of these strategies.

1. When first coding the conditional statement to detect when the object reaches the window's right edge, after initializing the program with a **size(800, 600)** statement, students generally write: **if (x == 800)**. When we change either the starting x position or the increment so that x will leapfrog over 800, students modify this expression to **if (x > 800)**. Instructor then demonstrates the system variable **width**, and how it takes the value of the first argument to the **size()** method. Instructor then directs students to change the first size() argument from 800 to 700. Students observe that the object goes past the right edge, but eventually reappears in reverse direction. Instructor asks students to modify the program so that the object will ricochet at the right edge no matter what the width of the window is. Although several students will change the conditional expression to **if (x > width)**, a common error is for students to simply replace 800 with 700. Eventually, with enough prodding,

students make the correct change and come to understand the power of variables to make a program behave properly with varied input.

2. When teaching how the **random()** method operates, students are presented with an alternate way of coding the part of their program that changes the color of their objects once they ricochet. The original code is:

```
this.clr = color( random(256), random(256), random(256) );
```

The new code is:

```
color newColor = random(256);  
this.clr = color (newColor, newColor, newColor);
```

Without running the program, the question is put to the class whether the new code will have output equivalent to the original. After discussion, everyone tests the code and sees the different output, now limited to grayscale colors, rather than the full palette. Each student is asked to write a paragraph explaining why the new code results in different output. Students are then asked to modify the new code, all the while maintaining use of the **newColor** variable, and make it work (spoiler: use 3 variables). It is through these series of experiments that students learn to appreciate not just how the **random()** method works, but to recognize that misunderstanding of an API method can lead to logic errors because the programmer may use it incorrectly.

When teaching students how to reverse an object's direction, students are prompted to come up with several code fragments for reversing the sign of a number, for example:

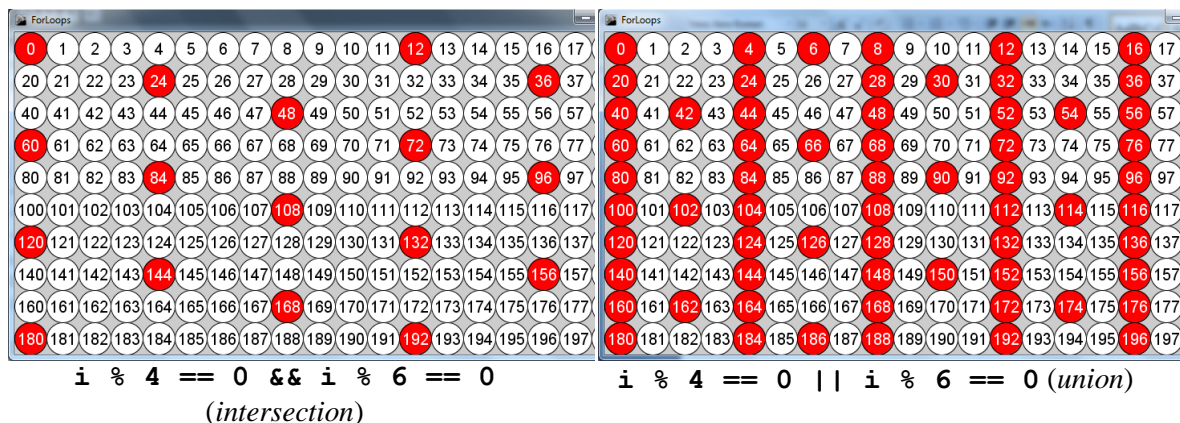
```
n = -n;  
n = n * -1;  
n *= -1;  
n = n - (2 * n); // which simplifies to the first statement
```

Counterexamples are also used in direct whole class instruction when introducing logical **AND** and **OR** expressions. **Venn diagrams** and **number lines** are used to graphically illustrate the difference between **AND** and **OR**, and to teach that **AND** corresponds to the *SET* concept of **INTERSECTION**, and **OR** corresponds to the *SET* concept of **UNION**.

Instructor distributes the Excel worksheet (at right) that only allows students to change the value in cell B2 (the divisor value). Through experimentation, students study the patterns of output when doing integer division (quotient) and modulus operations given **dividend** and **divisor** as inputs, and deduce that a modulus output of zero indicates that a dividend is evenly divisible by a divisor.

	A	B	C	D
1	DIVIDEND	DIVISOR	QUOTIENT	MODULUS
2	0	5	0	0
3	1	5	0	1
4	2	5	0	2
5	3	5	0	3
6	4	5	0	4
7	5	5	1	0
8	6	5	1	1
9	7	5	1	2
10	8	5	1	3
11	9	5	1	4
12	10	5	2	0
13	11	5	2	1
14	12	5	2	2

In direct whole class instruction, example practice problems using the 200-circle matrix program are demonstrated to get students started on the task. Below is the output showing the difference between **&&** (intersection) and **||** (union). Students also recognize that the **&&** expression is equivalent to $i \% 12 == 0$ and can be used to reveal the lowest common denominator.



Section 4. Rotating McClure Painting

Essential Question

How is a single method able to do different things?

Supporting Questions

What is **iteration**? Why use it?

How do **for-loops** allow you to do repetitive tasks or calculations?

What are the advantages to using a list (**array**)?

How are **members** of a list different from variables of the same type?

Can a Java list contain objects of **more than one type**?

What is the **connection** between iteration and lists?

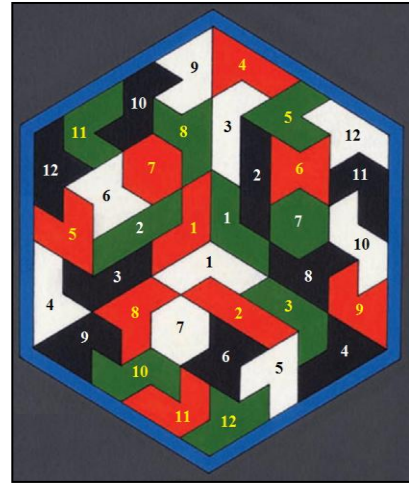
What kinds of programming errors cause **side-effects**?

How can programmers avoid coding **side-effects**?

How do you determine the **order** that you list **parameters** when defining a method?

Does the order that you do **transformations** (translate, rotate) matter?

Is **math** always involved when writing graphics programs?



Description

Students analyze a geometric image of a hexagonal painting. They recognize that the hexagon is composed of 3 identically shaped rhombuses (although component colors vary), and hypothesize that the program can draw the entire image by coding for the display of just one rhombus, then "rotating the drawing code" twice. Instructor gives students a helper-program that will create the code for a Java array of Points as they click on the 38 vertices of the 12 irregular polygons that make up each rhombus. Students splice this array into the beginning of their program, tweak vertex values for accuracy, and use the indexed points to write 12 methods for drawing the irregular polygons. Student write the methods by **bookending vertex()** method calls – which use indexed points as parameters – between **beginShape()** and **endShape(CLOSE)**. Students then encapsulate these 12 methods into a higher-level method called **drawRhombus()**, which

is placed in **setup()** [because the image does not (yet) move, there is no need to involve the **draw()** method at this point].

Students learn to code for the transformation by using the sequence: **translate()** - **rotate()**, which will **translate** the drawing plane's **origin** to the **center** of the figure, then **rotate** the drawing plane 120° in either direction before drawing the second and third rhombuses. Because *Processing's* **rotate()** method takes radians as input (rather than degrees), students learn the definition of **radian** and the common equivalents for standard angles (multiples of 30° and 45°). They are also shown the **radians()** method, that lets them simply wrap it around the more familiar degrees measures.

Students must also write a method **translatePoints()** that will translate each member of the Points array in the direction opposite to the translated origin in order to keep the hexagon center in the middle of the window. Prior to attempting this task, students gain a working knowledge of the **initialization**, **condition**, and **increment** parts of the regular **for-loop**.

To **paint** the rhombuses with the correct colors, students analyze the image for color patterns. They discern that there are 4 **sets** of 3 identically colored (**red**, **green**, **white**, **black**) polygons in each rhombus (1-5-7, 2-8-11, 3-10-12, 4-6-9). Each rhombus, however, colors the 4 sets differently. Student add four parameters to the **drawRhombus()** method. They then consider two methods for solving the problem: (a) leave the 12 polygon methods in number sequential order, and use 12 color-setting mode calls, one preceding each polygon method, or (b) regroup the 12 polygon methods according to their color set and precede each of the 4 groups with a color-setting **mode**

call. For further clarity, students create 4 **local** color variables for use as **arguments** in the **drawRhombus()** method call.

Finally, students make the image dynamic by moving the 3 **drawRhombus()** calls to **draw()**, and creating a global **angle** variable that is incremented at the end of **draw()**. This revisits the movement programming mechanism used in *Ricocheting Comets*, but for **angular**, rather than lateral, movement.

Key Assignments

1. Using the helper-program, students place a functioning code fragment for the **Points array** at the beginning of their McClure program.
2. Using the Points array's indexed point variables, students create 12 parameter-less methods for drawing the 12 irregular polygons in a single representative rhombus, and place these methods in a *working* higher-level user-defined method called **drawRhombus()**.
3. Students write a method called **translatePoints()** that recalculates the Points array coordinates so that the center point's coordinates is at the origin (0, 0).
4. Students write code for the **translation** and **rotation** transformations that allow one to draw the other two rhombuses with **drawRhombus()**.
5. Students add **4 color parameters** to **drawRhombus()** and modify the method body to paint each rhombus with the correct colors.
6. Students **declare, initialize, use** (with an additional **rotate()** call) and **update** a variable named **angle** that allows the image to rotate.

Teaching Strategies

Instructor uses **modeling** to help students understand the geometric transformation the program uses to draw the 2nd and 3rd rhombuses. The model likens the graphic drawing plane to a large sheet of paper, and the **drawRhombus()** method to a stamp. If one imagines that the paper does not move, then one must rotate the stamp to draw the 3 rhombuses. Implementing this would require the programmer to calculate a complete set of vertex coordinates for each of the 2 additional rotated rhombuses. Instead, the preferred method is to consider the **drawRhombus()** method as **fixed**, and to simply **rotate the sheet of paper** beneath it before "stamping" it. Instructor also makes an analogy to drawing a circle with a compass, using each of these procedures. Because the point of rotation is the paper's origin (top-left), a **translate** method needs to reposition the **origin** at the **center** of the hexagon prior to the rotation operation.

Using the 200-circle matrix program, Instructor uses guided discovery so that students see the effect of changing the **initialization**, **condition** and **increment** parts of the for-loop code that draws the red circles. In this way, students gain a working knowledge of how these 3 parts work together to perform an iterative task.

Instructor uses **guided discovery** to help students figure out how to write the **translatePoints()** method that will adjust the coordinates of the Points array so that the center point moves to (0, 0), and the remaining 37 points are translated an identical amount. Using a for-loop, students adjust point[0] and subtract its coordinates from the successive 37 points. Output shows that only point[0] has been modified. Instructor directs students to set the initialization part of the for-loop to 1 rather than 0. Students observe that the output is correct for all but the center point. Instructor asks students to

figure out why these **side-effects** occur. Students finally discover that they need to save off the original coordinates of point[0], then subtract these from all 38 points in the array. Students are thus made aware of the phenomenon of unintended side-effects that stem from altering a variable referred to **during** the performance of a task.

Section 5. Word Clouds

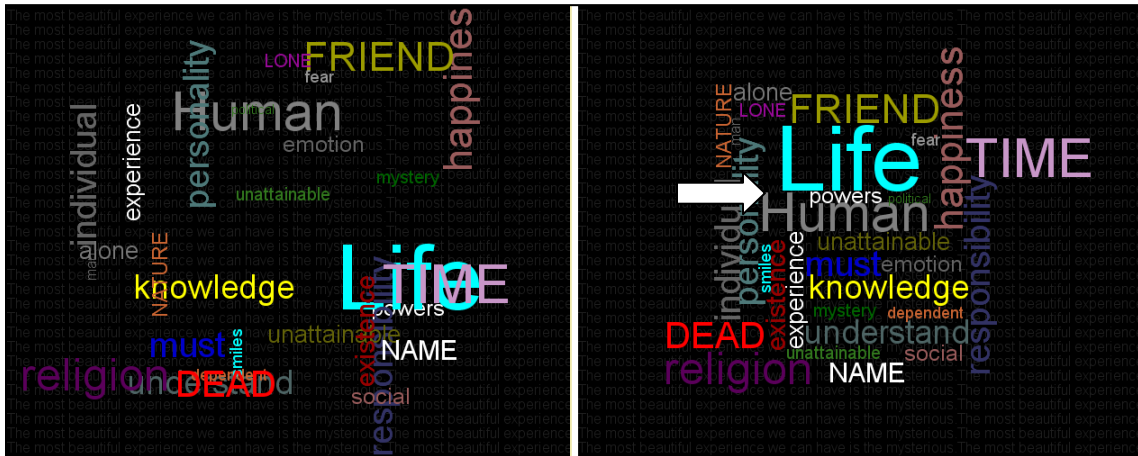
Essential Questions

What are the advantages to using **classes** in the organization of a program?
How does one isolate **side-effects** generated from transformation operations?

Supporting Questions

How is the **pushMatrix()-popMatrix()** combination similar in usage to the **beginShape()-endShape()** pair encountered in unit 3?
How does the **pushMatrix()-popMatrix()** combination prevent side-effects?
How do you use the **random()** method to place a word object along a window edge?
What are the advantages to using a **for-each** loop over a regular **for** loop? Are the two interchangeable?
How do you **synchronize** two or more events?
How do you write code to dynamically **alternate** between objects being in **motion** and then **at rest**?

Description



This unit teaches students how to use write programs that draw **text**. Students learn these new text methods, and are introduced to the **for-each** loop. They learn how to isolate transformation operations needed to render each word from having side-effects on subsequently drawn words by **bookending** commands between **pushMatrix()** and **popMatrix()** calls. The Word Cloud program intertwines these new concepts with the major programming concepts revisited from the first 3 units: **variables**, **conditional statements**, **Boolean expressions**, **arrays**, **classes**, **iteration** and **movement**.

Students spend time finding out about and experimenting with word clouds. They find lengthy pieces of text ranging from essays to state documents, and use them as input to any number of Internet word cloud programs referred by the Instructor. The instructor guides the class through the construction a simple program that shows how to **create fonts** and use them to **output text**. These methodologies are then encapsulated in a **DynamicText** class whose constructor takes a list of parameters for text, font, size, position, color, rotational angle and alignment. Students create an array of **DynamicText** objects, and output them using a **for-each** loop. Instructor demonstrates how to create a color-compatible background using text and a for-loop. Students use this code as a model to write a new program that will create a densely packed word cloud design using the most frequently occurring words in a student-chosen text passage.

To add motion, the instructor gives students a helper "*edges*" program to discover how to write code that will place each word at a random starting location on any of the four window edges. With instructor guidance, students discover a linear equation model for synchronizing the starting and ending times of all words from their initial to final locations. Lastly, students modify the program so that it cycles and spends equal time between two states: (a) text objects moving from random positions on the edges to their final positions, and (b) text objects remaining at the final positions to allow time for appreciation of the final static design.

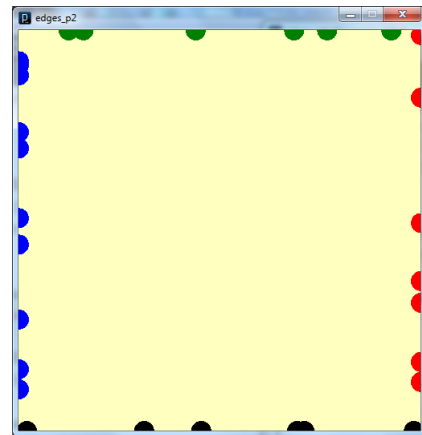
Key Assignments

1. After whole class instruction, students build a sample program that can output text of varying colors, size, font, rotational angle, alignment, and position.

2. Students build a program that outputs a static Word Cloud design.
3. Students modify their programs to output a dynamic Word Cloud where words appear at random positions on the window's 4 edges, then drift for 3-4 seconds to their final positions, where they come to rest for an equal period of time. Program cycles "forever" between these two states.
4. Using the techniques they learned in #3, students revisit the McClure painting program and modify it so that it will (a) alternate the direction of rotation, and (b) change background colors every time it begins to rotate in the opposite direction.

Teaching Strategies

Using the helper-program *Edges*, students examine two concepts: (a) randomly positioning (text) objects at the four edges of a window; and (b) mathematical variants for defining 4 random intervals, and their resulting constraints on programming style decisions.



Using whole class instruction, teacher guides students to discover what the x and y coordinates have to be if an object is to appear at any random position on the left edge: `x = 0; y = random(0, height);` Students sequester the code in a method called `leftEdge()`, then write similar the method bodies for `rightEdge()`, `topEdge()` and `bottomEdge()`.

Instructor next guides students to discover 2 basic variants for defining 4 random intervals of equal size:

```

float n = random(0,4);
if (n < 1) { leftEdge(); }
else if (n < 2) { rightEdge(); }
else if (n < 3) { topEdge(); }
else { bottomEdge(); }

float n = random(0,4);
if (0 <= n && n < 1) { leftEdge(); }
else if (1 <= n && n < 2) { rightEdge();}
else if (2 <= n && n < 3) { topEdge(); }
else { bottomEdge(); }

```

Students are asked to consider the two code fragments for simplicity and clarity.

They are then asked to swap lines, e.g. swap lines 2 and 3. Students discover that this has no effect on output for the second code fragment. However, in the first code fragment, no circles appear on the right edge, i.e. the `rightEdge()` method is never called. Students are asked to explain the phenomenon, and instructor illustrates the concept using (a) the number line, and (b) rearranging a sequence of filters/sieves with increasingly larger holes that are catching balls of various diameters, and so on.

To help explain saving/restoring of the drawing plane's **state** by **pushMatrix()**-**popMatrix()** – used by the program to allow text objects to rotate *independently* – instructor uses a camera metaphor, e.g. taking a snapshot of the drawing surface before any translation/rotation operations, performing the transformations, then restoring the prior state using the snapshot.

To derive expressions that allow the text objects to move (diagonally in most cases) from initial positions to final positions, instructor guides students to calculate a slope/intercept equation for both horizontal and vertical components of the motion. In this case, however, x and y are the dependent variables and *percent completion of motion* is the independent variable, with slope equal to the difference between final and starting coordinates, and the *y-intercept* equal to the starting coordinate. Instructor gives students

hints by asking what the x-coordinate would be at 0%, 100%, 50%, 25% (in that order) and so on (Note: although we are calling the variable "*percent*" for ease of instructional discussion, in a strict sense, it is in fact the *fraction* of movement traveled). Students are thus guided to derive the equation for the x-coordinate (below). Once solved, students are directed to derive the expression for the y-coordinate using the same methodology.

```
float percent = this.timeCurrent / TOTAL_TIME;
float xCurrent =
    (this.xEnd - this.xStart) * percent + this.xStart;
```

To make the objects rest for an equal amount of time, instructor directs students to keep incrementing `timeCurrent` to twice the `TOTAL_TIME` before reverting back to zero. Students observe that this causes each text object to travel twice as far, specifically 100% beyond their final positions. The remedy is to simply cap *percent* at 100% for all values above 100%:

```
float percent = this.timeCurrent / TOTAL_TIME;
if (percent > 1.0) {
    percent = 1.0;
}
float xCurrent =
    (this.xEnd - this.xStart) * percent + this.xStart;
```

Section 6. CodingBat: Boolean logic, Strings and Arrays (PART II: BUILDING PROGRAMMING SKILLS)

Description

CodingBat is a free site of live coding problems to build coding skill in Java... The coding problems give immediate feedback, so it's an opportunity to practice and solidify understanding of the concepts. The problems could be used as homework, or for self-study practice, or in a lab, or as live lecture examples. The problems ... have low overhead: short problem statements (like an exam) and immediate feedback in the browser.

- Codingbat.com/about.html

... Implicit in this is a [central] CodingBat idea: don't add complexity by making a problem which is realistic or has a motivating back-story. Practice problems do not need to be realistic. Instead, you want the description to be short and clear, and you want to have lots of problems so the student can work lots of repetitions (like exercise at a gym), building skill and confidence.

- Codingbat.com/authoring.html

At this point, students have had limited practice with most foundational programming concepts within a (hopefully) motivating dynamic art context. Although this should have given students a general framework for how computer programming can be applied, at this point, they need to begin to acquire basic programming competence. The intent is that students will not be learning disembodied skills, but rather will be learning to hone and expand their skill applying specific programming concepts they've already encountered within meaningful contexts.

Students now spend 6-8 weeks solving problems in 4 CodingBat modules: Logic-1, String-1, Array-1 and Array-2. Logic-1 covers Boolean variables, use of conditional statements (IF-ELSE, IF-ELSIF-ELSE, etc.), nested IF-ELSE statements, and common introductory logic problems. String-1 covers the use of the methods **length**, **substring**, **startsWith**, **endsWith**, **isEmpty**, **equals** and **equalsIgnoreCase**, as well as the logic of how to access string index positions from the start, end or middle of a string.

Array-1 covers simple problems in array creation, indexing and swapping of values.

Array-2 covers iteration through the members of the array, touching upon operations such as: searching; determining aggregate values; locating specific subsequences; and comparing adjacent member items.

The purpose of the module is to give students real programming competence using basic programming control structures and concepts. Because there are both simple and sophisticated ways of writing code to solve these problems, instructor requires that students first use the simpler, clearer (and longer) coding styles until satisfied that students understand the underlying logic and programming mechanisms. Instructor then shows students how and why the more sophisticated (and shorter) coding styles are equivalent.

Because solutions to CodingBat problems are rife throughout cyberspace, students are only given credit when they pass 4 custom/teacher-written quizzes, one for each module. These custom quizzes have the same format and style as all other CodingBat problems, and are accessible from the teacher's individual CodingBat home page.

In addition, the solutions to some of the problems involve programming issues that will arise in later projects, e.g. circular buffers (a clock). Because the CodingBat website simply glosses over these issues, the unit devotes significant time to exploring different ways of thinking about how one might model and program such systems, and requires expository assignments where students must *clearly* define the problem and explain how to code the solution.

Key Assignments

1. Logic-1 Module and Custom Test
2. String-1 Module and Custom Test
3. Array-1 Module and Custom Test
4. Array-2 Module and Custom Test

Teaching Strategies

Solving the problems in CodingBat is a major hurdle for all students. There are many ways/styles to write code that will solve the problems, and the solutions provided in the *Warm-Up* and *Help* sections of the website do not provide the necessary scaffolding required for most high school freshmen. Therefore, the teacher uses direct instruction to help students understand how to solve the problems in the *Warm-Up* section. Although each problem involves some new aspect or issue, the problem below can be used to illustrate the teaching strategies used:

The parameter weekday is true if it is a weekday, and the parameter vacation is true if we are on vacation. We sleep in if it is not a weekday or we're on vacation. Return true if we sleep in.

The instructor first demonstrates how to build a 2-D table that represents all 4 cases:

	vacation	!vacation
weekday	T	F
!weekday	T	T

Instructor then shows several ways to code the solution:

```
public boolean sleepIn(boolean weekday, boolean vacation) {
    return !weekday || vacation;
}

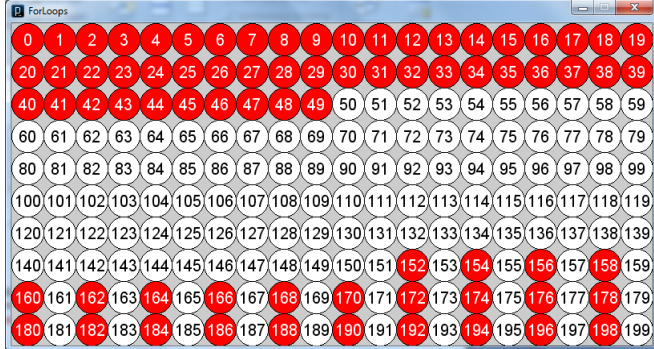
public boolean sleepIn(boolean weekday, boolean vacation) {
    if (vacation) {
        return true;
    }
    else {
        return !weekday;
    }
}
```

```

public boolean sleepIn(boolean weekday, boolean vacation) {
    if (vacation) {
        if (weekday) {
            return true;
        }
        else {
            return true;
        }
    }
    else {
        if (weekday) {
            return false;
        }
        else {
            return true;
        }
    }
}

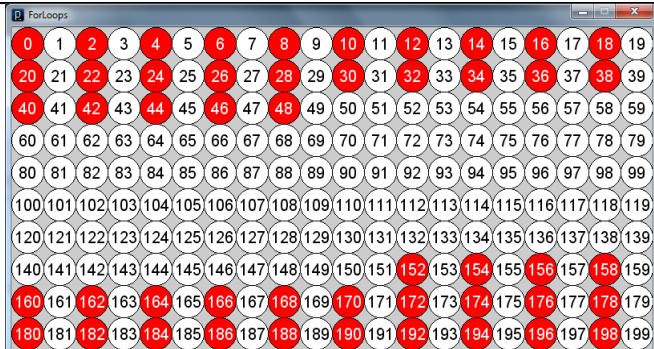
```

For students who initially struggle, the last solution is longer, but simpler to understand because each of the 4 possible combinations is represented. Once students see the code working, the instructor can guide them to realize that the first **if (vacation)** statement can be simplified because it always returns true, i.e. weekday is irrelevant to the return value.



`i < 50 || i > 150 && i % 2 == 0`

`(i < 50 || i > 150) && i % 2 == 0`



Some problems in CodingBat can involve complex Boolean expressions which combine the `&&` and `||` operators. The **200-circle matrix program** can show how `&&` has higher precedence than `||`. The expression

```
i < 50 || i > 150 && i % 2 == 0
```

yields the pattern above top, demonstrating that `&&` binds tighter than `||`, even though the expression is evaluated from left-to-right. When one adds parentheses to force the `||` to be evaluated first, as in

```
(i < 50 || i > 150) && i % 2 == 0
```

the pattern is as shown above bottom. The moral of the story is ALWAYS use parentheses to clarify the intention of the programmer.

Sometimes CodingBat problems inadvertently present common programming issues.

The problem below is an example:

We have a loud talking parrot. The "hour" parameter is the current hour time in the range 0..23. We are in trouble if the parrot is talking and the hour is before 7 or after 20. Return true if we are in trouble.

The Boolean expression for the time period between 8 pm and 7 am (non-inclusive), although written exactly as stated in the problem, is not intuitive because it spans the 0/24 boundary in the circular buffer representation of a clock. Normally a time interval between an earlier and later time is written using an AND expression, e.g.

`6 <= hour && hour <= 12`, analogous to how one would write a range using an

algebraic expression: `6 <= hour <= 12`. However, for intervals that span the boundary, the expression is nonsensical:

```
20 < hour && hour < 7
```

As illustration, the instructor uses Venn diagrams and the number line to demonstrate how a number cannot simultaneously be in two disjoint sets. One solution is to view the interval as the *union of two intervals* on either side of the boundary:

$$(20 < \text{hour} \ \&\& \ \text{hour} < 24) \ || \ (0 \leq \text{hour} \ \&\& \ \text{hour} < 7)$$

However, because the range of times is limited to 0-23, there is no need for the conditions in red. The expression simplifies to:

$$20 < \text{hour} \ || \ \text{hour} < 7$$

Because *students will revisit such boundary problems in the cross-curricular units*, they are required to write clear, but short, answers to all 4 of the following questions (in general, students who cannot do so do not yet have the abstraction abilities necessary to succeed in a programming course).

1. What is the normal expression for a specific time period on a 24-hour clock?
(e.g. between 12 and 18)
2. What is the problem with a time period that spans the 0/24 boundary?
3. What is the full expression for a time period that includes both sides of the boundary?
Explain how you get this expression.
4. Why can you simplify this full expression by dropping the (0 <= hour) and the (hour < 24)?

Section 7. Nested For-Loops, Regular Patterns, and T-Tables

Description

This is a short unit that introduces students to **nested for-loops** and a methodology for solving specific kinds of problems where these loops are used. Thirteen problems from the exercises section at the end of **Chapter 2** in *Building Java Programs* were adapted for this unit. The problems involve using **nested** for-loops to produce regular patterns of lined text output. In order to solve the problems, students must use inductive reasoning to determine linear expressions that describe all lines in the output, using line number as the independent variable. Students proceed by creating a T-table describing the number of different categories of characters/numbers for each line. They then graph the categories against line number and determine the **slope** of the resulting line. Plugging in the **slope** and any single **point** into the **slope-intercept equation** allows one to calculate the **y-intercept**. One now has a **slope-intercept expression** for each category. These are used in the **conditional** part of the **inner** for-loops, and at times for the output character itself, if it's a number. Below is an example pattern, a T-table, and the nested for-loops where the derived expressions are used.

```
* | | | |
** | | | |
*** | | | |
**** | | | |
***** | | | |
*****
```

line	#*	#
1	1	5
2	2	4
3	3	3
4	4	2
5	5	1
6	6	0
Algebraic expressions	line	6 - line

```

for (int line = 1; line <= 6; line++) { // outer loop
    // 1st inner loop prints asterisks
    for (int a = 0; a < line; a++) {
        out.printText("*");
    }
    // 2nd inner loop prints vertical bars
    for (int v = 0; v < 6 - line; v++) {
        out.printText ("|");
    }
    // after printing all the characters on the line,
    // go to the beginning of the next line
    out.printTextLine();
}

```

Key Assignments

13 Problems, including T-tables, graphs, Boolean expressions, and working code.

Teaching Strategies

Instructor uses whole-class direct instruction to go through the example described in the outline.

Instructor also has students reverse the process, i.e. trace through the code for several similar problems and show the output in both a T-table and console table for each iteration and output statement. An example of such a reverse problem is shown below.

```

for (int line = 1; line <= 5; line++) {
    for (int sp = 1; sp <= 5-line; sp++) {
        output.PrintText(" ");
    }
    for (int n = 1; n <= 2*line-1; n++) {

```

```

    output.PrintText(line);
}

output.PrintTextLine(); // Enter Key
}

```

Line	# spaces	# Number	Number
1			
2			
3			
4			
5			
Expression	5-line	2*line-1	line

Line 1									
Line 2									
Line 3									
Line 4									
Line 5									

Section 8. The Right to Vote
(PART III: INTERMEDIATE-LEVEL PROJECTS)

Essential Questions

How do you write a program to **simulate** an election, both the **marking** and **counting** of thousands of ballots?

How is a program like this similar to software used to tally **optical scan ballots**?

Supporting Questions

Describe the **flood fill** algorithm?

Define **recursion**. What's the danger inherent in using recursion?

What is the difference between global and local variables?

How would one decide whether to use a global vs. a local variable?

Description

To give students background, they begin the unit by examining Palm Beach, Florida's "butterfly ballot" from the Nov. 7, 2000 presidential election. Students watch the film 2008 HBO film *Recount* about the electoral chaos in Florida which was resolved on Dec. 12, 2000 by the U.S. Supreme Court decision that gave the election to George Bush. Students also watch the 2004 HBO film *Iron-Jawed Angels* which tells the story of the suffragist Alice Paul in the 8 years preceding the passage of the 19th Amendment. Students write an essay about the film in response to the prompt described in Key Assignments.

ELECTORS for PRESIDENT and VICE PRESIDENT	(REPUBLICAN)	3 →	●	← 4	(REFORM)
	GEORGE W. BUSH - President DICK CHENEY - Vice President				PAT BUCHANAN - President EZOLA FOSTER - Vice President
	(DEMOCRATIC)	5 →	●	← 6	(SOCIALIST)
	AL GORE - President JOE LIEBERMAN - Vice President		X		DAVID McREYNOLDS - President MARY CLAHOLLIS - Vice President
	(LIBERTARIAN)	7 →	●	← 8	(GREEN)
HARRY BROWNE - President ART OLIVER - Vice President		X		RALPH NADER - President WINONA LADUKE - Vice President	

At the start of the unit's programming section, students are instructed to figure out a way to mark the ballot above so that the entire white space within a circle is blackened. Students opt for what they know: using the **ellipse()** method to fill in the circle. However, because the border around the circles is pixelated, the rendering either leaves some pixels unmarked, or draws over gray pixels outside of the circle's boundary. At this point, Instructor introduces the **recursive Flood Fill** algorithm. Students reorder the recursive calls in the **floodFill()** method in a helper program that slows down the sequential filling in of the pixels; this allows students to see that the direction in which pixels are being drawn corresponds to the order of the recursive calls. Students use this information to write the body to a method named **markBallot()** that completely fills in a white circle for a single candidate. Students write a second method **markBallotX()** method that instead draws an X centered on a random location in the white circle, and consider what criteria should be used to determine the voter's intent, i.e. how many pixels in the white circle need to change color.

Students implement an **Election** class that marks and counts ballots. The method **markBallots()** creates thousands of ballots (**Ballot** class objects) and uses the (revisited) **random()** method to set the parameters for the percentages of the ballots that will be marked for each candidate. They will also use **random()** to mark a certain percentage of the ballots in some invalid way, such as for more than one candidate, or for no clear candidate choice.

Students then implement the **tallyElection()** method, which iterates through the ballot utilizing a **countBallot()** method that determines which candidate the voter selected. The **countBallot()** method in turn must employ a **readBallot()** method that

uses (revisited) **nested for-loops** to iterate through rectangular regions of the ballot image's pixels. They consider three algorithms for dealing with invalid ballots when implementing **countBallot()**: (a) Increment global variables for each candidate as marks are encountered. When a ballot marked for two or more candidates is determined to be invalid after it has already incremented their candidates' totals, it will be read a second time to decrement and correct those same variables. (b) Examine a ballot first to determine if it is valid. If so, read it a second time. (c) Read a ballot only once, but use local variables in the **countBallot()** method to first collect counts for all candidates. If the ballot is valid, use the local variable to increment the corresponding global variable.

Key Assignments

1. Students write an essay about the two films *Recount* and *Iron-Jawed Angels*, in response to the prompt: *The protagonists in both films used many strategies in their efforts to reach their goals, both of which concerned voting rights. Describe key strategies that each group used in order to try to attain their goal and how successful each of those strategies was.*
2. Students implement a strategy of their own design to **mark a ballot** for a candidate.
3. Students implement the **flood-fill** algorithm for fully marking a circle.
4. Students implement a method for **marking** the circle with an **X**.
5. Students implement an **Election** class with a method that returns an array of thousands of ballots marked with specified **percentages** for various candidates.
6. Students implement a certain percentage of **invalidly** marked ballots.

7. Students implement the **readBallot()** method that can determine the candidate(s) marked on a ballot.
8. Students implement a **countBallot()** method that throws out invalidly marked ballots and correctly increments the tally for the candidate marked.
9. Students implement the **tallyElection()** method that counts all the votes and reports the election results.

Teaching Strategies

Counterexamples, guided discovery, revisiting concepts and experimentation.

When presenting the **flood-fill** algorithm, a helper-program is distributed to help give students an intuitive feel for **recursion's** depth-first approach. The program performs the **floodFill()** method, but instead of drawing the pixels in real time, saves them in an array for drawing them slowly so that students can observe the sequence. The multiply-recursive method **floodFill()** appears as follows:

```
void floodFill(int x, int y, color targetClr) {
    MyPoint pt = new MyPoint(x, y);
    color clr = get(x, y);
    if (clr == targetClr && !contains(pts, pt)) {
        pts.add(pt);
        floodFill(x, y-1, targetClr);    // 1
        floodFill(x, y+1, targetClr);    // 2
        floodFill(x-1, y, targetClr);    // 3
        floodFill(x+1, y, targetClr);    // 4
    }
}
```

As students experiment with rearranging the numbered lines, they can observe the different directions in which the pixels are drawn.

The **readBallot()** method that examines a ballot's pixels **revisits/reuses** the **nested for-loop** iterative control structure that students learned in the previous unit. The method

looks at the circle next to each candidate and compares **white** pixels on the unmarked ballot with corresponding pixels on the marked ballot – if the corresponding pixel is a different color, the voter marked that candidate:

```
// returns candidate voted for (3-8) or 0 if invalid ballot
// x, y is the top left corner of a square bounding the
circle
// wh is the number of pixel rows/cols to process
boolean readBallot(int candidate) {
    int x = 453;
    int y = 34 * candidate - 68;
    int wh = 25;
    for (int row = x; row <= x + wh; row++) {
        for (int col = y; col <= y + wh; col++) {
            color clrU = img.get(col, row); // Unmarked Ballot
            color clrM = get(col, row); // Marked Ballot (screen)
            if (clrU == color(255) && clrM != clrU) {
                return true;
            }
        }
    }
    return false;
}
```

Instructor should point out the similarity of the structure of this nested for-loop to the problems from Section 7. Instructor then should make the point that in both cases, the code is processing a **2-D space** / **matrix** row-by-row, from top-left to bottom-right.

Section 9. Around the World in 24 Days

Essential Questions

How can software be used to study and solve problems in **Human Geography**?

Supporting Questions

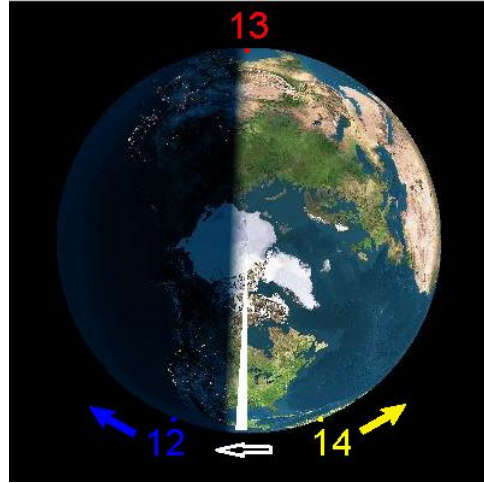
Why is there a need for the **International Dateline**?

How can you use the **sin** and **cos** functions to position objects around the edges of a circle?

Why does one need to treat the area around a **circle's 0°/360° boundary** differently from the rest of the circle?

How can one write a program that uses **the same code** to handle both a stationary and rotating Earth?

What types of discrepancies arise when one tries to use a **discrete model** to represent a **continuous system**? What are some mechanisms to deal with these?



Description

Students build a simulation of a rotating Earth in order to model the phenomenon described in Jules Verne's *Around the World in 80 Days* of an east-bound traveler who circumnavigates the world and experiences one day more than an observer remaining at the starting point. Three observers are placed on the surface, two of whom circumnavigate the globe in opposite directions: (a) an East-bound traveler (yellow); (b) a West-bound traveler (red); and (c) a stationary observer (white). After 24 days, the two travelers return to the starting point to rejoin the stationary observer. The east-bound traveler will have seen 25 sunrises, and the west-bound traveler will have seen 23 sunrises. The simulation sheds light on the reason for the establishment of the International Date Line.

Students begin by downloading 96 satellite images of Earth using the **View from Earth** website (<http://www.fourmilab.ch/cgi-bin/Earth>). These represent snapshots taken

over a 24-hour period spaced at 15-minute intervals. So that the surface of the Earth is half in shadow, the date chosen is either the Spring or Autumnal Equinox with Latitude = 90°N as if the satellite is positioned over the North Pole. Longitude is arbitrary, but 72°E was chosen so that Los Angeles is at the top of the simulation.

Students load the images into an array and implement the animation using a circular queue, which displays a stationary Earth with a moving *terminator* (the boundary line separating day and night). Students are already familiar with Processing's 2-D transformation operations from the Word Cloud and McClure units. They similarly implement rotation by translating the coordinate system origin to the center of the window, performing the rotation, and translating the origin back to the top left corner. As in the Word Cloud unit, students again bracket transformations between **pushMatrix()** and **popMatrix()** to independently rotate several objects simultaneously. The final rotation effect is that the Earth rotates and the **terminator** is stationary. A toggle variable controls rotation.

Earth, **Sunrise** and **Traveler** classes are implemented. A conditional expression for enabling a stationary traveler to detect a sunrise is initially expressed using normalized degree measurements to keep traveler and sunrise angles within the same range. The conditional expression is modified as more cases are accommodated, culminating with a solution for the edge condition at $0^{\circ}/360^{\circ}$. The final expression implements a **sector-point** intersection model.

Movement for travelers is implemented using a **speed** instance variable which is positive for traveling west; negative for traveling east; or 0 for no movement. Students discover that sunrise detection breaks down for moving travelers: at some point during

their circumnavigations - depending upon starting values for sunrise and traveler - the East traveler misses a sunrise and the West traveler clocks a double sunrise. An analogy to an escaping prisoner avoiding detection by a moving flashing searchlight is made. The problem is solved by narrowing or expanding the **sector** by the traveler's speed, and students consider the side-effects that occur when representing a **continuous** system with a **discrete** model.

Key Assignments

1. Students download 96 images of Earth and create an animation showing a complete 24-hour light cycle. The **animation** is that of a stationary Earth and a moving solar terminator.
2. Students implement an **option** for a stationary terminator and a rotating Earth.
3. Students implement an **Earth class** that does the bookkeeping involved in tracking and incrementing its angular position.
4. Students implement a **Traveler class** – although the first Traveler object created is stationary. Like the Earth, the Traveler keeps track of its angular position, and, when the earth rotates, changes its angular position at an identical rate to maintain the same location on the Earth. A Traveler object displays as a **number**, indicating the number of sunrises it has "seen".
5. Students implement a **Sunrise class** to keep track of the terminator's angular position.
6. Students implement a **normalize()** method that keeps all angles in the range $0 \leq \text{angle} < 360$.

7. Students implement **Traveler** methods **seeSunrise()** and **incSunrises()**. The **seeSunrise()** method revisits the boundary problem students first encountered in CodingBat's **parrotTrouble** problem.
8. Students implement **traveling** for a **Traveler** object, using a **speed** variable.
9. Students **correct** the **seeSunrise()** method to account for the longer or shorter sector width needed to detect a sunrise when a **Traveler** object moves east or west, respectively. The length of the sector's arc is adjusted by the **Traveler** object's **speed**.

Teaching Strategies

Modeling, counterexamples, guided discovery and experimentation.

To familiarize students with animation concepts, at the start of the unit, students write a program that animates Eadweard Muybridge's galloping horse photographs. The animation shows that all four feet of a horse are simultaneously off the ground at one point during a gallop cycle. Instructor stresses that this phenomenon is easier to grasp by seeing it in context within an animation rather than as a single still photograph.

Students revisit the edge/boundary problem they encountered when crafting the conditional expressions for CodingBat's **parrotTrouble** problem.

When implementing **display()** for the **Traveler** object, students are instructed to keep the number at the same position "height" above the Earth no matter the **Traveler**'s angular position. This involves explicitly calculating the left and top coordinates for the **text()** method (mathematically centering the number about its (x, y) position), then using the API's **textAlign()** method. When students asked why they had to bother calculating the left and top coordinates, only to comment out the code, the instructor tells them that

these are the same calculations the `textAlign()` method makes to center text horizontally and vertically.

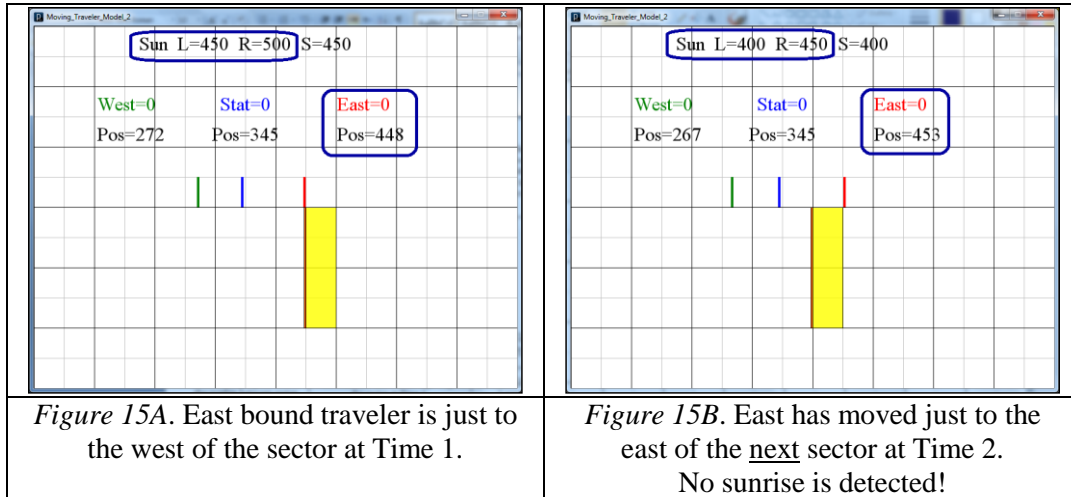


Figure 15A. East bound traveler is just to the west of the sector at Time 1.

Figure 15B. East has moved just to the east of the next sector at Time 2. No sunrise is detected!

To *model* the side-effect of using a **discrete model** with a **constant sector width** for traveling objects, the instructor distributes a *Moving Traveler* program that illustrates how, if the coordinates are inauspicious, an east-bound traveler can miss detection of a sunrise (**Figure 15**) and how a west-bound traveler can detect the same sunrise twice (**Figure 16**). The program shows the overlap between sector and traveler and records a sunrise event at these junctions.

The program also allows the user to widen or narrow the sector in order to illustrate how this adjustment can correct the detection errors (**Figures 17 and 18**). Note that the adjustment to the sector width occurs at the tail end (eastern edge), and the amount of the adjustment is the **distance** the traveler moves during an interval.

To assess understanding, students are instructed to write two paragraphs describing how each of the two errors occur and how each is corrected by an appropriate sector width adjustment.

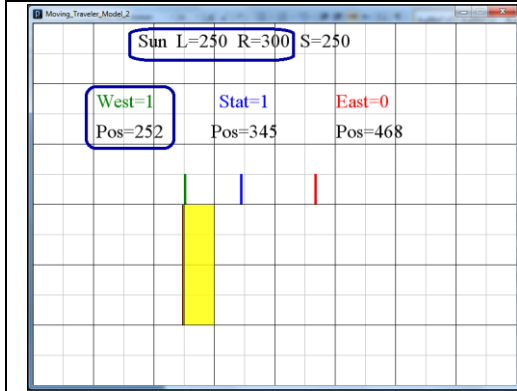


Figure 16A. West bound traveler is at the west edge of the sector at Time 1. A sunrise event is detected.

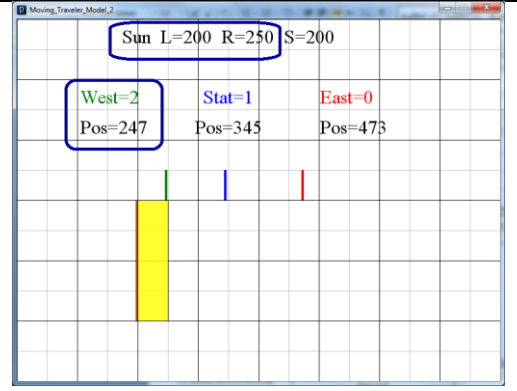


Figure 16B. West has moved to the east edge of the next sector at Time 2. A 2nd sunrise event is detected!

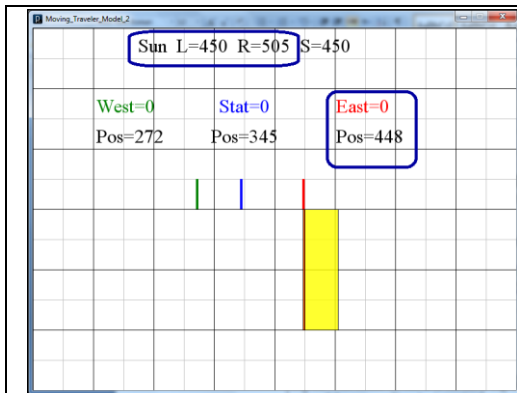


Figure 17A. East bound traveler is to the west of the widened sector at Time 1. No change in behavior.

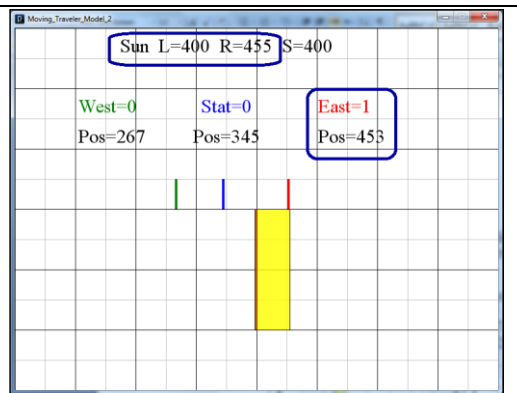


Figure 17B. East is within the widened sector at its east edge at Time 2. A sunrise event is now detected!

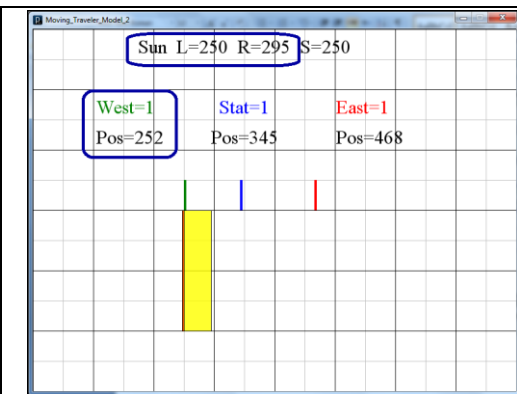


Figure 18A. West bound traveler is at the west edge of the narrowed sector at Time 1. A sunrise event is detected.

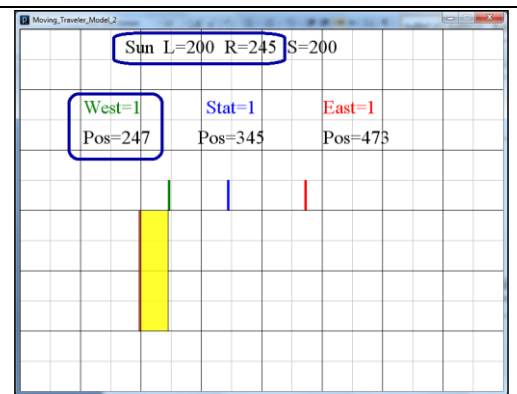


Figure 18B. West is now just outside the east edge of the next sector at Time 2, because the sector has been narrowed at this end. No 2nd sunrise event is detected!

Section 10. Galileo's Revolution and Astronomy

Essential Questions

How can a **software model of the Solar System** help us understand astronomical phenomena such as the phases of Venus, retrograde motion of Mars, and the infrequency of solar eclipses?

Supporting Questions

Why was the **discovery of the phases of Venus** so controversial *and* so significant historically?

How does **texture mapping** work?

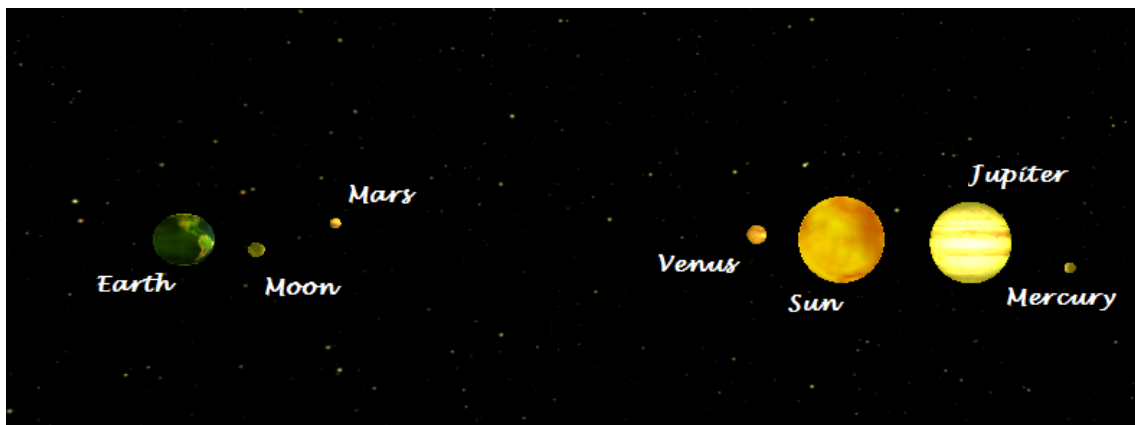
What are the math functions needed to describe an **elliptical orbit** in 2D?

What adjustment does one need to make to add a 3rd dimensional vertical movement to an orbit to implement **ecliptic tilt**?

How do you **position the camera** to view simulations of astronomical events?

Description

To provide background, the instructor guides students through the use of planetarium software, such as the proprietary Orion *Starry Night* or a free downloadable equivalent. Viewing the skies from any arbitrary location on Earth (such as one's home city) and using time-lapse settings, students view close-ups of Mercury and Venus as they go through their Moon-like phases, and observe Mars in retrograde over a period of roughly six months every two years.



Students then read and discuss Bertolt Brecht's play *Life of Galileo*, and write an essay in response to several possible prompts described in Key Assignments. Major

themes are (a) faith vs. doubt; (b) integrity vs. personal ambition; and (c) societal responsibility.

Students build a Copernican/heliocentric simulation of the inner solar system planets plus Jupiter, in order to view astronomical observations from various perspectives. The simulation replicates the **phases of Venus, Mars in retrograde**, and the **infrequency of solar eclipses**, both partial and total. Students create a **3-D** simulation and use **transformations** in 3 dimensions to position spherical objects representing the sun, planets and moon. Although the simulation cannot be to scale – because the planets would be too small to see – **distances** from the sun and planet **sizes** are consistent relative to one another. For similar reasons, elliptical **orbits** are approximated as circles. **Orbital tilts** for each planet and the moon are implemented relative to the ecliptic plane. The add-on *Processing* library **shapes3d** is used to add **texture mapped skin images** to the surface of the planetary spheres and to position the entire simulation within a surrounding static sphere whose skin is a **star map**. Students write methods to position and aim the **camera** in any direction in order to view the model not only from Earth, the planets and the sun, but from lateral and overhead views of the solar system. Students write methods to move the camera through a 3-D space. Finally, students write a second program to simulate the **Ptolemaic**/geocentric solar system model in order to understand what phenomena the model did and did not account for, and so help explain its nearly 2,000 year longevity.

Key Assignments

1. Students write an essay about the play *Life of Galileo* in response to several possible prompts:

a. A recurrent theme in the play is FAITH vs. DOUBT. In Scene 1, Brecht discusses this as it relates to science. In subsequent scenes, the references to faith/doubt are related to religion and the political/social order (the nobility ruling over the peasants). Discuss Brecht's ideas about faith and doubt as they come up during the course of the play.

b. Galileo uses his intelligence for three things: (a) trying to find a way to live well and be comfortable, (b) searching for scientific truth, and (c) trying to stay alive and out of trouble with the authorities. Discuss Galileo's use of cunning (shrewdness, cleverness, deception) to do all three as he negotiates the demands of the various authorities from the church and state (including the university and the city) that oppose him.

c. In Scene 7, the Little Monk argues that scientific truth should be abandoned for the sake of the peasants. Discuss his rationale (reasoning/reasons) for this stance and Galileo's vigorous response. Discuss the connection between this scene and the 2 lines at the end of Scene 12:

*Unhappy is the land that has no hero.
Unhappy is the land that needs a hero.*

2. Students write a 3-D program that places a yellow sphere (**sun**) in the center of the window using transformations.
3. Students use 3-D transformations to implement a **planet** revolving around the sun in a circular orbit.
4. Students implement the 4 terrestrial planets **Mercury**, **Venus**, **Earth** and **Mars**, and the gas giant **Jupiter** using diameters and orbital radii relative to each other. The

planets will be many times larger than scale, and all planets will orbit in the same ecliptic plane.

5. Students implement **inclinations** for all planets except Earth, i.e. their orbits are tilted relative to the ecliptic plane.
6. Students implement the Earth's **Moon** revolving around the Earth (diameter relative to Earth, but orbital radius not).
7. Students implement **rotation** of planets, sun and moon about their axes.
8. Students implement Earth's **axial tilt** (23°).
9. Students collect texture mapped images of the planets, sun and moon, and implement **texture mapping** so that they are drawn with realistic looking surfaces.
10. Students collect an appropriate texture mapped image of the **stars** to wrap on a super-large sphere that encloses the entire simulation.
11. Students write methods to position the **camera above** the ecliptic plane and to its **frontal side**.
12. Students implement methods to **position** the camera on each of the **moving planets** pointing to the sun.
13. Students implement a method to view **Mars** from Earth.
14. Students implement a **second** method to view Mars from Earth, but **fixing** the camera on a point far beyond Mars in order to view Mars' motion in retrograde.
15. Students implement methods to control the **speed** and **direction** of the simulation so that the phenomena of solar eclipses can be viewed.
16. Students implement methods for **moving** the **camera** through space, rotating left and right, up and down, and moving forwards and backwards.

Teaching Strategies

Counterexamples, guided discovery and experimentation.

The heliocentric (Copernican) model of the solar system allows students to view the complete cycle of Venus's phases as seen from Earth. The companion program that simulates the geocentric (Ptolemaic) epicycle model demonstrates the impossibility of observing both a completely dark and fully lit Venus. When students subsequently read Bertolt Brecht's play *Life of Galileo*, they learn that it was this single celestial observation – made possible by the newly invented telescope – that was a pivotal point in the further erosion of papal power, already weakened by the Reformation. Although Galileo himself was put under house arrest for the remainder of his life, his discoveries further loosened the Church's capacity to impede the pace of science during the Renaissance.

On the pedagogic level, this unit taken in its entirety extends and clarifies students' understanding of events 400 years old. It does so through the use of student-written software that is able to clarify the true nature of a celestial phenomenon, one whose logical implications had huge historic, social and political ramifications. Because of its many facets, the unit contains multiple points at which students can make engaging connections.

If placing problems in real-world contexts answers the question "How can I use this knowledge?", providing historical and social contexts allows students to ask "What is the human/societal impact?" At this point, students step back, gain perspective and look at the big picture to observe how their work can be used and misused. This unit brings up powerful ethical questions for scientists and engineers – such as the ethical role of the scientist – that dwarf such standard fare concerns as intellectual property.

For implementing the movement of the camera to view the simulation from different perspectives, students use "dummy" **Planet** class variables named **eyePlanet** and **centerPlanet** to hold the position and direction of the camera, respectively. In this way, students are introduced to the use of references variables that can hold values referring to objects. Implementing a way to observe Mars in retrograde from Earth is done in two ways. One is to simply assign Earth to **eyePlanet** and Mars to **centerPlanet**, the effect of which is to keep Mars statically fixed in the center of the viewport, while the star background moves in reverse direction relative to Mars. The second is to point the camera at Mars, then project a **vector** far beyond Mars (say 100 times the Earth-Mars distance), and capture that position to store in **centerPlanet**. The effect of this second strategy is to fix the camera direction on a static star background, allowing one to observe Mars move and reverse directions. The calculation of this distant position is done with **vectors** (see below).

To implement keystroke-driven camera movement for **left** and **right rotation**, students first consider rotation around an axis parallel to the y-coordinate axis (students will later implement rotation around an axis in any direction using matrices). Students are introduced to the *arctan* function to derive the angle that the camera vector projects onto the X-Z coordinate plane. Students learn, however, that the angles returned from this function are **doubly ambiguous** because tangent values are the same for quadrant pairs I and III, and II and IV. Therefore in order to map the correct angle, students learn that they need to also utilize the signs of the **cosine** and **sine** values in the calculation. Students use an **Excel** spreadsheet to quickly see that the **arctan** returns values in the range $-\pi/2$ to $\pi/2$ (-90° to 90°). They then expand this spreadsheet to show

corresponding **sine**, **cosine** and **tangent** values for all 360° , and the (differing) angle values returned by **arctan**. Using the data from each quadrant, students write a camera method that correctly maps degrees based on **arctan**, **cosine** and **sine** values as angular position is incremented or decremented.

To implement keystroke-driven camera movement **forwards** and **backwards**, students need to project component vectors onto all three axes, the same technique used to calculate a distant point when viewing the retrograde motion of Mars. Students will recognize that calculating the magnitude of these component vectors is similar to calculating the slope between two points. To calculate the slope between any two points, students know that it doesn't matter which point is subtracted from the other as long as one is consistent, i.e. $\Delta x = x^2 - x^1$ and $\Delta y = y^2 - y^1$ OR $\Delta x = x^1 - x^2$ and $\Delta y = y^1 - y^2$. This is because the signs of the two differences cancel each other when the two are positioned as a ratio: $\Delta y / \Delta x$. For example, a line defined by the origin and a point in quadrants I or III yields two positive differences or two negative differences, resulting in a positive slope. A line similarly defined in quadrants II and IV yield one positive and one negative difference, resulting in a negative slope.

With the component vectors, however, the order in which one subtracts the points is critical because (a) we are calculating the effective contribution of each vector separately, and (b) a vector has not only magnitude but direction. The instructor therefore has students calculate the extrapolated point using both possible orientations, so that students discover that the correct order is **destination** value minus **source** value – or for the camera variables, **centerX** minus **eyeX**, etc. Students are asked to write a short paragraph explaining how to do these calculations.

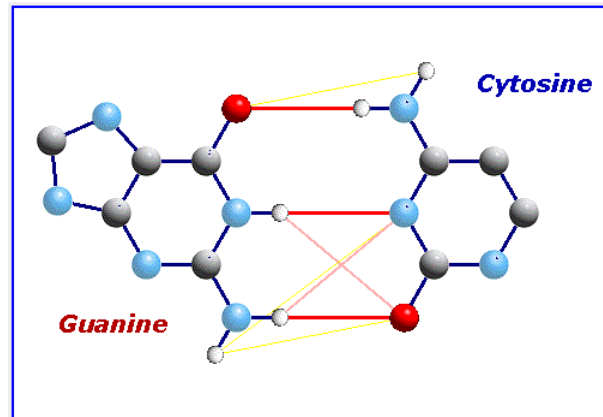
Section 11. Molecular Modeling and DNA

Essential Questions

How can we use Molecular Modeling software to explain how the opposing strands of a DNA double helix are **structurally** held together?

How can Molecular Modeling software help explain how the hydrogen bonds between the DNA bases provide the mechanism by which **genes** are faithfully **replicated**?

How can we use Molecular Modeling software to help us understand how **point mutations** arise?



Supporting Questions

What kinds of social and emotional obstacles might hinder **scientific collaboration**?

Which **geometry concepts** are needed to draw the DNA bases?

How can you use **sine** and **cosine** functions to determine the positions of atoms at irregular angle positions?

What are the advantages/disadvantages in keeping many separate, but corresponding, lists of properties rather than a list of objects, each of which contains all the properties?

What are the advantages in using **superclasses** and **subclasses**?

How can the **additive trigonometry identities** be used to implement **rotation**?

What are advantages/disadvantages to **implementing** translation and rotation using **math**, as opposed to the **transformation methods**?

Is there a way to implement **reflection** using Processing's transformation methods?

How do you program the GUI for **selection** and **movement** of objects using the **mouse**?

How do you implement **multiple selection**?

Description

Students build a 2-D molecular modeling program to examine the hydrogen bonding, between purine and pyrimidine bases, that holds the two anti-parallel strands of the DNA double helix together.

The unit begins with students familiarizing themselves with the freely available 3-D molecular modeling program **MolSoft ICM- Browser**, and exploring ways to configure the four DNA bases Adenosine, Guanine, Cytosine and Thymine within the program. The molecule files will be downloaded from the **NYU Library of 3-D**

Molecular Structures. Students then proceed to calculate the angles of the pyrimidine (hexagon) and imidazole (pentagon) rings and use **BYOB** (UC Berkeley) to correctly position each base's ring and functional group atoms. During this process, students design their code to reflect the biochemical nomenclature of the molecule's major features. They also abstract shared features of the molecules into common methods for building pyrimidine rings, imidazole rings, and adding functional groups at any of the 6 pyrimidine atoms.

Once they are familiar with the structure of the 4 bases, students go about building the 2-D molecular modeling program in **Processing**. They use the **sine** and **cosine** functions to create a method to position atoms using polar coordinates. They use getter methods to encapsulate the x- and y-coordinates of each atom. These methods become the central repository for calculating translated, rotated, and mirrored coordinates for each atom. Students use a geometry proof to find the additive angle formulas for sine and cosine. These are used to derive the rotation formulas for x- and y-coordinates, which are then implemented in the program.

Students study the chemistry of polar bonds and hydrogen bonds. They write methods for deciding which hydrogen atoms are **electropositive** and which nitrogen and oxygen atoms are **electronegative**. The optimal distance for intermolecular hydrogen bonds is indicated by color and line thickness. Students program the GUI for object selection with mouse, control key and lassoing. Move, rotate, and mirror-image actions are driven by mouse events.

At unit's end students *use their programs* to display normal A-T and G-C pairings. They also perform **predictive** tasks, i.e. find configurations for rare A-C and G-T pairings, which represent point mutation situations.

To anchor this project in a social setting, students study the **BBC** film *Double Helix*, which relates the little known story of the discovery of the DNA double helix by Watson and Crick, who used the x-ray diffraction data of the biophysicist Rosalind Franklin for building their model. Although her data was crucial to their calculations, which won them the Nobel Prize, they did not acknowledge her contribution until long after her death. Students write an essay about the film in response to the prompt described in Key Assignments.

Key Assignments

1. Lesson 1: Students study the **geometry** of **hexagons** and **pentagons** in order to write a BYOB program that correctly draws the angles and bond lengths for the 4 DNA base *in pyrimidine and purine ring nomenclature atom order*.
2. Lesson 2: Students organize and simplify their BYOB programs by using iteration and sequestering duplicate code into parameterized methods that have meaningful biological and/or chemical significance, e.g. **drawPyrimidine()**, **drawImidazole()**, **addAmine()**.
3. Lesson 3: Students write a *Processing* program – organized along the principles learned in the BYOB lessons – that correctly displays classes for the 4 DNA bases at the origin (translated to the center of the drawing window).
4. Lesson 4: Students modify their *Processing* program to implement polymorphism, using the common parent class Molecule.

5. Lesson 4: Students implement **translation**, so that the user can move/reposition the DNA bases. Students program the GUI to use the **Arrow** keys for gross movement, and the **Ctrl + Arrow key combinations** for finer movement.
6. Lesson 4: Students complete a **geometric proof** for the **Additive Trigonometric Identities**, and use these to derive expressions for implementing 2-D **rotation**.
7. Lesson 4: Students implement **rotation**. The **PageDown** and **PageUp** keys are used for rotation clockwise and counter-clockwise, respectively, with the option of using them in combination with the **Control key** for finer movement.
8. Lesson 4: Students implement **reflection**, so that users can flip molecules horizontally. The 'M' key (for "mirror") will toggle the state of the reflected molecule.
9. Lesson 5: Students implement **mouse events** in the GUI. Students implement single and multiple mouse **selection** using (a) **clicking**; (b) clicking in combination with the **Shift** and **Control** keys; and (c) **lassoing**. They implement the **mouse wheel** for rotation. They implement **drag-and-drop**.
10. Lesson 6: Students implement **electropositive** and **electronegative** chemical properties into the bases, so that they can form – and display – **intermolecular hydrogen bonds**.
11. Lesson 6: Students use the program to show the normal hydrogen bonds between A-T and G-C, and to discover at least one configuration each for point-mutation-causing hydrogen bonds between A-C and G-T.
12. Lesson 7: Students write an essay about the film *Double Helix* (a.k.a. *Life Story*) in response to the prompt:

The film *Double Helix* takes place in the years 1951-1953. At the beginning of the story, Dr. Rosalind Franklin returns from Paris to London to take a research position at King's College. There, as one of the few women researchers, she experiences first-hand the effects of a work place imbued with sexist attitudes and where men have traditionally been in charge. One of the effects of this suffocating environment is that she feels isolated in her work.

a) Think of scenes where Franklin feels, or is in fact, isolated or marginalized (not treated seriously or equally). Contrast these scenes with those where Franklin finds ways out of her isolation to form connections with other supportive characters.

(b) In the film, Franklin's character resists a system that puts her at a disadvantage. Think of scenes where she pushes back and how effective her efforts are in terms of successfully getting the changes (in behavior, or legally) that she might have wanted. Analyze these actions/efforts and discuss reasons why they might have been either effective or ineffective.

(c) At the end of the film, Bragg tells Franklin: "This race, this winning and losing, it's not the way I was taught to do science." This remark speaks to the main characters' motivations for doing science. Compare and contrast the motivations of Crick, Watson, Franklin and Wilkins. Remember that the characters' motivations refer not just to abstract issues about the pursuit of science, but equally – if not more – about what they enjoy about their day-to-day work

Teaching Strategies

Counterexamples, guided discovery, experimentation and **CONNECTIONS**.

One central strategy is the use of cross-curricular concepts, especially geometry and biology, to write a program that has both descriptive and predictive value vis-à-vis the bonding between the anti-parallel strands of DNA. Students must connect concepts from other disciplines in order to write an accurate 2-D molecular modeling program. Students will also see that CS is an engineering discipline, one that can be used to solve problems in other academic fields.

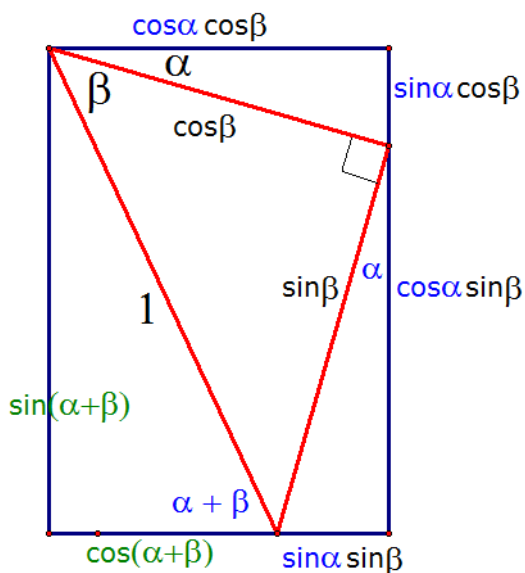


Figure 19. Derivation of Additive Trigonometric Identities

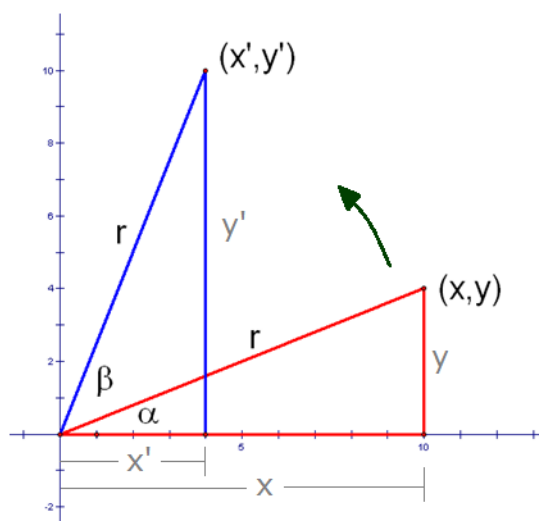


Figure 20. Derivation of Formulas for Calculating New Coordinates after Rotation about the Origin

One powerful example of making **CONNECTIONS** to other disciplines is the section where students implement molecule rotation. Changing the coordinates of a molecule's atom is an alternate means for rotating objects as opposed to using *Processing's* transformations. To do so, students learn – through guided discovery – a **geometric proof** for the Additive Trigonometric Identities, $\sin(\alpha + \beta)$ and $\cos(\alpha + \beta)$. Because the prerequisite for the course is proficiency in Algebra 1, students will have already taken concurrently the lion's share of a Geometry course – or an Algebra 2 course – by the time this final unit is encountered near the end of the school year. Moreover, in the *Around the World* unit, students have already been learned about and used the **sin** and **cos** functions. The proof involves nothing more than using the definitions of **sin** and **cos** on a specific right triangle, equating opposite sides, then isolating $\sin(\alpha + \beta)$ and $\cos(\alpha + \beta)$ (**Figure 19**). Once students complete the proof, they **use** the identities in another geometrically constructed diagram that sets up the theory for rotating a single point about the origin (**Figure 20**). Students can now derive the formulas for the x- and y-

coordinates (x_2, y_2) for a point (x_1, y_1) that is rotated about the origin by substituting in the additive trig identities, then substituting in the definitions of **sin** and **cos** ($x' = x\cos\beta - y\sin\beta$ and $y' = y\cos\beta + x\sin\beta$). Finally, students implement these simple formulas in their program and visually confirm that the molecules they've constructed rotate when the programs are run. Students are thus able to see that theoretical math does indeed have concrete applications.

REFERENCES

- Abu-Rabia, S., & Sanitsky, E. (2010, June). Advantages of Bilinguals Over Monolinguals in Learning a Third Language. *Bilingual Research Journal*, 33(2), 173 - 199.
doi:10.1080/15235882.2010.502797
- Altshuler, D., Pollara, V., Cowles, C., Van Etten, W., Baldwin, J., Linton, L., & Lander, E. (2000, Sep 28). An SNP map of the human genome generated by reduced representation shotgun sequencing. *Nature*, 407(6803), 513-516. Retrieved from www.nature.com
- Alvarado, C., & Dodds, Z. (2010, March 10-13). Women in CS: An Evaluation of Three Promising Practices. *SIGSCE '10 Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, 57-61.
doi:10.1145/1734263.1734281
- Amirtha, t. (2014, April 21). *This Is Your Brain On Code, According To Functional MRI Imaging*. Retrieved from Fast Company:
<http://www.fastcompany.com/3029364/this-is-your-brain-on-code-according-to-functional-mri-imaging>
- Applin, A. G. (2001). Second Language Acquisition and CS1: Is * == ** ? *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, 174-178. doi:10.1145/366413.364579
- Arditti, A., & Skirble, R. (2010, April 27). *In Schools, a Way to Keep Language From Getting in the Way of Science*. Retrieved from Learning English, Voice of America: <http://learningenglish.voanews.com/content/in-schools-a-way-to-keep-language-from-getting-in-the-way-of-science-92244839/117660.html>

- Azemi, A., & D'Imperio, N. (2011). New Approach to Teaching an Introductory Computer Science Course. *ASEE/IEEE Frontiers in Education Conference* (pp. S2G-1 to S2G-6). Rapid City, South Dakota: IEEE.
doi:10.1109/FIE.2011.6142988
- Bahier, D. J. (2005). *Teaching Secondary and Middle School Mathematics*. Boston: Pearson: Allyn and Bacon.
- Bailey, N., Madden, C., & Krashen, S. D. (1974). Is there a 'natural sequence' in adult second language learning? *Language Learning*, 24(2), 235–243. doi: 10.1111/j.1467-1770.1974.tb00505.x
- Balsim, I., & Feder, E. (2008). The Synthesis of Mathematical Foundations with Real World Applications in Computer Science Education. *Proceedings of the 2008 International Conference on Frontiers in Education: Computer Science and Computer Engineering, FECS*, 154-160. Retrieved from <http://dblp.org/rec/html/conf/fecs/BalsimF08>
- Beck, R., Burg, J., Heines, J., & Manaris, B. (2011). Computing and Music: A Spectrum of Sound. *SIGCSE'11 - Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, p 7-8. doi:10.1145/1953163.1953171
- Bell, T., Witten, I. H., & Fellows, M. (2006). *Computer Science Unplugged: An enrichment and extension programme for primary-aged children*. Christchurch: Creative Commons.
- Bennedsen, J., & Caspersen, M. E. (2007, June). Failure Rates in Introductory Programming. *Inroads - The SIGCSE Bulletin*, 39(2), 32-36.
doi:10.1145/1272848.1272879

- Bennedsen, J., & Caspersen, M. E. (2008). Abstraction Ability as an Indicator of Success for Learning Computing Science? *International Computing Education Research (ICER'08)* (pp. 15-25). Sydney, Australia: Association for Computing Machinery (ACM). doi:10.1145/1404520.1404523
- Bennedsen, J., & Caspersen, M. E. (2012, June). Persistence of elementary programming skills. *Computer Science Education*, 22(2), 81-107.
doi:10.1080/08993408.2012.692911
- Berent, G. P., Kelly, R. R., & Porter, J. E. (2008, June). Deaf Learners' Knowledge of English Universal Quantifiers. *Language Learning*, 58(2), 401-437.
doi:10.1111/j.1467-9922.2008.00445.x
- Bialystok, E. (1978). A theoretical model of second language learning. *Language Learning*, 28(1), 69-83. doi:10.1111/j.1467-1770.1978.tb00305.x
- Black, E. (2001). *IBM and the Holocaust*. Dialog Press. Retrieved from <http://www.ibmandtheholocaust.com/>
- Bloch, S. (2014, Oct 14). SIGSCE Listserv (Member Forum). *Communication Posting*.
- Bloomfield, L. (1933). *Language*. Chicago: University of Chicago Press.
- Blum, L., Frieze, C., Hazzan, O., & Dias, M. (2006). A Cultural Perspective on Gender Diversity in Computing. (*long version of a paper presented at SIGCSE 2006*). Retrieved from <http://www.cs.cmu.edu/~lblum/PAPERS/CrossingCultures.pdf>
- Blum, L., Frieze, C., Hazzan, O., & Dias, M. B. (2006, March 1-5). Culture and Environment as Determinants of Women's Participation in Computing: Revealing the "Women-CS Fit". *SIGCSE '06 Proceedings of the 41st ACM Technical*

- Symposium on Computer Science Education*, 22-26.
- doi:10.1145/1121341.1121351
- Brahier, D. J. (2005). *Teaching Secondary and Middle School Mathematics* (2 ed.). Boston, MA: Pearson Education Inc.
- Bransford, J., Sherwood, R., Vye, N., & Rieser, J. (1986, October). Teaching Thinking and Problem Solving: Research Foundations. *American Psychologist*, 41(10), 1078-1089. doi:10.1037/0003-066X.41.10.1078
- Brought, G., Wahls, T., & Eby, L. M. (2011, Feb). The Case for Pair Programming in the Computer Science Classroom. *ACM Transactions on Computing Education (TOCE)*, 11(1), Article No. 2. doi:1921607.1921609
- Bringsjord, S., & Ferrucci, D. A. (2000). *Artificial Intelligence and Literary Creativity: Inside the Mind of BRUTUS, a Storytelling Machine*. Mahwah, NJ: Lawrence Erlbaum Associates.
- Brown, R., & Kumar, A. (2013). The Scientific Method: Reality or Myth. *Journal of College Science Teaching*, 42(4), pp. 10-11. Retrieved from <http://www.jstor.org/stable/43631913>
- Caristi, J., Sloan, J., Barr, V., & Stahlberg, E. (2011). Starting a Computational Science Program. *SIGCSE'11 - Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, p 3-4. doi:10.1145/1953163.1953167
- Carter, D. (2007). *AP Computer Science Teacher's Guide*. New York: College Board.
- Carter, D. (2012, March). Highlights of CS Survey, State-by-State Analysis. *CSTA Voice*, Vol. 8(1), pp. 4-5. Retrieved from

- http://csta.acm.org/Communications/sub/CSTAVoice_Files/csta_voice_03_2012.pdf
- Carter, L. (2006, March 1-5). Why Students with an Apparent Aptitude for Computer Science Don't Choose to Major in Computer Science. *SIGCSE '06*. Houston, Texas. doi:10.1145/1124706.1121352
- Carter, L. (2007). Work in progress - Introduction to computers: An interdisciplinary approach. *Proceedings - Frontiers in Education Conference, FIE*, S1J1-S1J2. doi:10.1109/FIE.2007.4417820
- Chesterfield, R., & Chesterfield, K. (1985). Natural Order in Children's Use of Second Language Learning Strategies. *Applied Linguistics*, 6(1), 45-59. doi:10.1093/applin/6.1.45
- Chomsky, C. (1969). *The Acquisition of Syntax in Children from 5 to 10*. Cambridge, MA: MIT Press.
- Chomsky, N. (1959). On Certain Formal Properties of Grammars. *Information and Control*, 2, 137-167. doi:10.1016/S0019-9958(59)90362-6
- Chomsky, N. A. (1957). *Syntactic Structures*. The Hague/Paris: Mouton.
- College Board. (2015). *AP Data: National Report*. Retrieved from College Board: <https://research.collegeboard.org/programs/ap/data>
- Computer Science 131. Computing for Poets*. (n.d.). Retrieved from Wheaton College Catalog, Computer Science: http://wheatoncollege.edu/catalog/comp_131/
- Conway, C. M., Karpicke, J., & Pisoni, D. B. (2007). Contribution of Implicit Sequence Learning to Spoken Language Processing: Some Preliminary Findings With

- Hearing Adults. *Journal of Deaf Studies and Deaf Education*, 12(3). Retrieved from <http://www.jstor.org/stable/42658884>
- Corder, S. P. (1967, November). The Significance of Learner's Errors. *International Review of Applied Linguistics in Language Teaching*, 5(4), 161-170.
- CSTA Standards Task Force. (2011). *CSTA K-12 Computer Science Standards*. New York: CSTA / ACM. Retrieved from http://csta.acm.org/Curriculum/sub/CurrFiles/CSTA_K-12_CSS.pdf
- Cuny, J. (2011, May). Transforming Computer Science Education in High Schools. *Computer*, 44(6), 107-109. doi:10.1109/MC.2011.191
- Daly, T. (2009, Fall). Using introductory programming tools to teach programming concepts: A literature review. *The Journal for Computing Teachers*, 1-6. Retrieved from <http://www.iste.org/jct>
- Denner, J., Werner, L., Sampe, S., & Ortiz, E. (2014). Pair Programming: Under What conditions Is It Advantageous for Middle School Students? *Journal of Research on Technology in Education*, 46(3), 277-296. Retrieved from <http://ejournals.ebsco.com/direct.asp?ArticleID=4CE28DBF1FBE0968C7D0>
- DNA Learning Center. (n.d.). *The public Human Genome Project: mapping the genome, sequencing, and reassembly. 3D animation*. Retrieved from <http://www.dnalc.org/view/15477-The-public-Human-Genome-Project-mapping-the-genome-sequencing-and-reassembly-3D-animation-.html>
- Dodds, Z., & Erlinger, M. (2013, July 1-3). MyCS: Building a Middle-years CS Curriculum. *ITiCSE'13*, 330. doi:10.1145/2462476.2465611

- Doom, T., Raymer, M., Krane, D., & Garcia, O. (2003, August). Crossing the Interdisciplinary Barrier: A Baccalaureate Computer Science Option in Bioinformatics. *IEEE Transactions on Education*, Vol. 46(No. 3). doi:10.1109/TE.2003.814593
- Dulay, H. C., & Burt, M. K. (1973). Should We Teach Children Syntax? *Language Learning*, 23(2), 245-258. doi:10.1111/j.1467-1770.1973.tb00659.x
- Eglash, R. (2003). *Transformational geometry and iteration in cornrow hairstyles*. Retrieved from Culturally Situated Design Tools (Rensselaer Polytechnic Institute): http://csdt.rpi.edu/african/CORNROW_CURVES/index.htm
- Fellows, M., & Parberry, I. (1993, January). SIGACT Trying to Get Children Excited About CS. *Computing Research News*, 5(1), 7. Retrieved from <https://larc.unt.edu/ian/pubs/crn1993.pdf>
- Fendel, D., Resek, D., Alper, L., & Fraser, S. (2008). *Interactive Mathematics Program*. Key Curriculum Press. Retrieved from http://www.mathimp.org/general_info/intro.html
- Frieze, C., Quesenberry, J. L., Kemp, E., & Velazquez, A. (2012, July 31). Diversity or Difference? New Research Supports the Case for a Cultural Perspective on Gender Diversity in Computing. *Journal of Science Education and Technology*, 21(4), 423–439. Retrieved from <http://www.jstor.org/stable/41674471>
- Fulton, B. S. (2005). *Managing the Math Classroom for Maximum Success*. Milville, CA: Teacherto Teacher Press. Retrieved from http://www.tttpress.com/uploads/2/0/4/2/20424731/managing_the_math_class.pdf

- Garner, S., Haden, P., & Robins, A. (2005). My Program is Correct But it Doesn't Run: A Preliminary Investigation of Novice Programmers' Problems. *Proceedings of the 7th Australian Conference on Computing Education, ACE'05*, 42, 173-180.
Retrieved from <http://dl.acm.org/citation.cfm?id=1082446>
- Goldweber, M., Barr, J., Clear, T., Davoli, R., Mann, S., Patitsas, E., & Portnoff, S. (2013, March). A Framework for Enhancing the Social Good in Computing Education: A Values Approach. *ACM Inroads*, 4(1).
doi:10.1145/2432596.2432616
- Goldweber, M., Little, J., Cross, G., Davoli, R., Riedesel, C., von Kinsky, B., & Walker, H. (2011). Enhancing the Social Issues Components in our Computing Curriculum Computing for the Social Good. *CMIInroads, Vol. 2*(No. 1).
doi:10.1145/1929887.1929907
- Goode, J., Chapman, G., & Margolis, J. (2012, June). Beyond Curriculum: The Exploring Computer Science Program. *ACM Inroads*, 3(2), pp. 47-53.
doi:10.1145/2189835.2189851
- Gray, J. (2013, May). CS Principles Update. *CSTA Voice*, 9(2). Retrieved from http://csta.acm.org/Communications/sub/CSTAVoice_Files/csta_voice_05_2013.pdf
- Hallet, R., & Venegas, K. (2011, Spring). Is Increased Access Enough? Advanced Placement Courses, Quality, and Success in Low-Income Urban Schools. *Journal For The Education Of The Gifted*, 34(3), 468-487.
doi:10.1177/016235321103400305

- Holley, K. A. (2009, July 15). Understanding Interdisciplinary Challenges and Opportunities. *ASHE Higher Education Report, Vol. 35(2)*, 1-131.
- Hour of Code*. (2015). Retrieved from Hour of Code: <https://hourofcode.com/us#>
- Hurson, A., & Sedigh, S. (2010). Transforming the Instruction of Introductory Computing to Engineering Students. *2010 IEEE Transforming Engineering Education: Creating Interdisciplinary Skills for Complex Global Environments*. doi:10.1109/TEE.2010.5508834
- Impagliazzo, J., & McGettrick, A. (2007, October 10-13). New Models for Computing Education. *37th ASEE/IEEE Frontiers in Education Conference*, F3H-11 - FSH-16. doi:10.1109/FIE.2007.4417991
- Ivanitskaya, L., Clark, D., Montgomery, G., & Primeau, R. (2002, Winter). Interdisciplinary Learning, Process and Outcomes. *Innovative Higher Education, Vol. 27(No. 2)*, 95-111. Retrieved from <http://link.springer.com/article/10.1023/A%3A1021105309984>
- Kane, J., & Mertz, J. (2012, January). Debunking Myths about Gender and Mathematics Performance. *Notices of the American Mathematical Society, Vol. 59(No. 1)*, 10-21. doi:10.1090/S1088-9477-2012-00790-4
- Kelleher, C., & Pausch, R. (2005, June). Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Computing Surveys, 37(2)*, 83-137. doi:10.1145/1089733.1089734
- Kelleher, C., Pausch, R., & Kiesler, S. (2007, April 28-May 3). Storytelling Alice Motivates Middle School Girls to Learn Computer Programming. *CHI 2007*

Proceedings • Programming By & With End-Users.

doi:10.1145/1240624.1240844

Klein, J. T. (2005, Summer/Fall). Integrative Learning and Interdisciplinary Studies. *Peer Review*, 9-10. Retrieved from
http://www.academia.edu/755632/Integrative_learning_and_interdisciplinary_studies

Kliebard, H. (2004). In H. Kliebard, *The Struggle for the American Curriculum 1893-1958*. New York and London: RoutledgeFalmer.

Konidari, E., & Louridas, P. (2010, March). When Students are not Programmers. *ACM Inroads*, 1(1), 55-60. doi:10.1145/1721933.1721952

Krashen, S. (1981). *Second Language Acquisition and Second Language Learning*. Oxford: Pergamon Press, Inc.

Krashen, S. (1982). *Principles and Practice in Second Language Acquisition*. Oxford: Pergamon Press Inc.

Krashen, S. (1985). *The Input Hypothesis: Issues and Implications*. Harlow: Longman.

Krashen, S. (1998). Comprehensible Output? *System*, 26(2), 175-182.
doi:10.1016/S0346-251X(98)00002-5

Krashen, S. (2004). *The Power of Reading: Insights from the Research* (2nd ed.). Libraries Unlimited.

Kuhn, T. S. (1996). Anomaly and the Emergence of Scientific Discoveries. In T. S. Kuhn, *The Structure of Scientific Revolutions* (3rd ed., p. 59). Chicago and London: The University of Chicago Press.

- Kumar, D. (2013, September). The Changing, Not Evolving Pedagogy of CS1. *ACM Inroads*, 4(3), 36-37. doi:10.1145/2505990.2505994
- Kummerfeld, S. K., & Kay, J. (2003). The neglected battle fields of Syntax Errors. *Proceedings of the fifth Australasian Conference on Computing Education*, Adelaide, Australia: Australian Computer Society, Inc; 03, 20. Retrieved from <http://dl.acm.org/citation.cfm?id=858416>
- Kuntze, M. (2004). *Literacy acquisition and deaf children: A study of the interaction between ASL and written English*. ProQuest Dissertations and Theses. Retrieved from <https://searchworks.stanford.edu/view/5685734>
- Kurmas, Z. (2011, May 17). *The Deep End of the Pool*. Retrieved from Atomic Spin: Atomic Object's blog on Software Design & Development: <http://spin.atomicobject.com/2011/05/17/the-deep-end-of-the-pool/>
- Lahtinen, E., AlaMutka, K., & Järvinen, H.-M. (2005, June 27–29). A Study of the Difficulties of Novice Programmers. *Proceedings of the 10th annual ITiCSE conference*, 14-18. doi:10.1145/1151954.1067453
- Leinweber, L. (n.d.). *WATOR. Predator-Prey Simulation. a java applet*. Retrieved from Larry's Cerebral Snack Bar: <http://www.leinweb.com/snackbar/wator/>
- Linn, M. C., & Dalbey, J. (1985). Cognitive Consequences of Programming Instruction: Instruction, Access and Ability. *Educational Psychologist*, 20(4), 191-206. doi:10.1207/s15326985ep2004_4
- List of African-American inventors and scientists*. (2016, April 5). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/List_of_African-American_inventors_and_scientists

- Lyster, R., & Ranta, L. (1997). Corrective Feedback and Learner Uptake. *Studies in Second Language Acquisition*, 19(1), 37-66. Retrieved from http://people.mcgill.ca/files/roy.lyster/Lyster_Ranta1997_SSLA.pdf
- Margolis, J., Estrella, R., Goode, J., Holme, J. J., & Nao, K. (2008). *Stuck in the Shallow End: Education, Race, and Computing*. Cambridge, MA: MIT Press.
- Margolis, J., Fisher, A., & Miller, F. (1999/2000, Winter). Caring about connections: gender and computing. *Technology and Society Magazine, IEEE*, Vol. 18(No. 4), 13-20. doi:10.1109/44.808844
- Margolis, J., Ryoo, J., Sandoval, C., Lee, C., Goode, J., & Chapman, G. (2012, December). Beyond Access: Broadening Participation in High School Computer Science. *ACM Inroads*, 3(4), 72-78. doi:10.1145/2381083.2381102
- Martin, B., & Hearne, J. D. (1990, Jan). Transfer of Learning and Computer Programming. *Educational Techonology*, 41-44. Retrieved from <http://eric.ed.gov/?id=EJ405758>
- Mattern, K., Shaw, E., & Ewing, M. (2011). *Advance Placement Exam Participation: Is AP Exam Participation and Performance Related to Choice of College Major?* Retrieved from <http://professionals.collegeboard.com/profdownload/pdf/RR2011-6.pdf>
- Mayer, R. E., Dyck, J. L., & Vilberg, W. (1986, July). Learning to Program and Learning to Think: What's the Connection? *Communications of the ACM*, 29(7), 605-610. doi:10.1145/6138.6142

- McGettrick, A., Boyle, R., Ibbett, R., Lloyd, J., Lovegrove, G., & Mander, K. (2005). Grand Challenges in Computing: Education—A Summary. *The Computer Journal*, 48(1), 42-48. doi:10.1093/comjnl/bxh064
- Menze, B. H., & Ur, J. (2012, March 19). Mapping patterns of long-term settlement in Northern Mesopotamia at a large scale. *PNAS Plus*. doi:10.1073/pnas.1115472109
- Monti, M. M., Parsons, L. M., & Osherson, D. N. (2012). Thought Beyond Language: Neural Dissociation of Algebra and Natural Language. *Association for Psychological Science*, 23(8), 914-922. doi:10.1177/0956797612437427
- Morgan, R., & Klaric, J. (2007). *AP Students in College: An Analysis of Five-Year Academic Careers*. Retrieved from Research Report No. 2007-4: http://professionals.collegeboard.com/profdownload/pdf/072065RDCBRpt07-4_071218.pdf
- Mouny, J. L., Pucci, C. T., & Harmon, K. C. (2014, July). How Deaf American Sign Language/English Bilingual Children Become Proficient Readers: An Emic Perspective. *Journal of Deaf Studies and Deaf Education*, 19(3), 333-346. doi:10.1093/deafed/ent050
- Myles, F. (2010, July). The development of theories of second language acquisition. *Language Teaching*, 43(3), 320-332. doi:10.1017/S0261444810000078
- Nassaji, H. (2012). The Relationship Between SLA Research and Language Pedagogy: Teachers' Perspectives. *Language Teaching Research*, 16(3), 337-361. doi:10.1177/1362168812436903

- National Council of Teachers of Mathematics. (2000). *Executive Summary: Principles and Standards for School Mathematics*. Retrieved from NCTM:
http://www.nctm.org/uploadedFiles/Math_Standards/12752_exec_pssm.pdf
- National Council of Teachers of Mathematics. (2000). *Principles and Standards for School Mathematics*. NCTM, Inc.
- Newell, W. H. (1994, Summer). Designing Interdisciplinary Courses. *New Directions for Teaching and Learning*(No. 58). doi:10.1002/tl.37219945804
- Newmark, L. (1966). How not to interfere in language learning. *International Journal of American Linguistics*, 32, 77-87. Retrieved from
<https://ncela.ed.gov/rcd/bibliography/BE015215>
- Ohta, A. S. (2001). *Second language acquisition processes in the classroom: Learning Japanese*. Mahwah, NJ: Lawrence Erlbaum Associates.
- Parnin, C. (2014, April 23). *Scientists Begin Looking at Programmers' Brains: The Neuroscience of Programming*. Retrieved March 12, 2016, from Huffington Post :
http://www.huffingtonpost.com/chris-parnin/scientists-begin-looking-_b_4829981.html
- Parr, S., Byng, S., & Gilpin, S. (1999, February). Talking About Aphasia: Living with Loss of Language After Stroke. *Journal of Advanced Nursing*, 29(2), 27.
doi:10.1046/j.1365-2648.1999.0918e.x
- Perfetti, C. A., & Sandak, R. (2000, Winter). Reading Optimally Builds on Spoken Language: Implications for Deaf Readers. *Journal of Deaf Studies and Deaf Education*, 5(1), 32-50. doi:10.1093/deafed/5.1.32

- Portnoff, S. (2012). Teaching HS Computer Science as if the Rest of the World Existed: Rationale for a HS Pre-APCS Curriculum of Interdisciplinary Central-Problem-Based Units that Model Real-World Applications. *SIGCSE '12 Proceedings of the 43rd ACM technical symposium on Computer Science Education*.
doi:10.1145/2157136.2157210
- Pulimood, S., Shaw, D., & Lounsberry, E. (2011, March 9–12). Gumshoe: A Model for Undergraduate Computational Journalism Education. *SIGCSE'11*, 529-534.
doi:10.1145/1953163.1953314
- Radev, D., & Levin, L. (2009, September). Engaging High School Students in Interdisciplinary Studies: Expanding the Pipeline. *Computing Research News*, 21(4). Retrieved from
http://cra.org/crn/2009/09/engaging_high_school_students_in_interdisciplinary_studies/
- Rankin, Y., Gooch, A., & Gooch, B. (2008, Feb 28 - March 3). The Impact of Game Design on Students' Interest in CS. doi:10.1145/1463673.1463680
- Reeve, J. (2014). *Understanding Motivation and Emotion* (6th ed.). Wiley.
- Robertson, S. A., & Lee, M. P. (1995, December). The Application of Second Natural Language Acquisition Pedagogy to the teaching of Progranuning Languages - A Research Agenda. *SIGCSE Bulletin*, 27(4), 9-20. doi:10.1145/216511.216517
- Rosser, S. (1990). *Female Friendly Science: Appiying Women's Studies Methods arid Theories to Attract students*. New York, NY: Pergamon.
- Round-robin tournament*. (2016, Jan 13). Retrieved from Wikipedia:
http://en.wikipedia.org/wiki/Round-robin_tournament

- Roy, B. C., Frank, M. C., DeCamp, P., Miller, M., & Roy, D. (2015, October 13). Predicting the birth of a spoken word. *PNAS*, *112*(41), 12663–12668.
doi:10.1073/pnas.1419773112
- Roy, D. (2011, March). *The birth of a word (Transcript, Minute 06:31)*. Retrieved from TED:
http://www.ted.com/talks/deb_roy_the_birth_of_a_word/transcript?language=en
- Russell, S., & Norvig, P. (2009). *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall.
- Ruthmann, A., Heines, J., Greher, G., Laidler, P., & Saulters, C. (2010, March 10-13). Teaching Computational Thinking through Musical Live Coding in Scratch. *SIGCSE'10*. doi:10.1145/1734263.1734384
- Sadée, W. (n.d.). *Pharmacogenomics - Using Genetic Information: The Human Genome Project and SNP Mapping*.
- Schulte, C. (2012, November 8-9). Uncovering Structure behind Function – the experiment as teaching method in computer science education., (pp. 40-47). Hamburg, Germany. doi:10.1145/2481449.2481460
- Science Courseware.org, CSU. (n.d.). *Geology Labs Online: Virtual Earthquake*. Retrieved from Virtual Courseware: Earthquake:
<http://sciencecourseware.org/eec/Earthquake/>
- SDSC Biology Workbench*. (n.d.). Retrieved from UCSD, San Diego Supercomputer Center, Biology Workbench.: <http://workbench.sdsc.edu/>
- Selinker, L. (1972, Jan). Interlanguage. *International Review of Applied Linguistics in Language Teaching*, *10*, 209-231. Retrieved from

- [http://www.degruyter.com/dg/viewarticle/j\\$002firal.1972.10.issue-1-4\\$002firal.1972.10.1-4.209\\$002firal.1972.10.1-4.209.xml](http://www.degruyter.com/dg/viewarticle/j$002firal.1972.10.issue-1-4$002firal.1972.10.1-4.209$002firal.1972.10.1-4.209.xml)
- Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., . . . Brechmann, A. (2014). Understanding Understanding Source Code with Functional Magnetic Resonance Imaging. *ICSE '14 Proceedings of the 36th International Conference on Software Engineering*, 378-389. doi:10.1145/2568225.2568252
- Simms, L., & Thumann, H. (2007, Summer). In Search of a New, Linguistically and Culturally Sensitive Paradigm in Deaf Education. *American Annals of the Deaf*, 152(3), 302-311. Retrieved from <http://gupress.gallaudet.edu>
- Skinner, B. (1938). *The Behavior of Organisms*. New York: Appleton-Century-Crofts.
- Smarkusky, D., Propert, P., Stancavage, S., Plociniak, R., Eagan, R., & Nichols, A. (2011, October 20-22). Physics in Motion: An Interdisciplinary Project. *SIGITE '11*, 33-38. doi:10.1145/2047594.2047602
- Spohrer, J. C., & Soloway, E. (1986, July). Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM*, 29(7), 624-32. doi:10.1145/6138.6145
- STEM + Computing Partnerships Program Solicitation NSF 16-527*. (2016, April 3). Retrieved from National Science Foundation: <http://www.nsf.gov/pubs/2016/nsf16527/nsf16527.htm>
- Stross, R. (2008, November 16). What Has Driven Woment Out of Computer Science? *New York Times*. Retrieved from <http://www.nytimes.com/2008/11/16/business/16digi.html>

- The Joint Task Force on Computing Curricula, A. (2013, December 20). *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. New York, NY: ACM.
- Torrance, N., & Olson, D. (1987). Development of the metalanguage and acquisition of literacy: A progress report. *Interchange, A Quarterly Review of Education*, 18(1), 146-146. Retrieved from <http://link.springer.com/article/10.1007%2FBF01807066>
- Turner, K. (2016, March 17). *Why students are throwing tons of money at a program that won't give them a college degree*. Retrieved from The Washington Post: <https://www.washingtonpost.com/news/the-switch/wp/2016/03/17/why-students-are-throwing-tons-of-money-at-a-program-that-wont-give-them-a-college-degree/>
- Watson, C., & Li, F. W. (2014). Failure Rates in Introductory Programming Revisited. (*Innovation and Technology in Computer Science Education (ITiCSE '14)*) (pp. 39-44). Uppsala, Sweden: Association for Computing Machinery (ACM).
doi:10.1145/2591708.2591749
- Watson, J. (1925). *Behaviorism*. New York: Norton.
- Werner, L., Denner, J., Campe, S., Ortiz, E., DeLay, D., Hartl, A., & Laursen, B. (2013, March 6-9). Pair Programming for Middle School Students: Does Friendship Influence Academic Outcomes? *SIGCSE '13, Proceeding of the 44th ACM technical symposium on Computer Science Education*, 421-426.
doi:10.1145/2445196.2445322
- Willis, A., & Conrad, J. (2008, June). Design Of Intelligent Spacecraft: An Interdisciplinary Engineering Education Course. *Proceedings of the 2008 ASEE*

Conference. Retrieved from

http://webpages.uncc.edu/~jmconrad/Publications/ASEE2008_SpacecraftDesignCourse_Final.pdf

Wilson, C., Sudol, L., Stephenson, C., & Stehlik, M. (2010). *Running on Empty: The Failure to Teach K-12 Computer Science in the Digital Age*. ACM, CSTA.

Retrieved from <http://runningonempty.acm.org/>

Woolfolk, A. (2004). *Educational Psychology* (9th ed. ed.). Boston: Pearson, Allyn and Bacon.

Yardi, S., Krolikowski, P., Marshall, T., & Bruckman, A. (2008, October 1). An HCI Approach to Computing in the Real World. *Journal on Educational Resources in Computing*, 8(3), 1-20. doi:10.1145/1404935.1404938

APPENDIX A

A CRITIQUE OF THE ECS CURRICULUM

The ECS survey course took as its starting point topics from CSTA's 2004 *Level 2* and 2007 *Level 3 Objectives and Outlines*, part of CSTA's 2003/2006 *A Model Curriculum for K–12 Computer Science*, and then filled these "slots" with lessons. In doing so, the units may have a common theme in the abstract, but their individual pieces are disparate and unrelated. Absent from ECS's instructional strategies and pedagogy, as laid out in its *Instructional Philosophy*, is the key principle underlying the delivery of content in any mathematics course, or in any field for that matter: *building* knowledge concept by concept.

An effective mathematics curriculum focuses on important mathematics that will prepare students for continued study and for solving problems in a variety of school, home, and work settings. *A well-articulated curriculum challenges students to learn increasingly more sophisticated mathematical ideas as they continue their studies.* (National Council of Teachers of Mathematics, 2000)

Unit 1

The first unit, *Human-Computer Interaction* (HCI), opens with a lesson that familiarizes students with computer hardware, followed by a lesson on Internet searching. Neither topic is given treatment any different from what one would find in a typical computer literacy course.

Next follows a "*visualizing data*" lesson that uses computer tools to explore geometrical patterns in the crafts of various western Native American tribes. The lesson is on a web site developed by Ron Eglash that showcases lessons from his work in ethno-mathematics¹⁹. The lessons use online software to demonstrate advanced mathematical concepts expressed in indigenous cultural artifacts. A narrowly constrained image editor

¹⁹ The intersection of math and culture.

allows users to select from a list of image-details drawn from images of the artifacts. Students are tasked with trying to replicate the designs on a Cartesian grid using an iterative tool that allows them to input numbers and colors.

The lesson's content is typical of a domain for programming pattern exercises done at the beginning of several introductory programming courses to flesh out how nested loops operate. In a programming context, the number of iterations done by the outer loop is simply the number of lines. The number of iterations done by the inner loop is calculated from the point-slope form of the equation of a line, which in turn is derived from a multi-columned Algebra-1 style T-table fitted with line pattern data (See Part 2, Chapter 2, Section 7). None of these underlying mathematics or CS concepts is even hinted at; the content of the lesson is simply the domain. Devoid of any programming, algorithmic or mathematical context, this lesson has no relation to CS.

Pedagogically, students are left on their own to simply guess at ways to generalize from the patterns (pure discovery), an inefficient and frustrating use of instructional time and a wasted opportunity to make the mathematical connections. Moreover, an appropriate starting lesson on iteration would have instead focused on the workings of single loops, not the complex interactions of nested loops. That students might come away from this lesson with any coherent concepts, insights or skills, let alone deep algorithmic understanding, is not credible.

Lastly, and parenthetically, *data visualization* (the title of the lesson) concerns itself with information representation: a graphic display of entire datasets – particularly ones that are non-visual or do not readily lend themselves to visual representations – in such a way that elucidates non-obvious relationships in the data or its subsets. The lesson

has nothing to do with this topic. The absence of attention to relevance, planning, designing, and thinking – let alone deep reflection – regarding this lesson is unfortunately typical of most of the course's disorganized content. Instead, the authors' preferred method of lesson development is to simply insert material developed by other educators without modification, adaptation or extension.

What follows next is a primary-school level lesson on sequential commands, illustrated by tasks such as making a sandwich, taking a gag test, and drawing stick figures. These might have value as quick engagement activities to illustrate the concept that a computer is unable to interpret the intent behind incomplete instructions. The lesson ends without students seeing a computer run programs with such logical bugs.

The unit concludes with another primary-school level lesson, this time on the Turing test, drawn from *CS Unplugged*²⁰ (Bell, Witten, & Fellows, 2006). Although the Turing test is historically significant, its present-day value is primarily as a hackneyed, layman sound bite, warranting only quick mention at the start of a college-level *Artificial Intelligence* (not HCI) course:

AI researchers have devoted little effort to passing the Turing Test, believing that it is more important to study the underlying principles of intelligence than to duplicate an exemplar. The quest for "artificial flight" succeeded when the Wright brothers and others stopped imitating birds and started using wind tunnels and learning about aerodynamics. Aeronautical engineering texts do not define the goal of their field as making "machines that fly so exactly like pigeons that they can fool even other pigeons." (Russell & Norvig, 2009)

ECS' authors might have instead chosen a more contemporary (ca. 1999) and topical reference for AI, e.g. a lesson on a story-generating machine named BRUTUS that uses a measure called "literary creativity" (Bringsjord & Ferrucci, 2000). A more complex discussion – and one more rigorously suitable for high school students – of the types of

²⁰ The curriculum's subtitle is: "*An enrichment and extension programme for primary-aged children*".

considerations needed to judge the software implementation of an artificial author might include "weak" vs. "strong" creativity/originality, psychological tests such as the Torrance Tests of Creative Thinking, and the types of properties one would use to score a story, such as imagery, style and story structure. Members of the BRUTUS research team went on to create WATSON, the computer that, in 2011, defeated reigning *Jeopardy* champion Ken Jennings.

The first unit contains faulty and irrelevant content choices, confirming earlier doubts about the subject-matter competency of the authors. Likewise, the use of the title *Human-Computer Interaction* for a unit comprising ways that people "interact" with computers only in the most trivial sense, rather than how *HCI* is understood in a CS context, underscores this shortcoming. The unit might actually have been better served using a true 6-week high school level HCI course that had already been developed at Georgia Institute of Technology and field-tested on a demographic similar to that of ECS (Yardi, Krolikowski, Marshall, & Bruckman, 2008)²¹. Finally, the use of unedited and unadapted primary and middle-school level teaching materials speaks to the authors' lack of confidence in the capabilities of high school students, especially upperclassmen. Although ECS was given UCOP "a-g" approval in 2009, nothing in this first unit remotely qualifies as college preparatory material.

²¹ The focus of the project-based course was the creation a Touch-Screen Digital Desktop. The researchers were cautious, however, to characterize this introductory course as computing, not computer science.

Unit 2

The second unit fills another slot in CSTA's Objectives, *Problem Solving*. The unit's intent is to introduce students to algorithms. Topics covered are: (1) the handshake and fencepost problems; (2) the mathematics of cornrow braiding; (3) binary numbers (*CS Unplugged*); (4) linear/binary search; (5) sorting algorithms (*CS Unplugged*); (6) minimal spanning tree (*CS Unplugged*).

The unit begins with a lesson that takes students through the "four steps" of problem-solving after George Pólya's 1945 book *Hot to Solve It: A New Aspect of Mathematical Method*. These are: (1) Understand the problem; (2) Make a plan to solve the problem; (3) Carry out the plan; (4) Review and reflect. Ripped out of context, however, and presented as an overly simplistic and prescriptive recipe with no more elaboration than what appears above, the "steps" are useless in practice, insipid trial-and-error instructions with no more educational value than the instruction "Get out your pencils." Pólya, a Stanford mathematics professor, had actually laid out a host of heuristics for solving complex mathematical problems, but crucially assumed extensive and relevant *domain knowledge* when attacking new problems.²²

Another troubling problem is that the ECS authors uncritically adopted CSTA's erroneous assumption that solving problems is a general teachable skill. However, any credentialed teacher knows from his/her basic Educational Psychology course that the ability to solve problems is – and algorithms themselves are – highly domain-specific;

²² The top-down imposition of a recipe-like framework is reminiscent of the popularization of the "scientific method", which has been called "a type of scientific urban myth", because of the unspoken assumption of exclusivity while ignoring the principal roles of theory and serendipity in the history of scientific discovery (Brown & Kumar, 2013). At least the scientific method is actually useful in an investigative cycle. Observations and/or analysis of data patterns that spur hypotheses that can be tested experimentally, and have the potential to lead to a predictive understanding of observed phenomena and theories about underlying realities.

and that good problem-solvers draw upon prior experience and knowledge of particular domains (Woolfolk, 2004) (Bransford, Sherwood, Vye, & Rieser, 1986). From a psychological vantage point, problem-solving is a complex phenomenon, described by Gestalt theorists with notions like "restructuring", "insight" and "entrenchment"; and by cognitivism with a reliance on domain knowledge and heuristics. In all of these, although conditions that facilitate the crucial moments of insight can be listed, there are no satisfactory explanations for how such insights arise. The *incubation* phenomenon, setting aside a problem after being unable to find a solution, with a solution later popping into one's mind (like a forgotten detail that appears after the fact), argues that problem-solving may be a largely subconscious process.

Another deficiency is that the unit introduces algorithms, but does not ask students to implement them by writing programs, running them, and modifying them in order to study the problems in depth and solidify understanding. This is another wasted opportunity. This flawed pedagogy is by design, as several of the topics are drawn verbatim from *CS Unplugged Teachers Guide*, whose subtitle is "*An enrichment and extension programme for **primary-aged children***". Although any of the *CS Unplugged* activities would be plausible as engagement activities at the beginning of a lesson, the ECS team made no attempt to extend these problems for high-school level students. This is as irresponsible as teaching Algebra 1 students the content found in a 3rd-grade mathematics course, the antithesis of what the ECS claims to be doing in their

Instructional Philosophy section:

... a curriculum organized around major concepts that students are expected to know deeply. Teaching must engage students in active reasoning about these concepts. In every subject, at every grade level, instruction and learning must

include commitment to a knowledge core, high thinking demand, and active use of knowledge.

When I was involved with ECS in academic year 2009-2010, having attended the training at UCLA in the previous August, I found inadequate written instructions, explanations and resources – for both teachers and students – for many of the problems in the unit. The most overt example was a problem called "Youth Soccer League Activity" scheduled on Day 5 in ECS versions 1.0 and 2.0. It allowed students 35 minutes to create a schedule for each of 6 teams to play against every other team twice in as few days as possible. Instruction beforehand asked only that students calculate how many games would take place, as if that was all the information required for the solution. They were then expected to find a solution using pure discovery learning. While preparing the lesson and naively attempting a solution, after only a few minutes, I found that I was running into dead ends no matter what I tried. As ECS provided absolutely no documentation, I turned to the Internet, where I found that the solution – *round-robin tournament scheduling* – is neither obvious nor intuitive. The algorithm can create a schedule of $n-1$ days for an even number of teams, or n days for an odd number of teams (Round-robin tournament, 2016). An alternate algorithm uses *Berger tables*. If one wanted to make connections, the algorithm also seems to bear some relationship to the *magic square* problem often seen in geometry courses. Moreover, the solution for an even number of teams is different enough from that for an odd number of teams that students will be unable to extrapolate from one to the other. I devised an Excel spreadsheet demonstrating the solution for both even and odd cases, emailed both it and a discussion of the problem to the ECS team, and posted the information on the teacher support e-board. I heard nothing back. The lesson was dropped in ECS version 3.0.

Chief among Unit Two's drawbacks, starting with ECS version 3.0, is a 3-day lesson that has nothing to do with CS, and is an astoundingly misguided attempt at inclusion that instead ends up stereotyping African-American students. The lesson uses an online ethno-mathematics resource on the fractal nature of cornrow braids (Eglash, 2003), but the lesson is used as-is: the ECS team did nothing to adapt it or show how it might credibly be related to CS. The impetus to include a lesson with African-American content is consistent with one of ECS' major strategies, *Building on Students' Funds of Knowledge and Cultural Wealth*. However, the inclusion of this lesson in a CS course is patronizing and culturally insensitive at best, and naively racist at worst. That ECS administrators proudly referenced this lesson in an article published in *ACM Inroads*, a national CS education magazine, calls into question their judgment²³ (Margolis, et al., 2012). Simply put, the lesson is tokenistic, even setting aside its complete absence of rigor. The not-so-hidden message is that African-Americans' contribution to CS is so lacking that one's only recourse is to use a *mathematics* lesson *about* African fashion rather than intellectual contributions *by* black computer scientists. It would be one thing were such resources unavailable. However, an internet search for "black scientists, inventors and mathematicians" yields 18,700 results; and the Wikipedia page "List of African-American inventors and scientists" has no shortage of computer scientists (List of African-American inventors and scientists, 2016).

This example can be seen as laughable, inappropriate, quite frankly offensive, or any combination of the above. However, it does have parallels to a similar ideological battle for how to resolve the issue of disparities in female participation in CS (Part 2,

²³ The most common initial reaction to being shown the article's illustrations was raised eyebrows, with a follow-up as diplomatically phrased as possible: "Is this a joke?"

Chapter 1, Section 5). Some CS educators have suggested that curricula centered on story-telling using IDEs like *Alice* or *Scratch* be used to attract girls to computing

(Kelleher, Pausch, & Kiesler, 2007). However, others have warned:

The implications are that women do not need handholding or a "female friendly" curriculum in order for them to enter and be successful in CS or related fields, nor is there need to change the field to suit women. To the contrary, ***curricular changes***, for example, ***based on presumed gender differences can be misguided***, particularly if they do not provide the skills and depth needed to succeed and lead in the field. ***Such changes will only serve to reinforce, even perpetuate, stereotypes and promote further marginalization.*** (Blum L. , Frieze, Hazzan, & Dias, 2006)

Historically, the idea for the ECS survey course followed a failed attempt to increase the number of *APCS-A* offerings in LAUSD by the Los Angeles based group *Computer Science Equity Alliance (CSEA)*, started in 2004. The results of this two-year effort seemed successful on the surface:

The number of courses increased from 11 to 23, and the number of students enrolled in Advanced Placement Computer Science increased from 225 to 611 students. Not only did the total number of students increase, but also the enrollment of girls quadrupled, Latinos quintupled, and African Americans doubled. (Goode, Chapman, & Margolis, 2012)

There is a big difference, though, between participation in a course and successfully learning its content.

Students who complete AP courses do not necessarily experience success in college (Geiser & Santelices, 2004). Although taking a course may not improve student outcomes, the College Board (2005) found a strong correlation between passing AP exams and having academic success in college. As such, providing more AP courses in an urban school setting might not automatically ensure better college outcomes. (Hallet & Venegas, 2011)

As it turned out, CSEA's effort may have simply filled up classes with students who were academically unprepared – and were almost certainly taught by instructors who lacked subject-matter competence – as the less-than-forthright admission below acknowledges.

Yet, underlying these increased enrollment numbers was a tension that the **programming-centric** focus of the AP course and the advanced, college-level status was not an accessible point of entry for most students. Three years into this work, we recognized the need for a **foundational high school course introducing students to the major concepts of the field of computer science, and a course that had the potential to engage the diverse populations of Los Angeles schools.** (Goode, Chapman, & Margolis, 2012)

The problem with this conclusion is that it is faulty. What is needed to make the AP course "an accessible point of entry" is not a foundational survey course, but rather a foundational programming course. Moreover, stating that the authors "recognized a need for a foundational course" is disingenuous; admitting that they had no idea how to effectively teach high school students to program would have been a more honest assessment. There is also an inherent pedagogic contradiction in a survey course claiming to introduce the field's major concepts if students have no programming foundation. In a college sequence, the study of algorithms follows introductory programming for good reason. Pedagogically, this allows students to implement the algorithms, observe the execution of the code, and to make permutations that can deepen understanding.

The study of algorithms divorced from programming is also irrational in a practical sense. Algorithms are ways for *machines* to accomplish tasks. They provide no particular insight, value or efficiency for humans performing such tasks manually. Such concerns are echoed in the literature and are explicitly discussed in a critique of "unplugged" curricula in *Appendix D*.

The Goals section of ECS states:

Exploring Computer Science is designed to introduce students to the **breadth** of the field of computer science through an exploration of engaging and accessible topics....The goal of *Exploring Computer Science* is to develop in students the computational practices of algorithm development, problem solving and programming within the context of problems that are **relevant to the lives of today's students.**

Rather than "breadth", though, the algorithms in Unit 2 have simply been dissected out of end-of-unit problem sets in a first year college programming curriculum. As such, they are isolated domains with no particular relevance to CS, chosen by their original authors because their optimal solutions were opportunities for applying the particular PL concepts being studied.

That criticism aside, though, no thought seems to have been given as to what educational value they might have as standalone topics when taught to students who have no programming experience. To illustrate, the study of sorting algorithms can deepen understanding of the tension between memory space and theoretical execution time. Outside of this context, sorting algorithms and methods are hardly pressing 21st-century concerns. Additionally, minimal spanning trees certainly have their uses in engineering applications, but they do nothing to expose students to the breadth of algorithmic applications used in contemporary subfields of CS. Similarly, the fencepost and handshake problems are exercises one might find at the end of an initial chapter in a mid-20th-century introductory programming course, hardly major topics in any case.

Rather one might better argue that the most compelling contemporary topics in CS are found in disciplines like Artificial Intelligence, Big Data, Machine Learning and Bioinformatics. Concepts like gene discovery or gene assembly, neural networks, predicting social behaviors, genetic algorithms and computer vision might be better candidates for a non-programming survey course intended to generate 21st-century interest... but not competence. Indeed, it would be beyond the capabilities of non-programming students to understand the workings and theoretical underpinnings of a single *perceptron*, let alone a neural network. Designing a persuasive and rigorous high

school (or college-level, for that matter) survey course is not as simple as whittling down an introductory programming course.

Unit 3

The Web Design unit is the one unit in the course with grade-level content. It is well-thought out and sequenced, introducing students immediately to web-related social and security issues, and then to the technical nuts and bolts: the programming systems HTML, CSS and Javascript; the production tools Photoshop and Lightbox; and ready-made scripts for accordion menus and sliding images. The only criticism, and a serious one, is that – per Jan Cuny's criterion for distinguishing computer literacy from computer science (Cuny, 2011) – the unit is about being a consumer/user of technology production software, and not about being a technology creator. Compressed into six weeks, it's not credible to believe students will be able to understand HTML, CSS or Javascript on anything but a shallow level, let alone that they will be able to independently design, or build programming code, into web pages.

Unit 4

Unit 4 is an introductory programming unit. The decision to use Scratch to introduce programming to high school students immediately precludes them from learning about methods, parameters and hierarchical program organization (topics which students in my freshman pre-AP programming course begin delving into the first week). The programs students work on are games and storytelling; there is no application or connection to real world problems or situations. The content is middle-school level, at

best. Even so, there can be no realistic expectation that students will learn the uses of variables, conditionals, boolean logic, or timers in any depth or complexity in 7 weeks. There is no attempt to teach iteration, recursion or classes/objects, although in earlier versions, iteration and lists were taught in a later Python unit.

Unit 5 (versions 2.0, 3.0) / Unit 6 (versions 4.0, 5.0)

The Robotics programming curriculum used in Unit 5 was designed by a middle-school teacher (Michelle Hutton, The Girl's School, Mountain View, CA) for 8th grade students. It's excellent, for middle-school students.

Unit 6 (versions 2.0, 3.0) / Unit 5 (versions 4.0, 5.0)

Unit 6 has gone through several incarnations. In the 2009 ECS 2.0 version it was a Python programming unit, translated virtually verbatim from the first lessons of the ICT APCS-A curriculum, for which LAUSD has a district-wide license. This is a serious programming unit, with ambitious, but unrealistic, goals for teaching in 6 weeks(!) what a typical APCS-A course might take a full semester to teach in depth. What the ECS team soon found out is that Python has a steep learning curve and comes with a not so user-friendly IDE. In two year-long periods (2005-6, 2008-9) of teaching Python to 15-year-old high school students, one study concluded:

... the same factors that have made Python such a successful programming language in the market have presented our students with specific learning obstacles...

...Our experience shows that the success of a language in a professional setting does not predict success as a teaching tool. (Konidari & Louridas, 2010)

It was no surprise when Python disappeared from ECS two years later, replaced in versions 4.0 and 5.0 by a "Computing and Data Analysis" unit. The software used is described in an ECS proprietary manual:

Deducer is a graphical interface designed to work with R (a free, data analysis software environment for statistical computing and graphics) and allow users to perform data analysis without programming. The underlying language of R can be seen at each step, which enables students to learn about R if they are interested, but typing R commands at the command line is not required.

The topics in the unit utilize elementary statistical concepts, plots, maps, histograms and the like, analyzing many kinds of data, including text. As with the Web Design unit, this unit falls under the category of literacy and production, not technology creation, as students simply use the software to analyze data, rather than writing R programs. This would be a good unit in a mathematics curriculum to showcase real-world applications for statistics concepts, but, as in Units 1 and 2, the topics are not appropriate content for a computer science course.

APPENDIX B

THE INFLUENCE OF THE APIG ON CSTA POLICY

The *APIG* managed to carve out a seat for itself in the CSTA, as evidenced in their 2011 publication *CSTA K-12 Computer Science Standards* (CSTA Standards Task Force, 2011). Just as literary analysis of the Bible uncovers different groups of authors, *Standards* reveals at least two sets of author groups in the last ten years, with polar opposite philosophical views on the role of programming in the secondary CS curriculum. On the one hand, section 2 of *Standards* initially attests to the central and motivating force of programming:

Children of all ages love computing. When given the opportunity, young students enjoy the sense of mastery and magic that programming provides. Older students are drawn to the combination of art, narrative, design, programming, and sheer enjoyment that comes from creating their own virtual worlds. Blending computer science with other interests also provides rich opportunities for learning. Students with an interest in music, for example, can learn about digital music and audio. This field integrates electronics, several kinds of math, music theory, computer programming, and a keen ear for what sounds beautiful, harmonious, or just plain interesting. (p. 2)

This paragraph was copied verbatim from an earlier publication, CSTA's 2003/2006 *A Model Curriculum for K-12 Computer*. This passage, together with one from *Model Curriculum's* foreword (p. V) – also repeated in *Standards*, Section 2.5, "*Computer Science Can Engage All Students*" – makes clear that K-12 CS is about teaching students to program:

Pedagogically, computer programming has the same relation to studying computer science as playing an instrument does to studying music or painting does to studying art. In each case, even a small amount of hands-on experience adds immensely to life-long appreciation and understanding, even if the student does not continue programming, playing, or painting as an adult. Although becoming an expert programmer, a violinist, or an oil painter demands much time and talent, we still want to expose every student to the joys of being creative. The goal for teaching computer science should be to get as many students as possible

enthusiastically engaged with every assignment. Instead of writing the same old mortgage calculation program, have students design and write programs that control their cell phones or robots, create physics and biology simulations, or compose music. Students will want to learn to use conditionals, loops, and parameters and other fundamental concepts just to make these exciting things happen. (p. 5)²⁴

In section 4.2 (*Strands*), however, an anti-programming philosophy unprecedented in CS education is articulated:

Almost since its inception, computer science has been hampered by the perception that it focuses exclusively on programming. This misconception has been particularly damaging in grades K–12 where it often has led to courses that were exceedingly limited in scope and negatively perceived by students. It also fed into other unfortunate perceptions of computer science as a solitary pursuit, disconnected from the rest of the world and of little relevance to the interests and concerns of students.

We address these concerns by distinguishing five complementary and essential strands throughout all three levels in these standards. (p. 9)

Interestingly, the authors cite no research studies to support a causal relationship between programming as exclusive subject matter content and the so-called "damage" they allege, which, though unstated, includes persistently low enrollments and severe demographic disparities. Propped up with nothing more than opinion, belief and underlying agenda²⁵ to justify it, this alternative stance initiated a radical shift away from the central role of programming in CS secondary education.

Although the two points-of-view could hardly be more contradictory, the authors of *Standards* make no attempt to reconcile the competing narratives. Instead, they simply

²⁴ To those who have actually taught secondary CS, this last statement is wishful thinking, conditional on the academic readiness of the student. With near unanimity, students below grade-level have great difficulty learning to program.

²⁵ The agenda being that the authors could conceive of no framework for how to effectively teach programming to most grade-level secondary students.

coexist as a philosophical and political contradiction, notwithstanding the schizophrenic implications for inconsistent and antithetical policies.

The authors of *Standards* do not cite evidence of causation (or even correlation) between programming and the negative K-12 outcomes they claim occur because none exists. There is evidence at the postsecondary level, however, suggesting that (a) pedagogical strategies, e.g. not providing useful contexts for introductory programming concepts and skills, may be detrimental; and (b) a social/educational micro-environment inclusive of traditionally underrepresented students is helpful (Blum L. , Frieze, Hazzan, & Dias, 2006). Two colleges have, in fact, successfully reversed their gender imbalances using such strategies. The CS department at Carnegie-Mellon University (CMU) made several alterations: (a) it changed its admissions policies by deemphasizing prior programming experience; (b) it provided social support for female students; (c) it offered different introductory courses for students consistent with the degree of prior programming experience; and (d) it eventually began to provide contexts for introductory programming concepts by showing examples of their application in the world at large. Harvey Mudd College (HMC) has created different tracks for its beginning students based on prior programming experience and modified its introductory courses so that the subject matter occurs within broader contexts, with less emphasis on the nuts and bolts of programming (Alvarado & Dodds, 2010). A third HMC introductory course frames instruction within a biological context. Similar strategies are unlikely to have a similar impact at less elite colleges. Secondary sites, though, are even less likely to benefit from such practices.

APPENDIX C

THE APIG'S IMPACT ON CSTA'S CURRICULAR FRAMEWORK

The anti-programming pronouncement that appears in *Standards* (Appendix B) is accompanied by a curricular model consistent with that ideological position. Features of the *original* framework that it replaced, articulated in *Model Curriculum*, are summarized below:

As a basis for describing a model curriculum for K–12 computer science, we use the following definition of computer science as an academic and professional field.

Computer science (CS) is the study of computers and algorithmic processes, including their principles, their hardware and software designs, their applications, and their impact on society.

In our view, this definition requires that K–12 computer science curricula have the following kinds of elements: programming, hardware design, networks, graphics, databases and information retrieval, computer security, software design, programming languages, logic, programming paradigms, translation between levels of abstraction, artificial intelligence, the limits of computation (what computers can't do), applications in information technology and information systems, and social issues (Internet security, privacy, intellectual property, etc.). (p. 2)

Section 3 of *Standards* at first reiterates its support for this rigorous and traditional definition:

Computer Science, on the other hand, spans a wide range of computing endeavors, from theoretical foundations to robotics, computer vision, intelligent systems, and bioinformatics. The work of computer scientists is concentrated in three areas:

- designing and implementing software,
- developing effective ways to solve computing problems, and
- devising new ways to use computers.

For the purposes of this document, we rely heavily on the definition of computer science [*above, previous citation*] provided in the original *ACM/CSTA Model Curriculum for K–12 Computer Science*, as we believe that this definition of

computer science has the most direct relevance to high school computer science education. (p. 6)

...If these standards are widely implemented and these goals are met, high school graduates will be prepared to be knowledgeable users and critics of computers, as well as designers and builders of computing applications that will affect every aspect of life in the 21st century.

However, section 4 of *Standards*, in contrast, proposes a much simplified and poorly-conceived framework of five curriculum *strands*²⁶: (1) *Computational Thinking*, (2) *Collaboration*, (3) *Computing Practice and Programming*, (4) *Computers and Communication Devices*, and (5) *Community, Global, and Ethical Impacts*. The most significant change to the earlier traditional model occurs in the 3rd strand, *Computing Practice and Programming*, and is described in Section 4.2.3. Here, the anti-programming camp seems to have succeeded in its quest to have CSTA remove programming from its central position in secondary CS education:

The use of computational tools is an essential part of computer science education at all levels. While this is traditionally branded as “Information Technology,” it is impossible to separate IT from the other four strands in computer science. Computing practice at the K–12 level must therefore include the ability to create and organize web pages, explore the use of programming in solving problems, select appropriate file and database formats for a particular computational problem, and use appropriate Application Program Interfaces (APIs), software tools, and libraries to help solve algorithmic and computational problems.

...Because computing is often misperceived as only programming, it is especially important for students to understand the broad array of opportunities computer science knowledge can provide across every field and discipline. (p. 11)

Programming is now subsumed under a broader *Computing Practice* heading as just one activity among many, in which students are to merely "explore the use of programming in

²⁶ A curriculum *strand*, in the language of K-12 education, is a content area that is vertically aligned, i.e., where student knowledge is revisited, grows and deepens over the course of several grade levels. The National Council of Teachers of Mathematics (NCTM) has devised an exemplary model spanning the K-12 period – from counting through pre-Calculus – consisting of 10 strands (which they call standards): Number and Operations, Algebra, Geometry, Measurement, Data Analysis and Probability, Problem Solving, Reasoning and Proof, Communication, Connections, and Representation.

solving problems", as opposed to acquiring competence in the use of one or more programming languages. Practice is characterized as the use of "computational tools," later characterized in the sections detailing the specific standards as "web programming design tools" or "productivity/multimedia tools", tasks that fall under the realm of Computer Literacy. Completely absent is any mention of teaching programming basics. And no mention is made for how students will use "Application Program Interfaces (APIs)" and "libraries" in the absence of high-level programming instruction.

In addition, the section again warns – in case the reader missed it earlier – that *"computing is often misperceived as only programming"*, claiming further – and incorrectly – that *"it is impossible to separate IT [i.e. programming] from the other four strands in computer science."* This is sheer nonsense; the dependency, in fact, runs in the opposite direction. Programmability is universally acknowledged to be the core activity that enables and informs other CS topics.

The strand *Computational Thinking* deserves some mention. In section 4.2.1, the authors acknowledge *"that there is, as yet, no widely agreed upon definition of computational thinking,"* yet plow ahead with one recently developed:

CT is an approach to solving problems in a way that can be implemented with a computer. Students become not merely tool users but tool builders. They use a set of concepts, such as abstraction, recursion, and iteration, to process and analyze data, and to create real and virtual artifacts. CT is a problem-solving methodology that can be automated and transferred and applied across subjects. The power of computational thinking is that it applies to every other type of reasoning. It enables all kinds of things to get done: quantum physics, advanced biology, human–computer systems, development of useful computational tools.

The first sentence explicitly connects CT to implementation (i.e. programming) and limits its application to computers: this sounds a lot like programming. However, the paragraph subsequently makes the broad claim that CT can be applied *"to every other*

type of reasoning," though what this means even in an inexact way is unclear. At the section's outset, a similar unsupported claim is made: "*The study of computational thinking enables all students to better conceptualize, analyze, and solve complex problems by selecting and applying appropriate strategies and tools, both virtually and in the real world*" (p. 9). These statements have several false implications.

Consider the application of CT to the task of sorting a deck of cards. A person using any of the sorting algorithms in the CS canon would be ignoring the vastly greater sensory input, brain power, and strategies people have at their disposal for doing such tasks. An iterative or recursive solution makes up for the computer's deficits in these areas. For a person to sort a deck of cards using a computing algorithm would amount to a colossal and inefficient waste of time repeating comparisons of cards many fold times more than required. Furthermore, if a problem were so huge or complex that only a computer solution could reliably solve it, it's unlikely a person or team of people could perform the same operations without error and within a reasonable period of time.

Even when the authors state that CT can be applied to a particular problem, they wade cavalierly into territory outside their expertise. *Model Curriculum* states:

Students should be able to use computer science skills (especially algorithmic thinking) in their problem-solving activities in other subjects. One simple example is the use of logic for understanding the semantics of English in a language arts class. There are many others.

In reality, computational linguistics is far from simple. While problems in phonology and syntax are amenable to algorithmic solutions, those in semantics are actually not so easily solved. Semantic algorithms would make little sense in a 21st-century curriculum because they have been abandoned in favor of machine learning techniques. The latter

uncover patterns in huge datasets of structured and unstructured language assembled from a wide range of contexts.

However, the main problem with the definition is the claim that CT can enhance problem-solving, reasoning and *general thinking ability*, outside of the specific context of computer programming. This is echoed in section 5.1 (Level 1 standards):

We agree with teachers who believe that students at this age ought to begin thinking algorithmically as a general problem-solving strategy. Thus, it makes sense to develop more teaching strategies that encourage students to engage in the process of visualizing or acting out an algorithm. Seymour Papert's pioneering experiments in the 1970s corroborate this belief, and his seminal work *Mindstorms* and related curricula provide many more examples of how elementary students can be engaged in algorithmic thinking. (p. 12)

Papert's book appeared in 1980, followed shortly by others (Bork, Nickerson) making similar claims. However, one research study carried out a few years later to test these claims could establish no such correlation (Mayer, Dyck, & Vilberg, 1986). Instead, this study criticized Papert's reliance on "case studies and testimonials" – as opposed to experiments or research – and stated that "there have been very few relevant research studies and almost no convincing support of this connection." Specifically, these researchers debunked three assertions:

1. Learning a programming language will enhance a person's thinking skills
2. Certain thinking skills will enhance the learning of programming
3. Pre-training on certain thinking skills will enhance the learning of programming

Other research studies also revealed that "problem-solving skills appear to be much more discipline specific than had first been thought" (Linn & Dalbey, 1985).

When enhancement was found, it did not impact problem-solving ability: "Logo programmers outperformed other students on meta-cognitive tasks but showed no

differences in areas of cognitive development or logical thinking" (Martin & Hearne, 1990).

Claims for augmenting problem-solving ability are reminiscent of 19th-century "mental discipline", a theory that employed both "mind-as-a-muscle" and "transfer of training" metaphors. Edward Lee Thorndike was the first to disprove "mental discipline" in a 1901 publication summarizing experiments that failed to show such transfer; he concluded: "*Improvements in any single mental function need not improve the ability in functions commonly called by the same name*" (Kliebard, 2004). Although the principle of "mental discipline" had been a mainstay of American education throughout the 19th century, it soon lost credibility.

The idea that CT can improve or aid the thinking process is further belied by the fact that the activity is difficult to perform correctly even in its native domain; the code-test-debug software engineering cycle amply testifies to the difficulty that even experienced programmers have when trying to code straightforward programs correctly. Simply stated, CT is an unnatural way of thinking, its sole raison d'être being to train programmers to write software programs that computers, with their constrained set of operations, can execute to solve problems that are beyond human capabilities and limitations. It has little to no value or application outside of the context of CS.

That the authors of *Standards* make such a general claim so cavalierly might be forgivable if not for "problem solving" being a major topic in standard Educational Psychology classes, and one that every credentialed K-12 teacher studies. Discussions in such courses about successful problem solving underscore: (a) the importance of domain specific knowledge; (b) the ability to access portions of that knowledge relevant to the

problem at hand; and (c) the fact that algorithms are domain-specific, and that many/most problems are not solvable by application of algorithms (Woolfolk, 2004) (Bransford, Sherwood, Vye, & Rieser, 1986).

The decision to put forward *Collaboration* as a strand is bizarre. First, "collaboration" as understood in a university CS context involves human-computer interaction (HCI), specifically interfaces that facilitate online interaction and communication (The Joint Task Force on Computing Curricula, 2013). It is not group work independent of computer networks. Second, collaboration in a secondary setting is a pedagogic strategy – not academic content – and one commonly employed in K-12 classrooms irrespective of subject area. Although collaborative strategies are mentioned in the *CTE Anchor* standards and the *Common Core ELA Speaking and Listening* standards, the goals are neither content-related nor teamwork, but rather "*building on others' ideas and expressing their own clearly and persuasively*". In this context, collaboration is just one of many learning skills to support academic learning. The skills are not subject specific in the slightest.

The authors also make two specious claims: (a) "*significant progress is rarely made in computer science by one person working alone*"; and (b) "*new programming methodologies such as pair programming emphasize the importance of working together*" (Section 4.2.2, p. 10). The first assertion is contradicted by the hardly "rare" and well-known examples of individuals who have conceived innovative ideas, and then founded and grew major start-ups (e.g. Jobs, Gates, Zuckerberg). New insights and theories invented by academics and theoreticians (Turing, Dijkstra, Hopper) were again individual efforts. The second assertion about "pair programming" in both industrial and

educational contexts ignores mixed findings about its effectiveness, and the particular circumstances under which it demonstrates advantages. As stated at the end of Part 1, Chapter 2, Section 1, there are no studies showing the effectiveness of pair programming in secondary classrooms²⁷. Proposing *Collaboration* as a strand is an inept attempt to fill the void created by the unnecessary removal of programming competence with a "strand" entirely devoid of academic content.

²⁷ The same section cited *recent* studies showing benefits for middle-school students under many pairing conditions – and particularly so when the two participants are friends – but not all (Werner, et al., 2013) (Denner, Werner, Sampe, & Ortiz, 2014).

APPENDIX D

THE APIG FALLACIES OF COMPUTER-FREE CS INSTRUCTION

CSTA's current standards model is divided into 3 levels: Level 1 Grades K-6, Level 2 Grades 6-9, and Level 3 Grades 9-12. It's unclear what basis the authors have for creating standards for Levels 1 and 2, when experience with teaching CS at these pre-secondary levels is sparse. Consider Standard 2 for Computational Thinking, Level 1, Grades 3-6 (L1:6:CT):

2. Develop a simple understanding of an algorithm (e.g., search, sequence of events, or sorting) *using computer-free exercises*.

This standard was inspired by the popular *Computer Science Unplugged: An enrichment and extension programme for primary-aged children* (Bell, Witten, & Fellows, 2006), an elementary curriculum that decouples the learning of algorithms from computer programming. However when this curriculum was taught to middle and secondary students, it was found wanting:

For CS unplugged activities are some evaluative reports available: Feaster et. al. ([10], p.252) conclude that “the program had no statistically significant impact on student attitudes toward computer science or perceived content understanding.” Taub et.al. [35] yield similar results (p.24) and found the following explanation (similar to [36]): “only some of the objectives were addressed in the activities, [...] the activities do not engage with the students’ prior knowledge and [...] most of the activities are not explicitly linked to central concepts in CS” (p.1) ... (Schulte, 2012)

By the same token, CS professors at Harvey Mudd College who authored a middle school curriculum (*MyCS*) using certain aspects of *CS Unplugged* cautioned:

MyCS requires no resources other than a computer lab. Many of its activities, inspired by *CS Unplugged*, do not even require a lab of machines, though we do not believe the skill-building necessary for the development of a justifiably confident computational identity is possible without hands-on creation at a computer. (Dodds & Erlinger, 2013)

Comments like the above illustrate the tendency – perhaps desperation – of CS educators for novel teaching strategies before their effectiveness has been corroborated by research. Moreover, there are two objections one could make to the *CS Unplugged* approach solely on principle: (1) virtually all students enter a computer class hungry to extend their knowledge and gain new skills and competencies; one would need very convincing pedagogic reasons to set aside this natural inner motivation in order to purposely forego the use of a computer when teaching computer-related concepts; and (2) computer-free activities are not ends in themselves, but rather starting points to engage students in acquiring new concepts and skills, especially in a setting beyond primary school.

Advocates of computer-free instruction often utilize a quote attributed to Edsger Dijkstra, a preeminent computer scientist, who said: "*Computer science is no more about computers than astronomy is about telescopes.*" The quote actually originated in an article by two computer scientists at the University of British Columbia (Fellows & Parberry, 1993) and was originally intended to remind its audience about the foundation of discrete mathematics underlying much of theoretical CS. Notwithstanding, the article did make two claims: "*Science is not about tools. It is about how we use them and what we find out when we do*" and "*computer science is not about computers – it is about computation.*"

This exclusionary logic, however, is debatable. In his seminal work, *The Structure of Scientific Revolutions*, Thomas S. Kuhn recognized the crucial role of instrumentation for scientific discovery and the development of theories to account for experimental anomalies: "*The decision to employ a particular piece of apparatus and use*

it in a particular way carries an assumption that only certain sorts of circumstances will arise. There are instrumental as well as theoretical expectations, and they have often played a decisive role in scientific development." (Kuhn, 1996). In other words, science is not entirely an abstract realm existing independently of the tools it uses for exploration. Tools construct and frame the conceptual structure in which scientists think about their discipline. Certainly in astronomy, the development of increasingly sophisticated telescopes and the collection of the particular types of observations/data that they are able to detect – often obtained and interpreted by software – constrain and inform the types of ideas that are generated. Similarly in CS, the evolution of progressively more complex computers and peripherals has spurred the birth of ever new sub-disciplines in the field (e.g. parallel computing, computer graphics).

It is worth repeating Taub's criticism of *CS Unplugged* that "*most of the activities are not explicitly linked to central concepts in CS.*" The twelve activities in *CS Unplugged*, although topics that may be addressed somewhere in the first two years of a university CS program, have little connection to one another, are presented with little depth, and their contexts/applications are simplified academic exercises of trivial value. They certainly form no cohesive whole and one wonders about the value in teaching students such content and the prospects for retention if they will not re-encounter and build on the topics on a regular basis.

As an example, consider the decision to include binary numbers in this curriculum. While certainly important when presenting such concepts as round-off error, bit operations on image data, and compression, *CS Unplugged's* "Secret Message" activity for practicing binary-to-decimal conversion has no real-life counterpart. The

authors were apparently unwilling to use binary numbers greater than 31, which effectively ruled out even the trivial mention of its connection to the ASCII code. Had they done so, students might have discovered binary patterns, such as the bit that distinguishes uppercase letters from lower-case ones. One wonders, then, about the authors' reasons for its inclusion. Those who experienced the "New Math" curricula of the 1960s might remember practicing similarly useless base-conversion lessons ("inspired" by computer architecture) that existed in isolation from the rest of the math curriculum. Even worse, in addition to practicing conversions with bases 2 and 8, exercises also included superfluous bases like 3-7 and 9.

As an alternative, what might be the objection to postponing CS study proper until later grades, and instead direct efforts towards students acquiring proficiencies in Computer Literacy (CL) areas. As it stands, there is already much room for greater rigor in CL education, as current curricula limit themselves to teaching office and multimedia productivity software and Internet skills. For a start, one might imagine including software applications that not only support, but extend what students already learn in their science and math classes, using educational software like *Geometer's Sketchpad* or *GeoGebra*, *Biology Labs On-Line*, and *Virtual Courseware for Earth and Environmental Science*, or actual science tools such as *SDSC Biology Workbench*, *MEGA6 (Molecular Evolutionary Genetics Analysis)* and the *NCBI* biomedical and genomic databases. By doing so, instructors might generate in their students an appreciation for the crucial role CS currently plays in subfields of other disciplines.

APPENDIX E

DIFFERENCES BETWEEN NATURAL AND PROGRAMMING LANGUAGES

Some of the differences between natural languages (NLs) and programming languages (PLs) are discussed below:

Vocabulary

Although NLs evolve over time, they have a large, fixed lexicon. PLs, on the other hand, contain an exceedingly small set of *reserved* words, but an infinite set of non-standardized *identifiers* (lexical items). "Non-standardized" means that names of specific syntactic items (variables, methods, classes, etc.) can be invented at the whim of the individual programmer, along with their meaning (semantics) within the context of the program.

Intelligibility

Utterances in a single NL are understood by all of its speakers. In contrast, a PL's non-existent lexicon means that programs written using the synthesized idiosyncratic vocabulary of one programmer may be unintelligible to other readers. This necessitates the teaching of compiler-unenforced programming style recommendations such as programming principles, program organization, naming and indentation conventions, and other types of guidelines.

Purpose

NLs facilitate two-way communication for any human purpose. A pidgin is a utilitarian, rump language made up of elements of two or more natural languages, spoken by no one as a mother tongue, and used as a common, intermediary language for either basic communication or a specific purpose (e.g. trade). A high-level PL occupies such a

middle space between human and machine language and functions primarily as a unidirectional pidgin for humans to write instructions for computers to compile and execute, with due attention given to writing a well-organized program that other humans can read with little difficulty. One discerns the quality of the program logic not by a language response, but by the computer's behavior (i.e. output). Runtime output by itself, however, provides no insight into the quality of the program's organization.

No Spoken Analogue

A NL is acquired without conscious effort simply by listening and speaking. Its written form is – at the most basic level – a notational system for encoding the spoken language²⁸. Although acquiring reading and writing skills requires substantial effort, the implicit precondition is that the corresponding spoken language already exists in a learner's brain. In an L2 learning process, although the spoken language does not already exist in a learner's brain, the student still has recourse to: (a) initially map L2 features and vocabulary onto an existing first language (L1) structure, and (b) use spoken L2 input as a source of new data in the building of writing and reading skills.

A PL has no spoken counterpart. Learning to read and write a language without speaking proficiency – and the accompanying and extensive prior knowledge of the language – is not a trivial task. The closest analogy is arguably Deaf students learning a hearing (non-signed) language. Deaf students in the United States learn English as a second language almost exclusively through literacy. Because they have limited or no access to auditory and oral data, they must instead mediate the building of English skills using their native language, American Sign Language (ASL). As might be expected of a

²⁸ A written language's most comprehensive form can be said to be an expansive version of the total linguistic knowledge of all of the speakers – past and present – of the standard dialect.

task that is many times more difficult than for their hearing counterparts, Deaf students have historically had low achievement in reading and speaking (as well as in mathematics, employment and earning levels) (Simms & Thumann, 2007). These difficulties echo the high failure rates of CS1 students. Interestingly, Deaf educators often refer to the situation of hearing ESL learners as analogous to that of Deaf students learning English (Berent, Kelly, & Porter, 2008). Nonetheless, a minority of Deaf people are able to achieve native or near-native fluency in written English. A strong correlation between proficiency in ASL and English literacy has repeatedly been demonstrated (Kuntze, 2004); and studies looking at the limitations of attempting to learn a writing system with no access to its language system have consistently emphasized the importance of "mastery of a primary language" (i.e. ASL) for achieving reading competence in a second language (Perfetti & Sandak, 2000). Qualitative research has identified possible conditions and strategies that may help Deaf children become literate, but no conjectures have been articulated to explain how this might occur (Mounty, Pucci, & Harmon, 2014).

A PL is Visual Language

If a PL has no spoken counterpart, that is, if it exists only in written form, then the corollary is that it is a visual language, like ASL. An alternate way of saying this is that a PL is a spatial language. ASL is mediated by a 3-dimensional space. A PL exists in two dimensions on a flat surface, and the spatial arrangement of components can encode meaning. The clearest example is using a systematic indentation scheme to delineate scope, mandatory in Python, and optional in languages that utilize braces to bracket blocks of statements, but which greatly enhances semantic clarity and organization.

One of the chief properties of space is direction, and ASL makes extensive use of this characteristic. One key use of direction is the representation of time: a forward motion of the palm signifies future tense while a backwards motion indicates past tense. Or, were a speaker to sign about an interaction with a friend, he/she might set up a point in space to represent the friend, then directionally sign the verb "give" by moving a hand from that point to oneself (or vice versa) to indicate the subject and object. In most PLs, the clearest use of direction is the assignment statement: a quantity evaluated on the right side of an equal sign is assigned to a variable on the left. Interestingly, the operators \leftarrow and \rightarrow in the R programming language allow for both leftward and rightward assignment, respectively.

Spatially Non-Linear

NLs consist of a temporally sequential phonological signal, that is, meaningful language units follow one after another. The signal in spoken NLs is auditory and the units are phonemes and morphemes. The visual signal in signed NLs is likewise sequential, but the language units are signs, and the phonological components are handshake, location, movement and orientation. What is different, however, is that signed NLs also utilize space. *"Time is arguably the primary foundation for audition, with sounds changing rapidly in particular ways over time, whereas space is the primary referent for vision, with visual objects defined by size and shape."* (Conway, Karpicke, & Pisoni, 2007).

The auditory signal in a spoken NL is temporally linear, as is its written form. The sequence of signs in a visual signal for a signed NL is also temporally linear, but spatially non-linear as individual signs are constructed utilizing all directions of a 3-D

space. Likewise, reading a PL is temporally linear. However, they are also spatially non-linear because programs can be (should be) organized hierarchically,. Method calls, for example, require positional jumps to other locations, with eventual returns to the jumping-off point. The same might be said to a lesser degree of loops, particularly when they contain conditional statements.

Precision

Because much of the meaning in speech is derived from context, NLS can be spoken imprecisely and ungrammatically – with half-completed sentences and changes of direction – and yet still be comprehensible. Although the intent of a program may be completely clear to other programmers, minute syntax errors will introduce either logical or compiler errors. Certainly intelligent compilers can be built to correct many such errors, but as things currently stand, machines cannot implement a programmer's intent or meaning without properly formed syntactic markers. This does not mean, however, that one programmer's bug-filled code will necessarily be unintelligible to another.

Semantics

The number of syntactic features in NLS is *at least* an order of magnitude larger than the number of syntax structures in PLs, which number around a dozen. Syntactic elements also carry a semantic component. In English, for example, the suffix "-ed" and its variants are verbal markers for past tense. Because syntactic components in a PL are much fewer in number, they are semantically much more diffuse and ambiguous. As an example, the meaning of an if-statement is the overly broad concept "conditionality".

One mechanism PLs employ to compensate for their amorphous semantics and to narrow down statements to specific functionalities/meanings is by assembling

combinations / blocks composed of primitive syntactic components. For maximum intelligibility, these blocks may be encapsulated into methods with user-defined names and reused as if they were primitive elements themselves. PLs thus not only allow for the creation of vocabulary items, but for new syntax elements as well.

Due to the strikingly different ways that NLs and PLs handle syntax, there are virtually no syntactic components in NLs that have PL counterparts. The impact on students is that one very early, crucial and often-used L2 learning strategy – reference to analogous syntax structures in L1 – is simply not available to students learning a PL. Each and every syntax component in a PL is foreign in all senses of the word.