Blocking Progress? Transitioning from Block-based to Text-based Programming

David Weintrop[*], Connor Bain[#], Uri Wilensky[#]
dweintrop@uchicago.edu, connorbain2015@u.northwestern.edu, uri@northwestern.edu
Center for Connected Learning and Computer-based Modeling
* UChicago STEM Education, University of Chicago
# Learning Sciences & Computer Science, Northwestern University

Abstract
Block-based programming languages are becoming increasingly common in introductory computer science classrooms, from kindergarten through grade twelve. One oft-cited justification for this switch to block-based environments is the idea that the concepts and practices developed using these introductory tools will prepare learners for future computer science learning opportunities. In this paper, we present data investigating how a five-week introductory curriculum in which high school students used either a block-based, a text-based, or a hybrid blocks/text programming environment, affects how the students approach the more traditional Java curriculum that immediately follows. Using a quasi-experimental design, we show that there are subtle differences in programming practices between students that worked in block-based environments compared to students who worked in isomorphic text-based and hybrid blocks/text environments. This paper also provides an example of using computational logging as a methodological approach for understanding emerging programming practices.

Introduction

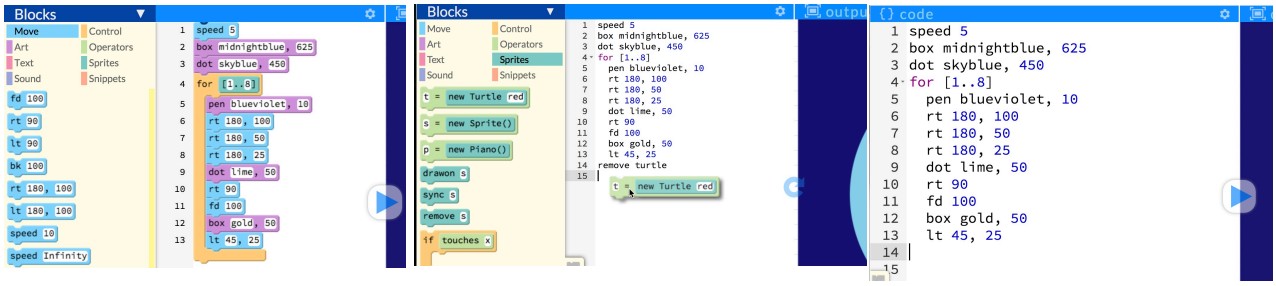
Block-based programming is quickly becoming the way that younger learners are introduced to programming and to computer science in general. Led by the popularity of tools such as Scratch (Resnick et al., 2009), Snap! (Harvey & Mönig, 2010) and Code.org's suite of Hour of Code activities (*Hour of Code*, 2013), millions of kids are engaging with programming through drag-and-drop graphical tools. Due in part to the success of such tools at engaging novices in programming, these environments are increasingly being incorporated into curricula designed for high school computer science classrooms, like Exploring Computer Science (Goode, Chapman, & Margolis, 2012), the CS Principles project (Astrachan & Briggs, 2012), and Code.org's curricular offerings (*Code.org Curricula*, 2013). Many uses of block-based tools in formal educational contexts presuppose that such tools will help prepare students for later instruction in text-based languages (Armoni, Meerbaum-Salant, & Ben-Ari, 2015; Brown, Mönig, Bau, & Weintrop, 2016; Dann, Cosgrove, Slater, Culyba, & Cooper, 2012; Powers, Ecott, & Hirshfield, 2007). While work has been done focusing on learning with block-based tools (e.g. Franklin et al., 2017; Grover & Basu, 2017; Weintrop & Wilensky, 2015), little empirical work has rigorously tested the transition to text-based languages in classroom settings. This question is of great importance given the growing role of block-based tools in K-12 education. As one student in the study said: "*I can guarantee that the transition between languages will be hard to do.*"

This paper seeks to understand if and how introductory programming tools (block-based, text-based, and hybrid blocks/text) prepare learners for the transition to conventional text-based languages. To answer this question, we conducted a quasi-experimental study in three high school computer science classrooms, comparing isomorphic block-based, text-based, and hybrid

block-based programming environments. To understand the differences, we investigate programming practices (like frequency of compilation and size of changes between consecutive compilations) and patterns of errors in authored programs.

Data Collection

The data presented in this paper are part of a larger study comparing introductory programming modalities at a selective enrollment public high school in a Midwestern city (Weintrop, 2016). We followed students in three sections of an elective introductory programming course for the first 15 weeks of the school year. Each section spent the first five weeks of the course working in one of three introductory programming environments: a Text condition, a Blocks condition, and a Hybrid condition (in this condition, students could browse through a Blocks library, but when dropped on the canvas, these blocks morphed into text) (Figure 1). All three environments are extensions of the Pencil Code platform (Bau, Bau, Dawson, & Pickens, 2015). Each class then spent the remaining 10 weeks programming in Java. A single teacher who had previously taught the course many times before taught all three sections.



(a) (b) (c)

Figure 1. The three forms of the programming environment used in the study: (a) block-based, (b) hybrid blocks/text, and (c) text.

In total, 90 students participated in the study (approximately 30 students per section), with 75 students identifying as male and 15 identifying as female. The students participating in the study were 41% White, 27% Hispanic, 11% Asian, 10% African American, and 11% Multi-racial and ranged from freshman to senior in age. Roughly half of these students spoke a language other than English in their households and 58.6% of the school-wide student body comes from economically disadvantaged households.

This paper analyzes the programs written by students in the Java portion of the class to understand the lasting impact the various introductory modalities had on students' programming ability. Each classroom computer was instrumented so that a call to compile a program[1] would send a copy of that Java program, along with the complier output, to a remote server controlled by the researchers. In this way, each student-authored program and each run of the program was logged. The following analysis, broken down into a series of questions, uses this data to explore different aspects of the stated research question.

Findings

[1] This was a call to `javac` executed from the command line by the student.

The findings section of the paper is broken down into a number of guiding questions revealing different aspects of learner practices and outcomes as they transitioned from introductory tools to the Java programming language.

Are there differences in how often students attempted to run their programs?

Students in the Blocks condition ran the `javac` command an average of 142.3 times (SD = 67.1). The same statistic for the Text condition was 130.9 (SD = 61.1) and for the Hybrid condition was 150.9 (SD = 79.2) . The results of an *F*-test (F(2, 80) = .594, p = .55) on the three samples shows no statistical significance, meaning in aggregate, there was no difference in the number of calls to `javac` based on the introductory modality students used. Figure 2 shows the average number of compilations for each student per day across the three conditions by week[2]. This chart includes both successful compilations as well as calls the resulted in an error.

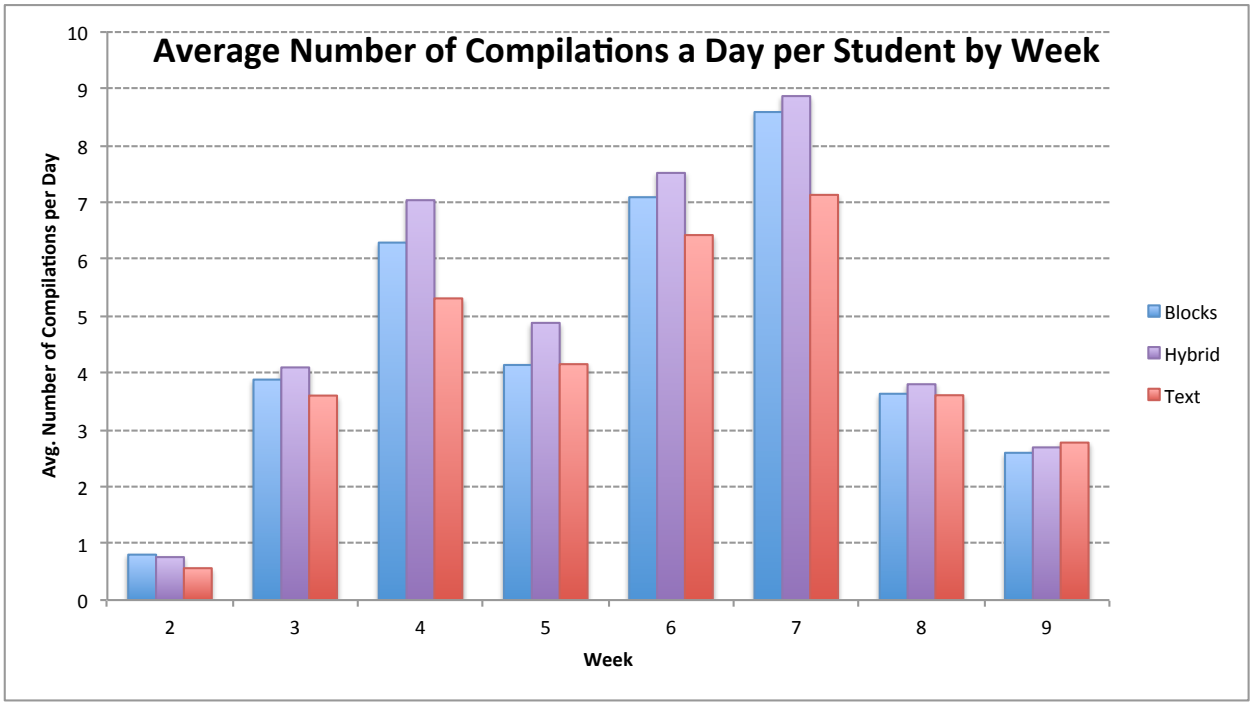


Figure 2. The average number of javac calls per student per day grouped by week.

This figure shows spikes (like weeks 4 and 7) and dips (like in weeks 5, 8, and 9), over the course of the ten-week curriculum. The shifts in per-week runs from week-to-week are largely explained by the in-class activity for the week. For example, in week 7, students were introduced to the `char` variable type through an assignment where they were asked to write a short program, then try and run it will different values to see what would happen. As a result, there was a spike in week 7 as these types of assignments (that would have student call `javac` over and over again) were not the norm.

[2] Unless otherwise specified, all charts in this section show per-student averages to control for the fact that not all classes had the same number of students. In addition, the charts do not include week 1 and week 10 because students did not compile or run any programs in those weeks.

Are there differences in how error prone/successful the students' compilations are?

While the previous analysis included all calls to `javac`, we now separate these calls into 'successful' and 'failed' in order to look for systematic patterns across the three conditions. Figure 3 uses the same compilation data as the previous chart, but now only includes successful compilations. The pattern in this figure largely matches the data from the previous figure.

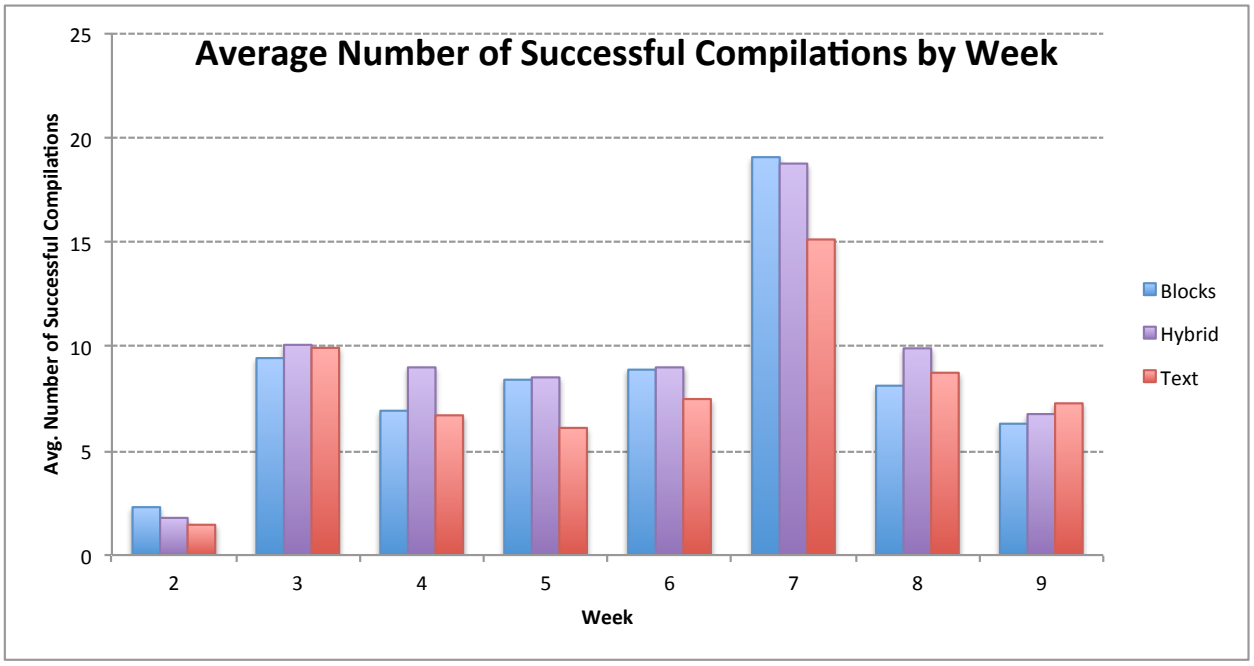


Figure 3. Average number of successful compilations by condition.

Students in the text condition frequently had the lowest average number of successful compilations, while students from the Hybrid condition showed the highest number of successful compilations. Since the students were all working on the same assignments in the same programming language (Java), the difference in successful runs does suggest a subtle difference in programming practice based on introductory modality. The higher frequency of successful compilations of Block and Hybrid students implies these students had higher levels of success in writing syntactically valid programs. One explanation for this pattern is that students in these conditions were more likely to compile their program at intermediate steps along the way. In other words, as they were writing their program, they would check to see if the portion they had written was correct before continuing with the next portion. A second potential explanation is that students these conditions ran their programs more frequently once they were completed, thus inflating their successful compilation counts. A third possible explanation is as simple as the fact that students in the Text condition did not call `javac` as often as students in the other two classes. In fact, it is possible Text students took longer to author programs in the first place which may have caused them to spend less time exploring and playing with the program they had authored.

Are there differences in the amount of code added in between successful compilations?

Along with compilation patterns, we are also interested in composition patterns. To measure the distance between two programs, we use the Levenshtein distance between the texts of the two programs. Levenshtein distance captures the minimum number of single-character

edits (i.e. insertions, deletions or substitutions) required to change one string into the other. Table 1 shows the results of this analysis. The columns capture the size of the Levenshtein distance between consecutive successful programs, while the cells show the average number of occurrences of that distance per student. The lower the number, the less often a program with that distance from the previous successful compilation was run by a student. For example, the left-most column that contains numbers shows that, on average, students in the Blocks condition complied a program that was identical to the last program they complied 7.00 times over the course of the 10 weeks, while the Hybrid condition recompiled programs an average of 7.40 times and the Text condition only did this 6.16 times.

Table 1. The frequency of successful compilations with a given Levenshtein distance from the last successful compilation of the same program.

	Levenshtein Distance								
	0	1	2	3	4	5 - 10	11 - 25	26 - 100	> 100
Blocks	7.00	3.37	5.70	1.33	2.37	4.30	3.52	6.56	3.33
Hybrid	7.40	3.68	5.88	1.84	2.60	4.64	4.72	6.48	3.76
Text	6.16	3.00	5.58	1.23	2.13	3.77	3.48	5.87	2.55

The students in the Text condition made fewer large changes to their programs and also re-ran their programs without making any changes less often than the other two conditions. These numbers tell the same story as Figure 2, which shows the Text group to have called `javac` least often. The most likely explanation for this is that students in the Text condition are slower to author programs in Java, but this study did not collect keystroke data, which is the data source needed to provide strong evidence for this outcome. The data does not show that fewer compilations is a result of the students making larger sets of changes between runs.

Are there differences in the number and type of errors encountered?

Just as we can glean information from patterns found in successful compilation calls, we can also look at novice programmers emerging understanding by looking of programming at the errors they encounter. Every time a student makes a call to `javac`, our system records all errors reported by the Java compiler. The plurality of compilation errors produced by the Java compiler has been documented as both a source of difficulty for novices (Nienaltowski, Pedroni, & Meyer, 2008; Traver, 2010) as well as an opportunity for improving introductory programming environments (Flowers, Carver, & Jackson, 2004; Hristova, Misra, Rutter, & Mercuri, 2003). Table 2 provides on overview of the frequency of failed compilations per student as well as information about the number of errors per failed javac call broken down by condition.

Table 2. High-level descriptive patterns of failing compilations and errors over the course of the entire 10 week period.

	Failed `javac` calls per student	Compilation errors per student	Compilation errors per failed `javac` call
Blocks	75.11	165.78	2.23
Hybrid	80.04	212.04	2.5
Text	69.55	164.26	2.21

Unfortunately, the compiler often does not (and at times cannot) provide meaningful error messages to the programmer. For instance, a missing ';' could result in the error message "`expected ';' on line 11`" or by the rather generic message "`not a statement`". In addition, many error messages are class specific (e.g. "`Class names, 'VarRefConcate', are only accepted if annotation processing is explicitly requested`"). In order to make the analysis more meaningful, errors were grouped into more broadly defined error types. For example, the class name error above was classified as an "`Incorrect javac Call`" as that is the most common cause of that particular error.

Figure 4 shows the ten most frequently found errors encountered, grouped by condition. The values in this chart are reported on a per-compilation basis to control for how often students chose to compile as well as the fact that the three conditions did not have the same number of students.

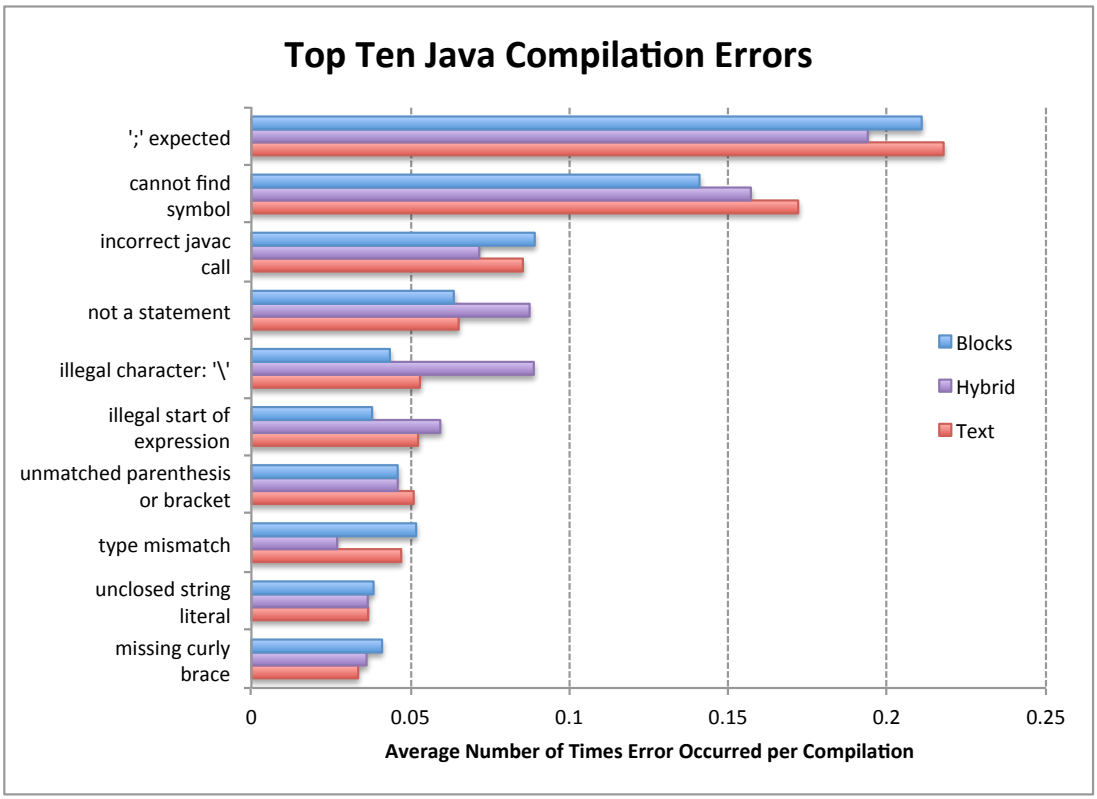


Figure 4. The ten most frequently encountered Java errors, grouped by condition.

The most common error was: "`';' expected`", which is seen when students forget to end a statement with a semi-colon, a syntactic requirement of Java. The second most common error: "`cannot find symbol`", occurs when students try and use a variable before it has been defined. Neither of these two errors are possible in the introductory modality, as semi-colon terminators were not required and variables do not need to be instantiated before they are used. It is important to note that regardless of introductory modality, novices frequently encounter these two errors. Both Jadud (2005) and Jackson et al. (2004) identified these two mistakes as the most frequently encountered in their data.

Looking across the ten errors, we see that half of the ten most frequently occurring errors were seen least often by students in the Blocks condition. One possible explanation for this

outcome is that because Java code is so unlike the block-based modality the students had spent the first five weeks working in, they were more attentive to the specific syntax they were being forced to use. Text and Hybrid students on the other hand, were already accustomed to manipulating a text-based language, but were used to an entirely different syntax, so they may have assumed a higher level of similarity across the text-based languages resulting in more errors.

Conclusion

While block-based languages have exploded in popularity, little work has been done to show that students learning in these environments are effectively transitioning to more traditional text-based languages like Java. This paper is meant to be a step towards trying to understand how the modality a learner uses in an introductory course impacts their approach to programming in a professional language. Even with the relatively small sample size, we were able to take computationally logged data and qualitatively analyze it in order to produce several reasonable explanations. Our analysis shows minor deviations in the programming patterns adopted by novices as they transition from introductory tools to Java based on modality, but few large or statistically significant patterns. This suggests the differences that emerge are more nuanced and more-learner specific than can be captured in the coarse grained analysis provided here. Nevertheless, this paper provides evidence towards a counter-narrative suggesting that one modality is not inherently 'better' than another. Instead, modality (block-based, text-based, or hybrid blocks/text interfaces) is just one dimension of a more complex learning context that shapes learners emerging attitudes and understandings.

References

Armoni, M., Meerbaum-Salant, O., & Ben-Ari, M. (2015). From Scratch to "Real" Programming. *ACM Transactions on Computing Education (TOCE)*, *14*(4), 25:1-15.

Astrachan, O., & Briggs, A. (2012). The CS principles project. *ACM Inroads*, *3*(2), 38–42.

Bau, D., Bau, D. A., Dawson, M., & Pickens, C. S. (2015). Pencil Code: Block Code for a Text World. In *Proceedings of the 14th International Conference on Interaction Design and Children* (pp. 445–448). New York, NY, USA: ACM.

Brown, N. C. C., Mönig, J., Bau, A., & Weintrop, D. (2016). Future Directions of Block-based Programming. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (pp. 315–316). New York, NY, USA: ACM.

Code.org Curricula. (2013). Code.org. Retrieved from http://code.org/educate

Dann, W., Cosgrove, D., Slater, D., Culyba, D., & Cooper, S. (2012). Mediated transfer: Alice 3 to Java. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (pp. 141–146). ACM.

Flowers, T., Carver, C. A., & Jackson, J. (2004). Empowering students and building confidence in novice programmers through Gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual* (p. T3H/10-T3H/13 Vol. 1). https://doi.org/10.1109/FIE.2004.1408551

Franklin, D., Skifstad, G., Rolock, R., Mehrotra, I., Ding, V., Hansen, A., Weintrop, D. & Harlow, D. (2017). Using Upper-Elementary Student Performance to Understand Conceptual Sequencing in a Blocks-based Curriculum. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 231–236). New York, NY, USA: ACM. https://doi.org/10.1145/3017680.3017760

Goode, J., Chapman, G., & Margolis, J. (2012). Beyond curriculum: the exploring computer science program. *ACM Inroads*, *3*(2), 47–53.

Grover, S., & Basu, S. (2017). Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic (pp. 267–272). ACM Press. https://doi.org/10.1145/3017680.3017723

Harvey, B., & Mönig, J. (2010). Bringing "no ceiling" to Scratch: Can one language serve kids and computer scientists? In J. Clayson & I. Kalas (Eds.), *Proceedings of Constructionism 2010 Conference* (pp. 1–10). Paris, France.

Hour of Code. (2013). Code.org. Retrieved from http://code.org/learn

Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. In *ACM SIGCSE Bulletin* (Vol. 35, pp. 153–156). ACM.

Jadud, M. C. (2005). A first look at novice compilation behaviour using BlueJ. *Computer Science Education*, *15*(1), 25–40.

Nienaltowski, M.-H., Pedroni, M., & Meyer, B. (2008). Compiler error messages: What can help novices? In *ACM SIGCSE Bulletin* (Vol. 40, pp. 168–172). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=1352192

Powers, K., Ecott, S., & Hirshfield, L. M. (2007). Through the looking glass: teaching CS0 with Alice. *ACM SIGCSE Bulletin*, *39*(1), 213–217.

Resnick, M., Silverman, B., Kafai, Y., Maloney, J., Monroy-Hernández, A., Rusk, N., … Silver, J. (2009). Scratch: Programming for all. *Communications of the ACM*, *52*(11), 60.

Traver, V. J. (2010). On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction*, *2010*.

Weintrop, D. (2016). Modality Matters: Understanding the Effects of Programming Language Representation in High School Computer Science Classrooms. Ph.D. Dissertation. Northwestern University, Evanston, IL.

Weintrop, D., & Wilensky, U. (2015). Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (pp. 101–110). New York, NY, USA: ACM. https://doi.org/10.1145/2787622.2787721