The Education Column

by

Juraj Hromkovič and Dennis Komm

ETH Zürich, Switzerland and PH Graubünden, Chur, Switzerland
juraj.hromkovic@inf.ethz.ch and dennis.komm@phgr.ch

The Problem with Debugging in Current Block-based Programming Environments

Juraj Hromkovič and Jacqueline Staub
Department of Computer Science, ETH Zürich
Universitätstrasse 6, 8092 Zürich, Switzerland
{juraj.hromkovic, jacqueline.staub}@inf.ethz.ch

Abstract

Programming is a highly creative activity that cultivates problem-solving skills, but also requires a high degree of precision. The Logo philosophy empowers novice programmers to become successful problem solvers who are capable of dealing with mistakes. The widespread emergence of block-based programming languages has lead to an active prevention of certain classes of errors while others still prevail. Rather than providing support, most block-based interfaces, to some extent, abandon learners in the difficult task of troubleshooting. We present a block-based programming environment that supports autonomous troubleshooting. Programming competences are not restricted to writing programs only. Correcting, modifying, and extending the functionality of previously-written programs is equally important and should not be neglected. Learning by productive failure is an unavoidable part of education.

1 Why Debugging is Important

Programming is a creative activity that develops constructive problem-solving skills and which finally has found its way into public school education. By programming, students communicate with a machine and instruct it what to do. For this purpose, they learn a programming language which essentially is a language that the machine “understands” [20]. Unlike humans, however, machines are not able to interpret ambiguities or handle imprecise statements. As a consequence, even the smallest flaws can cause a program to result in unexpected behavior or fail entirely. Making mistakes is natural and troubleshooting is one of the most crucial parts of a programmer’s skill set [13].

Under the term *debugging* we understand a systematic process that is aimed at finding the reason why a given program does not work as intended and fixing

the underlying cause [8]. Although the primary goal of debugging is simply to fix bugs, we consider the path of reaching this goal equally important: while analyzing novices' debugging patterns, Perkins et al. [18] found that tinkering is a widespread practice among novice debuggers. Generally, novices' debugging process often seems to lack systematicity [14] and especially locating errors is challenging for inexperienced programmers [13]. Fostering debugging skills explicitly, on the other hand, has shown to have a positive impact on learners' conceptual understanding of programming [2, 9].

The birth of the Logo programming language in 1967 marked the dawn of a profound philosophy of debugging for educational purposes which we will subsume under the term *Logo philosophy* [1, 6]. Papert expressed a meta-cognitive vision for programming education that emphasizes the need for learners to "rethink their thinking and to learn about their own learning" [16]. Debugging is an essential component of this philosophy as the following *Mindstorms* excerpt shows [17]:

> What we see as a good program with a small bug, the child sees as "wrong", "bad", "a mistake". School teaches that errors are bad; the last thing one wants to do is to pore over them, dwell on them, or think about them. [...] The debugging philosophy suggests an opposite attitude. Errors benefit us because they lead us to study what happened, to understand what went wrong, and, through understanding, to fix [the program]. Experience with computer programming leads children more effectively than any other activity to "believe in" debugging.

Confronting students with problems and allowing them to fail is relevant far beyond the scope of programming and computer science only. Van Lehn et al. [12] provide strong evidence for delaying external support during instruction and letting students learn by failure. This form of impasse-driven learning is associated with concept understanding [22]. While students are trying (and failing) to solve unstructured problems, they build relevant mental models that can later be used for more efficient direct instruction. This concept of *productive failure* has shown clear learning benefits for students [7].

Programming provides a good framework for students to learn by means of the productive failure approach. Programs are sequences of commands that are executed one after the other and thus the effect of a program can be anticipated and planned. Programs are typically written with a clear expectation of what exactly will happen during execution. Due to mistakes, however, reality and expectation may diverge unexpectedly. Quite often, this outcome puzzles programmers and leaves them with the challenging task of finding out where an error crept in. Narrowing down the exact location of an error has proven to be especially hard for novice programmers [13].

Scholars have tried different measures (e.g., pedagogical interventions via didactic methods, carefully chosen analogies, and technological tools) to support novices' troubleshooting and error localization throughout the past 50 years. This article highlights some technical debugging methods and contrasts them with today's common practice in block-based programming. We present a block-based programming environment that is specifically designed for novices to acquire debugging skills by handling logical errors. We intend to raise awareness and hope to spark more active research in the domain of debugging and block-based programming.

2 Two Different Perspectives on Debugging

Logo not only revolutionized the domain of programming education but it also brought forth a variety of debugging mechanisms, strategies, and tools that were devised with the intent of fostering novice programmers' debugging skills. With the dawn of block-based programming, debugging practices have shifted away from the deliberate and active exposure to errors towards a more preventive form of error handling. In this section, we present the key attributes of both of these perspectives and hint at their respective implications on novice programmers.

2.1 Debugging in Accordance with the Logo Philosophy

The term "Logo" characterizes not only the name of a family of programming languages but also a philosophical interpretation of how learning is best achieved [1]. This philosophy is deeply anchored in the ideas of *constructivist* learning by Jean Piaget (i.e., "learning by doing") and the *constructionist* visions of Seymour Papert (i.e., "learning by making"). Within the context of these theories, programming simply represents a *tool for learning*. The Logo philosophy is based on the rationale of making students iteratively construct, analyze, and refine their own learning artifacts. While programming, students develop creative solutions to given problems. They learn to formally describe the resulting algorithm as a program whose execution they can delegate to a computer. Errors can creep in at any point during this process and students must learn to deal with unexpected results.

2.1.1 The Logo-Way of Handling Errors

Programming errors can occur in virtually all conceivable situations and the underlying causes are manifold. With the Logo philosophy (or closely connected with it), numerous debugging mechanisms have been invented with the aim of supporting novice programmers in error handling and developing their debugging

skills. Three of these ideas (creating observable models of computation, providing advanced error diagnostics, and integrating process-related debugging mechanisms directly into the language) will be presented separately.

- **Introducing the Turtle: An Observable Model of Computation**

  Programmers need to express their ideas as sequences of commands that are subsequently interpreted and executed. Exactly how this interpretation and execution takes place, however, verges on pure magic for most novices. Numerous models of computation exist and some of them have been created specifically with the intent of helping novices peek inside this magic blackbox of program execution [3]. One of the first such models was the Logo turtle with which programmers can draw geometric shapes onto a screen. Thanks to the turtle's observable behavior, errors become obvious and programmers learn to systematically test their programs.

- **Developing Advanced Error Diagnostics for Novices**

  As many other programming languages, Logo knows at least three different types of errors: (i) Structural syntax errors arise due to grammatical inconsistencies that can be detected by the parser. (ii) Structural semantic errors are typically detected at runtime and lead to early termination of program execution. (iii) Logical semantic errors are neither detected by the parser nor at runtime and their characterizing feature is an unexpected program behavior. Several techniques allow novice programmers to cope with such errors more easily, from custom error messages, to in-line error highlighting, all the way to tailor-made debuggers.

- **Extending the Language with Built-In Debugging Mechanisms**

  Logo originally came with its own debugging vocabulary that was directly incorporated into the language. Via the `pause` command, for example, the execution of Logo programs could be interrupted at any point in time. This allows the existing program variables and intermediate results to be examined using an interactive mode of programming. The `trace` command is useful for tracing the program flow across multiple layers of abstraction. Several of these functionalities add to the underlying concept of *Read-Eval-Print-Loop* (REPL) that Logo inherited from Lisp.

In summary, the Logo philosophy is characterized by its wide range of different debugging mechanisms that supports troubleshooting with novice programmers. The focus of the Logo approach is on helping learners understand and make use of the complex and abstract concepts of automatic information processing.

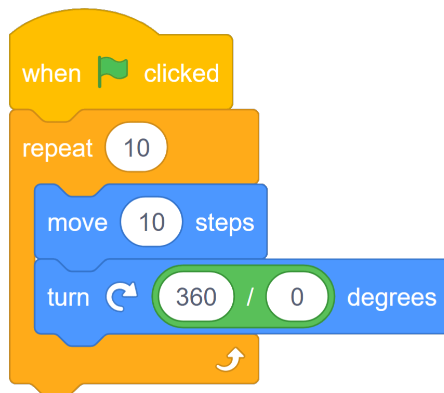

Figure 1: Example of a (silently failing) runtime error in Scratch. The result of a division by zero is mathematically undefined.

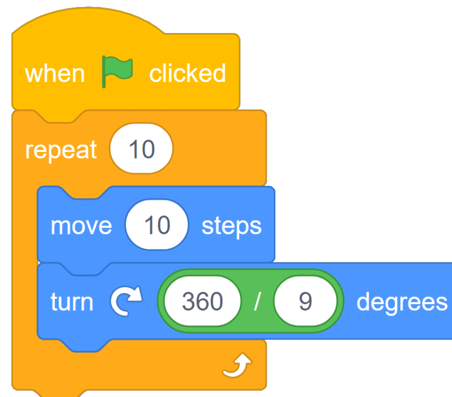

Figure 2: Example of a logical error in Scratch. In a decagon (polygon with 10 corners of equal angle), each angle must be 360/10 degrees.

2.2 Debugging and Block-Based Programming

After more than three decades, Logo had to slowly hand over its undisputed monopoly to the newer type of block-based programming languages; a class of languages that allows programmers to write programs by snapping command blocks in a graphical user interface. Using this feature, it is possible to prevent syntactic errors almost entirely from happening [10].

Although many block-based interfaces still contain traits of the Logo turtle, the approach of how block-based environments handle errors is different from the original Logo philosophy. Although certain classes of errors can be eliminated using a block-based interface, the issue of debugging is far from resolved: besides syntactic errors there are two more classes of errors which must be considered, independently of whether the chosen environment uses a block- or text-based interface, namely (i) logical errors and (ii) runtime errors. Both of these error classes are semantic in nature and cannot be handled considering only the syntactical attributes of a programming language. Figures 1 and 2 show two examples of such errors in Scratch. The example on the left results in a runtime error (i.e., division by zero is mathematically undefined) while the example on the right turns out to be logically incorrect.

One troubleshooting strategy used by many block-based environments consists in using "failsoft commands" which essentially result in runtime errors to be treated like logical errors. Rather than stopping execution, as is usually the case with runtime errors, many block-based programming environments swallow runtime errors and resume execution without informing the user. The concept of failsoft

commands became popular with the Scratch programming environment more than ten years ago [10]. By now, it is common practice among many well-known and broadly-used block-based programming platforms such as Snap, Code.org., and even Blockly. We question the usefulness of failsoft commands. Error localization is known to be one of the hardest parts of debugging. Intentionally hiding relevant information from users thus seems contra-productive from a debugging point of view.

3 Our Approach

Logo was originally designed to provide a broad variety of supporting features to facilitate error handling. This can be observed in a plethora of different mechanisms from syntax (e.g., Logo is particularly light on syntax with a bare minimum of syntactic elements [21]) to the role of the turtle [17] and the question of how to best phrase compiler error messages for beginners to understand [5, 19].

The argument for introducing block-based programming was inherently motivated by error handling, too. Syntax errors were claimed to distract novices from algorithmic aspects and for certain age ranges handling a keyboard was considered an additional challenge by itself. For those factors, block-based programming has proven to be a useful tool. Still, block-based programming cannot help against all errors and dedicated mechanisms for handling runtime errors and logical errors are still required.

XLogoOnline is a tailor-made programming environment for novice programmers from kindergarten to grade 6. Students at the lower end of this age range typically are not able to read and write yet and their formal education in mathematics has only just begun. In order to still allow students to dive into the world of algorithms and programming, XLogoOnline provides a block-based programming interface for students between kindergarten and grade 4. This interface was extended with three forms of support for troubleshooting: (i) a small command set was chosen not to admit any runtime errors, (ii) the environment includes the possibility for physical program execution to reduce struggles with change of perspective, and (ii) an exercise collection tool was developed which verifies student solutions and reports some logical errors. These three forms of troubleshooting will be elaborated individually.

3.1 Runtime Errors not Possible

Young children between 6 and 8 years old can learn to program despite their challenges in reading and writing and limited formal mathematics background. XLogoOnline provides a block-based interface which allows students to navigate a

turtle on a rectangular grid. For this purpose, students are provided with six block commands:

1. **forward** moves the turtle ahead. Each of these movements is unparameterized and uses an internal unit distance of 100 pixels, i.e., one grid cell.
2. **back** moves the turtle backwards. Each of these movements is unparameterized and uses an internal unit distance of 100 pixels, i.e., one grid cell.
3. **left** rotates the turtle 90 degrees to the left. The rotation angle is fixed and aligns with the underlying rectangular grid.
4. **right** rotates the turtle 90 degrees to the right. The rotation angle is fixed and aligns with the underlying rectangular grid.
5. **setpencolor** exchanges the pen with one of six possible colors. The selection involves a drop-down menu.
6. **repeat** allows any sequence of commands to be executed repeatedly. Students choose the number of repetitions by providing a positive integer parameter. Arithmetic operations are disallowed.

None of these six commands has the possibility to throw an unexpected runtime exception. This means that only logical errors can still occur and must be handled by programmers.

3.2 Physical Program Execution

One of the known logical difficulties Logo novices initially struggle with is connected to the concept of perspective. All of the four movement and rotation commands mentioned in Section 3.1 are interpreted from the perspective of the turtle, rather than the perspective of the programmer. This means that programmers have to imagine themselves in the position of the turtle in order to figure out whether to turn to the left or to the right.

This change of perspective is sometimes hardly even noticed by students, especially if the different perspectives align (as in Figure 3). There are, however, also cases which are notoriously hard for young programmers due to a clash of perspective. Figure 4 shows such a case where the ladybug's perspective differs from the reader's perspective. A clash of perspective may result in confusion between left and right.

In order to tackle this problem, we provide not only a virtual turtle but a physical turtle in addition as shown in Figure 8. This approach was originally proposed by Papert in the early days of Logo. Back then, the turtle was usually considered

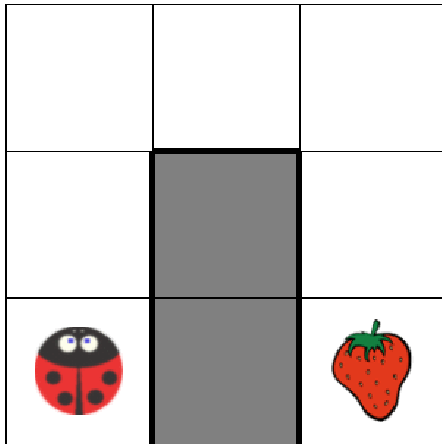

Figure 3: The change in perspective may not be noticed as much because the ladybug's perspective is the same as the programmer's perspective at the beginning.

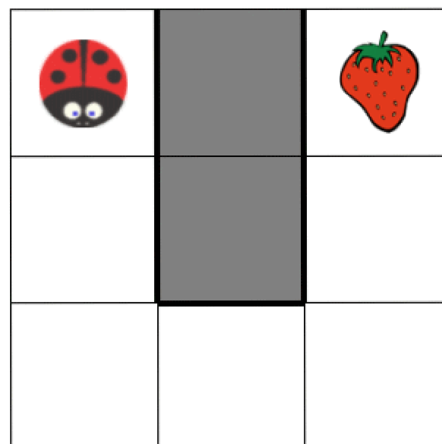

Figure 4: The ladybug's perspective deviates from that of the programmer. As a result, the required change of perspective entails additional cognitive effort.

a physical robot instead of a virtual agent on the screen. Physical robots have the advantage that a change of perspective comes naturally by simply changing one's position relative to the robot. This is something many young novices do unconsciously and which seems to improve their performance.

3.3 Exercise Collection with Automatic Verification

Programming is a precise form of communication; in order to instruct a computer to do something for us, we need to express ourselves in a way that is unambiguous and instructive enough for a mindless machine to "understand". Novice programmers must learn to deal with this required level of precision. One way of reaching this goal involves providing novices with age-appropriate exercises that challenge their conceptual understanding. We have developed an exercise collection with predefined tasks (some examples are shown in Figures 5 to 7). Student solutions to these tasks are automatically verified for correctness and the result is reported back the user.

Tasks differ in terms of what students are allowed to do in their solution. Some tasks contain fields that students are not allowed to cross. Forbidden fields may be marked visually sometimes (e.g., Figure 5) while other times, their status can only be inferred from the exercise text (e.g., Figure 6). In addition to forbidden fields, walls can be used to separate neighboring fields which would otherwise be

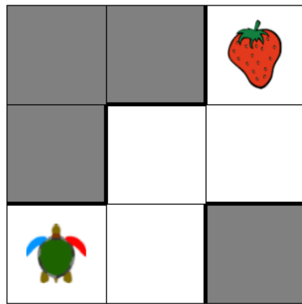

Figure 5: Find the strawberry without passing over a forbidden field.

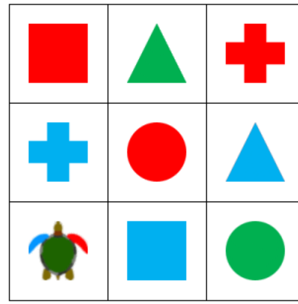

Figure 6: Collect all blue shapes without crossing over a red field.

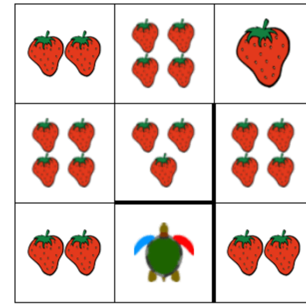

Figure 7: Collect 18 strawberries but without crossing a wall.

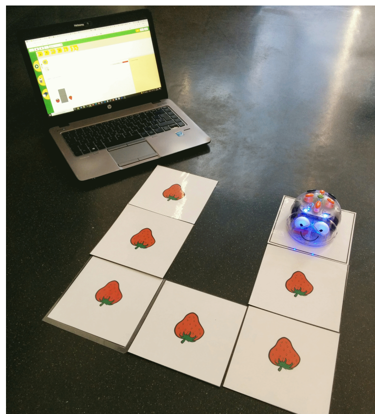

Figure 8: Students who struggle with changing perspectives benefit from executing their programs on a physical robot.

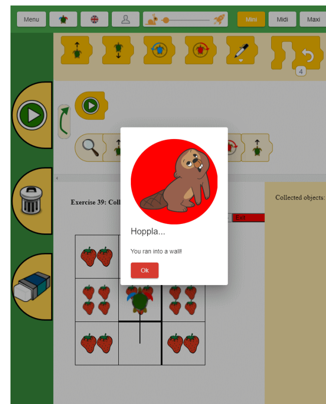

Figure 9: Error dialog that pops up whenever one of the underlying rules is violated.

connected. One such example is shown in Figure 7. Not respecting one of these conditions leads to an error dialog, as shown in Figure 9.

4 Summary and Conclusion

Not only programming itself but also dealing with errors needs to be learned. Block-based programming environments offer one-sided support that focuses mainly on the elimination of syntactic errors. There are several error classes (e.g., runtime errors and logical errors) that are not affected by the use of blocks and that are oftentimes not handled in block-based programming environments. Teaching pupils how to approach problems and making them learn from failure is of utmost

importance for pupils' cognitive development. Whether block- or text-based, it is crucial that learners leave school not as passive consumers of technology but that they learn to create their own solutions and deal with errors – one of the most natural ways of doing so consists in teaching them how to program.

References

[1] Harold Abelson. Logo for the Apple II *Peterborough, NH: BYTE/McGraw-Hill*, 1980

[2] C.-F. Chiu and H.-Y. Huang. Guided Debugging Practices of Game Based Programming for Novice Programmers. *International Journal of Information and Education Technology*, 5:343–347, 01 2015.

[3] B. du Boulay, T. O'Shea, and J. Monk. The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices. *International Journal of Man-Machine Studies*, 14(3):237–249, 1981.

[4] S. Fitzgerald, G. Lewandowski, R. McCauley, L. Murphy, B. Simon, L. Thomas, and C. Zander. Debugging: Finding, Fixing and Flailing, a Multi-Institutional Study of Novice Debuggers. *Computer Science Education*, 18(2):93–116, 2008.

[5] M. Forster, U. Hauser, G. Serafini, and J. Staub. Autonomous Recovery from Programming Errors Made by Primary School Children. In *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*, pages 17–29. Springer, 2018.

[6] Juraj Hromkovic, Dennis Komm, Regula Lacher, and Jacqueline Staub. Teaching with LOGO philosophy. In *Encyclopedia of Education and Information Technologies*, Springer, 2019.

[7] M. Kapur and K. Bielaczyc. Designing for Productive Failure. In *Journal of the Learning Sciences*, 21:45–83, 12 2012.

[8] C. Kim, J. Yuan, L. Vasconcelos, M. Shin, and R. Hill. Debugging During Block-Based Programming. *Instructional Science*, 46, 10 2018.

[9] M. J. Lee and A. J. Ko. Comparing the Effectiveness of Online Learning Approaches on CS1 Learning Outcomes. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, page 237–246, New York, NY, USA, 2015. Association for Computing Machinery.

[10] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick. Scratch: a Sneak Preview [Education]. In *Proceedings. Second International Conference on Creating, Connecting and Collaborating through Computing, 2004.*, pages 104–109. IEEE, 2004.

[11] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):1–15, 2010.

[12] K. VanLehn, Toward a theory of impasse-driven learning. *In Learning issues for intelligent tutoring systems* pages 19-41. Springer, New York, NY, 1988.

[13] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander. Debugging: a Review of the Literature from an Educational Perspective. *Computer Science Education*, 18(2):67–92, 2008.

[14] T. Michaeli and R. Romeike. Improving Debugging Skills in the Classroom: The Effects of Teaching a Systematic Debugging Process. In *Proceedings of the 14th Workshop in Primary and Secondary Computing Education*, WiPSCE'19, New York, NY, USA, 2019. Association for Computing Machinery.

[15] N. Nusen and A. Sipitakiat. Robo-Blocks: a Tangible Programming System with Debugging for Children. In *Proceedings of the 19th international conference on computers in education. Chiang Mai*, pages 1–5, 2011.

[16] S. Papert. *Teaching Children Thinking (LOGO Memo)*. Massachusetts Institute of Technology, USA, 1971.

[17] S. A. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic books, 2020.

[18] D. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons. Conditions of Learning in Novice Programmers. *Journal of Educational Computing Research*, 2:37 – 55, 1986.

[19] C. Solomon, B. Harvey, K. Kahn, H. Lieberman, M. L. Miller, M. Minsky, A. Papert, and B. Silverman. History of Logo. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020.

[20] J. Staub, M. Barnett, and N. Trachsler. Programmierunterricht vom Kindergarten bis zur Matura in einem Spiralcurriculum. *Informatik Spektrum*, 42(2):102 – 111, 2019-04.

[21] Jacqueline Staub Programming in K–6: Understanding Errors and Supporting Autonomous Learning. *ETH Zurich*, Thesis, 2021.

[22] C.-Z. Yen, P.-H. Wu, and C.-F. Lin. Analysis of Experts' and Novices' Thinking Process in Program Debugging. In K. C. Li, F. L. Wang, K. S. Yuen, S. K. S. Cheung, and R. Kwan, editors, *Engaging Learners Through Emerging Technologies*, pages 122–134, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.