In Proceedings of the 8th International Confernce on Human Factors and Ergomonics, 2017, Springer.

Assessing the Effectiveness of Emoticon-Like Scripting in Computer Programming

Angelos Barmpoutis[1], Kim Huynh[1], Peter Ariet[1] and Nicholas Saunders[1]

[1] University of Florida, Digital Worlds Institute, Gainesville, Florida 32611, United States of America
angelos@digitalworlds.ufl.edu, {huynhtina123,pjpariet,password1}@ufl.edu

Abstract. In this paper a new method is proposed for learning computer programming. This method utilizes a set of human-readable graphemes and tokens that interactively replace the grammatical tokens of programming languages, using a concept similar to emoticons in social media. The theoretical framework of the proposed method is discussed in detail and two implementations are presented for the programming language ECMAScript (JavaScript). The results from user testing with undergraduate students show that the proposed technique improves the student's learning outcomes in terms of syntax recall and logic comprehension, in comparison to traditional source code editors.

Keywords: Computer Education · Programming Languages · Human Factors · Interaction Design · Emoticons · Graphemes

1 Introduction

In many ways, learning to program can be challenging for beginners of all ages. This is often attributed to poor self-efficacy, limited prior experience with computers, or inability to relate personal experiences to abstract programming concepts [1]. In turn, these challenges often result in beginners to regard programming as "boring and difficult" [2]. Since the early 1960's, there have been a number of tools developed to address these issues and overall focused on making programming accessible for everyone. With the wide array of modern Computer Science (CS) educational tools available, identifying the benefits and challenges associated will be necessary to gain insight into the varying effectiveness of each tool [3, 4,5]. Currently, the two most notable types of interactive tools for computer science education are Tangible User Interfaces (TUI) and Graphical User Interfaces (GUI).

TUI systems are in the form of physical objects and environments, which can represent or be augmented by digital information [6]. The biggest advantage of TUI systems is that they build directly on existing knowledge and experience from the real world [4]. One example of a TUI system is Robo-Blocks, which enable users to program a moving robot by connecting blocks that contain embedded technology, where each block represents a certain command [7]. Studies have shown that the use of Robo-Blocks show considerable potential in helping children learn how to debug, and further develop problem solving skills [5,7]. Another example is Tern, which is a set of wooden blocks, where each block represents a basic command that can be connected with others in order to create a program [8]. Through minimizing the amount of resources used in creating Tern, it provides a practical way for children to have real-time interactions with CS concepts [9]. Studies have shown that children had noticeably high hands-on interaction when using Tern, which indicates that it can encourage children to take more active roles in learning [8,9]. However the major disadvantage of TUIs is their disconnect from programming languages and thus cannot be used beyond an early stage of learning.

GUI programming tools employ virtual graphic components as means for interaction with a computer, which have attracted in general a wider demographic variety of users than TUIs, since GUIs have greater flexibility in content and have the capability to improve user's understanding at a university level. One example of a GUI system is Alice, which is a programming environment that enables novices to develop 3D environments using drag and drop scripts [10,11]. Studies on the use of Alice in introductory CS courses have shown that the average student scores were consistently higher, when compared to those in courses using Java [10]. Scratch is another example of a GUI system, which allows users to program "interactive stories, games, and animations" through an online platform [12]. User studies indicate that middle school and college students found Scratch to be a reliable way to introduce beginners to the basics of programming, because it removed the complexity of syntax [12,13]. However, in the same studies, a number of students did not favor Scratch because it underestimates the detail and complexity of more comprehensive programming languages. While GUIs can provide a reliable starting point, the students should transition into a more advanced programming language [12], which is the main limitation of this method.

In this paper, we propose a new educational framework that overcomes the problems of the aforementioned approaches by adding a human-readable layer on the top of existing programming languages. The proposed method is based on the use of emoticon-like typing that has become popular with social networks. Emoticons are visual representations that have one to one relationship with a corresponding combination of characters such as ":)". These can be perceived as visual interpretations of the corresponding characters that provide instant feedback to the user regarding the meaning associated with the typed code. The proposed framework utilizes a set of meaningful visual replacements of each grammatical token in a given programming language that appear instantly when complete valid tokens are typed.

The proposed method, dubbed Brain-Activating Replacement method (BAR), is based on the following three hypotheses: a) the immediate feedback given to the programmer can result in improved learning outcomes as it stimulates the brain to build one-to-one connections, b) the unique correspondence of each visual replacement, with a valid programming token re-enforces the learning of the syntax in an intuitive trial-and-error framework, c) the use of visual replacements remove visually the grammatical and syntactical details of a programming language and reveal to the users the logic of the program in the form of a pseudo code.

The developed framework was evaluated in a pilot study using 35 undergraduate students, and the results conclusively demonstrate the merits of the proposed method.

2 Methods

The smallest units in any writing system are known as graphemes [14]. Graphemes are not only the characters in a given alphabet but also the accents, punctuation marks, and other symbols that may be used in the corresponding writing system. Similarly, in any programming language a set of graphemes is used, which usually includes the graphemes of the Latin alphabet as well as other logical, mathematical, and structural symbols required for the needs of a particular programming language.

One or more graphemes can form morphemes, which are the smallest grammatical units in a particular language. One or more morphemes can function as components of a word, which along with various self-standing graphemes and morphemes are the smallest self-standing units in a language, also known as tokens. Morphemes contribute a particular meaning to a token when used as suffix, prefix, or as an intermediate compound of the token. For example, in computer programming the morphemes "+" and "=" can form the token "+=", or they can function as independent tokens by themselves. In this example, the token "+" denotes addition, the token "=" denotes assignment, and the token "+=" denotes "be increased by", which contains addition and assignment.

This hierarchical structure of written languages continues in higher levels in order to compose more complex constructions as ordered compositions of lower level units, such as sentences, paragraphs, sections, chapters, books, and book series. Similarly, in computer programming one or more tokens when used in the proper order can form a command. Subsequently, one or more commands can form a self-contained program, such as a function. Finally, one or more functions can compose larger structures such as classes, or an executable computer program.

Let Ω_0 be the set of graphemes in a written language. A morpheme can be expressed as an ordered set of graphemes as follows:

$$\mu = \{\gamma_1, \gamma_2, \gamma_3, \cdots : \gamma_i \in \Omega_0\} \quad \textbf{(1)}$$

If Ω_1 is the set of all morphemes in a written language, a token can be expressed similarly as:

$$t = \{\mu_1, \mu_2, \mu_3, \cdots : \mu_i \in \Omega_1\} \quad \textbf{(2)}$$

where $t \in \Omega_2$, which is the set of all tokens in a given language. Equations 1 and 2 can be generalized in order to account for higher grammatical levels as follows:

$$\{e_1, e_2, e_3, \cdots : e_i \in \Omega_{k-1}\} \in \Omega_k \quad \forall k > 0 \quad \textbf{(3)}$$

where e_i is an element of the previous hierarchical set. For example, in the case of k=1, e_i represents a grapheme, in the case of k=2, e_i represents a morpheme, in the case of k=3, e_i represents a token, etc.

2.1 Human-Readability of Programming Languages

One key difference between written natural languages and programming languages is that the former are constructed as representations of spoken languages, and for this reason, the formation of graphemes and morphemes imitates the corresponding phonemes of the spoken language. This correspondence between graphemes and phonemes makes a written text easy to be read and comprehended by speakers of the corresponding spoken language. However, such correspondence does not exist in programming languages, since their graphemes were not derived by a phonetic system but were specifically defined in order to facilitate the communication with computers. In that sense, the source code of a program is meant to be natively read by a computer rather than humans, who can in turn decipher the corresponding programming context but can only communicate it after having interpreted it to a human spoken language.

Therefore, the primary role of graphemes and tokens in a programming language is to serve as input to computers rather than as output to humans. This leaves a significant gap in the process of writing/reading a computer program, which is more evident in the case of beginner programmers who often try to read and comprehend a given text written in a computer programming language. For example the text “a+=2;” in many programming languages means “increase the value of a by 2”. By comparing the original phrase with the translated one, it is evident that the latter is easier to read and understand especially in the case of beginner programmers.

In this paper, a new text-editing process is proposed as a solution that bridges the aforementioned gap in computer languages. The proposed solution does not intend to form a new programming language but enhance the readability of existing ones by extending the traditional human-computer interaction of text editors.

Let us consider the following written sample: “not:(or:|!be:)” and its equivalent in another written language with different graphemes: “not☹or😐!be☺”. Obviously, the latter is easier to read, but the former is easier to write in the form of a typed text in a computer device. This example shows that there exist written languages that are primarily meant to be written (possibly to serve as an input to a computer system), and others that are primarily meant to be read. Furthermore, there exists a mapping that maps elements of the former language to elements of the latter:

$$M_k\colon (\Omega_k, \Omega_{k+1}) \to \Psi_k \qquad \textbf{(4)}$$

where Ω_k denotes the set of elements in the k^{th} hierarchical level of one language, and Ψ_k denotes the set of elements in the same hierarchical level of another language. According to the previous example, $M_k(":)", C) =$ “☺”, where k is the token-level and C denotes the context set $C = \{"not", ":(", "or", ":|", "!", "be", ":)"\}$, which is an element of the next hierarchical level $k+1$.

The role of the context Ω_{k+1} in Eq. 4 is to enable us to define context-depended mappings for a particular element Ω_k. For example, the token “=” in ECMAScript (JavaScript) programming language can be mapped to “be:”, unless it is followed by the token “[”, in which case it can be mapped to “be the following array:”. Finally, Eq. 4 can be further generalized in order to consider the context in different hierarchical levels as follows:

$$M_k\colon (\Omega_k, \Omega_{k+1}, \Omega_{k+1}, \cdots) \to \Psi_k \qquad \textbf{(5)}$$

Equations 4 and 5 can be used in order to provide readability to computer programming languages without altering in any way the programming languages themselves. A set of mappings M_k and a target language Ψ must be defined according to Eqs. 4 and 5. Different mappings may be defined in different hierarchical levels. For instance, the previous example in ECMAScript defined the following token-level mapping: {“a”, “=”, “2”} $\rightarrow$ {“a”, “be:”, “2”}. An additional command-level mapping can be defined in order to map {“a=2”} $\rightarrow$ {“Set a to 2”}. The corresponding mapping can be used as soon as the required input elements of the original language Ω are typed. For example, during the composition of a token, a grapheme-level mapping can be used. Once a complete token is composed, a token-level mapping can be employed in order to render the token. Similarly, a command-level mapping can be used as soon as a complete command is composed.

Obviously, the result of mappings can be modified or cancelled when the input that activated these mappings is changed. For example, when a previously typed command is being edited, the command-level mapping is cancelled, and a token-level mapping is used to render the tokens of this command. Similarly, when the user edits a previously typed token, the token-level mapping is cancelled, and a grapheme-level mapping can be activated to render this token.

2.2 Properties

The theoretical framework presented in the previous sections for interactively replacing the elements of an input written language Ω with elements of a target language Ψ has the following properties:

Preservation of Cardinality. The mapping from Ω to Ψ should preserve the number of elements in each level, i.e. N tokens in Ω should be mapped to N tokens in Ψ, M commands in Ω should be mapped to M commands in Ψ, etc.

Interpretative Replacements. The elements of the target language can employ graphemes, symbols, or other textual representations that interpret the corresponding elements of the original language. The replacements can be properly chosen based on the age, proficiency in Ω, or personal preferences.

Interactive Validation. The instant replacement of elements of Ω provides continuous feedback that validates the user’s input and offers a trial-and-error interface for text composition.

Authentic Reproduction. A text in Ψ cannot be reproduced unless the original text is re-typed in Ω, which requires that the user is capable of composing the original text in Ω. Therefore, it is an authentic reproduction since the user cannot blindly copy the text in Ψ in order to achieve the desired result.

Syntax Discovery. The combination of authentic reproduction and interactive validation facilitates the discovery of syntactical phenomena and is hypothesized to re-enforce learning by stimulating the brain to build connections between the programming language (Ω) and its interpretation (Ψ).

Identity Mapping. Traditional text editors can be considered special cases of the proposed framework in which the identity mapping is used between Ω and Ψ. Source code editors that utilize color-coding are also special cases, in which the graphemes in Ψ differ from those in Ω only in their color properties and not in their structure.

The aforementioned properties extend the traditional text editing process by introducing new interactive features that do not currently exist in source code editors, but the general audience has established familiarity with them through emoticon scripting in social media and text messages.

3 Implementation

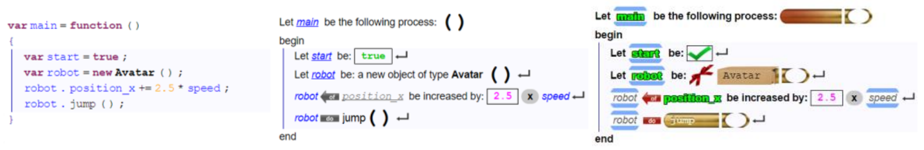
The proposed method, dubbed Brain-Activating Replacement (BAR) method, was implemented for ECMAScript (JavaScript) in two different target languages. The first implementation was designed for beginner programmers and employs iconic graphemes and visual metaphors such as nametags for the names of variables and pipes for representing functions. The second implementation was designed for more mature users and employs more discrete iconic graphemes with primarily text-based token replacements. For each implementation, a set of 88 BAR-tokens was designed. Table 1 shows a sample list of the tokens from the target language designed for beginner programmers.

Table 1. A sample list of the BAR-tokens created for beginner programmers. The table shows 19 out of the 88 types of tokens implemented for ECMAScript (JavaScript).

variable	*name*	variable being modified	**name**
number	123.45	string	*'a string'*
true, false	✓ ✗	null, undefined	☐ ???
`;`	↵	`method()`	method ◯
`=`	**be:**	`==, !=, ===`	**is equal to, is not, strictly is**
`var`	**Let**	{ }	**begin, end**

As shown in Table 1, all data values are visualized inside rectangular boxes that resemble fields in an electronic form. Comparison and logical operators as well as structural markers are replaced by textual interpretations, such as “begin”, “is not”, etc. An example of JavaScript source code visualized in the two implemented target languages is shown in Fig. 1.

By comparing the output of the two implementations as shown in Fig. 1, it is evident that the implementation designed for beginners is more colorful and iconic, compared to the one designed for mature audiences, which is primarily text-based and can be read almost as a continuous text: “Let main be the following process: Begin. Let start be true. Let robot be a new object of the type Avatar. Position_x of robot be increased by 2.5 x speed. Robot do jump. End.” One interesting observation is that the tokens “=” and “.” were interpreted differently based on their context as defined in Eq.5. For example “.” was interpreted as “do” when followed by a method and as “of” when followed by a property.



```
var main = function ()
{
    var start = true ;
    var robot = new Avatar () ;
    robot . position_x += 2.5 * speed ;
    robot . jump () ;
}
```

```
Let main be the following process: ( )
begin
    Let start be: true ↵
    Let robot be: a new object of type Avatar ( ) ↵
    robot of position_x be increased by: 2.5 x speed ↵
    robot do jump ( ) ↵
end
```

```
Let main be the following process:
begin
    Let start be: ↵
    Let robot be: Avatar ↵
    robot of position_x be increased by: 2.5 x speed ↵
    robot do jump ↵
end
```

Fig. 1. Side-by-side comparison of the two implemented target languages for beginners (*right*) and for more mature users (*middle*). The corresponding JavaScript source code is on the left.

The two implementations (Fig 1 middle and right) and the traditional implementation (Fig. 1 left) were developed in JavaScript using the open-source library VisiNeat, which is licensed under BSD 2-clause by the University of Florida. The developed BAR-enabled text editor is available in the VN JavaScript Studio and was used in a series of experiments discussed in the next section.

4 Experimental Results

The proposed BAR method was tested during the academic semester of Spring 2017, using students volunteers from the on-line and on-campus undergraduate program of Digital Arts and Sciences at the Digital Worlds Institute at the University of Florida, with partial support from the University of Florida On-Line Institute and the grant PRDSP024 from the University of Florida Provost's office.

In total 35 students (22 female) used the developed text editor to complete 6 programming assignments for a period of 7 weeks. During this period, the proposed method was adopted as the primary method of instruction; hence, all the content delivered to the students as part of lectures or additional material was in the form of BAR-tokens. It should be noted that the students were allowed to choose between the two developed BAR-enabled text editors (Fig. 1 middle and right) and a regular source code editor (Fig. 1 left), or transition between the editors as they wished.

The students in this program focus on the theory and practice of interactive digital media, and, as part of their curriculum, learn programming fundamentals. Although the majority of the students have limited prior experience in programming, others have taken prior programming classes or practiced programming on their own. It was anticipated that the proposed framework would have different effect on the students based on their programming level, as it is demonstrated later in this section.

4.1 Manual Observation of Student Coding Patterns

During the course of this experiment, the keystrokes performed by the participating students within the developed text editors were recorded in order to manually inspect the coding patterns of the students with or without the proposed framework. After systematic inspection of the recorded sequences of keystrokes, the following scenarios were noted.

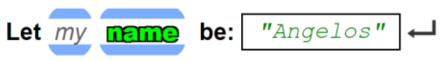


Fig. 2. This figure shows an example of erroneous use of spaces within the name of a variable. The corresponding BAR-tokens provide instant visual feedback in order to self-correct this error. In this case, 2 nametags are shown and only the second one is affected by the assignment.

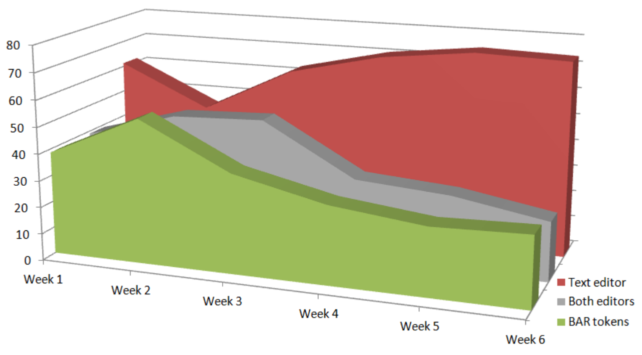


Fig. 3. The percentage of the students who used the proposed editor and/or traditional code editor during this experiment. The plot shows the transitions of the students between editors.

The students who typed their assignments in a BAR-enabled text editor were able to identify and correct syntax errors as well as discover unknown syntax rules on their own before the compilation of their code. A common mistake was the use of spaces within the name of a variable as shown in Fig. 2. The corresponding BAR tokens provided instant visual feedback that assisted the students to identify and correct their mistakes.

On the other hand, students who made similar mistakes in a traditional text editor were not able to identify their mistake prior to compilation. What is more, there were several instances where the students could not understand their mistake even after receiving an "unexpected identifier" error message from the compiler.

Another mistake that was observed several times is the incorrect use of the equal sign in an attempt to test equality instead of the double equal sign "==", which is the appropriate token to be used in this case. The problem in this scenario is that this is a logical rather than a syntax error; hence, the compiler does not provide any error message to help the programmers identify their mistake. However, in the case of BAR tokens, there is a clear difference between the correct and incorrect case of testing equality. More specifically, in the correct test of equality the target code reads "if(score is equal to 100)", which corresponds to the source code "if(score==100)". On the other hand, when the source code is "if(score=100)", the target code reads "if(score be: 100)", which reveals the incorrect logic. It has been recorded several times in the collected data that the students who used the BAR-enabled editor were able to identify the difference between these two statements and correct their logic without further assistance.

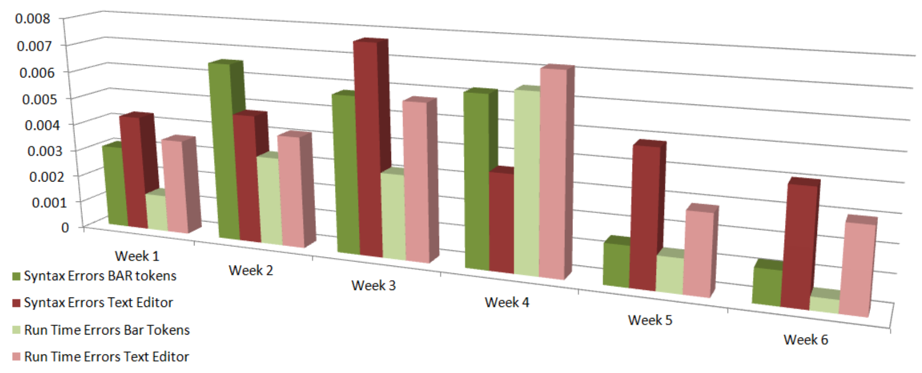


Fig. 4. The recorded number of syntax and run-time errors per keystroke per person during this experiment. The results are reported separately for the proposed and traditional text editors.

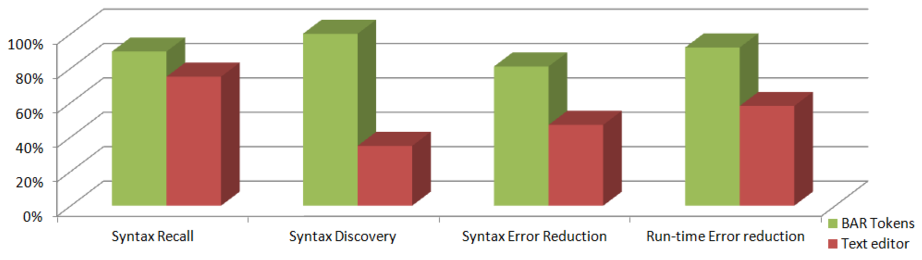


Fig. 5. The percentages of successful syntax recall, new syntax discovery, and error reduction that correspond to the proposed method versus traditional text editors.

4.2 Quantitative Analysis

Several types of data were recorded during this experiment, including the type of text editor used in each keystroke and the error messages generated by the system (syntax errors and run-time errors), in addition to the keystroke sequences produced by each student.

Fig. 3 shows the transitions of the students between the provided text editors during the data collection period. As expected, a gradual transition from the BAR-enabled editor to the traditional editor was observed as the competency of the students in programming increases, although a small transition back to the BAR-enabled editor was also observed towards the end of the data collection period. It should be noted that a significant percentage of the students completed their programming assignments using both editors.

The syntax and run-time error messages generated by each student are shown in Fig. 4. The data were normalized by the number of keystrokes per student in each type of editor. Although the absolute value of the reported numbers depend on the particular level of difficulty of each week's programming assignment, we can compare the data across categories. By observing the data in Fig. 4, it should be noted that the number of run-time errors generated by the students who used the BAR-enabled editor is less than the corresponding number of errors from the traditional text editor, and this is true throughout the entire data collection period. A similar pattern was observed for syntax errors with the exception of weeks 2 and 4. This observation may indicate that the effect of the proposed method on logic understanding is stronger than the effect on syntax recall, since the majority of the run-time errors are typically associated with logic errors.

At the end of the data collection period, the students were asked to recall particular syntax rules as well as try to comprehend unfamiliar syntactical structures. Fig. 5 shows that 90% of the students who used BAR-tokens were able to successfully recall the syntax compared to 75% for the case of a traditional text editor. This result suggests that the students who see the typed source code are not able to remember the syntax as effectively as the students who do not see the typed code.

Furthermore, only 30% of the students who read an unfamiliar sample of code in a conventional text editor were able to comprehend it. The same task in the BAR-enabled editor was reduced to the study of a text in plain Engish, which was comprehended by all students.

Finally, Fig. 5 reports that the overall reduction of errors using the BAR method was larger than the corresponding errors in a traditional text editor. If the reduction of errors is assumed to be correlated with the learning outcomes, then the results may suggest that higher learning outcomes can be achieved using the proposed method.

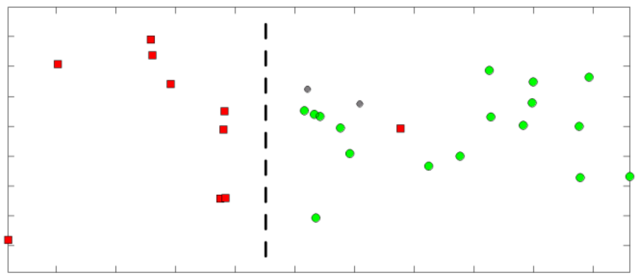


Fig. 6. The distribution of the students based on their responses on the TAM questionnaires as shown on the dominant eigen-plane of the response data. The students can be linearly separated based on their perception regarding the improvement of their learning performance, depicted in green and red for positive and negative responses respectively.

4.3 Perceived Usefulness and Perceived Ease of Use

Finally, at the end of the data collection period, the proposed method was evaluated using the technology adaptation model (TAM) [15]. The original questionnaires of the TAM model were extended in order to capture the perception of the students regarding the effect of the proposed technique on syntax recall and logic comprehension.

In order to evaluate the effect of the proposed method as a function of the student's competency in programming, each student's programming level was assessed through programming questions. Principal component analysis of the responses on the TAM survey showed that the students were divided into two groups based on their perception regarding the effect of BAR-tokens on their learning performance (Fig. 6).

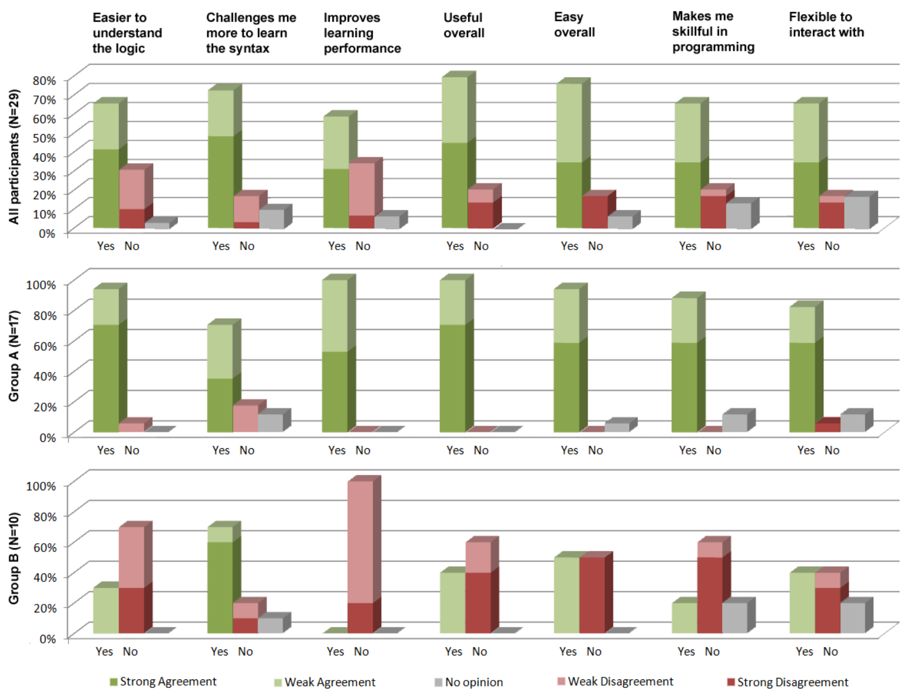


Fig. 7. The students' responses on the key questions of the extended TAM survey [15]. The results are also shown separately based on the students' perception regarding the improvement of their learning performance (*middle row: positive response, bottom row: negative response*).

The group of students who believed that the proposed method did not improve their learning outcomes includes more than 70% of the students who were identified as advanced programmers. Therefore, the independent analysis of the two groups can give us insights on how students of different programming levels perceived the usefulness and ease of use of the proposed method.

Fig. 7 shows that the majority of students on both levels agreed that the proposed method challenges them more to learn the syntax. This result aligns with the "authentic reproduction" property of our theoretical model as discussed in Sec. 2.2.

Furthermore, the students in group A, who are predominantly beginners in programming, agreed on the majority of the questions, as they found that they could understand the programming logic easier with the proposed method. Furthermore, they believed that the proposed method improved their learning performance and can make them skillful in programming. They also found that the proposed method is useful, easy, and flexible to interact with.

On the other hand, the students in group B, who are predominantly more experienced programmers, appeared to be divided on several questions regarding the flexibility and ease of use of the proposed method. As it was expected, the effect of the proposed method on the more advanced students was limited, which is reflected on the students' perception on the ease of use and usefulness of the proposed method.

5 Conclusions

This paper presented a novel method for learning computer programming, dubbed Brain-Activating Replacement method, which is based on the hypothesis that the interactive replacement of syntactical tokens in programming languages with human-readable tokens, facilitates the building of stronger connections between the source code and its logical meaning. Two implementations of the proposed method were presented and tested in a pilot study. The results suggested that the proposed framework increases the learning outcome of the students. The observed benefit is stronger in the case of beginner programmers, whose performance has improved in terms of syntax recall and logic comprehension, compared to the performance achieved using traditional text editors for source code editing.

References

1. Lahtinen, E., Ala-Mutka, K., & Järvinen, H. M.: A study of the difficulties of novice programmers. ACM SIGSCE Bulletin 37(3), 14--18 (2005)
2. Jenkins, T.: On the difficulty of learning to program. In Proceedings of the 3rd conference of the LTSN Centre for Information and Computer Sciences, Vol. 4, pp. 53--58 (2002)
3. Busch, T.: Gender differences in self-efficacy and attitudes towards computers. Journal of Educational Computing Research, 12(2), 147--158 (1995)
4. Jacob, R.J.K., Girouard, A., Hirshfield, L.M., Horn, M.S. Shaer, O., Solovey, E.T., and Zigelbaum, J.: Reality-Based Interaction: A framework for PostWIMP interfaces. In Proc. of the SIGCHI conference on Human factors in computing systems, pp. 201--210 (2008)
5. Sapounidis, T., Demetriadis, S., & Stamelos, I.: Evaluating children performance with graphical and tangible robot programming tools. Personal and Ubiquitous Computing 19(1), 225--237 (2015)
6. McNerney, T. S.: From turtles to Tangible Programming Bricks: explorations in physical language design. Personal and Ubiquitous Computing, 8(5), 326--337 (2004)
7. Sipitakiat, A., and Nusen, N.: Robo-Blocks: designing debugging abilities in a tangible programming system for early primary school children. In Proceedings of the 11th International Conference on Interaction Design and Children, pp. 98--105 (2012)
8. Horn, M. S., Solovey, E. T., Crouser, R. J., & Jacob, R. J.: Comparing the use of tangible and graphical programming languages for informal science education. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 975--984 (2009)
9. Horn, M. S., & Jacob, R. J.: Tangible programming in the classroom with tern. In CHI'07 extended abstracts on Human factors in computing systems, pp. 1965--1970 (2007)
10. Moskal, B., Lurie, D., & Cooper, S.: Evaluating the effectiveness of a new instructional approach. ACM SIGCSE Bulletin, 36(1), 75--79 (2004)
11. Sykes, E. R.: Determining the effectiveness of the 3D Alice programming environment at the computer science I level. J. of Educational Computing Research 36(2), 223--244 (2007)
12. Malan, D. J. and Leitner, H.H.: Scratch for Budding Computer Scientists. ACM SIGCSE Bulletin 39(1), 223--227 (2007)
13. Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M.: Learning computer science concepts with scratch. Computer Science Education, 23(3), 239--264 (2013)
14. Coulmas, F. The Blackwell Encyclopedia of Writing Systems, Blackwell, pp. 174 (1996)
15. Davis, D. F.: Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. MIS Quarterly 13(3), 319--340 (1989)