Learning Programming Languages as Shortcuts to Natural Language Token Replacements

Angelos Barmpoutis
University of Florida
Gainesville, Florida
angelos@digitalworlds.ufl.edu

ABSTRACT

The basic knowledge of computer programming is generally considered a valuable skill for educated citizens outside computer science and engineering professions. However, learning programming can be a challenging task for beginners of all ages especially outside of formal CS education. This paper presents a novel source code editing method that assists novice users understand the logic and syntax of the computer code they type. The method is based on the concept of text replacements that interactively provide the learners with declarative knowledge and help them transform it to procedural knowledge, which has been shown to be more robust against decay. An active tokenization algorithm splits the typed code into tokens as they are typed and replaces them with a pre-aligned translation in a human natural language. The feasibility of the proposed method is demonstrated in seven structurally different natural languages (English, Chinese, German, Greek, Italian, Spanish, and Turkish) using examples of computer code in ECMAScript (JavaScript).

CCS CONCEPTS

• **Applied computing → Education**; • **Software and its engineering → Software notations and tools**; **General programming languages**;

KEYWORDS

Computer Education, Source Code Editors, Non-CS majors, Computer Programming

ACM Reference Format:
Angelos Barmpoutis. 2018. Learning Programming Languages as Shortcuts to Natural Language Token Replacements. In *18th Koli Calling International Conference on Computing Education Research (Koli Calling '18), November 22–25, 2018, Koli, Finland*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3279720.3279721

1 INTRODUCTION

The relationship between natural and computer programming languages has been a topic of research from multiple perspectives. Both types of languages obey to specific sets of rules that form the grammar of each language and use distinct bodies of words, known as vocabularies. The key differences, however, are their flexibility in expressing a message in multiple different ways and the context in which they are utilized. Although natural languages can be used in many different contexts, programming languages are primarily used to precisely describe a set of actions to be performed by a computing device.

Natural language can be translated to computer code under certain conditions, especially if the structure of the natural language text is restricted so that there is no ambiguity in its content[21, 23]. For example, a spoken variation of Java has been defined in [5] in the form of a precisely structured English language. Similarly, a small set of predefined pseudo-code phrases have been used to interactively generate computer code in [17].

On the other hand, machine translation of a computer code to a natural language, such as English, can be achieved using statistical machine translation algorithms [13, 30] and has many applications including understanding the logic of the program by reading its pseudo-code and generating automatic documentation in a natural language. For the latter application, summary comments can be generated to outline the structure of a program, such as the list of the defined methods or functions and their corresponding input and output [10, 16, 27, 38].

Furthermore, coding can be used to visualize natural language in the form of storytelling [24]. Conversely, storytelling can provide useful insights about coding [18]. The correspondence between natural and programming languages has also been utilized in educational context. Learning to code has assisted students in learning English as a foreign language [28].

The present paper introduces a novel method for assisting beginners to learn programming languages through the immediate translation and replacement of the code being typed with phrases from a natural language. Computer education is a well-studied research subject. Learning computer programming involves acquisition of several skills, including problem solving, computational thinking, and information encoding in programming language syntax. Several methods have been developed to help students acquire computer programming skills using age appropriate tools [14], such as tangible user interfaces (TUI), including robotics [35, 44] and block-coding interfaces (BCI) [8, 31, 46]. Although these tools have been very successful in early stages of learning, especially for acquiring problem solving and computational thinking skills, transitioning to text editing interfaces (TEI), also known as typeface interfaces, could be problematic for beginners, as it requires encoding in programming language syntax [26, 32]. TEIs constitute

Koli Calling '18, November 22–25, 2018, Koli, Finland

ACM ISBN 978-1-4503-6535-2/18/11…$15.00
https://doi.org/10.1145/3279720.3279721

the final learning step that introduces learners to professional programming languages, which provide greater flexibility in content and can be used to teach more advanced concepts [26, 29].

In order to ease the transition from BCIs to TEIs, various solutions have been developed. A hybrid technique between BCIs and TEIs is frame-based coding, which offers an extended block-coding interface using frames that resemble TEIs [6, 32, 45]. Another approach is to split the interface into a TEI panel and a code visualization panel, which can present the execution of the code as an explanatory animation [7]. A popular solution is to offer simplified special-purpose application programming interfaces (APIs) within light TEI environments that resemble professional integrated development tools [12, 34].

This paper presents an alternative solution to this problem, optimizing first exposure to TEI environments, using a set of translated phrases from a natural language that have been pre-aligned with the syntax of a programming language. The goal of this solution is to train learners in encoding procedural information using professional computer language syntax, a necessary skill in computer programming. Hence the emphasis here in not on the acquisition of other skills that are majorly independent from any specific programming language, such as problem solving and computational thinking. Therefore, the proposed solution is complimentary to the aforementioned TUI and BCI techniques that can be used by learners before transitioning to TEIs.

The proposed method is based on a multi-layer active tokenization algorithm that identifies individual tokens as soon as they are typed by the users and replaces them with interpretative textual overlays in the native language of the user. The produced translated version of the computer code can be navigated and edited by intuitively editing the underlying programming code; hence the proposed text editor trains the users to code in a professional programming language while they read the logic of the code in their natural language. The feasibility of the method is demonstrated in 7 structurally different natural languages. Finally, the proposed method was tested in a preliminary study using 88 students of ages between 10-15 years old.

2 THEORETICAL FRAMEWORK

The process of learning a programming language involves the acquisition of the skill to transcribe information using a particular well-defined encoding scheme, among other skills as discussed in section 1. These encoding schemes are mappings between a specifically encoded input (computer source code) and the corresponding procedural information that the programmer intends to transcribe.

The learning of encoding mappings has been well studied outside of the scope of programming languages due to its applicability in several forms of human-machine interaction and its direct effects in the human brain. Like any form of learning, it relies on neuroplasticity, which is evident not only in gray matter but also in white matter, as it has been shown in individuals during the learning of Morse code [36]. In this case, the neuroplasticity is due to the encoding of the mapping between an input code such as ".-.." and the corresponding letter "L". The questions that arise are: When is the neuroplasticity adequate so that the learner can read

Table 1: Stages of learning (derived from [19])

#	Stage	Knowledge	Application
1	Acquiring	Declarative	Slow
2	Consolidating	Decl. & Procedural	Moderate
3	Tuning	Procedural	Fast

".... . .-.. .-.. —" and natively perceive it as "HELLO" and vice versa, and how can we optimize the learning process?

It has been shown that system design plays significant role in the process of learning such mappings. More than a billion users have become proficient in typing input such as "4433555555666" for encoding the message "hello" and achieved notable speeds (20 words per minute) within their first 400 min. of experience using multi-tap text messaging in 4×3 cellphone keypads [25]. Within the same period of exposure even higher speeds (26 words per minute) can be achieved by users who learn how to encode the same message as "11001 11000 00111 00111 11111" using 5-bit binary mappings by performing the corresponding 5-finger gestures [40]. These findings are in agreement with studies in computer science education that have shown that novices can learn programming languages such as Java, which borrows several keywords from English with similar effort as programming languages that use random ASCII keywords [39].

The learning of all of the above mappings can be interpreted using the theory of learning in three stages by Kim at al. [19], which is based on the theory of Fitts [11], Anderson [1], Rasmussen [33] and VanLehn [42]. According to this theory, learning is achieved in three stages, as outlined in Table 1. In the first stage, the learner acquires knowledge about a particular mapping (e.g. the instructor/instructional material says that "a != b" is the syntax for testing inequality). The acquired declarative knowledge is consolidated in the second stage. For example, the learner is reading scripts that contain various instances of the aforementioned syntax and is trying to understand their logic and replicate them. In the final stage, the learner can use the syntax fluently due to procedural knowledge that has been acquired by experience. According to Kim at al., during the third stage of learning, declarative knowledge may degrade especially with lack of use of the particular mapping. Nevertheless, the learner can still perform the task if all the knowledge is proceduralised (hence used natively) or is available in the environment and thus not forgotten with time [19]. For example, if the learner reads in the text editor the script: "a != b" but fails to understand its meaning due to declarative memory failure, the text editor (i.e. the system) should provide the missing declarative knowledge: "a is not equal to b".

In all of the aforementioned examples the mapping is between an unknown domain (i.e. the subject of learning), such as programming syntax, Morse code, multi-tap sequences, binary finger gestures, etc. and a known domain, such as a description of the instructed actions in the learner's native language (for example German), or characters from the learner's native language (for example the Latin alphabet). It is known that people establish a one-to-one correspondence between the elements of the two domains by unconsciously building a mapping according to the alignment of relational structure [9],

known as relational isomorphism [15]. According to Gentner's systematicity principle [15], the deeper the corresponding relational structure shared by the two domains, the more likely it is that people will construct a relational mapping.

In the case of programming languages, the domain elements are the individual tokens that compose syntactically correct source code, while in the case of natural languages, the domain elements are the individual words or phrases with self-contained meaning. The systematicity principle can be applied by establishing a strong one-to-one relational structure between the two domains [9, 15]. Since the programming language domain cannot be altered (i.e. the syntactic rules have been previously established) the only available degree of freedom of the aforementioned relational structure lies in the formation of the natural language counterparts. For example, the syntax "a . b" can be mapped to the English phrase "b of a" or to the alternative phrase "a's b". The latter, however, corresponds to a stronger one-to-one relational structure. Hence, according to this theory, it is more likely that learners will develop a stronger declarative memory by mapping "a . b" to "a's b" [9, 15].

Therefore, in order to reinforce learning, the system (i.e. the text editor) should be able to support the learner's declarative memory [19] by producing one-to-one mapping between the programming language tokens and the corresponding words or phrases from a natural language, also known as straight alignment [43]. In such case, the tokens of programming languages become the medium for encoding procedural information no differently from encoding letters with Morse code or computer tasks with keyboard shortcuts. When a user learns to close an application using Command-W (Mac) or Alt-F4 (Windows), the initial declarative knowledge becomes procedural knowledge with use. It is well-known that procedural memory is more robust than declarative memory [19]. Hence, if a student learns that a particular combination of characters, such as "!=" (in C-like languages), "/=" (in Fortran 90), or "~=" (in Matlab), is the shortcut for the phrase "is not equal to", this knowledge can remain robust against decay when it is proceduralized through the shortcut interaction.

The following section demonstrates that the design of such system is feasible by establishing straight alignment between computer code in ECMAScript (JavaScript) syntax and various natural languages.

3 SYSTEM DESIGN

The framework presented in this paper requires a computer code to be mapped to a natural language using specific theory-driven constraints, as discussed in section 2, that enable interactive replacement of each token in a computer script. It should be noted that the verb "translate" does not properly describe this mapping process, as it implies that the result should be a flawless text in a natural language, which, however, may not be possible due to the theoretical constraints. In this section these constraints are defined and are employed for the design of the proposed system using 7 natural languages.

3.1 Mapping constraints

In statistical machine translation, word or text alignment is the process of matching the words or phrases of equivalent meaning between two versions of the same text in different languages [20]. For a detailed discussion on the text alignment problem the reader is referred to [43]. It is possible in translations of short and simple sentences to have straight alignment between the corresponding texts. For example, the phrase "It is good" and its German translation "Das ist gut" have straight alignment because the first words of each phrase have equivalent meaning, and the same is for the second and third words.

In the case of mapping computer code to natural language, there are different possible alignments that can be produced. For example, the natural language could simply follow the syntax of the computer code by "reading" it symbol after symbol, such as "player dot trophies equals five", which corresponds to the code "player.trophies=5;" (using [5]). The goal of this translation is not to explain the logic of each command as in pseudo-code but to create a straightly aligned spoken version of the code.

On the other hand, a translation can be in the form of explanatory statements that describe the logic of each command, such as "substitute 5 for trophies", which corresponds to the code "trophies=5;" (using [13]). In this case, the alignment between the two texts is not straight, because the translated words follow the proper order of the natural language and not the order of the programming tokens. Furthermore, the symbol "=" was translated by [13] as "substitute" and not as "equals", which was the translation suggested by [5].

It is evident from the previous examples that the characteristics of the translation in need depend on the nature of the application. For the educational application discussed in this paper, the derived mappings should satisfy the following constraints:

(1) The number of aligned words, phrases, or symbols in the mapped text must be equal to the number of tokens in the corresponding computer code. This constraint will enhance relational isomorphism by creating one-to-one mapping [15].
(2) The alignment between the computer code and the natural language should be straight. This constraint will enforce the creation of a strong shared relational structure between the two mapped domains [9].
(3) The composed natural language text should be as grammatically correct as possible and contain simple phrases that describe the logic of the underlying code. With this constraint the system will contain the necessary declarative knowledge [19].

The first two constraints will enable the interaction of emoticon-like typing within a source code editor as shown previously in [4]. More specifically, due to the first condition, each programming token can be replaced by a word, phrase, or symbol that will cover the original token in a similar manner as typing emoticons through keyboard shortcuts in social media. Furthermore, the sequence of replacements will be arranged in the same order as the corresponding tokens according to the second condition. This will reinforce learning of the underlying computer programming syntax as discussed in section 2 and will also enable users to traverse and edit the original code by performing the same actions to the translated text.

Finally, the third constraint will allow users to build one-to-one connections between computer code and its logic, as described by the natural language replacements. Subsequently, syntax errors

Figure 1: Example of text replacements in English (En), Chinese (Zh), German (De), Greek (El), Italian (It), Spanish (Es), and Turkish (Tr) in alphabetical order. The corresponding JavaScript (JS) tokens and English replacements are shown on the top.

in the computer code will be reflected in the translated phrases. Erroneous translations can work as a hint that helps users identify and correct their mistakes in the computer code.

3.2 Practical implications

The aforementioned theory-derived constraints were employed in order to map the unique tokens in JavaScript to words or phrases from the following 7 natural languages: English, Chinese, German, Greek, Italian, Spanish, and Turkish. The selection of the languages was made so that the three most popular languages are included (Chinese, Spanish, and English) and others based on the availability and willingness of faculty and students from various countries to participate in this project. The mapping to each language was led by faculty and post-doctoral fellows from Computer Science and relevant departments, who were also assisted by their students. All participants were native speakers of the corresponding language and had expertise in computer programming. The list of the previously described constraints was given to each team along with the list of all unique tokens and syntax patterns in JavaScript that they had to map.

Figure 1 shows an example of computer code along with the corresponding mappings as defined by the participating groups. In all cases there is a one-to-one mapping to the corresponding programming tokens. However, this is not always possible for every expression in a given computer programing language and a corresponding natural language, and there are several practical implications that were noted as a result of this exercise and are discussed in this section.

3.2.1 Straight alignment. Straight alignment is not always feasible between computer code and a natural language due to strict syntax constraints in some natural languages, such as the placement of the verb within a sentence. Table 2 summarizes the results from the alignment of JavaScript tokens to the 7 aforementioned natural languages. Straight alignment was possible across all participating natural languages for several types of tokens, such as assignment operators, logical operators, and control flow statements. However, straight alignment was not possible for relational operators (in Turkish), as well as object-oriented expressions (in Greek, Italian, and Spanish). Details on these specific issues are provided in the rest of this section.

Table 2: Feasibility of straight alignment

Type of tokens	Examples	Straight align.
assignment operators	=, +=	100%
relational operators	==, !=	85.7%
logical operators	||, &&	100%
control flow statements	if, while	100%
object-oriented expressions	a.b	78.5%

3.2.2 Simple alignment graph. In the example of Fig.1 the keyword "var" has been replaced by "Let" in English and a similar expression "Deje que" in Spanish, an equivalent expression that means "Define" (Shēngmíng) in Chinese and (Tanimla) in Turkish, and "Let's define" (*Ας ορίσουμε*) in Greek. By observing the text replacements, it is evident that it is not always possible to establish one-to-one alignment between one programming token, such as "var", and a single word in a given natural language, such as "Let". In this case the users will establish in their mental models an association between a programming token and a phrase. This association can be enhanced by avoiding large phrases, hence by maintaining simplicity in the established alignment graph.

Furthermore, in the case of German, although straight alignment was successfully achieved using grammatically correct mappings, it was done by avoiding subordinate clauses (that would inconveniently require the verb to be placed at the end of the sentence) through the use of "Definiere:" in the beginning of the sentence. Similarly, in Spanish, a slightly better translation "Sea flag igual a verdadero" was avoided, because this would map the symbol "=" to the phrase "igual a", which does not provide the correct declarative information regarding the assignment operator, hence the mapping to the word "sea" was preferred instead as shown in Fig.1.

3.2.3 Translating logic not keywords. Although most programming languages include keywords in English such as "if", "for", and "this", they should not be directly translated to a natural language in order to satisfy the previously defined constraint (3). The underlying logic should be used instead in their translation. For example, when translating computer code to English, the token "for" could be aligned with "iterate" or "iterate for", and the token "this" could be aligned with the phrase "this object", etc.

3.2.4 Translating meaning not symbol names. The various symbol-based tokens such as "=", "==", and "===" should not be aligned with the corresponding symbol's name but with a word or phrase that explains their meaning. In this example "be:" or "become:" should be used as the replacement of the single equal sign instead of its name, i.e. "equal sign". Similarly, the double equal sign could be replaced by "is equal to" and the triple equal sign could be aligned with "is strictly equal to".

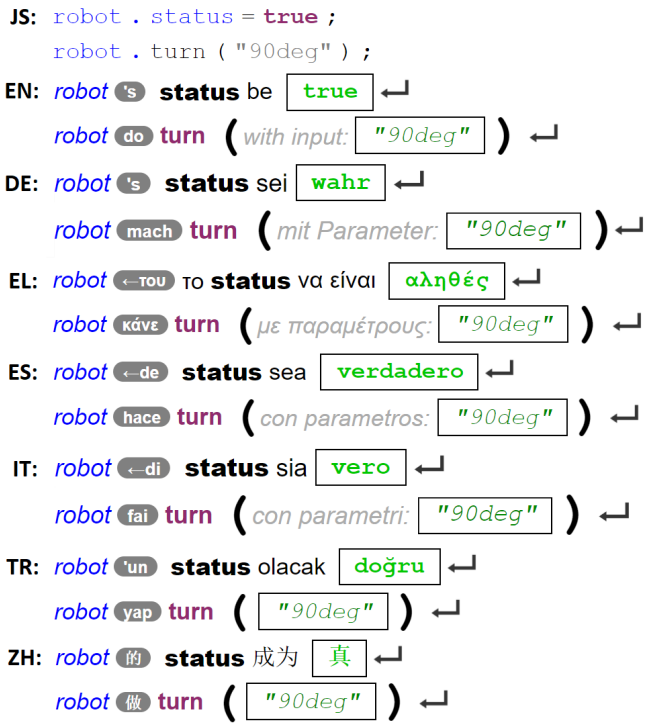

Figure 2: Context-based text replacement example that shows problematic straight alignment in the use of genitive case in languages with noun declension. (In alphabetical order except from JavaScript and English, which are shown on the top.)

3.2.5 Use of articles. In programming code the names of variables are typically nouns (or composite phrases that contain nouns) defined by the programmer. In several natural languages the nouns have gender properties and may be accompanied by an article in the gender of the noun. In this case, the variable names should also be accompanied by a default article for non-native words, such as neuter, in order to compose grammatically correct code translations. In the example of Fig. 1 the variable name "flag" is aligned with "$\tau o\ flag$" in the Greek translation, where "τo" is the article that precedes the non-native (not translated) noun "$flag$".

3.2.6 Genitive case. In several object-oriented programming languages the relation between objects and their properties is denoted by the use of a symbol between them, such as "." in the expression "object.property". In natural languages this corresponds to the so-called genitive case. In English such relationship can be expressed either as "property of the object" or "object's property". In order to satisfy the previously defined constraint (2), the latter phrase must be used as it corresponds to a straight alignment with the computer code, in which "." is aligned with "'s" (Fig. 2). Similarly, in Chinese straight alignment can be achieved by using the logogram "de", and in Turkish by using "'um", as shown in the same figure. However, straight alignment is not possible in the Greek, Italian, and Spanish translations, as the words "τov", "di", and "de" respectively indicate ownership in the opposite direction, i.e. it should be used in front of the noun "robot." In this case, all three constraints cannot be satisfied simultaneously without introducing an artificial directional symbol, as shown in the corresponding examples of Fig. 2. It should be noted that "'s" is used in German language in certain phrases (although not always) and for this reason it was conveniently used in the mapping of the token ".", as in the English translation.

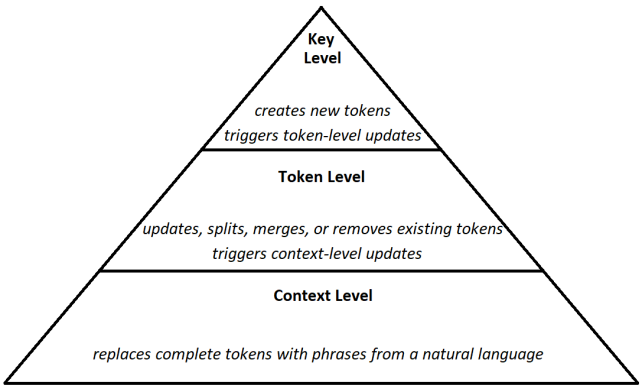

Figure 3: Illustration of the proposed three-level active tokenization method.

3.3 Implementation

The computer code mapping framework established in the previous section can be implemented using an active tokenization algorithm that identifies the individual programming tokens and replaces them with their translation in a natural language as soon as they are typed by the user.

In this algorithm, the typed text is not treated as a string of characters but as a sequence of individual token elements. Each token element is a data structure that consists of the type of the token (such as name, number, string, etc.), the corresponding computer code, and its textual replacement and can be either in edit or rigid mode. By default all tokens are in rigid mode and show their textual replacements (as in Figs. 1 and 2). Only one token at a time can be in edit mode and shows its underlying code that can be edited by the user. Insertion, deletion, merging, or splitting of tokens can be performed by the active tokenization algorithm that consists of three layers as shown in Fig. 3.

3.3.1 Key Level. The top level is the entry point of the algorithm, which is triggered every time the user types a new character (including control characters, such as backspace). The top level is responsible for initializing the current token when its first character is typed (function *initializeToken* in Alg. 1). For example, if "a" is typed in a new token, it is initialized as a token of type "name." While, if "5" is typed instead, it is initialized as a token of type "number", etc.

3.3.2 Token Level. The second level is responsible for updating, splitting, merging, and deleting existing tokens based on the new character being typed and the existing content of the current token (function *appendCharacters* in Alg. 1). For example, if "a" is typed at the end of an existing token that contains the number "5", it

Algorithm 1: Active Tokenization Algorithm

```
Input: List of tokens l, current token element t, new typed
       character ID c, caret position i
Output: The updated list of token elements l
p=getPreviousToken(l,t);
if length(t.code)=0 then
    if isBackspace(c) then
        editToken(p);
        removeToken(l,t);
    else
        initializeToken(t,c);
    end
else
    if length(t.code)=i then
        appendCharacters(t,c);
    else
        if i=0 and isBackspace(c) then
            editToken(p);
            removeToken(l,t);
            appendCharacters(p,t.code);
        else
            newCode=modifiedCode(t.code,c,i);
            resetToken(t);
            appendCharacters(t,newCode);
        end
    end
end
```

Table 3: Examples of context-based replacements

Token	English replacement	When followed by
=	be	a value
=	be the following	the name "function"
=	be the following array	"["
=	be the following object	"{"
.	's	a name
.	do	a name followed by "("

will trigger the splitting of the token into two tokens "5" and "a", because the latter is not allowed to be appended to the former.

3.3.3 Context Level. The completion/modification of existing tokens triggers the third level that is responsible for rendering the tokens that go into rigid mode based on their context. Table 3 shows examples of context-based replacement rules for the tokens "=" and ".". For instance, the token "." can be replaced by the English word "do" if followed by the call of a method as shown in the example of Fig. 2, which reads "robot do turn (with input: 90 deg)". In the same example, the token "(" has been replaced by "(with input:". This context-based replacement is triggered when the left parenthesis follows a name token, such as "turn" in this example.

The proposed method was implemented in JavaScript as a web-based source-code editor that allows the user to choose the language of preference for the text replacements (between the 7 available languages discussed in this paper). The editor allows the user to interact with the typed code in a "playful" manner that resembles the typing of emoticons in social media [4]. In this case, each token is being replaced by a text instead of an icon or a set of icons as shown in [2]. An advanced example of a complete computer program rendered by the proposed source code editor using text replacements in English is shown in Fig. 4. It should be noted that the user in this example has typed the JavaScript code shown in Fig. 4a. However, while typing each token has been immediately replaced by the corresponding textual replacement as shown in Fig. 4b. Due to space limitations this paper does not provide the corresponding examples in other languages. The readers are encouraged to visit the website of the project (provided in section 6) and type the script in Fig. 4a using their preferred set of token replacements in other languages.

By observing Fig. 4, several correspondences between code and English can be identified. For example, the tokens "{", "}", "==", "+=", and "//" have been replaced by "begin", "end", "is equal to", "be increased by:" and "Note:" respectively. Furthermore, the token "=" has been replaced by "be:" or the phrase "be the following" when precedes the keyword "function". The latter has been replaced by "procedure:".

The developed source code editor was tested by a small-scale study that is presented in the next section.

4 PILOT USER STUDY

A pilot study was performed in order to assess the effectiveness of the proposed TEI as an environment that can assist beginners practice computer coding in comparison to conventional TEIs. The tested hypothesis was that the proposed environment with text-based token replacements improves students' learning outcome in comparison to a conventional environment without token replacements.

4.1 Study design

As discussed previously in section 1, learning computer programming is a complex process that depends on several parameters, such as the learner's competency in problem solving, prior exposure to procedural thinking (see detailed discussion in [22]), as well as general attitude towards STEM areas, and many other factors including basic knowledge of English language [28]. In order to avoid unintentional bias from all of these external factors, 8 different measures were taken in the design of this study. More specifically, various measures were taken during the environmental setup and several other constraints were imposed to the user enrollment procedure and are presented in detail in this section.

4.1.1 Subject Enrollment. Although the vision of the proposed project is to assist learners from all nationalities without any discrimination, it is expected that non-English speakers could potentially benefit more when English-based programming tokens are replaced with explanatory phrases in their native language. This is a clear advantage of the proposed method over traditional TEIs. Hence any comparative study using non-English speakers will introduce a natural bias in favor of the proposed method.

4a) What is typed in the proposed editor.

```
var Fibonacci = function ( )
{
  // This function prints the first 20 numbers of the sequence
  var number = 0 , previous1 = 0 ,
  previous2 = 0 , i = 1 ;

  while ( i <= 20 )
  {
    // We calculate the next number in the sequence
    if ( i == 1 ) number = 1 ;
    else if ( i == 2 ) number = 1 ;
    else number = previous1 + previous2 ;

    // We print out the number
    println ( number ) ;

    // Finally, we shift the values
    i += 1 ;
    previous2 = previous1 ;
    previous1 = number ;
  }
}
```

4b) What is shown instead.

```
Let Fibonacci be the following procedure: ( without input )
begin
  Note: This function prints the first 20 numbers of the sequence
  Let number be: 0 , previous1 be: 0 ,
   previous2 be: 0 , i be: 1 ↵

  while ( i is less than or equal to 20 )
  begin
    Note: We calculate the next number in the sequence
    if ( i is equal to 1 ) number be: 1 ↵
    else if ( i is equal to 2 ) number be: 1 ↵
    else number be: previous1 + previous2 ↵

    Note: We print out the number
    println ( with input: number ) ↵

    Note: Finally, we shift the values
     i be increased by: 1 ↵
     previous2 be: previous1 ↵
     previous1 be: number ↵
  end
end
```

Figure 4: An advanced example that shows the English-replaced JavaScript code of a program that prints the first numbers of the Fibonacci sequence. The two panels show the code that was typed in the proposed editor (top) and what was shown instead (bottom).

For this reason, in order to create the conditions for an unbiased study, it was decided to disadvantage the proposed method by allowing only native English speakers to participate in this study among other constraints which are listed below:

(1) Students whose native language was not English were not eligible to enroll in this study.
(2) Students with prior experience in computer coding using TEIs were not eligible to enroll in this study. This constraint was necessary in order to ensure (to the extend possible) that all users had similar prior knowledge on the subject before the study. This was determined with a pretest questionnaire that was given to the subjects upon enrollment.
(3) All subjects were enrolled in schools in the same region of the same country at the time of this study. The reason for this constraint was to avoid unnatural differences caused by significant variations in school curricula and state standards across larger geographical regions.
(4) All subjects were equally distributed in the control group (conventional TEI) and study group (proposed TEI) based on their gender, school level (grade), and school (current enrollment). Such even distribution was necessary in order to avoid any bias related to these factors. The corresponding demographic information was also collected upon enrollment as part of the pretest questionnaire.

4.1.2 Environmental Setup. Additionally, the following measures were taken to avoid bias due to the experimental setup:

(1) The same programming language was used (taught, practiced, and tested) by both study and control groups during this experiment.
(2) The same instructional material was used in both study and control groups (number of slides, content of slides, given examples, and practice questions).
(3) The same amount of time was given to both study and control groups for instruction, practice, and test.
(4) The same compiler was used by both control and study groups in order to ensure that the error messages had exactly the same information and verbiage.

4.2 Study execution

The experimental setup described in section 4.1 was implemented and executed as a short-term pilot study using 88 students of ages between 10-15 years old, who volunteered to participate. All subjects were enrolled in three schools in Alachua county, Florida, were native speakers of English, and had no prior experience in TEI environments for source code editing or programming syntax in general. The study took place in the computer facilities of the Digital Worlds Institute at the University of Florida in the period between March-July 2017 on different days based on the availability of each participating school. Before the study, the enrolled subjects were split into two similarly sized groups of similar age, gender, and school mix as described in section 4.1.

One of the two groups was used as the control group (N=46), and the other one was the study group (N=42). During the study, the students of each group were exposed to the same basic programming principles, namely variables and conditionals using JavaScript syntax for approximately 45 minutes. During this time the students

Table 4: Levels of student success per metric

Method	Metric 1	Metric 2	Metric 3	Metric 4
Nat.Language	52.38%	97.62%	80.95%	90.48%
Comp. Code	17.39%	93.48%	50.00%	54.35%
Increase	201.19%	4.43%	61.90%	66.48%
χ^2	11.9624	0.8675	9.2180	14.0898
p	0.00054	0.35164	0.00239	0.00017

of each group practiced using different text editors: the study group used the proposed editor with natural language replacements in English, and the control group used a conventional source code editor without replacements. Both editors used the same compiler (JavaScript compiler of Google Chrome) with the same error messages as explained in section 4.1. At the end of the study all students had to complete the same programming assignment in JavaScript with pen and paper based on the material that was covered in the study. The reason for performing the final assessment in pen-and-paper fashion was to test if the students could recall the syntax and logic of the programming language without the assistance of the compiler or the proposed natural language replacements. The programming assignment involved the declaration of a few variables and the use of conditionals to test their values based on the material covered in the study.

The student solutions to the programming assignment were quantitatively evaluated for accuracy using four different metrics: 1) The submission had no logical or syntax errors, 2) The variable declarations had no errors, 3) The conditionals had no errors, 4) There were no other errors (examples of observed errors in this category are: "var if" followed by a correct conditional, incorrect variable reference as "var name" inside a conditional, unbalanced braces, etc.)

Table. 4 shows the percentages of student success using the four established metrics. According to the data in the first column (percentage of submissions without errors), the students who practiced programming using the proposed editor performed better compared to the students who used conventional editors. More specifically, the use of natural language replacements increased the success of the students by a factor of 3, i.e. the number of students who successfully completed the assignment tripled (52% vs. 17%).

Overall, the detailed percentages from all metrics indicate that the proposed method improved the student success, especially in conditionals (metric 3) and other errors (metric 4). More specifically, the percentage of success was increased by 4.4%, 61.9%, and 66.5% in terms of variable declarations, conditionals, and other syntactical structures respectively.

In order to evaluate the statistical significance of each finding, a chi-square test of independence was performed for each metric. More specifically, the null hypothesis was that the two categorical variables, namely the employed TEI and the student performance in a particular metric, are independent. The alternative hypothesis was that the two categorical variables are dependent, i.e. the student performance depends on the employed TEI. The calculated critical values (χ^2) for each metric are shown in table 4 as well as the corresponding probability values (p). Based on the calculated values, the null hypothesis is rejected ($p < 0.01$) for metrics 1,3,4, while there is no significant statistical difference observed in metric 2.

5 DISCUSSION

After studying the behavior of users while interacting with the proposed educational tool, the following observations were made:

(1) The proposed environment is not a simple text editor but a space for active experimentation, in which the users type and expect an immediate reaction. Through the continuous alternation of action (typing) and cognition (observation of the token replacement), the learners pass through the three learning stages [19]. This process leads to abstract conceptualization, i.e. reinforces learning through a try-and-error environment. For example, if a student intends to test equality in the computer code, he or she expects to see a text replacement "is equal to." But if the text replacement suddenly appears as "be:" instead, it indicates an erroneous token that was typed by the user. Subsequently, students who practiced using the proposed editor during our study had improved learning outcomes.
(2) One of the key advantages of this framework is that it does not constitute an intermediate learning step but a terminal one, as the users learn to type correct code in the syntax of a professional programming language.
(3) The proposed editor promotes good coding habits with regards to the names of properties and methods defined by the students in their code. The students tend to use nouns and verbs for the properties and methods of their objects, so that the translated code will read better in conjunction with "'s" and "do" respectively.

However, due to the narrow scope of this paper there are several issues and limitations that need to be addressed in the future, some of which are listed below:

(1) The study presented in section 4 was pilot in nature, hence it was limited in duration as well as number of participants. The results only indicate that there was a positive effect in the user's first experience with TEIs. Although it is known that the outcome of the first experience has a substantial and lasting effect on participants' subsequent behavior [37, 41], a long-term study should be undertaken in the future.
(2) Section 3 did not investigate how different verbiage could be used for different target audiences. More effective mappings could possibly be developed for specific populations, such as college educated adults, adults without college education, high school students, middle school students, etc. This is an interesting topic that should be addressed in a future publication.
(3) This framework has not been tested on non-English speakers in order to avoid introducing bias. Investigating how programming languages are biased towards English, and how this affects learning is an interesting topic that has been discussed in the past [28] and could be further investigated using the proposed framework.
(4) The proposed framework was demonstrated for the syntax of one programming language only (JavaScript). Identifying

mappings for different syntax, such as for Python, R, or other programming languages is a critical topic that should be addressed in the future.

(5) Finally, this paper did not address the possible effects from coupling the proposed framework with other methods such as BCI followed by the proposed TEI or TUI followed by the proposed TEI as it was recently assessed in a different study [3].

6 DISSEMINATION

In order to increase the dissemination of the proposed method and its impact on the widest possible audience, the tool presented in this paper has been made available online at the following address: https://research.dwi.ufl.edu/projects/emoticoding/.

The tool includes a source-code editor with active tokenization and various sets of token replacements including plain JavaScript, English, Chinese, German, Greek, Italian, Spanish, and Turkish. With the assistance of early adopters new sets are under development for Korean, and Indonesian.

7 CONCLUSIONS

In this paper a novel method was presented for assisting beginner programmers in their first exposure to source code editors. The contributions of this paper were threefold: a) a framework for aligning computer code to natural languages was presented and demonstrated using seven structurally different languages, b) an implementation of the proposed framework was developed using an active tokenization algorithm, which interactively replaces individual programming tokens with words or phrases from a natural language, c) the results from a pilot study were discussed, which indicated that the proposed method improved the students' performance. In the future, text replacements from more languages will be included in order to make the proposed tool available to a wider audience. Finally, a large scale study will be performed in order to assess the effect of the proposed method during a year-long curriculum in different countries using non-English students.

ACKNOWLEDGMENTS

This project was in part funded by the University of Florida College of the Arts.

The author would like to thank Marcial Abrahantes from the University of Florida, Monica Berti and Giuseppe Celano from the University of Leipzig, Kryscia Daviana Ramírez Benavides from the University of Costa Rica, Muhammet Demirbilek from the Suleyman Demirel University, Liyan Liu from Zhengzhou University, and their students for assisting with the translations and providing insightful comments regarding the code alignment in their respective languages.

Finally, the author would like to express his appreciation to the students who voluntarily participated in this study and their parents or legal guardians for their consent as well as enthusiasm for this project.

REFERENCES

[1] J.R. Anderson. 1982. Acquisition of cognitive skill. *Psychological Review* 89, 4 (1982), 369–406.

[2] A. Barmpoutis. 2018. Integrating algebra, geometry, music, 3D art, and technology using emoticoding. In *Proceedings of the 8th IEEE Integrated STEM Education Conference*. 30–33.

[3] A. Barmpoutis and K. Huynh. 2019. Name Tags and Pipes: Assessing the Role of Metaphors in Students' Early Exposure to Computer Programming Using Emoticoding. *Advances in Human Factors in Training, Education, and Learning Sciences* 785 (2019), 194–202.

[4] A. Barmpoutis, K. Huynh, P. Ariet, and N. Saunders. 2018. Assessing the Effectiveness of Emoticon-Like Scripting in Computer Programming. *Advances in Intelligent Systems and Computing* (2018), 63–75.

[5] A. Begel and S. Graham. 2005. Spoken programs. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. 99–106.

[6] N. Brown, A. Altadmri, and M. Kölling. 2016. Frame-Based Editing: Combining the Best of Blocks and Text Programming. In *International Conference on Learning and Teaching in Computing and Engineering*. 47–53.

[7] Sanja Maravić Čisar, Robert Pinter, and Dragica Radosav. 2011. Effectiveness of program visualization in learning java: a case study with jeliot 3. *International Journal of Computers Communications & Control* 6, 4 (2011), 668–680.

[8] L. de Oliveira Brandão, Y. Bosse, and M. A. Gerosa. 2016. Visual programming and automatic evaluation of exercises: An experience with a STEM course. In *Frontiers in Education Conference (FIE)*. 1–9.

[9] V. Diehl and D.D. Reese. 2010. Elaborated metaphors support viable inferences about difficult science concepts. *Educational Psychology* 30, 7 (2010), 771–791.

[10] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. 13–22.

[11] P.M. Fitts. 1964. *Perceptual-motor skill learning*. Academic Press, 243–285.

[12] J. Freeman, B. Magerko, and R. Verdin. 2015. EarSketch: A web-based environment for teaching introductory computer science through music remixing. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 5–5.

[13] H. Fudaba, Y. Oda, K. Akabe, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. 2015. Pseudogen: A Tool to Automatically Generate Pseudo-Code from Source Code. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 824–829.

[14] F. J. García-Peñalvo, A. M. Rees, J. Hughes, and et al. 2016. A survey of resources for introducing coding into schools. In *Proceedings of the 4th International Conference on Technological Ecosystems for Enhancing Multiculturality*. 19–26.

[15] D. Gentner. 1983. Structure mapping: A theoretical framework for analogy. *Cognitive Science* 7 (1983), 155–170.

[16] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*. 35–44.

[17] L. HaoWen, L. Yin, and L. Wei. 2013. Design of online pseudo-code editor and code generator. In *IEEE Conf. Anthology*. 1–3.

[18] A. Joelc and W. Andrew. 2012. 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*. 643–64.

[19] J. W. Kim, F. E. Ritter, and R. J. Koubek. 2013. An integrated theory for improved skill acquisition and retention in the three stages of learning. *Theoretical Issues in Ergonomics Science* 14, 1 (2013), 22–37.

[20] P. Koehn. 2009. *Statistical Machine Translation*. Cambridge University Press.

[21] N. Kushman and R. Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *North American Chapter of the Association for Computational Linguistics (NAACL)*.

[22] E. Lahtinen, K. Ala-Mutka, and H. M. Järvinen. 2005. A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin* 37, 3 (2005), 14–18.

[23] T. Lei, F. Long, R. Barzilay, and M. Rinard. 2013. From natural language specifications to program input parsers. In *Association for Computational Linguistics (ACL)*.

[24] H. Liu and H. Lieberman. 2005. Metafor: Visualizing stories as code. In *10th International Conference on Intelligent User Interfaces*. 305–307.

[25] Kent Lyons, Thad Starner, Daniel Plaisted, James Fusia, Amanda Lyons, Aaron Drew, and EW Looney. 2004. Twiddler typing: one-handed chording text entry for mobile phones. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 671–678.

[26] D. J. Malan and H. Leitner. 2007. Scratch for Budding Computer Scientists. *ACM SIGCSE Bulletin* 39, 1 (2007), 223–227.

[27] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *Program Comprehension (ICPC) 2013 IEEE 21st International Conference on. IEEE*. 23–32.

[28] J. Moreno-León and G. Robles. 2015. Computer programming as an educational tool in the English classroom a preliminary study. In *IEEE Global Engineering Education Conference (EDUCON)*. 961–966.

[29] C. Morgado and F. Barbosa. 2012. A structured approach to problem solving in CS1. In *Proceedings of the 17th ACM annual conference on Innovation and*

technology in computer science education. 399–399.
[30] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 574–584.
[31] I. Ouahbi, F. Kaddari, H. Darhmaoui, A. Elachqar, and S. Lahmine. 2015. Learning basic programming concepts by creating games with scratch programming environment. *Procedia-Social and Behavioral Sciences* 191 (2015), 1479–1482.
[32] Thomas W. Price, Neil C.C. Brown, Dragan Lipovac, Tiffany Barnes, and Michael Kölling. 2016. Evaluation of a Frame-based Programming Editor. In *ACM Conference on International Computing Education Research*. 33–42.
[33] J. Rasmussen. 1986. *Information processing and human-machine interaction: an approach to cognitive engineering*. Elsevier.
[34] C. Reas and B. Fry. 2014. *Processing: A Programming Handbook for Visual Designers and Artists*. Cambridge: MIT Press.
[35] T. Sapounidis and S. Demetriadis. 2016. Educational Robots Driven by Tangible Programming Languages: A Review on the Field. In *International Conference EduRobotics 2016*. 205–214.
[36] Lara Schlaffke, Alexander Leemans, Lauren M Schweizer, Sebastian Ocklenburg, and Tobias Schmidt-Wilcke. 2017. Learning morse code alters microstructural properties in the inferior longitudinal fasciculus: a DTI study. *Frontiers in human neuroscience* 11, 383 (2017), 1–9.
[37] H. Shteingart, T. Neiman, and Y. Loewenstein. 2013. The role of first impression in operant learning. *Journal of experimental psychology. General* 142, 2 (2013), 476–488.
[38] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proc. International Conference on Automated Software Engineering (ASE)*. 43–52.
[39] A. Stefik and S. Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Transactions on Computing Education* 13, 4 (2013).
[40] Adrian Dan Tarniceriu, Pierre Dillenbourg, and Bixio Rimoldi. 2013. The effect of feedback on chord typing. In *Proceedings of The Seventh International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*.
[41] M. P. Uysal. 2014. Improving First Computer Programming Experiences: The Case of Adapting a Web-Supported and Well-Structured Problem-Solving Method to a Traditional Course. *Contemporary Educational Technology* 5, 3 (2014), 198–217.
[42] K. VanLehn. 1996. Cognitive skill acquisition. *Annual Review of Psychology* 47 (1996), 513–539.
[43] Jean Véronis. 2013. *Parallel Text Processing: Alignment and use of translation corpora*. Vol. 13. Springer Science & Business Media.
[44] D. Wang, L. Zhang, C. Xu, H. Hu, and Y. Qi. 2016. A tangible embedded programming system to convey event-handling concept. In *Proceedings of the TEI'16: Tenth International Conference on Tangible, Embedded, and Embodied Interaction*. 133–140.
[45] D. Weintrop and U. Wilensky. 2017. Between a Block and a Typeface: Designing and Evaluating Hybrid Programming Environments. In *ACM Interaction Design Conference*. 183–192.
[46] B. Wohl, B. Porter, and S. Clinch. 2015. Teaching Computer Science to 5-7 year-olds: An initial study with Scratch, Cubelets and unplugged computing. In *Proceedings of the Workshop in Primary and Secondary Computing Education*. 55–60.