Language Migration in non-CS Introductory Programming through Mutual Language Translation Environment

Yoshiaki Matsuzawa
Graduate School of Informatics,
Shizuoka University
3-5-1 Johoku, Nakaku, Hamamatsu
Shizuoka, Japan
matsuzawa@inf.shizuoka.ac.jp

Takashi Ohata
Graduate School of Informatics,
Shizuoka University
3-5-1 Johoku, Nakaku, Hamamatsu
Shizuoka, Japan
ohata@sakailab.info

Manabu Sugiura
Keio University
5322 Endo, Fujisawa
Kanagawa, Japan
manabu@sfc.keio.ac.jp

Sanshiro Sakai
Graduate School of Informatics,
Shizuoka University
3-5-1 Johoku, Nakaku, Hamamatsu
Shizuoka, Japan
sakai@inf.shizuoka.ac.jp

ABSTRACT

In the past decade, improvements have been made to the environments used for introductory programming education, including by the introduction of visual programming languages such as Squeak and Scratch. However, migration from these languages to text-based programming languages such as C and Java is still a problem. Hence, using the OpenBlocks framework proposed at the Massachusetts Institute of Technology, we developed a system named BlockEditor, which can translate bidirectionally between Block (the block language used here) and Java. We conducted an empirical study of this system in an introductory programming course taken by approximately 100 university students not majoring in computer science. When students were given opportunities to select the language to solve their programming assignments, we traced their selection by tracking working time with BlockEditor or Java for each individual student. The results illustrate the nature of the seamless migration from Block to Java, and show that there is great diversity in the timing and speed of migration to Java by each individual. Additionally, we found that students with low self-evaluation of their skill chose to use Block at a significantly higher rate than did students with high self-evaluation. This suggests that BlockEditor can act as scaffolding for students by promoting mixed programming between Block and Java in their migration phase.

SIGCSE'15, March 4–7, 2015, Kansas City, MO, USA.

http://dx.doi.org/10.1145/2676723.2677230.

Categories and Subject Descriptors

D.1.7 [**Programming Techniques**]: Visual Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques; K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

Design, Human Factors, Measurement

Keywords

Programming Education; Visual Programming; Language Migration; Scaffolding

1. INTRODUCTION

In the past decade, environments have been developed by researchers for the purpose of supporting introductory programming education. The most popular approach is to use a visual programming language such as Squeak [8] or Scratch [19]. The visual programming approach enables beginning learners to build their program by mouse operations, which eliminates typing and makes grammatical errors impossible. In this paper, we focus on the building-block approach [13], a visual programming approach exemplified by languages such as Squeak Etoys and Scratch (hereinafter, block languages' generally and Block for our specific environment). Using block languages, we can create programs by building with blocks [1] that represent elements of the programming language (e.g., commands, branching, and variable declaration), putting these blocks together in a jigsaw puzzle style [13]. Grammatical errors are prevented while programming in block languages because it is not possible

[1]In the Squeak community, what we called blocks here are called tiles [10]

to connect blocks unless those blocks can connect grammatically.

The main purpose of introductory programming courses is not to develop an understanding of the grammar of particular programming languages but rather to develop the computational problem-solving skills that facilitate so-called "computational thinking" [22]. In 2002, the United Nations Educational, Scientific and Cultural Organization described the task of such courses as "designing a task-oriented algorithm," where programming was defined in the following way.

> Programming at this level is not a technical subject. By and large, it means changing a task you can "do for your yourself" into one which can be "done by others". This means describing a task as a procedure in sufficient and complete detail so that another person or a device can perform it precisely and repetitively [20].

Researchers and developers of visual programming languages believe that the visual programming approach is an effective way to support learners in developing computational thinking during introductory programming courses. In theory, by using a visual approach, learners can focus on their problem-solving tasks and thereby come to understand programming concepts [5] more easily than with text-based programming. However, no clear evidence of the advantages has been shown in literature [11]. Additionally, there have been no reports that learners can successfully transform their visual programming skills into text-based programming skills.

Hence, in this research we propose an environment that supports learners in migrating from a visual language to a text-based language. The environment has two interfaces, one for a visual-block language (Block) and one for a text-based language (Java), as well as a system for bidirectional translation between Block and Java. We based the system on OpenBlocks[18] which was developed at the Massachusetts Institute of Technology (MIT). Learners can choose their language from between the two, and they can freely switch between languages to solve programming tasks. We hypothesized that learners would choose the Block language first; this language would then act as scaffolding [23] during their learning of programming, and students would gradually migrate to Java on their own schedule.

2. RELATED WORK

There have been many reports on using block languages for introductory programming education [12, 15], as well as reports on the development of new block languages [8, 19, 3, 2, 6]. Researchers agree that block languages feel familiar to beginners because they can avoid grammatical errors while programming. However, several researchers have proposed text-based educational programming languages, such as Dolittle [9]. These researchers argue that visual programming approaches have disadvantages for learners, and particularly that students have difficulty in moving to text-based programming after their introductory programming course.

There have been some studies on migration from block-based languages to text-based languages. Pasternak [17] proposed offering the ability to translate a program written in Block to a program written in Java. Google Blockly [6] can be translated into multiple languages (e.g., into Javascript or Python) from its block representation. In 2012, Dann et al. tackled the problem of figuring out how students can transfer their knowledge of Alice3 (a block-based language) to Java programming, and they reported that experience with a block language acted as helpful scaffolding for students learning to program in Java [4]. A key difference between our study and previous studies is that in those studies, the environments offered translation in only one direction: from block to text. We used a bidirectional translation environment. In 2008, Warth et al. proposed a bidirectional translation system, "TileScript," between a block language and JavaScript [21], although that research verified only that the technical requirements could be met to implement the prototype system. They did not propose a block editing system that could be used in actual education, nor did they evaluate the use of such a system.

The present research is an attempt to raise the "ceiling of programming" up to practical programming levels. Harvey et al. [7] proposed a functionality they called BYOB ("Build Your Own Block") in Scratch. This functionality expands the descriptive capabilities of the block language. The purpose of the project is to support a block language suitable for a wide range of users, from beginners to professionals. They stress the importance of structured programming, as discussed in the book *Structure and Interpretation of Computer Programs (SICP)* [1], and the support that block languages offer to structured programming. Although we fundamentally agree with the importance of structured programming, our suggested approach is to implement it by migrating to a language suitable for practical use.

3. TOOL DESIGN

3.1 Basic Function and Interface

We have developed a tool named BlockEditor. This tool was designed as a support for creating programs in the Block language, and it was embedded into the programming environment that we used for an introductory programming course whose target language is Java. The resulting environment has two interfaces, one for the (visual) Block language and one for the textual Java language, as well as a system for bidirectional translation between the two languages. Figure 1 shows the user interface for the environment.

The BlockEditor is shown at the left bottom of Figure 1. Users can build a program by using blocks in the center pane of the BlockEditor. Then, users can take blocks from the right pane of the BlockEditor. When a user presses the button labeled "save as Java," the Block program is translated to Java. The translated program is shown in the Java development window, which can be seen at the right top of the figure. When users edit the Java program that results from translation and then save it, the Java program is automatically translated back into a Block program and shown in the BlockEditor. The developed program can be compiled and run from either window.

3.2 Design of Block and Translation

We based the developed system on OpenBlocks [18], which was developed at MIT. We made some modifications to the original OpenBlocks in order to accommodate the Java translation. Although the translation function does not cover the entire grammar for Java, enough grammar is cov-

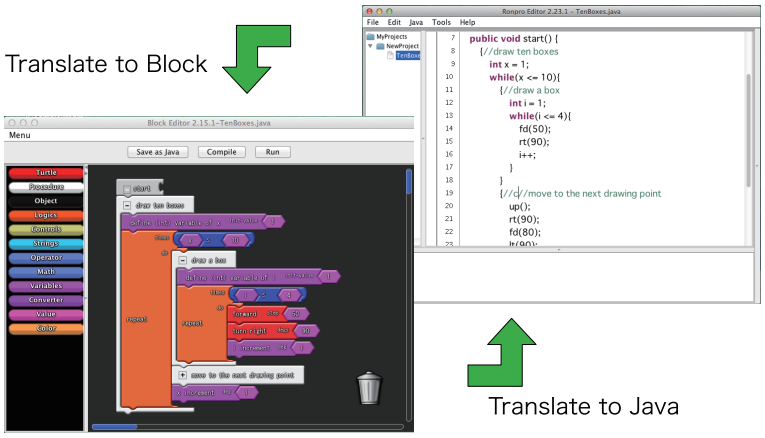

Figure 1: Overview of the proposed environment.

(a) context menu for variable definition

(b) context menu for object variable definition

Figure 2: Context menu interface of BlockEditor

ered for the tasks that are assigned to beginners in introductory programming courses.

Command blocks for Turtle Graphics [16] were designed to mirror those embedded in the sample of OpenBlocks. Because we developed the Turtle Graphics library in Java, the blocks for the Turtle Graphics library can be translated into Java. For example, as shown in Figure 1, the block "forward 100" can be translated into "fd(100);", and the block "turn right 90" can be translated into "rt(90);".

The primitive variable types available for blocks are the same as in Java. It is not typically considered necessary to distinguish between `int` and `double` in block languages, as a result of which only one number type is considered in general block languages such as Scratch and Squeak. However, unifying the numeric types is difficult because of technical considerations such as translation, and the intent is for learners to come to understand Java primitive types. The current version of the system provides the `int`, `double`, `boolean`, and `String` types as primitive types. We created a block type for variable definitions, and users can take obtain blocks for reference to the variable and for assigning values into the variable by using a pull-down menu, as shown in Figure 2(a)

Decision blocks (i.e., conditionals) and repetition blocks are each represented by only one type of block. This means that we convert all repetition structures in Java (e.g., `while` statements, `for` statements, and `foreach` statements) into a single block type, the "repeat" block, and conditional structures are similarly merged to one block type, the "conditional" block. Once the Java program is converted into the block language, information about the original representation is lost. Therefore, for example, if the user writes a program using a `for` statement in Java and this is subsequently translated into a repetition block, then translation back to Java will result in a `while` statement, even though the meaning and effects of the program have not changed.

The blocks to define functions (methods) have the same design as in the original OpenBlocks. Because OpenBlocks does not have blocks for object-oriented programming, we implemented a richer set of blocks to allow defining, creating, and using the objects. We can define variables for the defined types (classes), and we can obtain call-method blocks from a pull-down menu, as is shown in Figure 2(b).

The system has offers bi-directional translation between Block and Java. It is technically implemented through a block description file formatted according to the XML format specified for OpenBlocks. In the case of conversion from Block to Java, the system generates a description file for the blocks by calling a function in OpenBlocks. Then, the system reads the file and converts the result into Java code. In case of a conversion from Java to Block, the system converts the Java source into an abstract syntax tree; the system generates the block description file in the XML format defined for OpenBlocks. BlockEditor shows no blocks when a user tries to convert a Java program that contains syntax errors. Although we could build some of the blocks by ignoring parts of the errors, we thought that this might confuse learners about the errors.

4. EMPIRICAL STUDY METHOD

4.1 Research Question and Hypothesis

We conducted an empirical study in our introductory programming class. The goal of the study was to evaluate

the effectiveness of the proposed environment (BlockEditor). The research question was "How do students select their languages when given the opportunity to select the language they use to solve their programming assignments?" Students were given the opportunity to select the language they used to solve their programming assignments, which could be completed in Block or Java. We traced their selection by tracking working time with BlockEditor and with Java for each individual student. We began with two hypotheses:

Hypothesis I Learners will choose to use Block first, which will act as scaffolding [23] for learning programming, and then learners will gradually migrate to Java on their own schedule.

Hypothesis II Students with low self-evaluation of their skills will use Block at a higher relative frequency than students with higher self-evaluation.

4.2 Educational Environment Descriptions

An experimental evaluation of the tool was conducted throughout the introductory programming course at our university. The course was designed for art students, rather than for computer science students. Therefore, the objective of the course is to develop an understanding of task-oriented programming, which is independent from any language. Approximately 100 students participated in this course; two lecturers and six teaching assistants conducted the class.

In this class, all students were assigned 3–5 tasks per week. The given tasks were all to build a program (there were no paper tests, and no tasks such as filling in a missing part of a program). Students were given one of the following sets of instructions for how to solve each task:

- B (Block Task):Students MUST use BlockEditor to build a program.
- J (Java Task):Students MUST use Java to build a program.
- A (ANY):Students can select the language used to build a program, and this choice can be freely changed.

The number of assignment types for all weeks are shown in Table 1. We assigned at least one Block task and one Java task for each of the first 4 weeks in order to give students exposure to both languages; we assigned only ANY tasks in the last part of the course.

4.3 Method of Data Collection

The actual effort time was recorded and provided by PPV [14]. The working time was calculated by summing up the time and excluding periods of longer than 5 min with no user operation. We defined the two calculated times as follows:

Tb Working Time with BlockEditor

Tj Working Time with Java

Then, we calculated the BlockEditing Rate (Rb) by using the following equation.

$$Rb = \frac{Tb}{Tb + Tj} \quad (1)$$

Table 1: Curriculum of the introductory programming course.

Week	Contents	B	J	A
1	Setup & Guidance	-	-	-
2	Introduction of Turtle Graphics	1	1	3
3	Variables, Conditionals	1	2	2
4	Loops, Nested Loops	1	2	4
5	Creating Animation	1	2	2
6	Handling Key and Mouse Input	0	0	2
Middle	Your Own Product	-	-	1
7	Console & Calculation (Expense Sharing)	0	0	3
8	Console & Calculation (BMI Calculation)	0	1	5
9	Method (with Arguments)	0	1	3
10	Method (with Return Value)	0	2	4
11	Method (Recursion)	0	1	3
12	Collection	0	0	2
13	Sorting Algorithm	0	0	2
14	Dictionary	0	0	0
15	Your Own Product (in Teams)	-	-	-

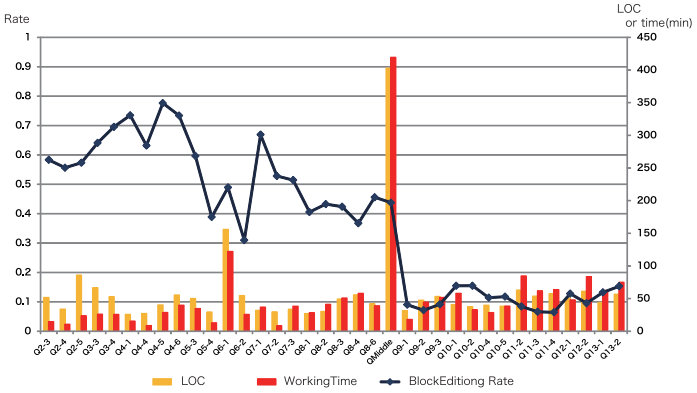

Figure 3: Relative rate of working with BlockEditor, total working time, and the lines of code (LOC) for each assignment.

5. RESULTS

5.1 BlockEditing Rate

The chart in Figure 3 shows the averages of the rate of BlockEditor use (Rb), lines of code (LOC), and the total working time (in minutes), for each task. All the ANY tasks assigned to students are shown in increasing chronological order.

The mean of Rb was 0.6 near the start of the course, 0.4 in the middle of the course, and 0.1 in the latter part of the course. The average rate was gradually decreasing, although the slope was steep down to around assignment Q9-1. This change is likely because QMiddle required a relatively longer program than the other tasks did. Students were asked to create their own game in the QMiddle assignment. As the average LOC reached 400, it became difficult to manage the program in BlockEditor.

We created a grid representation of the rate of BlockEdi-

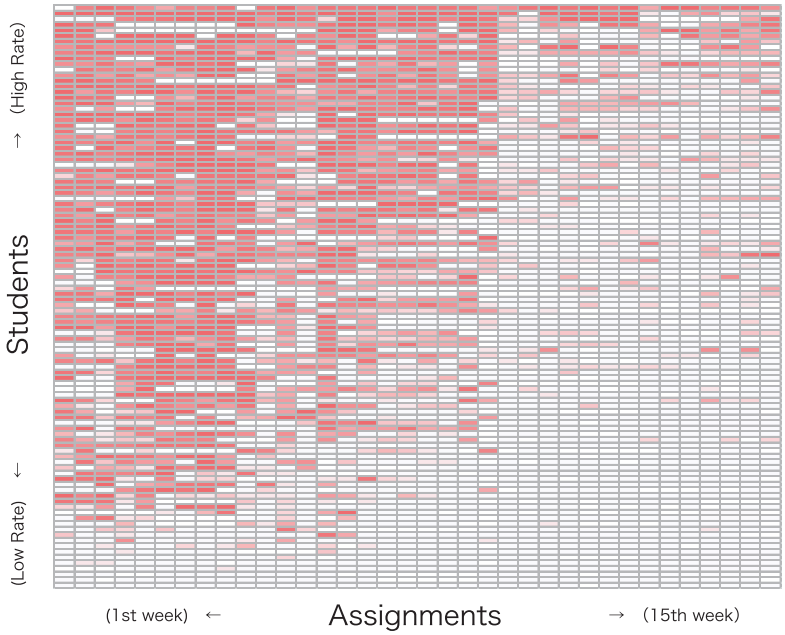

Figure 4: Grid representation of relative rate of working with BlockEditor for each student (color intensity indicates the rate of BlockEditor use).

tor use (Rb) to illustrate the nature of the seamless migration from Block to Java. The representation is shown in Figure 4. Each column represents one task assigned to students, arranged in chronological order. Each row represents one student; the rows are sorted by course-average Rb, with higher rows indicating higher average use. Each cell thus represents the value of Rb for a particular student completing a particular task. High-intensity color indicates a high Rb, and white indicates an Rb of 0 (i.e., using only Java).

We can observe that the bottom 10% of students (in terms of Rb) did not select BlockEditor at all, and the top 10% of students almost always used BlockEditor until the end of the course. The remaining 80% of students gradually migrated from Block to Java. We can observe the seamless migration both overall (by the number of students who used Java in the class) and at the individual level. Low-level rates at the individual level indicate that those students built their programs with both languages, which is noteworthy. The data show wide diversity in the schedule and speed of migration by each individual.

5.2 Self-evaluation and BlockEditor Selection

By questionnaire survey, we asked students to self-evaluate their programming skill level on a four-point scale (VG: very good; G: good; NG: not good; NGA: not good at all). We obtained answers for three levels (G, NG, and NGA); no student marked VG. We compared the three groups on block editing rate. The results are shown in Figure 5.

The descriptive statistics were as follows: G ($n = 11, sd = 0.25, avg = 0.25$), NG ($n = 47, sd = 0.20, avg = 0.41$), NGA ($n = 28, sd = 0.15, avg = 0.44$). Welch's t-test was conducted between groups G and NG and between groups G and NGA. The difference between G and NG was marginally significant ($t(13.29) = 2.04$, $.05 < p < .10$). The difference between G and NGA was significant ($t(13.15) = 2.39$, $p < .05$). These results show that students who evaluated their programming skills as low selected to use BlockEditor more often than did students who rated their skills as high.

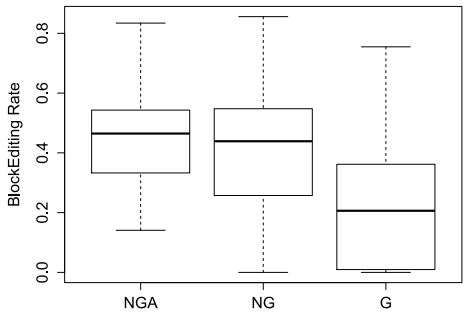

Figure 5: Distributions of rates of working with BlockEditor, by self-evaluated programming skill level.

6. DISCUSSION

The results indicate that the environments succeeded in promoting learners' seamless migration from Block to Java. Additionally, the results of the questionnaire given to students were similar. The illustration of the rate of selecting Block fades along the positive time axis, with the rate of use gradually dropping. These results therefore provide evidence that a block language can successfully act as scaffolding for students learning text-based programming. One

of the considerations when verifying this result is to discern whether the students had the meta-cognitive skills necessary to select the language appropriate to their learning. We have assumed that the students did have sufficient knowledge both because they were university students and because we exposed the students to both languages before offering a choice. Many students used Block only partially, although the teacher did not explicitly teach this usage. We consider that is an evidence that the students were aware of their own level of understanding.

The results indicated that there is little relation between the rate of using BlockEditing and length of program, as well as little relation between the use rate and working time. Whereas the factor of the programming experience, and individual differences have a high relationship with the rate of use. Both Hypothesis I (that learners will choose a block language first and then gradually migrate to Java) and Hypothesis II (that the block language will be used relatively more by students with low self-evaluation of their skills) were supported by the results.

A limitation of this research is that we analyzed only the language selection rate. This is not direct evidence that the learners' understanding is being enhanced. However, the same course was taught in the previous year entirely in Java using the same content. In the range of our observations, we confirmed that students' understanding level between the two classes was equivalent (or better understood by the class that used Block). The negative atmosphere in the classroom was dramatically changed into a positive one. We strongly believe that the results will encourage teachers who use a block language in introductory programming education.

7. REFERENCES

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs – 2nd ed.* MIT Press, 1996.

[2] J. C. Cheung, G. Ngai, S. C. Chan, and W. W. Lau. *Filling the gap in programming instruction: a text-enhanced graphical programming environment for junior high students.* SIGCSE' 09 Proceedings of the 40th ACM technical symposium on Computer science education, New York, NY, USA, 2009.

[3] S. Cooper, W. Dann, and R. Pausch. Teaching objects-first in introductory computer science. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, SIGCSE '03, pages 191–195, New York, NY, USA, 2003. ACM.

[4] W. Dann, D. Cosgrove, D. Slater, D. Culyba, and S. Cooper. Mediated transfer: Alice 3 to java. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 141–146, 2012.

[5] M. Fal and N. Cagiltay. How Scratch Programming May Enrich Engineering Education. In *2nd International Engineering Education Conference(IEEC2012)*, pages 107–113, 2012.

[6] Google Inc. Blockly:a visual programming editor. `http://code.google.com/p/blockly/`, referenced at 2013.03.17.

[7] B. Harvey and J. Monig. Bringing“ no ceiling ”to scratch: Can one language serve kids and computer scientists? *Constructionism 2010, Paris*, 2010.

[8] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of squeak, a practical smalltalk writtern in itself. *Proc. of ACM OOPSLA' 97*, page 318, 1997.

[9] S. Kanemune, T. Nakatani, R. Mitarai, S. Fukui, and Y. Kuno. Dolittle - experiences in teaching programming at K12 schools. In *Second International Conference on Creating, Connecting and Collaborating through Computing(C5)*, pages 177–184, 2004.

[10] A. Kay. Squeak Etoys authoring & media. *Viewpoints Research Institute Research Note*, 2005.

[11] C. Lewis. How programming environment shapes perception, learning and goals: logo vs. scratch. In *Proceedings of the 41st ACM technical symposium on Computer science education (SIGCSE '10)*, pages 346–350, 2010.

[12] C. Lewis. What do Students Learn About Programming From Game, Music Video, And Storytelling Projects? In *Proceedings of the 43rd ACM technical symposium on Computer science education (SIGCSE '12)*, pages 643–648, 2012.

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

[14] Y. Matsuzawa, K. Okada, and S. Sakai. Programming Process Visualizer: A Proposal of the Tool for Students to Observe Their Programming Process. In *Innovation and Technology in Computer Science Education (ITiCSE '13)*, pages 46–51, 2013.

[15] D. Ozoran, N. Cagiltay, and D. Topalli. Using Scratch In Introduction to Programming Course for Engineering Students. In *2nd International Engineering Education Conference(IEEC2012)*, pages 125–132, 2012.

[16] S. Papert. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc., New York, NY, USA, 1980.

[17] E. Pasternak. Visual programming pedagogies and integrating current visual programming language features. Master's thesis, Carnegie Mellon University Robotics Institute Master's Degree, 2009.

[18] R. V. Roque. Openblocks: An extendable framework for graphical block programming systems. *Master thesis at MIT*, 2007.

[19] Scratch Team Lifelong Kindergarten Group MIT Media Lab. Scratch -imagine.program.share- `http://scratch.mit.edu/`.

[20] UNESCO. ICT Curriculum for School / Program of Teacher Development. `http://unesdoc.unesco.org/images/0012/001295/129538e.pdf`, 2002.

[21] A. Warth, T. Yamamiya, Y. Ohshima, and W. Scott. Toward a more scalable end-user scripting language. In *Proceedings Second International Conference on Creating Connecting and Collaborating through Computing 2008*, pages 172–178, 2008.

[22] J. Wing. Computational Thinking. *Communications of the ACM*, 49(3):33–35, 2006.

[23] D. Wood, J. S. Bruner, and G. Ross. The role of tutoring in problem solving. *Journal of child psychology and psychiatry*, 17(2):89–100, 1976.