The International Journal of E-Learning and Educational Technologies in the Digital Media (IJEETDM) 5(3): 93-101
The Society of Digital Information and Wireless Communications (SDIWC), 2019 ISSN: 2410-0439 (Online)

Design of a Programming Environment for Non-Procedural Programming Languages using Blockly

Yuya Sano and Koji Kagawa
Kagawa University
2217-20 Hayashi-cho, Takamatsu, Kagawa 761-0396, JAPAN
kagawa@eng.kagawa-u.ac.jp

ABSTRACT

When learners study their second or third programming language, they have to learn its basic concepts and its grammar at the same time in a short period. This is a heavy burden for them. We adopt Blockly, a library for constructing a block-type programming environment for beginners, and try to build a programming learning environment for languages other than procedural ones by using Blockly's dynamic transformation behavior of blocks.

KEYWORDS

Web, Blockly, JavaScript, Programming Language, Programming Paradigm

1 INTRODUCTION

It is a heavy burden for learners of programming to learn the concepts and the grammar of the language at the same time. To reduce this burden, we need a learning environment where they can program without paying too much attention to the grammar. For this purpose, we can use block-based programming environments [1]. Among several candidates such as Waterbear [2] and Droplet [3], we adopt Blockly [4] due to its rich set of blocks and extensive documentation. However, it cannot cope with several languages to be learned in the advanced university courses as it is. In addition, because there are limited numbers of blocks whose shape can be changed dynamically, the flexibility of programming languages will be constrained by the shape of blocks.

Several programming languages such as C, Java, JavaScript, Python, Haskell, Prolog, Bison, Flex, and so on are used to present algorithms in the undergraduate and the graduate courses of the computer science program of the authors' university. Each language has its own specific grammar. It takes a fair amount of time for learners to understand the grammar enough to write source code using only text editors.

This research tries to broaden the scope of learning support in Blockly, allowing learners to quickly acquaint with their second or third programming language. Since blocks tend to get deeply nested in non-procedural languages, we must consider changing the shape of blocks dynamically. This paper reports several attempts to adapt Blockly to non-procedural programming languages.

The rest of this paper is organized as follows. First, Section 2 introduces Blockly briefly. Then, Section 3 explains the features of the languages we will tackle and discuss the design of the new blocks. Section 4 explains feedbacks from volunteer users. Section 5 presents the design of new blocks introduced after the feedbacks. Finally, we conclude and discuss future directions in Section 6 and 7.

2 BLOCKLY

Blockly [4] is a library from Google for building beginner-friendly block-based programming environments. Like Scratch [5], programming is performed by connecting blocks. Therefore, learners can author programs intuitively without being bothered by syntax errors. Since it is written in JavaScript and accompanying documents are sufficiently prepared, it is easy to create customized blocks. In addition, since we can construct a Web-based application with Blockly, there is no need for learners to install the environment in advance. By the initially provided library, programs created with Blockly can be converted into source

codes of the following five programming languages: JavaScript, Dart, Python, Lua, and PHP, at the time of writing. Moreover, by providing code generators, it can be easily extended to other programming languages.

2.1 Structure of Blockly Applications

The user interface of a typical Blockly application (shown in Figure 1) consists of the three components: the block menu, the workspace, and the source code viewer. If you press the source code tab, it switches to the component of the source code viewer. The source code viewer is a space for displaying the text-based representation of the created program. The block menu part is the space where the offered blocks are presented, and is usually displayed on the left side of the workspace.

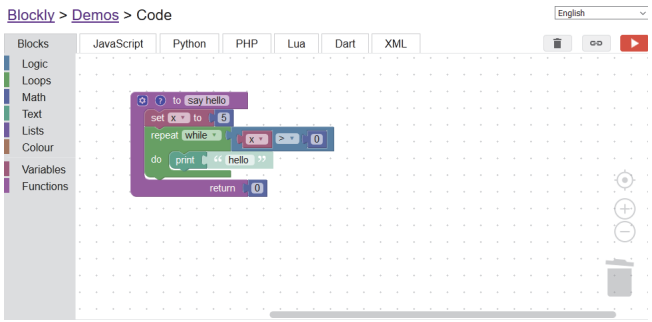

Figure 1. A Typical Blockly Application

Blockly blocks are roughly classified into two types of blocks (Figure 2). One is the type of blocks with a left connection (also called an output connection). This more or less corresponds to “expressions” in the programming language terminology. The other is the type of blocks with either a top or a bottom connection (also called a previous or a next connection). In many cases, a block has both a top and a bottom connection. This corresponds to “statements.” Therefore, in the following, we will use the term expression blocks and statement blocks, respectively.

Figure 2. An Expression Block (Left) and a Statement Block (Right)

2.2 Mutators

A mutator is an extension mechanism of Blockly to offer dynamic transformations of block shapes. It defines how to save the transformation of blocks in an XML format. A block that uses a mutator may have a gear-shaped icon at its upper left corner. When clicking the gear-shaped icon, a balloon-like dialog window appears nearby. The left half of the dialog window is a block menu and the right half is a mini-workspace. The screenshot at this moment is shown in Figure 2. The blocks in the dialog window determines the shape of the parent block.

If you extract a certain block from the menu component of the dialog window and attach it to the existing blocks of the mini-workspace component of the dialog, the parent block is transformed. Figure 3 shows the screenshot at that moment. As the shape of the block can be customized by mutators, the types of blocks that must be provided at the block menu component can be significantly reduced.

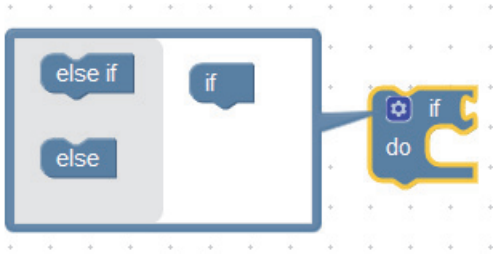

Figure 2. Mutator Block (Before Modification)

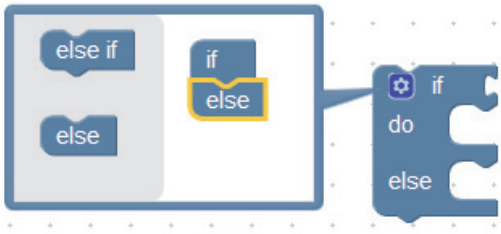

Figure 3. Mutator Block (After Modification)

3 DESIGN OF BLOCKS

In order to support C, Haskell [6], and Flex [7] by Blockly, we have made new block definitions. In this section, the newly implemented types of blocks are explained in more details.

3.1 Printf Block for C

The C language is a typical procedural language. Most of the original blocks of Blockly can be diverted straightforwardly. Then, Yamagata [8] designed some blocks for C to make backward translation from a C source code to a Blockly program possible. The blocks designed there include that for the printf function call. Since printf is a "varargs" function (i.e. a function which accepts a variable number of arguments), we must change the number of value inputs using the gear-shaped icon. A value input is a hole where you can insert an expression block. This is, however, rather tedious. Therefore, we have implemented a new mutation in the printf block for the C language, and enabled multiple arguments to be automatically embedded in a string to be output. It is offered in the input/output block category of the block menu. The image of the block is shown in Figure 4. The number of format specifiers (e.g. "`%d`", "`%f`") that start with the '`%`' character is detected in the string given as the first argument of printf, and the number of the value inputs is increased or decreased automatically. The detection is performed when the content of the string changes. Since users do not need to modify blocks by clicking on the gear-shaped icon, it saves time and effort on the dynamic transformation and simplifies the operation, thus reducing the burden on learners.

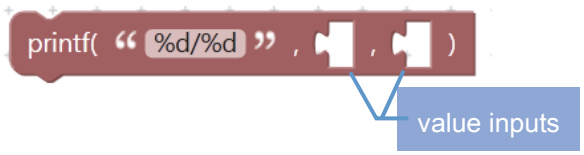

Figure 4. A Printf Block

3.2 Flex Blocks

Flex [7] is a lexer generator for C. (It should not be confused with Adobe Flex™, which is a software development kit based on Adobe Flash.) Flex generates a lexer written in C from pairs of a regular expression and a corresponding action described in C. For Flex, we have renewed the regular expression block based on the blocks implemented by Ozaki [9].

A new block for the character class is implemented with mutation, and it is offered in the regular expression block category of the block menu. The character class is an extended notation for regular expressions available in Flex and matches a single character out of several candidates. For example, we can use the notation `[abc]` to match an '`a`', a '`b`', or a '`c`', and use `[0-9]` to match a single digit. The negated character class is used to match a character that is not in the character class. For example, we can write `[^0-9a-fA-F]` to represent a character that is not used in the hexadecimal notation of integers.

It is a common misconception of regular expression beginners that they try to write regular expressions between '`[`' and '`]`' as in '`[^(ab|bc*)]`'. Actually, any character except '`^`', '`-`', '`\`' and '`]`' does not have a special meaning between '`[`' and '`]`' and therefore even special characters of regular expressions such as '`(`', '`)`', '`|`' and '`*`' are treated as the character itself. To avoid this misconception, the proposed block for the character class does not allow other regular expression to occur inside. It offers three kinds of sub-blocks in the mutator dialog (Figure 5). If you connect the range blocks as the sub-block, two input forms will appear, and you will be able to enter a character range in the parent block. If you add a single character sub-block, you can enter any single character in the parent block. For a special character sub-block, a drop-down menu appears, allowing you to select among six special characters.

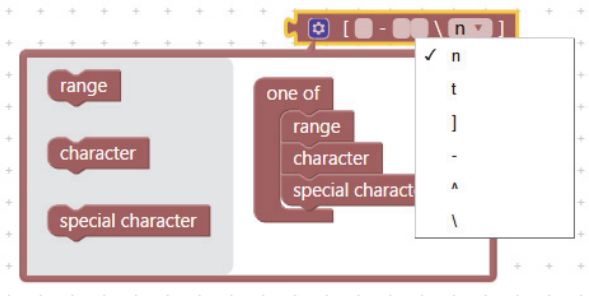

Figure 5. A Character Class Block

Another problem of Blockly blocks for Flex (and possibly for other expression-oriented languages) is that if we only used blocks with two value inputs for binary operators, blocks would tend to become too deeply nested and therefore too large to fit in a screen. This problem is serious in Flex since the concatenation (represented simply by juxtaposition of regular expressions) and the alternation (represented by the '|' operator) are very frequently used in regular expressions. To avoid this, in the concatenation blocks in Figure 6, the number of value inputs can be increased and decreased by the mutator dialog to represent multiply concatenated or multiply alternated regular expressions. We feel that this usage of mutators is still tedious, but for the moment we can think of no other ways to avoid this tediousness. In the repetition block, it is possible to select an operator among '+', '*', and '?' that indicates the meaning of repetition in the drop-down menu. The proposed blocks use the actual Flex keywords and operator symbols. This allows the user to program without being bothered by syntax errors.

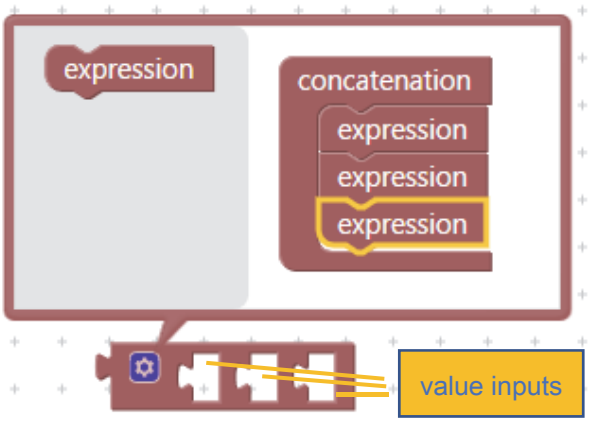

Figure 6. A Mutator Workspace for the Concatenation Block

3.3 Notation Switching

We have also implemented the notation switching facility in the blocks for Flex. There are two types of notations. One is the natural language notation (also called the Japanese notation) and the other is the one using programming language operators and keywords (called the Flex notation). Switching is performed by the select element of the HTML form.

In the Japanese notation, the meaning of the expression is easy for beginners to understand, then they can switch to the Flex notation. This makes it possible to understand the concept of each block part and the correspondence with the syntax of regular expressions.

3.4 List Comprehensions in Haskell

Haskell is a purely functional programming language and has a syntax rather different from procedural languages. We begin with the block for list comprehensions.

An example of list comprehensions is shown below.

```
foo n = [ (x,y) | x <- [0..n]
                , y <- [x..n]
                , mod (x + y) 2 == 1]
```

Here, a form such as `x <- [0..n]` is a generator. It means that the variable `x` is bound to one of the elements of the list `[0..n]`. A Boolean expression such as `mod (x + y) 2 == 1` is a guard. It represents a condition that holds among variables. Therefore, the expression `foo 3` evaluates to a list `[(0,1),(0,3),(1,2),(2,3)]`.

We represent both generators and guards as statement blocks. The list comprehension block is included in the list category of the block menu. The image of a list comprehension block is shown in Figure 7. The source code of the list comprehension consists of an expression, several generators and Boolean guards in a pair of square brackets. In the block for list comprehensions, an expression is inserted into the top value input and several blocks for generators and guards are inserted to the position below the expression.

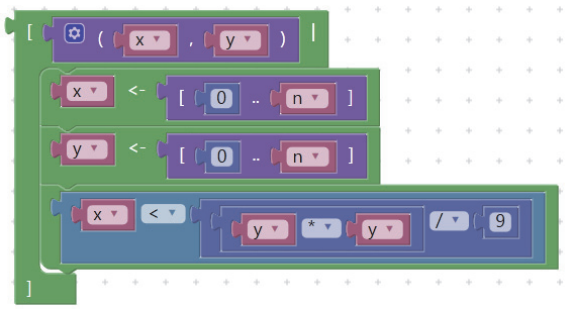

Figure 7. A List Comprehension Block

If you try to connect an expression block directly as a guard of the comprehension block, it cannot be connected due to mismatch of the form of the connection. Expression blocks, which are usually connected to value inputs, must be connected vertically here, Therefore, we offered a block (the bottom one in Figure 7) for this purpose. This block is, however, rather awkward. In the Blockly for procedural languages, this problem is not apparent because we can usually separate functions that have return values (and are connected to value inputs) and procedures that do not have meaningful return values (and are connected vertically). This is not the case for Haskell.

3.5 Sample Menu

For those who are new to Blockly, it is difficult to build blocks from the ground up and complete the executable program. We have offered a sample menu to make it easier for beginners to use this system. Several samples are offered at the top of the Web page. By pressing the menu, blocks prepared beforehand are displayed. Then, beginners can understand the system while arranging the program of the completed block.

4 EVALUATION

We asked five volunteer students in the computer science department to actually use the C language system and the Haskell language system, and then to answer to several questions. In the question "do you understand how to operate the system intuitively", in the C language system, we obtained a response "we can customize the completed program when we press a sample button." On the other hand, in the Haskell language system, there were a lot of negative responses: "I did not understand how to connect blocks or how to create a function".

In the question "evaluation of used blocks", there is an answer "I thought it was good to change the number of input expressions dynamically when '`%`' was contained" in the printf block of the C language system.

5 DESIGN OF NEW BLOCKS

After the evaluation, we decided to rearrange blocks for Haskell by adding some blocks and also dropping some. We also implement blocks for Prolog. In this section, we will show designs of newly implemented blocks for these languages.

5.1 Haskell Blocks

The blocks for Haskell were created for the "programming paradigm" class at the graduate school to which the authors belong. At the time of the evaluation, it did not cover all the syntax used in that class. Especially, it is necessary to support partial applications of functions, pattern matching in function definitions, do expressions, let expressions, and case expressions.

In Haskell, all functions are "curried." This means that a function with multiple arguments is represented as a function which takes a single argument and returns another function which takes the next argument. Therefore, we often use partial applications, that is, we supply fewer arguments than the total number of arguments and may pass the result to other functions. For example, if we define a function `bar` as follows:

```
bar x y = 2 * x + 3 * y
```

Then, we can use an expression `bar 5` as a function which takes an argument, multiply it with 3, and adds it to 10 (= $2 \cdot 5$). Moreover, a function can be freely stored in data structures such as lists. Therefore, a function which is defined to take *n* arguments may sometimes take

less than *n* and may sometimes take more than *n* arguments when it is used.

A function that takes other functions as arguments is called a higher-order function. This means that variables can be used in the position of a function. For example, the function `iterate` in the standard library is defined as follows.

```
iterate f a = a : iterate f (f a)
```

Here, the variable `f` is used as a function in the right-hand side of the definition.

Therefore, we change the block for function calls as follows. Functions are represented as simply shaped blocks without value inputs (similar to conventional variable reference blocks). Separately, we offer a block which is designated for function calls. This block has a mutator, and can change the number of arguments freely. In the block menu, we offer a combination of the above two kinds of blocks so that it can take the number of arguments given at the time of definition (Figure 8).

Figure 8. Function Call and Application Blocks

Haskell offers pattern matching. We can write patterns as well as simple variables in the position of arguments when defining functions. A pattern is an expression which only consists of data constructors, constants, variables, and wildcards. Unlike other languages that do not support pattern matching, a function definition consists of several alternative equations for different patterns. Patterns can be used, in addition to function definitions, in almost all places such as let expressions, lambda expressions, and generators in list comprehensions where variables can be bound. Accordingly, function definition blocks must provide value inputs for arguments instead of simple variable fields. Moreover, multiple function definition blocks for the same function must be allowed to present for different patterns (Figure 9).

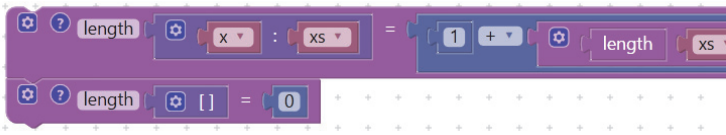

Figure 9. Function Definition Blocks

Haskell's let expressions are used for introducing locally-scoped identifiers. We use statement blocks to express variable definitions and function definitions and line up them in the block for let expressions (Figure 10). Haskell's case expressions are a syntax for pattern matching. We use a statement block to denote an alternative of the form *pattern* -> *expression* and place some alternative blocks in the block for case expressions.

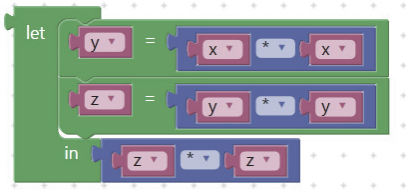

Figure 10. A Let Expression Block

We use do expression to define computations using monads [10]. Monads are especially used to represent computations such as input/output, state transformation, exception handling, and non-determinism in Haskell. Intuitively, a monad represents a set of computations which can be sequentially combined and, at the same time, can contain a value. A do expression consists of several *statement*s. And each statement is either a simple expression or a form *pattern* <- *expression*. The former is used for the case where the contained value of the monadic computation represented by the expression can be ignored. The latter is used when the contained value is necessary for the subsequent computation. Haskell's do expressions are syntactically similar to its list comprehensions. In both cases, we must connect expression blocks vertically. Though input/output functions such as `print` and `putStr` are used in do expressions in most cases, they can be also used as an argument of higher-order functions. We need both expression blocks and statement blocks for such functions.

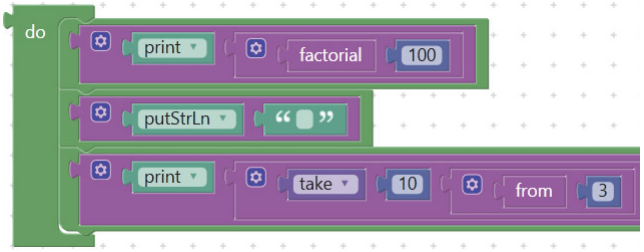

Figure 11. A Do Expression Block

Therefore, we provide expression blocks as primitives for such functions and a block to convert an expression block to a statement block (as in list comprehension blocks). Figure 11 shows an example usage of such blocks. In the block menu, we offer a combination of such blocks to cope with frequently used forms.

We do not offer blocks for where clauses and guards which are sometimes used in functions definitions. Though we tried to offer the blocks for them in the first evaluation, they rather caused confusion. Since they can be covered by combinations of other blocks, these blocks are withheld to reduce the kinds of blocks. We do neither offer blocks regarding types such as data declarations, class declarations, instance declarations, and type signatures. Since in Haskell programmers do not usually need to write types explicitly thanks to type inference, they are not necessary for beginners.

5.2 Prolog Blocks

The syntax of Prolog is not so complicated as other programming languages. Clauses are represented as procedure blocks without return values. Like Haskell functions, we must be able to put patterns in the place of arguments to support unification and multiple predicate definition blocks for the same predicate name must be allowed. We represent goals (the right hand side of the "：-" symbol in the text-based representation) by statement blocks (Figure 12).

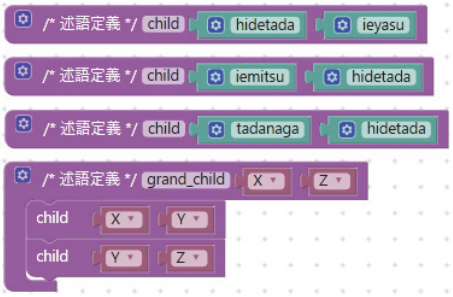

Figure 12. Prolog Blocks

Blocks for facts (clauses without goals) and rules (clauses with goals) can be switched by checking/unchecking "allow goals" in the mutator dialog.

Unlike Haskell's function definitions, blocks for predicate definitions are never used inside of other blocks. Therefore, they do not need to have top and bottom connectors. (The order of clauses is decided by the vertical position of blocks.)

The block for functors (names) is much like blocks for strings but with arguments. For convenience, we offer blocks for the functors which are already used in the workspace in the block menu.

6 CONCLUSIONS

In order to support the languages studied in undergraduate and graduate classes, we have designed new blocks for C, Flex, Haskell, and Prolog in the Web-based graphical programming editor implemented by Blockly. There, we have tried to design blocks so that the flexibility of programming language is not limited by the shape of blocks and we have used dynamic transformation so that there are not too many types of blocks. There remain, however, some awkwardness in the design of blocks with variable arity and blocks that must be connected both horizontally and vertically. On the other hand, the block for the printf function in C was highly evaluated, since it reduces burden on learners. It is necessary to further improve the design of blocks based on the obtained feedbacks and to confirm the effect of using block-based environments in actual classes.

7 FUTURE WORK

In order for the system to be effectively used in university classes, the following issues must be addressed.

7.1 Blocks for Other Languages

The languages supported so far are C, Flex, Haskell, and Prolog. We need to further evaluate newly added blocks. For Haskell, since we use several kinds of statement blocks, we need to be able to avoid putting such blocks in wrong places.

To support other languages learned in our university classes, we need at least block sets for Scheme and Java. For Java, it would be necessary to provide a designated block for the dot operator (“.”). There are already some block-based environments that can export Java source code [11, 12]. Since they are based on Alice and OpenBlocks, respectively, it makes sense to provide a Blockly-based environment.

Functional languages and logical languages have typical short examples such as Eratosthenes’s sieves, family trees, and list append. Object-oriented languages do not seem to have such convenient short examples. We need to define some classes, some fields, and some methods to show merits of object-oriented programming. On the other hand, block-based programs tend to be longer than their text-based counterparts. Therefore, they can easily become too large to fit in a single screen. Currently we consider that blocks may not be a suitable form to show merits of object-oriented programming language.

7.2 Frame-based Approaches

An alternative way to avoid syntax-related troubles in educational programming environments would be to use a frame-based approach as in Stride [13]. We are interested in whether frame-based editors can be extended to support non-procedural languages such as Flex and Haskell. Stride is a Java-like programming language and it uses frames at the statement level but does not seem to use frames at the expression level syntax.

7.3 Dual Mode Environments

Currently, our system does not have a facility to translate text-based representation of Flex, Haskell, and Prolog programs back into block-based representation. There are several programming environments that support bidirectional translation between block-based and text-based representation of programs [12, 14]. It would be possible to provide the backward translation feature to our system and investigate its educational effects.

7.3 Other Styles for Mutation

We use mutator dialogs for changing the arity of function definitions/calls in the Haskell system and the number of operands in the Flex system. This works but seems a little bit awkward. On the other hand, Snap! [15] and GP [16] seem to use an alternative style to increase/decrease the number of input holes (i.e. by using ◂ and ▸ icons). We would like to examine whether other styles including that of Snap! and GP are applicable to Blockly.

ACKNOWLEDGMENTS

We are grateful to Kenta Ohashi, Haruhiko Nishina, Yuya Asano, Takumi Ito, and Keisuke Ota for their feedback in the evaluation of the system.

This work is partially supported by JSPS KAKENHI Grant Number 15K01075.

REFERENCES

[1] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak, “Learnable programming: blocks and beyond,” Commun. ACM 60, 6, pp. 72-80. DOI: https://doi.org/10.1145/3015455, May 2017.

[2] D. Elza, “Waterbear: Welcome” URL: https://waterbearlang.com/, accessed Nov. 2019.

[3] D. Bau, “Droplet, a blocks-based editor for text code”, J. Comput. Sci. Coll. 30, 6, pp. 138-144, June 2015.

[4] E. Pasternak, R. Fenichel and A. N. Marshall, “Tips for creating a block language with Blockly,” 2017 IEEE Blocks and Beyond Workshop (B&B), Raleigh, NC, pp. 21-24, 2017. DOI: 10.1109/BLOCKS.2017.8120404.

[5] 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), 16, 2010.

[6] S. Marlow, "Haskell 2010 language report", Available online http://www. haskell. org/, 2010.

[7] G. T. Nicol, "Flex: the lexical scanner generator", Free Software Foundation, 1993.

[8] Y. Yamagata, and K. Kagawa, "A Web-based learning support system for the C language with automatic generation of Blockly programs", EdMedia + Innovate Learning 2018, Amsterdam, Netherlands, June 2018.

[9] Y. Ozaki, and K. Kagawa, "Development of a Flex programming environment using a Web-based graphical programming editor" (in Japanese), JSiSE2013, TF1-1, Sep. 2013.

[10] P. Wadler, "Comprehending Monads", In Proceedings of the 1990 ACM conference on LISP and functional programming (LFP '90). ACM, New York, NY, USA, 61-78.
DOI: http://dx.doi.org/10.1145/91556.91592, 1990.

[11] W. Dann, D. Cosgrove, D. Slater, D. Culyba, and S. Cooper, "Mediated transfer: Alice 3 to Java" *SIGCSE*. Vol. 12. pp. 141-146, 2012.

[12] Y. Matsuzawa, T. Ohata, M. Sugiura, and S. Sakai, "Language Migration in non-CS Introductory Programming through Mutual Language Translation Environment", In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, pp. 185-190.
DOI: https://doi.org/10.1145/2676723.2677230, 2015.

[13] M. Kölling, N. C. C. Brown, H. Hamza, and D. McCall, "Stride in BlueJ -- computing for all in an educational IDE", In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19). ACM, New York, NY, USA, pp. 63-69,
DOI: https://doi.org/10.1145/3287324.3287462, 2019.

[14] D. Weintrop, and N. Holbert, "From Blocks to Text and Back: Programming Patterns in a Dual-Modality Environment", In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17). ACM, New York, NY, USA, 633-638.
DOI: https://doi.org/10.1145/3017680.3017707, 2017.

[15] B. Harvey, and J. Mönig. "Snap! reference manual", URL: http://snap. berkeley. edu/SnapManual. pdf 2017.

[16] J. Maloney, J. Mönig, Y. Ohshima, M. Guzdial, and J. Lavalle, "GP Blocks" URL: https://gpblocks.org/, accessed Nov. 2019.